Lightweight Formal Methods for Component-based Software Development

by

Michihiro MATSUMOTO

submitted to
Japan Advanced Institute of Science and Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Supervisor: Professor Kokichi FUTATSUGI

School of Information Science
Japan Advanced Institute of Science and Technology

March 22, 2002

Abstract

In the thesis, we discuss (1) lightweight formal methods for component-based software called *LFMB* and *LFME*, (2) the component-based software development using LFMB or LFME called *CBDL*, and (3) the support tools of CBDL. Recently, component-based software development has gained in popularity. In the development, firstly we prepare a component library whose components produce basic functionalities. Then, by selecting components from the component library and by combining them using connectors, component-based software is developed. We call the parts of the software combining the components *connectors*. The reason for the popularity is that it can increase software productivity. To increase software productivity, components must be reused. The obstacles of component reuse are (a) a lack of a consensus about component usage between component developers and software developers, i.e. component users and (b) an architectural mismatch. We developed CBDL to eliminate the obstacles. CBDL is based on the Catalysis approach. The former obstacle is eliminated by specifying business models and component specifications by using UML diagrams with OCL descriptions and verifying consistency in the UML diagrams. The former technique is an idea of the Catalysis approach. The latter technique, i.e. LFMB and LFME, is a contribution of the thesis. Lightweight formal methods are formal methods such that they have target problems and they use simple logic that the verification of the problems can be executed automatically. The consistency verification in the UML diagrams are the target problems of LFMB and LFME. Behavioral logic and equational logic are the simple logic of LFMB and LFME, respectively. So, LFMB and LFME are lightweight formal methods. Because LFMB and LFME include automated verification methods of the consistency, component developers and software developers who are not familiar with formal methods can get the benefit of the verification by using the support tools of CBDL. The latter obstacle is eliminated by selecting *tree architecture* that we developed. We can regard the UML diagrams as specifications specified by *a language for programming-in-the-large*. So, in CBDL, moreover, we generate component-based software from the UML diagrams by combining the connectors specified in the UML diagrams and components of a component library. Note that the above consistency verification guarantees the correctness of the connectors. Because the support tools of CBDL are designed as client-server type software, component developers and software developers can access to the tools through Internet at any place at any time. To summarize, by using the support tools of CBDL, we can increase component reuse and correctness of component-based software.

Acknowledgements

First of all, I am grateful to my main supervisor Professor Kokichi Futatsugi who has not only provided valuable suggestions but also shown me the right direction of my study.

In addition, I thank LDL members, especially previous Associate Professor Takuo Watanabe, previous Associate Kazuhiko Ogata, previous Associate Răzvan Diaconescu, previous Associate Akira Mori, Associate Noriki Amano, and Dr. Shusaku Iida for helpful discussions about my study.

A part of my study was carried out as the projects that were supported by grant Support program for young software researchers 99-004 and Support program for young software researchers 01-006 from Information-technology Promotion Agency (IPA) and Research Institute of Software Engineering (RISE). So, I thank Information-technology Promotion Agency and Research Institute of Software Engineering. Also, I thank the project members, Mr. Yoshihito Katayama, Mr. Takanori Nakama, and Mr. Yoshiharu Hashimoto.

Finally, I thank my employer PFU Limited, which supports my life at JAIST.

Contents

Chapter 1

Component-based Software Development and Formal Methods

In the thesis, we discuss:

1. lightweight formal methods for component-based software called *LFMB (a Lightweight Formal Method using Behavioral specification)* and *LFME (a Lightweight Formal Method using Equational specification)*,
2. the component-based software development using LFMB or LFME called *CBDL (a Component-Based software Development approach using LFMB or LFME)*, and
3. the support tools of CBDL.

In Chapter 1, firstly, we discuss:

1. what component-based software development (abb. CBD) is,
2. what formal method is, and
3. what kinds of applications of formal methods to CBD have been studied.

Then, we discuss CBDL.

1.1 Component-based Software Development

Component-based software development has gained in popularity. Many developers use component technologies, for example, JavaBeans, COM, EJB, and CORBA[35].

In the development, firstly we prepare a component library whose components produce basic functionalities. Then, by selecting components from the component library and by combining them using connectors, component-based software is developed. We call the parts of the software combining the components *connectors*.

The reason for the popularity is that it can increase software productivity. To increase software productivity, components must be reused. The obstacles of component reuse are:

1. a lack of a consensus about component usage between component developers and software developers, i.e. component users and
2. an architectural mismatch.

Because component developers and software developers usually do not know one another, it is difficult to get the consensus about component usage without a support means. The software developers can not use the components without the knowledge about component usage. So, it is difficult to reuse components without the support means.

The Catalysis approach[12] is one of the support means. It uses UML diagrams with OCL descriptions[6] to help to get the consensus. The business concepts of the target domain are specified by using UML diagrams with OCL descriptions. The behavior of components are specified by using the business concepts. So, ambiguities about component usage are eliminated by the UML diagrams. We call the specifications specifying the business concepts *business models* and the specifications specifying behavior of the components *component specifications*.

The architectural mismatch problem[13] is the problem that components cannot be combined with components which have different software architectures. For example, user interface components which use X library cannot be combined with user interface components which do not use X library. So, it is difficult to reuse components without a common software architecture.

There are many researches about software architectures[2]. In CBD, we deal not with software but a software family. *Product line architecture* [3, 8, 30, 33] whose idea at least dates back to [31] is a software architecture for a software family. In the CBD whose software architecture is product line architecture, firstly, we fix a software family, i.e. a target domain and prepare a component library of the target domain. By selecting components from the component library and by combining them, component-based software of the target domain is developed.

We use architectural description languages (abb. ADLs) to specify software architectures. The idea of ADLs dates back to "programming in the large versus programming in the small"[9]. *Programming in the small* is programming for making modules. *Programming in the large* is programming for combining modules. The idea of programming in the large led to module interconnection languages (abb. MILs)[32] and ADLs.

From ADL specifications, we sometimes generates connectors that combine modules. *Generative programming*[8] is one of the generation techniques.

1.2 Formal Methods

Formal methods are approaches of software engineering that use mathematics. The main instruments of the formal methods are specification languages. There are many specification languages, for example, *Z*[34], *B*[24], *VDL*[4], *RAISE*[18], *OBJ3*[17], and *CafeOBJ*[10]. Because specifications specified by using the specification languages are mathematical notations, we can verify properties of the specifications by using mathematical logic. Some specification languages, for example, *B*, *VDL*, *RAISE*, *OBJ3*, and *CafeOBJ* have verification systems for their languages.

The benefits of formal methods are as follows:

1. we can get clear understanding of a target software in the process of specifying the target software and

2. we can verify properties of the target software using the verification systems.

Unfortunately, the verification usually needs human help. So, it is difficult for non-specialists of the verification to get the benefit of 2. Lightweight formal methods are formal methods such that:

1. they have target problems and
2. they use simple logic that the verification of the problems can be executed automatically.

Because support tools that use the simple logic execute the verification automatically, the non-specialists can get the benefit of 2 by using the support tools.

The formal method using *Alloy* [22] is a lightweight formal methods. The purpose of *Alloy* is to propose the smallest modeling notation that is easy to read and write and can be analyzed automatically.

1.3 Applications of Formal Methods to Component-based Software Development

As we discussed in Section 1.1, the solutions of eliminating the obstacles of component reuse are as follows:

1. support means for getting the consensus about component usage and
2. selection of a common software architecture.

As we discussed in Section 1.2, formal methods can be the support means. By specifying component usage by using specification languages, we can get the consensus. In fact, we can regard UML with OCL in the Catalysis approach as a specification language. Note that the solution of the Catalysis approach is nothing other than an application of formal methods to CBD.

We can verify properties of the specifications by using the verification systems. The verification helps to get the consensus, too.

To specify the common software architecture, we can use specification languages. Note that we can regard ADLs as specification languages.

Because ADLs are specification languages, we can verify properties of ADL specifications. Consider generation of connectors from the ADL specifications. By the verification, we guarantee the correctness of the connectors.

1.4 CBDL

We developed lightweight formal methods called *LFMB (a Lightweight Formal Method using Behavioral specification)* and *LFME (a Lightweight Formal Method using Equational specification)*. *CBDL* is a Component-Based software Development approach using LFMB or LFME.

CBDL is based on the Catalysis approach. By using the Catalysis approach, we get the benefit of 1 of formal methods. But, because the verification is out of the scope of the Catalysis approach, we can not get the benefit of 2 by only using the Catalysis approach.

In the Catalysis approach, UML diagrams are specified by a number of software engineers. So, there may be inconsistencies in the UML diagrams. To eliminate the inconsistencies, consistency verification is useful. Because most of software engineers are non-specialists of the verification, the verification should be executed automatically as far as possible. So, we developed lightweight formal methods complimenting the Catalysis approach called *LFMB* and *LFME*.

One of the main idea of the Catalysis approach is that behavior of an action is specified by using changes of attributes' values. Behavioral specification has a similar idea. When we specify a behavioral specification, we regard target software as the following black box:

1. it has a state,
2. it has operators called *actions* that changes the state, and
3. it has operators called *observations* that is used for observing the state.

So, when we specify components, the actions and the attributes of the Catalysis approach correspond to the actions and the observations of the behavioral specification, respectively. Moreover, we have studied behavioral specification[25, 26, 27, 28, 29]. Therefore, we developed a lightweight formal method using behavioral specifications called *LFMB*.

In the Catalysis approach, we can regard most of OCL descriptions as equations. Therefore, we developed a lightweight formal method using equational specifications called *LFME*.

The consistency verification is the target problems of LFMB and LFME. As we will discuss in Chapter 7 and Chapter 8, we can automatically execute the consistency verification. So, behavioral logic and equational logic are the simple logic of LFMB and LFME, respectively. So, LFMB and LFME are lightweight formal methods.

In LFMB and LFME, we use *AA-trees model* as a model of objects and actions. Because the Catalysis approach is not conscious of the verification, its model may not have sufficient information about the verification. *AA-trees model* is a refinement of the model that the specifiers are forced to specify the information.

Moreover, in CBDL, we generate connectors from the UML diagrams. Note that the correctness of the connectors is guaranteed by the verification. In CBDL, by combining the connectors and components of a component library, we generate component-based software. The common software architecture of components is *tree architecture*. There is a correspondence between AA-trees model of component specifications and tree architecture. Based on the correspondence, we generate the connectors. Because requirements of the target domain continue to change, the component library must continue to evolve. We developed *tree architecture* to simplify the evolution of the component library.

The main ideas of CBDL using LFMB or LFME are as follows:

1. a formal definition of AA-trees model by using behavioral specifications or equational specifications,
2. the consistency verification methods,
3. a formal definition of tree architecture by using behavioral specifications, and
4. the connector generation methods, respectively.

The thesis is organized as follows.

In Chapter 2, we discuss preliminaries. Firstly, we discuss the Catalysis approach that is the base of CBDL. Secondly, we discuss algebraic specification. Projection-style behavioral specification used in LFMB and equational specification used in LFME are categories of algebraic specification. Then, we discuss abstract reduction system. Abstract reduction system, especially, term rewriting system is used for the verification of LFMB and LFME. After that, we discuss a specification language CafeOBJ . We use CafeOBJ to specify projection-style behavioral specification and equational specification. Finally, we discuss JavaBeans and Servlets. The support tools of CBDL generate component-based software that is constructed from (1) JavaBeans or (2) JavaBeans and Servlets.

In Chapter 3, we discuss an overview of CBDL. CBDL is the component-based software development discussed in the thesis. Firstly, we discuss the process of CBDL and where formal methods apply to. Secondly, we discuss an overview of AA-trees model. AA-trees model is a refinement of the model of objects and actions of the Catalysis approach. In CBDL, we model business processes and behavior of components by using AA-trees model. Then, we discuss an overview of tree architecture. Tree architecture is the common software architecture of components in CBDL, which is used for avoiding architecture mismatch. After that, we discuss overviews of LFMB and LFME. LFMB and LFME are lightweight formal methods. In CBDL, we use LFMB and LFME for eliminating inconsistencies in the UML diagrams. Finally, we discuss an overview of connector generation. Based on the correspondence between AA-trees model of components and tree architecture, the support tools of CBDL generate the connectors.

In Chapter 4, we discuss tree architecture. Firstly, we discuss what tree architecture is. Then, we discuss the correspondence between AA-trees model of components and tree architecture.

In Chapter 5, we discuss AA-trees model. Firstly, we discuss what AA-trees model is. Then, we discuss how to specify AA-trees model by using UML diagrams with OCL descriptions.

In Chapter 6, we discuss a verification method of equational specification with $\neq$. Firstly, we discuss a complete deduction system of equational specification with $\neq$. Then, we discuss double term rewriting system with condition that is an implementation of the deduction system that uses term rewriting system.

In Chapter 7, we discuss LFMB. Firstly, we discuss projection-style behavioral specification. In LFMB, we use projection-style behavioral specification for formalizing AA-trees model. Secondly, we discuss how to formalize AA-trees model by using projection-style behavioral specification. Because AA-trees model is specified by using UML diagrams, then, we discuss how to translate UML diagrams into projection-style behavioral specification. Finally, we discuss how to verify consistency in the UML diagrams.

In Chapter 8, we discuss LFME. In LFME, we use equational specification for formalizing AA-trees model. Firstly, we discuss how to formalize AA-trees model by using equational specification. Because AA-trees model is specified by using UML diagrams, then, we discuss how to translate UML diagrams into equational specification. Finally, we discuss how to verify consistency in the UML diagrams.

In Chapter 9, we discuss comparisons between LFMB and LFME. Because we formalize the same AA-trees model in LFMB and LFME, for the same verification about the model, the results of LFMB and LFME must be the same. The common verification of LFMB

and LFME is refinement verification. So, firstly, we discuss a correspondence between refinement verification in LFMB and that in LFME. Because the verification results of LFMB and LFME are the same, we predicted that the logic of projection-style behavioral specification was equational logic. The prediction is true. Then, we discuss that the logic of projection-style behavioral specification.

In Chapter 10, we discuss connector generation. Based on the correspondence between AA-trees model of components and tree architecture, the support tools of CBDL generate the connectors. Because tree architecture is a software architecture of an abstract level, to generate the connectors, there must be a correspondence between tree architecture and a software architecture of an implementation level. Firstly, we discuss a software architecture of an implementation level that uses JavaBeans. We call the software architecture *JavaBeans implementation of tree architecture*. Then, we discuss how to generate the connectors of JavaBeans implementation of tree architecture. Finally, we discuss a software architecture of an implementation level that uses Servlets with JavaBeans. We call the software architecture *Servlets with JavaBeans implementation of tree architecture*.

In Chapter 11, we discuss the support tools of CBDL. We developed two groups of support tools of CBDL. One is the group of the support tool of CBDL using LFMB. The other is the group of the support tools of CBDL using LFMB and LFME. Firstly, we discuss the support tool of the former group. Then, we discuss the support tools of the latter group.

In Chapter 12, we discuss case studies. We did case studies by using the support tools. We did a case study of the domain of file transfer programs by using the support tool of CBDL using LFMB. Firstly, we discuss the case study. We did a case study of the domain of online bookstores by using the support tool of CBDL using LFMB and LFME. Then, we discuss the case study.

Finally, In Chapter 13, we discuss some conclusions.

Chapter 2

Preliminaries

2.1 The Catalysis Approach

The Catalysis approach[12] deals with all phases of CBD, which are *modeling phase*, *design phase*, and *implementation phase*. LFMB and LFME deal with only *the modeling phase* and *the design phase*. So, in this section, we summarize only those phases of the Catalysis approach.

In the modeling phase, we specify a business model of a target domain. In the design phase, we specify specifications of software that implements some parts of the business model.

An *action* of the Catalysis approach corresponds to a *UML use case*. An *attribute* is a function returning an object or a function returning a set of objects if its multiplicity is "1" or "$0 \ldots n$", respectively. Attributes are drawn by using two ways. One way is that an attribute is drawn in the middle part of a class box of a class diagram. Another way is that an attribute is drawn as a role of an association, which is a line drawn between class boxes. In the former case, multiplicity of the attribute is "1". In the latter case, multiplicity is drawn near the attribute's name. The effects of an action is described by changes between attributes' values immediately before and after the action has happened. Usually, the effects are specified by using OCL (the Catalysis extension). The most typical assertion for describing the effects is an equation whose form is

```
[object].[attribute] = F([object].[attribute]@pre)
```

where `[object].[attribute]@pre` and `[object].[attribute]` are the attribute values immediately before and after the action has happened, respectively, and `F` is a function.

Example 1 *Fig. 2.1 shows a usecase diagram.* buy *is an action.* Purchaser *and* Vendor *are objects, which participate in* buy *action. The effects of* buy *action are described by changes of values of those objects' attributes. Fig. 2.2 shows a class diagram that specifies attributes of the objects (classes). Attributes of a* Purchaser *object are* p-balance*,* p-possess*, and* vendor*. Attributes of a* Vendor *object are* v-balance*,* v-possess*, and* purchaser*. The multiplicity of* p-balance*,* p-possess*,* v-balance*, and* v-possess *is "*1*". The multiplicity of* vendor *and* purchaser *is "*$0 \ldots n$*".* p-balance *and* v-balance *are functions returning the balances of* Purchaser *and* Vendor *objects, respectively.* p-possess *and* v-possess *are functions returning the boolean values showing whether* Purchaser *and* Vendor *objects have a* Thing *object, respectively.* price *is a function returning the price of* Thing.

Figure 2.1: An action and objects

Figure 2.2: Classes and their attributes

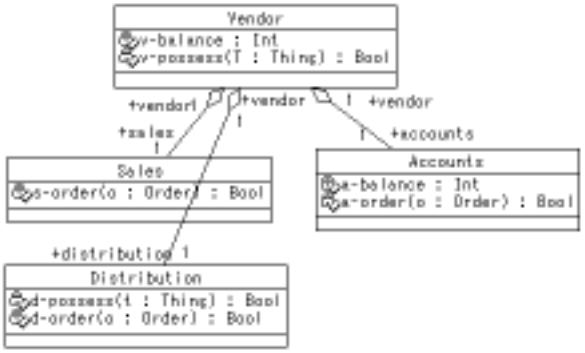

Figure 2.3: Decomposition of an object

buy *action is an action that a* Purchaser *object buy a* Thing *object from a* Vendor *object.* buy *action happens when the* Purchaser *object does not have the* Thing *object but the* Vendor *object has it.* buy *action cause payment for the* Thing *object and transportation of it. So, the precondition and the effects of* buy *action are specified by using OCL (the Catalysis extension) as follows:*

```
action (P : Purchaser, V : Vendor)::buy(T : Thing)
pre:  (P.vendor(V) = true) and
      (P.p-possess(T) = false) and
      (V.v-possess(T) = true)
post: (P.p-balance = P.p-balance@pre - T.price) and
      (P.p-possess(T) = true) and
      (V.v-balance = V.v-balance@pre + T.price) and
      (V.v-possess(T) = false)
```

The "pre:" description is the description of the precondition. The "post:" description is the description of the effects. □

There may be constraints on attributes' values. We call such constraints *static invariants.*

Example 2 *Consider a* Purchaser *object and a* Vendor *object in Figure 2.2. There is a constraint that those objects can not have the same Thing object at the same time. The constraint is specified by using OCL (the Catalysis extension) as follows:*

```
context (P : Purchaser, V : Vendor, T : Thing)
inv: (P.vendor(V) = true) and (P.p-possess(T) = true)
```

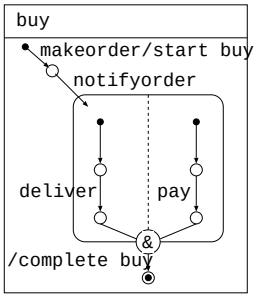

Figure 2.4: Decomposition of an action

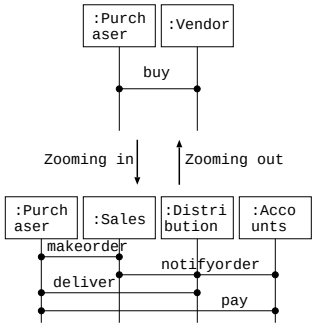

Figure 2.5: Zooming in and out

```
      => (V.v-possess(T) = false)
inv: (V.purchaser(P) = true) and (V.v-possess(T) = true)
      => (P.p-possess(T) = false)
```

□

Objects and *actions* may be decomposed. For example, a *Vendor* object and *buy* action (Fig.2.1) are decomposed (Fig.2.3 and Fig.2.4). We call the process that actions and objects are decomposed *Zooming in* and the process that actions and objects are composed *Zooming out* (Fig.2.5).

There are the following differences between the modeling phase and the design phase.

- In the modeling phase, actions represent something that happens between a set of objects, but in the design phase, actions represent messages or methods and
- in the modeling phase, there is no system boundary, but in the design phase, there are system boundaries.

2.2 Algebraic Specification

For algebraic specification, we introduce the notations used later and refer to [14, 36] for a more detailed presentation. Algebraic behavioral specification has many formalisms, for example [5, 11, 15, 20, 27]. In this section, we discuss our formalism of algebraic behavioral specification, which is used for defining projection-style behavioral specification in Chapter 7.

2.2.1 Signature, algebra, and term

Signature

Definition 1 *We let S^* denote the set of all lists of elements from a set S, including the empty list which we denote $[]$.* □

Definition 2 *Given a set S of* sorts*, an S-*sorted (or S-indexed) set *A is a family $\{A_s \mid s \in S\}$ of sets, one for each $s \in S$. We let $|A| = \cup_{s \in S} A_s$ and we let $a \in A$ mean that $a \in |A|$.* □

Definition 3 *Given a sort set S, then S-sorted* signature Σ *is an indexed family $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of sets, whose elements are called* operators, operation symbols, *or* function symbols. *A symbol $\sigma \in \Sigma_{w,s}$ is said to have* arity w, sort s, *and* rank $\langle w, s\rangle$. *In particular, any $\sigma \in \Sigma_{[],s}$ is called a* constant symbol. *We let $|\Sigma| = \cup_{w,s}\Sigma_{w,s}$ and we let $\Sigma' \subseteq \Sigma$ mean that $\Sigma'_{w,s} \subseteq \Sigma_{w,s}$ for each $w \in S^*$ and $s \in S$.* □

Algebra

Definition 4 *A* Σ-algebra M *consists of an S-sorted set also denoted M, i.e., a set M_s for each $s \in S$, plus*

1. *an element $\sigma_M \in M_s$ for each $\sigma \in \Sigma_{[],s}$, interpreting the constant symbol σ as an actual element, and*
2. *a function $\sigma_M : M_{s_1} \times \cdots \times M_{s_l} \to M_s$ for each $\sigma \in \Sigma_{w,s}$ where $w = s_1 \cdots s_l$ for $l > 0$, interpreting each operation symbol as an actual operation.*

Together, these provide an interpretation of Σ in M. We may sometimes write M_σ instead of σ_M, and also M_w instead of $M_{s_1} \times \cdots \times M_{s_l}$. The set M_s is called the carrier *of M of sort s.* □

Using the above notation we can write:
$M_\sigma : M_w \to M_s$.

Term

Definition 5 *Given an S-sorted signature Σ, then the S-sorted set T_Σ of all* Σ-terms *is the smallest set of lists over the set $|\Sigma| \cup \{\underline{(}, \underline{)}\}$ (where $\underline{(}$ and $\underline{)}$ are special symbols disjoint from Σ) such that*

1. *$\Sigma_{[],s} \subseteq (T_\Sigma)_s$ for all $s \in S$, and*
2. *given $\sigma \in \Sigma_{s_1 \cdots s_l, s}$ and $t_i \in (T_\Sigma)_{s_i}$ for $i = [1, \ldots, l]$ then $\sigma\underline{(}t_1 \cdots t_l\underline{)} \in T_{\Sigma,s}$.* □

Definition 6 *Given a Σ-term t,* subterms of t *are defined as follows:*

1. *t is a subterm of t, and*
2. *if $t = \sigma\underline{(}t_1 \cdots t_l\underline{)}$ then subterms of t_i are also subterms of t.*

In particular, any subterm of t except t is called a proper subterm of t. □

Term Algebra

Definition 7 *We can view T_Σ as a Σ-algebra as follows:*

1. *interpret $\sigma \in \Sigma_{[],s}$ in T_Σ as the singleton list σ, and*
2. *interpret $\sigma \in \Sigma_{s_1 \cdots s_l, s}$ in T_Σ as the operation which sends $t_1, \ldots, t_l$ to the list $\sigma\underline{(}t_1 \cdots t_l\underline{)}$, where $t_i \in T_{\Sigma,s_i}$ for $i = [1, \ldots, l]$.*

Thus, T_Σ is called the term algebra (over Σ). □

2.2.2 Homomorphism, equation, and satisfaction

Homomorphism

Definition 8 *An* S-sorted arrow *$f : A \to A'$ between S-sorted sets A and B is an S-sorted family $\{f_s \mid s \in S\}$ of arrows $f_s : A_s \to A'_s$. Given S-sorted arrows $f : A \to A'$ and $g : A' \to A''$, their composition $g\ f$ is the S-sorted family $\{g_s\ f_s \mid s \in S\}$ of arrows. Each S-sorted set A has an* identity arrow, *$1_A = \{1_{A_s} \mid s \in S\}$.* □

Definition 9 *Given an S-sorted signature Σ and Σ-algebras M and M', a* Σ-homomorphism *$hm : M \to M'$ is an S-sorted arrow $hm : M \to M'$ such that:*

1. *$hm_s(\sigma_M) = \sigma_{M'}$ for each constant symbol $\sigma \in \Sigma_{[],s}$ and*
2. *$hm_s(\sigma_M(e_1, \ldots, e_l)) = \sigma_{M'}(hm_{s_1}(e_1), \ldots, hm_{s_l}(e_l))$ whenever $l > 0$, $\sigma \in \Sigma_{s_1 \cdots s_l, s}$, and $e_i \in M_{s_i}$ for $i = [1, \ldots, l]$.*

The composition *$hm_2\ hm_1 : M \to M''$ of Σ-homomorphisms $hm_1 : M \to M'$ and $hm_2 : M' \to M''$ is their composition as S-sorted arrows.* □

Equation

Definition 10 *Σ is a* ground signature *iff $\Sigma_{[],s} \cap \Sigma_{[],s'} = \emptyset$ whenever $s \neq s'$, and $\Sigma_{w,s} = \emptyset$ unless $w = []$, i.e., iff it consists only of distinct constant symbols.* □

Definition 11 *The* union *of two signatures is defined by:*
$(\Sigma \cup \Sigma')_{w,s} = \Sigma_{w,s} \cup \Sigma'_{w,s}$.
A special case is union with a ground signature X. For this, we will use the notation:
$\Sigma(X) = \Sigma \cup X$,
but only in the case $|\Sigma|$ and $|X|$ are disjoint. So, the above equation may be rewritten as:
$\Sigma(X)_{[],s} = \Sigma_{[],s} \cup X_s$ *and*
$\Sigma(X)_{w,s} = \Sigma_{w,s}$ *when* $w \neq []$. □

Definition 12 *A* Σ-equation *consists of a ground signature X of* variable symbols *(disjoint from Σ) plus two $\Sigma(X)$-terms of the sort $s \in S$; we write such an equation in the form:*
$(\forall X)t = t'$. □

Definition 13 *A* conditional Σ-equation *consists of a ground signature X disjoint from Σ, a set of pairs (u_i, u'_i) $(i = [1, \ldots, k])$ of $\Sigma(X)$-terms, and a pair (t, t') of $\Sigma(X)$-terms; we write such a conditional equation in the form:*
$(\forall X)t = t'$ *if* $((u_1 = u'_1)$ *and* $\cdots$ *and* $(u_k = u'_k))$.
We call the part $((u_1 = u'_1)$ *and* $\cdots$ *and* $(u_k = u'_k))$ the condition part of the conditional Σ-equation *and use C to denote the condition part.* □

Satisfaction

Lemma 1 *Given a signature Σ, a ground signature X disjoint from Σ, a Σ-algebra M, and a map $as : X \to M$, there is a unique Σ-homomorphism $\overline{as} : T_\Sigma(X) \to M$ which extends as, in the sense that $\overline{as}_s(x) = as_s(x)$ for each $s \in S$ and $x \in X_s$. We call as an* assignment *from X to M.* □

We generally write as instead of $\overline{as}$ when there is no confusion.

Definition 14 *A* substitution *of Σ-terms with variables in Y for variables in X is an assignment $sb : X \to T_\Sigma(Y)$; we may use the notation $sb : X \to Y$. The* application *of sb to $t \in T_\Sigma(X)$ is $\overline{sb}(t)$. Given substituting $sb_1 : X \to T_\Sigma(Y)$ and $sb_2 : Y \to T_\Sigma(Z)$, their* composition *$sb_2\ sb_1$ (as substitutions) is the S-sorted arrow $\overline{sb_2}\ sb_1 : X \to T_\Sigma(Z)$.* □

Definition 15 *A Σ-algebra M* satisfies *a Σ-equation $(\forall X)t = t'$ iff for any assignment $as : X \to M$ we have $as(t) = as(t')$ in M. In this case we write:*
$M \models_\Sigma (\forall X)t = t'$. □

Definition 16 *A Σ-algebra M* satisfies *a conditional Σ-equation $(\forall X)t = t'$ if C iff for any assignment $as : X \to M$, if $as(u_i) = as(u'_i)$ for each $(u_i = u'_i) \in C$, then $as(t) = as(t')$ in M. In this case we write:*
$M \models_\Sigma (\forall X)t = t'$ *if* C.
A Σ-algebra M satisfies *a set E of conditional Σ-equations iff it satisfies each $ceq \in E$, and in this case we write:*
$M \models_\Sigma E$. □

Lemma 2 *Given a Σ-equation $eq = (\forall X)t = t'$, let $ceq = (\forall X)t = t'$ if $\emptyset$. Then for each Σ-algebra M, $M \models_\Sigma eq$ iff $M \models_\Sigma ceq$.* □

Consequently, we can regard any ordinary equation as a conditional equation with the empty condition, and we will feel free to do so hereafter. We generally omit the subscript Σ when there is no confusion.

2.2.3 Specification and model

Specification

Definition 17 An equational specification *is a pair (Σ, E), consisting of a signature Σ and a set E of conditional Σ-equations.* □

Model

Definition 18 *Given an equational specification (Σ, E), a* (Σ, E)-model *M is a Σ-algebra such that:*
$M \models_\Sigma E$. □

Definition 19 *Given an equational specification (Σ, E), a conditional Σ equation ceq. If for each* (Σ, E)-model *M,*
$M \models_\Sigma ceq$,
in this case, we write:
$E \models_\Sigma ceq$. □

Term Model

Definition 20 *Given a Σ-algebra M, a* Σ-congruence relation *on M is an S-sorted equivalence relation $\equiv = \{\equiv_s | \ s \in S\}$ on M, where each $\equiv_s$ is an equivalence relation on M_s such that for each $\sigma \in \Sigma_{s_1 \cdots s_l, s}$:*
$e_i \equiv_{s_i} e'_i$ for $i \in [1, \ldots, l]$ implies $\sigma(e_1, \ldots, e_l) \equiv_s \sigma(e'_1, \ldots, e'_l)$
for $e_i, e'_i \in M_{s_i}$. $\square$

Lemma 3 *Given a Σ-algebra M and a Σ-congruence $\equiv$ on M, then the* quotient *of M by $\equiv$, denoted $M/\equiv$, is also a Σ-algebra, in which $\sigma \in \Sigma_{[],s}$ is interpreted as $[\sigma]$, and $\sigma \in \Sigma_{s_1 \cdots s_l, s}$ with $l > 0$ is interpreted as the map sending $[e_1], \ldots, [e_l]$ to $[\sigma(e_1, \ldots, e_l)]$, for $e_i \in M_{s_i}$.* $\square$

Corollary 4 *Given an equational specification (Σ, E), the equivalence classes of Σ-terms modulo E form a (Σ, E)-model, hereafter denoted $T_{\Sigma,E}$. We call this (Σ, E)-model the* term model (over (Σ, E)). $\square$

2.2.4 A complete deduction system of equational specification

Definition 21 *Given an equational specification (Σ, E), the following* rules of deduction *define the Σ-equations that are* deducible (from E)*:*

1. (Assumption) *Each Σ-equation in E is deducible.*
2. (Reflexivity) *Each equation of the form*
   $(\forall X)t = t$
   is deducible.
3. (Symmetry) *If*
   $(\forall X)t = t'$
   is deducible, then so is
   $(\forall X)t' = t$.
4. (Transitivity) *If the equations*
   $(\forall X)t = t', \ (\forall X)t' = t''$
   are deducible, then so is
   $(\forall X)t = t''$.
5. (Congruence) *If $\theta, \theta' : X \to T_\Sigma(Y)$ are substitutions such that for each $x \in X$, the equation*
   $(\forall Y)\theta(x) = \theta'(x)$
   is deducible then given $t \in T_{\Sigma(X)}$, the equation
   $(\forall Y)\theta(t) = \theta'(t)$
   is also deducible.

6. (Substitutivity) *If*

$(\forall X)t = t'$ *if* C

is in E*, and if* $\theta : X \to T_\Sigma(Y)$ *is a substitution such that for each* $u_i = u_i' \in C$*, the equation*

$(\forall X)\theta(u_i) = \theta(u_i')$

is deducible, then so is

$(\forall X)\theta(t) = \theta(t')$.

When an Σ*-equation eq is deducible from* E*, we write:*
$E \vdash_\Sigma eq$. □

Lemma 5 *Given an equational specification* (Σ, E) *and a* Σ*-equation eq, then*
$E \models_\Sigma eq$ *iff* $E \vdash_\Sigma eq$. □

Lemma 6 *Given an equational specification* (Σ, E) *and a conditional* Σ*-equation*
$(\forall X)t = t'$ *if* C*,*
then
$E \models_\Sigma (\forall X)t = t'$ *if* C *iff* $E \cup C \models_{\Sigma(X)} (\forall \emptyset)t = t'$. □

2.2.5 Algebraic behavioral specification

Behavioral equation

Definition 22 *A* Σ-behavioral equation *consists of a ground signature* X *of* variable symbols *(disjoint from* Σ*) plus two* $\Sigma(X)$*-terms of the sort* $s \in S$*; we write such an equation in the form:*
$(\forall X)t \sim t'$. □

Algebraic behavioral specification

Definition 23 *Let* V *and* H *be sets of sorts such that* $V \cap H = \emptyset$*. Let* Ψ *be a* V*-sorted signature and* Σ *be a* $(V \cup H)$*-sorted signature such that* $\Psi \subset \Sigma$*. Let* E_Ψ *be a set of* Ψ*-equations and* E *be a set of* Σ*-equations and* Σ*-behavioral equations such that* $E_\Psi \subset E$*.*

We call a 4-tuple (V, Ψ, H, Σ) a hidden signature *iff it satisfies the following condition: for each* $\sigma \in \Sigma_{w,s} \backslash \Psi_{w,s}$*, exactly one sort in* H *occurs in* w*.*

We call a 6-tuple $(V, \Psi, E_\Psi, H, \Sigma, E)$ an algebraic behavioral specification *iff it satisfies the following conditions: (1)* (V, Ψ, H, Σ) *is a hidden signature and (2) for each* $eq \in E \backslash E_\Psi$*, at least one operator in* $\Sigma \backslash \Psi$ *occurs in eq.*

For the algebraic behavioral specification $(V, \Psi, E_\Psi, H, \Sigma, E)$*, we call sorts in* V visible sorts*, sorts in* H hidden sorts*, and* (Ψ, E_Ψ) a data specification. □

A visible sort and a hidden sort correspond to a data type and the set of states of a black box, respectively.

States of the black box are only observed by using the following observable contexts. Note that observations are for observing states of the black box and actions are for changing states of it.

Definition 24 *Let (V, Ψ, H, Σ) be a hidden signature. We call $\sigma \in \Sigma_{w,s} \backslash \Psi_{w,s}$* an observation *if $s \in V$. We call $\sigma \in \Sigma_{w,s} \backslash \Psi_{w,s}$* an action *if $s \in H$ and s coincides with the hidden sort occurring in w.* □

Definition 25 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be an algebraic behavioral specification. Let h be a hidden sort. Let v and v_i be visible sorts. Let x_i be a variable of sort v_i such that $x_i \neq x_j$ if $i \neq j$. Let $\diamond$ be a special h-sorted variable called* hole. *The set* AllOc_h *of* observable contexts *of sort h is inductively defined as follows: (1) for each observation $obs : v_1 \cdots v_k\ h \rightarrow v$ and for each variable x_is, $obs(x_1, \ldots, x_k, \diamond) \in \mathrm{AllOc}_h$ and (2) for each action $act : v_1 \cdots v_k\ h \rightarrow h$, for each element oc of AllOc_h, and for each x_is, $oc[act(x_1, \ldots, x_k, \diamond)] \in \mathrm{AllOc}_h$ where $oc[t]$ denotes the term obtained by substituting the term t for $\diamond$.* □

Satisfaction

Definition 26 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be an algebraic behavioral specification and M be a Σ-algebra. M* satisfies *a Σ-behavioral equation $(\forall X)t \sim t'$ iff for any assignment $as : X \rightarrow M$, for any observable context oc, we have $as(oc[t]) = as(oc[t'])$ in M. In this case we write:*
$M \models_\Sigma (\forall X)t \sim t'$. □

Definition 27 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be an algebraic behavioral specification and M be a Σ-algebra. M* satisfies *E iff it satisfies each $eq \in E$ and in this case we write:*
$M \models_\Sigma E$. □

Definition 28 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be an algebraic behavioral specification and M be a Σ-algebra. We call M* a hidden $(V, \Psi, E_\Psi, H, \Sigma, E)$-algebra *iff $M \models_\Sigma E$.* □

Definition 29 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be an algebraic behavioral specification and eq be an Σ-equation or Σ-behavioral equation. When $M \models_\Sigma eq$ holds for each hidden $(V, \Psi, E_\Psi, H, \Sigma, E)$-algebra M, we write:*
$E \models_\Sigma eq$. □

2.3 Abstract Reduction System

We introduce the notations used later and refer to [1] for a more detailed presentation.

2.3.1 Abstract reduction system

Definition 30 *An* abstract reduction system (ARS) *is a structure $A = (A, (\rightarrow_\alpha)_{\alpha \in I})$ consisting a set A and a sequence of binary relations $\rightarrow_\alpha$ on A, also called* reduction or rewrite relations. *In the case of just one reduction relation, we also use $\rightarrow$ without more. If for $a, b \in A$ we have $(a, b) \in \rightarrow_\alpha$, we write $a \rightarrow_\alpha b$ and call b a* one-step (α-) reduct *of a.* □

Definition 31 *The transitive reflective closure of $\rightarrow_\alpha$ is written as $\rightarrow_\alpha^*$. So $a \rightarrow_\alpha^* b$ if there is a possible empty, finite sequence of reduction steps $a = a_0 \rightarrow_\alpha a_1 \rightarrow_\alpha \cdots \rightarrow_\alpha a_n = b$. The element b is called an (α-) reduct of a. The transitive closure of $\rightarrow_\alpha$ is $\rightarrow_\alpha^+$. The converse relation of $\rightarrow_\alpha$ is $\leftarrow_\alpha$. The union $\rightarrow_\alpha \cup \rightarrow_\beta$ is denoted by $\rightarrow_{\alpha\beta}$.* □

Definition 32 *Let $A = (A, (\rightarrow))$ be an ARS, $\rightarrow$ is* confluent *if $\forall a, b, c \in A\ .\ \exists d \in A\ .\ (c \leftarrow^* a \rightarrow^* b \Rightarrow c \rightarrow^* d \leftarrow^* b)$.* □

Definition 33 *Let $A = (A, (\rightarrow))$ be an ARS, $\rightarrow$ is* terminating *if every reduction sequence $a_0 \rightarrow a_1 \rightarrow \cdots$ eventually must terminate.* □

Definition 34 *We say that $a \in A$ is a* normal form *if there is no $b \in A$ such that $a \rightarrow b$. Further, $b \in A$ has a* normal form *if $b \rightarrow^* a$ for some normal form $a \in A$. We call a a* normal form of *b.* □

2.3.2 Term rewriting system

Definition 35 *Given $t \in T_\Sigma(X)$, the set of* variables in *t, denoted $var(t)$ it the least ground signature $Y \subseteq X$ such that $t \in T_\Sigma(Y)$.* □

Notice that t is a ground term iff $var(t) = 0$. From now on, we will often just say "Σ-term" for what we were previously careful to call a "Σ-term with variables".

Definition 36 *Given a signature Σ, a conditional* Σ-rewrite rule *is a conditional Σ-equation $(\forall X)t_1 = t_2$ if C such that $var(t_2) \subseteq var(t_1) = X$, and $var(u) \subseteq var(t_1)$ and $var(v) \subseteq var(t_1)$ for each pair $\langle u, v\rangle \in C$. It follows that we can use the notation $t_1 \rightarrow t_2$ if C, which is unambiguous because X is determined by t_1. A* Σ-term rewriting system (Σ-TRS) *is a set of conditional Σ-rewrite rules; we may omit the prefix Σ when it is not needed, and we may denote such a system by (Σ, E).* □

Definition 37 *Given a Σ-term rewriting system (Σ, E), the* one-step rewriting relation *is defined for Σ-terms t, t' as follows:*

$t \Rightarrow t'$ iff there exists: a rule $(\forall X)t_1 \rightarrow t_2$ if C in E; a Σ-term $t_0 \in T_\Sigma(\{z\} \cup Y)$ having exactly one occurrence of the variable z; and a substitution $sb : X \rightarrow T_\Sigma(Y)$ such that:

$sb(u) = sb(v)$ for each pair $\langle u, v\rangle \in C$,

$t = t_0(z \leftarrow sb(t_1))$ and $t' = t_0(z \leftarrow sb(t_2))$.

In the case, the pair $\langle t_0, sb\rangle$ is called a match to *t* by the rule *$t_1 \rightarrow t_2$ if C. The* term rewriting relation *is the transitive reflexive closure of one-step rewriting relation, for which we write $t \Rightarrow^* t'$ and say that t* rewrites to *t'* (under (Σ, E)). □

2.4 CafeOBJ

In the thesis, we use a specification language CafeOBJ [10] because it supports behavioral specifications. The specification of natural numbers with successor and plus by CafeOBJ is as follows:

```
mod! NAT {   [ Nat ]
 op 0 : -> Nat   op s_ : Nat -> Nat
 op _+_ : Nat Nat -> Nat
 vars N1 N2 : Nat
 eq N1 + 0 = N1 .
 eq N1 + s N2 = s(N1 + N2) . }
```

In CafeOBJ , specifications are divided into modules which are declared by `mod!` or `mod*`. *Nat* which is surrounded by `[` and `]` is a *(visible) sort* which denotes a data type. `op`, `eq`, and `var` declare *an operator*, *an equation*, and *a variable*, respectively. For `op`, we call a list of sorts which occur between `:` and `->` *an arity* and a sort which occurs in the right of `->` *a sort (of the operator)*. For example, consider 0 operator. Its arity is $\emptyset$ and its sort is *Nat*. A signature Σ and a set of axioms E are the parts which are constructed from `op` and `eq` of a specification, respectively. We use (Σ, E) to denote *an equational specification*. We call a set with functions which correspond to operators of Σ *an Σ-algebra*. We call a symbol sequence of operators, ",", "(" and ")", like "0", "$s(0)$", and "$0 + s(s(x))$" *a term*. We call terms which do not have variables *closed terms*. We call the Σ-algebra whose set is constructed from closed terms *the term Σ-algebra*. We use T_Σ to denote this term Σ-algebra. We call the quotient of the term Σ-algebra by E *the quotient term Σ-algebra*. We use $T_{\Sigma,E}$ to denote this Σ-algebra. `!` of `mod!` declares that the model of this module is its quotient term Σ-algebra. On the other hand, `*` of `mod*` declares that the models of this module are all Σ-algebras which satisfy the equations of E.

2.5 JavaBeans and Servlets

2.5.1 JavaBeans

JavaBeans [1] are components programmed by using Java, which have the following interfaces:

1. *events* used for reporting change of the states of JavaBeans,
2. *properties* used for observing the states, and
3. *methods* used for calling inner functions of JavaBeans.

2.5.2 Servlets

Servlets are components programmed by using Java, which are used on Web servers.

[1] Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Chapter 3

An Overview of CBDL

We developed lightweight formal methods called *LFMB (a Lightweight Formal Method using Behavioral specification)* and *LFME (a Lightweight Formal Method using Equational specification)*. *CBDL* is a Component-Based software Development approach using LFMB or LFME.

CBDL is based on the Catalysis approach[12]. In the Catalysis approach, the software development process is divided into *modeling phase*, *design phase*, and *implementation phase*. For modeling phase and design phase, CBDL is a refinement of the Catalysis approach. But, for implementation phase, CBDL is an original approach.

By using the Catalysis approach, we get the benefit of formal methods that *we can get clear understanding of a target software in the process of specifying the target software*. But, because the verification is out of the scope of the Catalysis approach, we can not get the benefit that *we can verify properties of the target software using the verification systems* by only using the Catalysis approach.

In modeling phase and design phase of the Catalysis approach, UML diagrams are specified by a number of software engineers. So, there may be inconsistencies in the UML diagrams. To eliminate the inconsistencies, consistency verification is useful. Because most of software engineers are non-specialists of the verification, the verification should be executed automatically as far as possible. So, we developed lightweight formal methods complimenting the Catalysis approach called *LFMB* and *LFME*.

In LFMB and LFME, we use *AA-trees model* as a model of objects and actions. Because the Catalysis approach is not conscious of the verification, its model may not have sufficient information about the verification. AA-trees model is a refinement of the model that the specifiers are forced to specify the information.

We developed a software architecture for component-based software *tree architecture*. We developed it to simplify the evolution of the component library. In implementation phase of CBDL, we use tree architecture as the common software architecture of components.

In design phase of CBDL, we model components whose architecture is tree architecture by using AA-trees model.

Moreover, in CBDL, we generate connectors from the UML diagrams. Note that the correctness of the connectors is guaranteed by the verification. In CBDL, by combining the connectors and components of a component library, we generate component-based software. The common software architecture of components is *tree architecture*. There is a correspondence between AA-trees model of component specifications and tree architec-

ture. Based on the correspondence, we generate the connectors. Because requirements of the target domain continue to change, the component library must continue to evolve.

The main ideas of CBDL using LFMB or LFME are as follows:

1. a formal definition of tree architecture by using behavioral specifications,
2. a formal definition of AA-trees model by using behavioral specifications or equational specifications,
3. the consistency verification methods, and
4. the connector generation method, respectively.

Based on 2, UML diagrams with OCL descriptions are translated into behavioral specifications or equational specifications, respectively. In 3, by using the behavioral specifications or the equational specifications, the consistency is verified, respectively. In 4, based on the correspondence between AA-trees model of component specifications and tree architecture caused by 1 and 2, the connectors are generated.

In Chapter 3, firstly, we discuss:

1. what CBDL is,
2. what parts of CBDL formal methods apply to,
3. what tree architecture is, and
4. what AA-trees model is.

Then, we discuss overviews of LFMB, LFME, and connector generation.

3.1 CBDL

The process of CBDL is as follows:

1. fix a target domain,
2. analyze the target domain, specify the business process of the target domain, i.e. the business model, and verify properties of the business model,
3. find reusable components by analyzing the business model and specify behavior of the components,
4. develop the component library,
5. specify behavior of software, specify how to combine components to construct the software, and verify whether the combination of the components satisfies the behavior of the software, and
6. generate connectors based on the specification about how to combine components to construct the software and generate the software by combining components of the component library and the connectors.

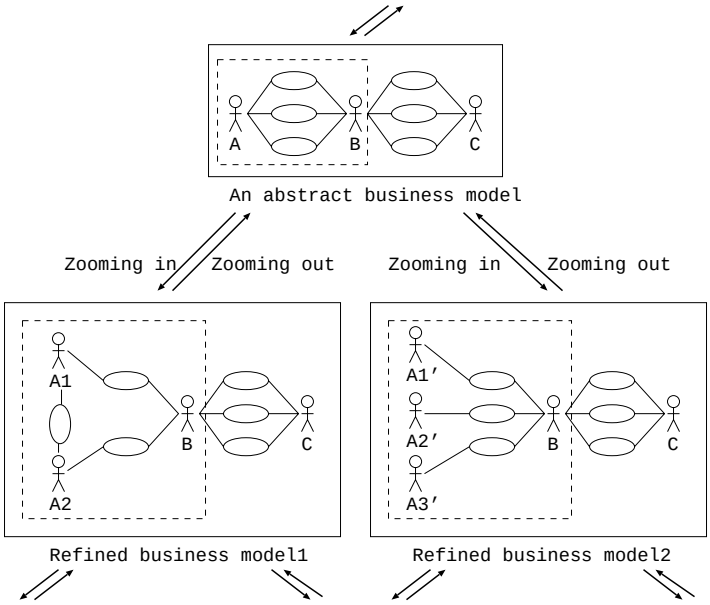

Figure 3.1: A hierarchical structure of a business model of a target domain

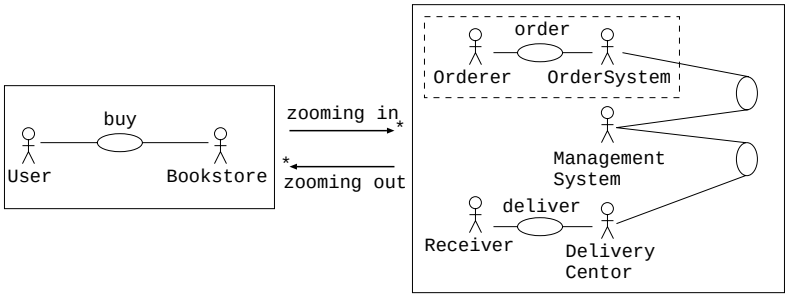

Figure 3.2: A component specification in a business model

Because requirements of the target domain continue to change, the part from 2 to 6 is iterated. The part from 2 to 4 is the development of the component library. The part from 5 to 6 is development of component-based software. 2 is the modeling phase. 3 and 5 are the design phase. 4 and 6 are the implementation phase.

The business model is a hierarchical model whose lower ranks are caused by zooming in and out (Fig. 3.1). An abstract level business process is specified in an abstract business model. More concrete level business processes are specified in refined business models. Because objects and actions may be decomposed in different ways, there may be different concrete level business models obtained from the abstract business model by zooming in. For example, in Fig. 3.1, *A* is decomposed into *A1* and *A2* in *refined business model1* and *A* is decomposed into *A1'*, *A2'*, and *A3'* in *refined business model2*. Note that the behavior of the parts surrounded by dotted lines in Fig. 3.1 at the abstract level is the same.

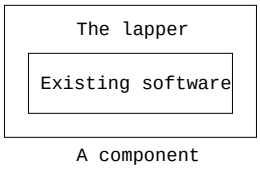

Figure 3.3: A component developed by using existing software

In a business model, not only the processes executed by software but also the processes executed by people are specified. For example, in Fig. 3.2, *buy* is decomposed into *order*, *deliver*, and so on. *Bookstore* is decomposed into *OrderSystem*, *ManagementSystem*, and *DeliveryCentor*. *OrderSystem* and *ManagementSystem* are software. But, *DeliveryCentor* are people. So, *order* is executed by software, but *deliver* is executed by people. Note that the parts surrounded by dotted lines in Fig. 3.1 corresponds to software and the user.

We specify the business model, behavior of the components, behavior of software, and how to combine components to construct the software by using UML diagrams with OCL descriptions. We translate the UML diagrams with OCL descriptions into algebraic specifications. Then, we verify consistency in the UML diagrams by using the algebraic specifications. Concretely, we verify properties of the business model and whether the combination of the components satisfies the behavior of the software by using the algebraic specifications.

The developers of the components of the component library do not need to develop the components from scratch. The components can be developed by using existing software, like free software or commercial software. At this time, firstly, we develop "the lapper" of the software and then we combine the software and the lapper (Fig. 3.3).

3.2 The Applications of Formal Methods in CBDL

In CBDL, formal methods apply to the following parts:

1. *specifications* of the business model, behavior of the components, behavior of software, and how to combine components to construct the software by using formal notations, i.e. UML diagrams with OCL descriptions,

2. *verification* of consistency in the UML diagrams, concretely, verification of properties of the business model and of whether the combination of the components satisfies the behavior of the software, and

3. *generations* of correct connectors.

1 helps to clarify requirements of business processes and behavior of components. 2 increases the correctness of the software. 3 guarantees correctness of the connectors.

We use simple logic in the verification that the verification can be executed automatically. So, the formal methods are lightweight formal methods. Because support tools that use the simple logic execute the verification automatically, non-specialists of the verification can get the benefit of the verification by using the support tools.

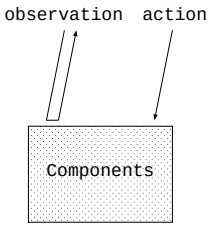

Figure 3.4: An event model of a component

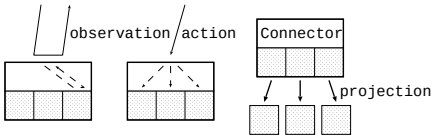

Figure 3.5: A composite component

3.3 Tree Architecture

Tree architecture is a software architecture of component-based software. There is a correspondence between AA-trees model of component specifications and tree architecture. Based on the correspondence, we generate connectors. Because requirements of the target domain continue to change, the component library must continue to evolve. We developed tree architecture to simplify the evolution of the component library.

3.3.1 Tree architecture

In tree architecture, we model components as *event models*, which are obtained by abstracting behavior of the components from the view point of event sequences.

An event model (Fig.3.4) is an object such that:

1. it has *a state*,
2. it has two kinds of events *observations* used for observing the state and *actions* used for changing the state,
3. information about the state is only obtained by using observations, and
4. the state is only changed by using actions.

From now on, we may call event models *components*.

We can regard an object obtained by combining components and the *connector* that combines the components as a component, too. We call an object satisfying the following conditions *a composite component* (Fig. 3.5):

1. for each observation of it, there exists a constructing component and a corresponding observation of the constructing component,
2. for each action of it, for each constructing component, there exists a corresponding action of the constructing component or the action does not influence the state of the constructing component, and
3. for each constructing component, there exists *a projection* from the state of it to the state of the constructing component.

Note that the structure of components is a tree whose nodes are components and whose branches are projections.

3.3.2 Evolution of a component library

Consider a composite component A whose constructing components stored in a component library are B and C. The main causes of the evolution is as follows:

1. a condition that combining B with C increases the performance of A and
2. a condition that a part *B1* of B can be reusable but the other part *B2* can not be reusable.

In the former condition, the evolution is that the specification of A and optimized components whose specifications are A are stored in the component library. In the latter condition, the evolution is that (1) the specification of B and components whose specifications are B are eliminated from the component library and (2) the specifications of *B1* and *B2* and components whose specifications are *B1* or *B2* are stored in the component library. Consider a composite component $\bar{A}$ that includes A. Note that the evolution does not influence to $\bar{A}$ outside A.

3.4 AA-trees Model of Objects and Actions

AA-trees model is a refinement of the model of objects and actions of the Catalysis approach. We model (1) the business process of a target domain and (2) behavior of components by using AA-trees model. The model of (1) is the business model and the model of (2) is the component specifications.

An object represents a cluster of information and functionality. There may be connections between objects. We call the connections *associations*. *An attribute* is a side of an association such that it has a name. *An action* represents anything that changes values of attributes of objects. We call the objects *the participants of the action*. Behavior of the action is specified by changes of the attributes of the participants.

From the viewpoint of the way how to relate to actions, objects can be divided into two groups. One group is a set of objects that participate in actions. Another group is the set of remained objects. We use the latter group to describe data structures. The most important difference between the two groups is that attributes' values of the objects corresponding to the former group may be changed by occurrences of actions, but those of the objects corresponding to the latter group are not changed by the occurrences. We call objects of the former group *agents* and objects of the latter group *data objects*.

Agents may be decomposed. We call an agent that is decomposed *a parent agent* and an agent that is a part of the parent agent *a child agent of the parent agent*. *Agent decomposition* means that all functionality of the parent agent are provided by the combination of the child agents. To describe the parent-child relations, we use associations between parent agents and child agents. We call the associations *projection-lift associations*. By iterating agent decomposition, we get *an agent tree* whose nodes are agents and whose branches are projection-lift associations.

Actions may be decomposed. We call an action that is decomposed *a parent action*. We call a sequence of actions obtained by action decomposition of the parent action *a child action sequence of the parent action* and elements of the sequence *child actions of the parent action*. Note that a parent action may have some child action sequences. When a parent action is decomposed, a participant of the action is decomposed, too. Each

participant of each child action is a participant of the parent action without the participant that is decomposed or a child agent of the participant. Action decomposition means that changes of all the attributes of all the participants immediately before and after the parent action has happened are the same as changes of those immediately before and after the sequence of the child actions has happened. By iterating action decomposition, we get *an action tree* whose nodes are actions and whose branches are parent-child relations.

AA-trees of AA-trees model are abbreviations of agent trees and action trees.

In component specifications, an action has exactly two participants. One is a participant that calls the action and the other is a participant that executes the action. We call the former participant *an interface of the action* and the latter participant *a component of the action*. Because actions are assigned to components, we may call the actions *methods of the components*.

3.5 An Overview of LFMB

In LFMB, we formalize AA-trees model of component specifications by using projection-style behavioral specifications. Projection-style behavioral specification is a kind of behavioral specification.

We specify behavior of components, behavior of software, and how to combine components to construct the software by using projection-style behavioral specifications. As consistency verification, we verify whether the combination of the components satisfies the behavior of the software by using the projection-style behavioral specifications. So, LFMB supports 3 and 5 of CBDL.

The target problem of LFMB is the above verification and the simple logic of LFMB is behavioral logic.

3.6 An Overview of LFME

In LFME, we formalize AA-trees model by using equational specifications.

We specify a business model, behavior of components, behavior of software, and how to combine components to construct the software by using equational specifications. As consistency verification, we verify properties of the business model and whether the combination of the components satisfies the behavior of the software by using the equational specifications. So, LFME supports 2, 3, and 5 of CBDL.

The target problem of LFME is the above verification and the simple logic of LFME is equational logic.

3.7 An Overview of Connector Generation in CBDL

We assign a component of tree architecture:

1. to *a function bean* and *an interface bean* that are JavaBeans or
2. to *a function bean* and *an interface servlet* that are a JavaBean and a Servlet.

Based on the correspondences, we generate connectors, i.e. (1) JavaBeans or (2) JavaBeans and Servlets.

Note that by the verification using the AA-trees model, we guarantee the correctness of the connectors.

Chapter 4

Tree Architecture

Tree architecture is a software architecture of component-based software. There is a correspondence between AA-trees model of component specifications and tree architecture. As we will discuss in Chapter 10, based on the correspondence, we generate connectors. Because requirements of the target domain continue to change, the component library must continue to evolve. We developed tree architecture to simplify the evolution of the component library.

4.1 Tree Architecture

Tree architecture is one kind of product line architecture [3, 8, 30, 33]. We fix a domain of a software family and make a component library for the domain. Software is developed by combining components selected from the component library and connectors.

We model components by using *event models*.

Objects obtained by combining components and connectors are components, too. So, tree architecture has a hierarchical structure of components (*"static structure"*).

The software family may evolve to adapt to feedback and newly added requirements. In this adaptation, firstly a component is divided into smaller components and then some of the smaller components are replaced by suitable components(*"dynamic structure"*). Note that the evolution of the software family causes the evolution of the component library.

4.1.1 Event models

Event models of components are obtained by abstracting behavior of the components from the view point of event sequences.

An event model (Fig.4.1) is an object such that:

1. it has *a state*,
2. it has two kinds of events *observations* used for observing the state and *actions* used for changing the state,
3. information about the state is only obtained by using observations, and
4. the state is only changed by using actions.

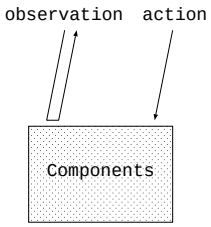

Figure 4.1: An event model of a component

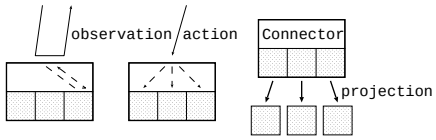

Figure 4.2: A composite component

Example 3 *Consider the component PUT-A that transfers A's files on the local machine to a remote machine. PUT-A has three observations* getremote*,* isinlocal*, and* isinremote *used for getting the current target remote machine's name, observing whether the specified file is in the local machine, and observing the specified file is in the specified remote machine, respectively. PUT-A has two actions* setremote *and* put *used for setting target remote machine and transferring the specified file to the target remote machine, respectively.* □

4.1.2 Static structures

We fix a domain of a software family and make a component library for the domain. The component library is divided by *component specifications* that specify the behavior of components.

We can regard an object obtained by combining components and connectors as a component, too. We call an object satisfying the following conditions *a composite component* (Fig. 4.2):

1. for each observation of it, there exists a constructing component and a corresponding observation of the constructing component,
2. for each action of it, for each constructing component, there exists a corresponding action of the constructing component or the action does not influence the state of the constructing component, and
3. for each constructing component, there exists *a projection* from the state of it to the state of the constructing component.

We call the part of a composite component that combines constructing components *connector*.

The static structure of tree architecture is that:

1. there is a fixed domain of a software family,
2. there is a component library for the domain,
3. the component library is divided by component specifications,
4. components are modeled by using event models, and
5. software of the domain is developed by combining components selected from the component library and connectors that combine components.

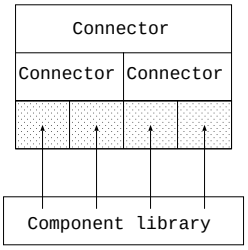

Figure 4.3: The structure of the software

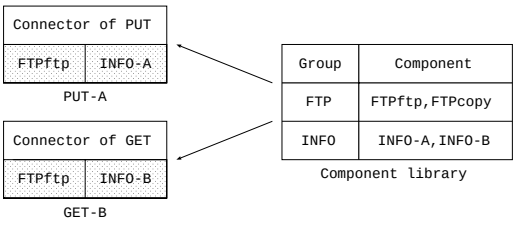

Figure 4.4: A software family and a component library

Software may have a hierarchical structure (Fig.4.3), like the ATM system [26].

Example 4 *Consider a software family of file transfer programs (Fig. 4.4). The software family includes PUT-A that transfers A's files on the local machine to a remote machine and GET-B that transfers B's files on a remote machine to the local machine. The component library is divided into FTP group that transfers files and INFO group that manages personal information, like user names and passwords. FTPftp and FTPcopy provide file transfer functions using FTP protocol and using copy command of OS, respectively. FTPftp and FTPcopy belong to FTP group. INFO-A and INFO-B provide management functions of A's personal information and B's personal information, respectively. INFO-A and INFO-B belong to INFO group. PUT-A is constructed from FTPftp, INFO-A, and the connector of PUT. GET-B is constructed from FTPftp, INFO-B, and the connector of GET.* □

4.1.3 Dynamic structures

A software family may evolve to adapt to feedback and newly added requirements. In this adaptation, firstly a component is divided into smaller components and then some of the smaller components are replaced by suitable components.

The dynamic structure of tree architecture is that:

1. the fixed domain of the software family evolves by adding pairs of new components and corresponding component specifications to the component library,

2. the new components are (1) parts of the replaced components or (2) components that provide new basic functionality, and

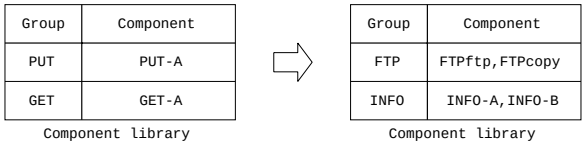

Figure 4.5: Evolution of a component library

3. pairs of component specifications of the replaced components in (1) and these components are eliminated from the component library.

Example 5 *Consider the software family constructed from PUT-A and GET-A. At this time, the library was divided by PUT and GET specifying behavior of PUT-A and GET-A, respectively, because the user was only A. Here, the new requirement that we wanted to support B occurred. To extend from this software family to the software family in Example 4, we replaced PUT and GET with FTP and INFODB (Fig. 4.5). By this replacement, the static structure of tree architecture evolved.* □

Chapter 5

AA-trees Model of Objects and Actions

AA-trees model is a refinement of the model of objects and actions of the Catalysis approach.

Because the Catalysis approach is not conscious of the verification, its model may not have sufficient information about the verification. AA-trees model is a refinement of the model that the specifiers are forced to specify the information.

We model (1) the business process of a target domain and (2) behavior of components by using AA-trees model. The model of (1) is a business model and the model of (2) is component specifications.

5.1 AA-trees Model

5.1.1 Objects, associations, and attributes

Objects

An object represents a cluster of information and functionality.

Associations

An association represents a connection between objects.

Fig. 5.1 shows objects and an association. The squares are objects and the line is an association.

Parameterized associations

A parameterized association is one whose each instance is an association such that its parameters are sets of objects.

Attributes and reverse attributes

An attribute is a side of an association such that it has a name. We regard the object connecting to the other side as the owner of the attribute. So, we call the attribute *an attribute of the object*. We regard the object connecting to the side as the value of the attribute. So, we call this object *the value of the attribute*.

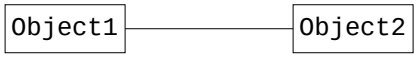

Figure 5.1: Objects and associations

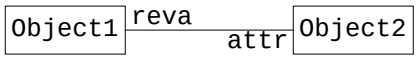

Figure 5.2: An attribute and the reverse attribute

Example 6 *Consider* attr *and* reva *in Fig. 5.2.* attr *is an attribute of* Object1 *and* reva *is an attribute of* Object2. *The value of* attr *is* Object2 *and the value of* reva *is* Object1. □

From now on, we only call associations whose both sides have names *associations*.

Consider an association and a side, i.e. an attribute. We call the other side *the reverse attribute of the attribute*.

Example 7 *Consider* attr *and* reva *in Fig. 5.2.* reva *is the reverse attribute of* attr *and* attr *is the reverse attribute of* reva. □

Parameterized attributes

A parameterized attribute is a side of a parameterized association such that it has a name. We regard the parameter of the parameterized association as the parameter of the parameterized attribute. We may call parameterized attributes *attributes*.

5.1.2 Actions

Actions

An action represents anything that changes values of attributes of objects. We call the objects *the participants of the action*.

5.1.3 Agents and data

Agents and data objects

From the viewpoint of the way how to relate to actions, objects can be divided into two groups. One group is a set of objects that participate in actions. Another group is the set of remained objects. We use the latter group to describe data structures. The most important difference between the two groups is that attributes' values of the objects corresponding to the former group may be changed by occurrences of actions, but those of the objects corresponding to the latter group are not changed by the occurrences. We call objects of the former group *agents* and objects of the latter group *data objects*.

For each set of agents, we can define the set of the indexes. Note that the indexes are data objects. From now on, we assume parameters of parameterized associations are sets of data objects. To satisfy the assumption, the specifier may need to define the sets of the indexes.

Data operators, data attributes, and agent attributes

Because we use data objects to describe data structures, we assume that values of attributes of data objects are data objects. Then, we regard attributes of data objects as

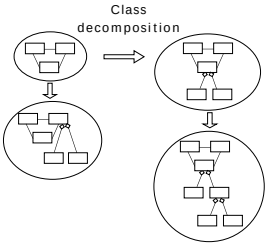

Figure 5.3: Agent decomposition

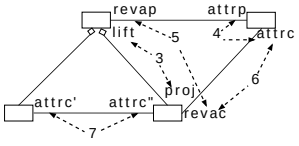

Figure 5.4: Equations added by agent decomposition

operators on the data. From now on, we call attributes of data objects *data operators* and we only call attributes of agents *attributes*.

From the above assumption,

1. there is no association between an agent and a data object and
2. attributes whose values are data objects do not have reverse attributes.

We call attributes whose values are data objects *data attributes* and attributes whose values are agents *agent attributes*.

5.1.4 Agent decomposition

Agent decomposition

Agents may be decomposed. We call an agent that is decomposed *a parent agent* and an agent that is a part of the parent agent *a child agent of the parent agent*.

Agent decomposition is refinement of a static structure (Figure 5.3). *Agent decomposition* means that all functionality of the parent agent are provided by the combination of the child agents. So, there should be a relation of each attribute of the parent agent to a combination of attributes of the child agents.

To describe the parent-child relations, we use the following *projection-lift associations*.

Projection-lift associations

We assume that there is exactly one association between a parent agent and a child agent. We call the association *a projection-lift association*, the attribute of the parent agent *a projection*, and the attribute of the child agent *a lift*.

To implement all functionality of the parent agent, the child agents must satisfy the following *data attribute constraint*, *agent attribute constraint*, and *reverse attribute constraint*.

Relations between a data attribute of a parent agent and data attributes of child agents

Each data attribute of a parent agent is implemented by using data attributes of child agents. So, for each data attribute of the parent agent, there should be a set of data attributes of child agents and a function on data F such that $pdatt = F(cdatt_1, \ldots, cdatt_n)$ where *pdatt* is the value of the data attribute of the parent agent and $cdatt_i (i \in I)$ are

the values of the data attributes of the child agents. We call the constraint *data attribute constraint*.

Relations between an agent attribute of a parent agent and an agent attribute of a child agent

Each agent attribute of a parent agent is implemented by using an agent attribute of a child agent. So, for each agent attribute of the parent agent, there should be an agent attribute of a child agent such that $paatt = caatt$ where *paatt* is the value of the agent attribute of the parent agent and *caatt* is the value of the agent attribute of the child agent. We call the constraint *agent attribute constraint*.

Relations between a reverse attribute of a parent agent and a reverse attribute of a child agent

If there is an association agent decomposition should preserve the association. Let *PA* and *PA'* be agents such that *PA* is decomposed. Consider an association between *PA* and *PA'*. Let *attrp* and *revap* be attributes of *PA* and *PA'*, respectively, such that those are the both sides of the association. Let *CA* be a child agent of *PA* and *attrc* be an attribute of *CA* such that $paatt = caatt$ where *paatt* is the value of *attrp* and *caatt* is the value of *attrc* (Figure 5.4). The preservation means that for *attrp-revap* association, there is *attrc-revac* association where *revac* is an attribute of PA'.

In other words, each reverse attribute of a parent agent is implemented by using a reverse attribute of a child agent. So, for each reverse attribute of the parent agent *revap*, there should be a reverse attribute of a child agent *revac* such that the owner of the former reverse attribute is the same as the owner of the latter reverse attribute. We call the constraint *reverse attribute constraint*.

Agent trees

By iterating agent decomposition, we get *an agent tree* whose nodes are agents and whose branches are projection-lift associations.

5.1.5 Action decomposition

Action decomposition

Actions may be decomposed. We call an action that is decomposed *a parent action*. We call a sequence of actions obtained by action decomposition of the parent action *a child action sequence of the parent action* and elements of the sequence *child actions of the parent action*. Note that a parent action may have some child action sequences.

When a parent action is decomposed, a participant of the action is decomposed, too. Each participant of each child action is a participant of the parent action without the participant that is decomposed or a child agent of the participant. *Action decomposition* means that changes of all the attributes of all the participants immediately before and after the parent action has happened are the same as changes of those immediately before and after the sequence of child actions has happened.

Consider a parent action. Then, consider two child action sequences of the action. Note that because we only compare changes of all the attributes of all the participants in action decomposition, changes of some attributes of some child agents may be different.

Parallel executions of child actions

Some of child actions can happen in parallel. Consider two actions *act1* and *act2*. Let *CPS* be the set of all the common participants of *act1* and *act2*. We deal with parallel executions of *act1* and *act2* as the interleave model that changes of all the attributes of all the elements of *CPS* immediately before and after the sequence *act1*,*act2* has happened is the same as those immediately before and after the sequence *act2*,*act1* has happened.

Action trees

By iterating action decomposition, we get *an action tree* whose nodes are actions and whose branches are parent-child relations. Note that because a parent action may have some child action sequences, a parent actions may have some action trees.

5.1.6 Classes

Classes

Classes are frameworks of objects. There are associations between classes and classes have attributes. We say that *an object is an instance of a class* if for each attribute of the class, the object has some attributes whose names are the same as the name of the attribute of the class.

Consider an attribute of a class. If each object of the class has only one attribute whose name is the same as the name of the attribute of the class, we say that *the multiplicity of the attribute is "1"*. If not, we say that *the multiplicity of the attribute is "*$0 \ldots n$*"*. Consider an association between classes. Let *attr* and *reva* be the names of the both sides of the association. If the multiplicities of *attr* and *reva* defined in the above are "1" and "$0 \ldots n$", respectively, we say that *the multiplicity of* attr *is "*$0 \ldots n$*"*. This is an exception of the above definition. Note that multiplicity of an association is the same as multiplicities of the attributes that are the both sides of the association.

Consider an attribute whose multiplicity is "$0 \ldots n$". We regard the attribute as a parameterized attribute whose parameters include the class corresponding to the set of values of the attribute and whose values are boolean values such that:

1. if a value of the attribute is assigned to the parameter corresponding to the class, the value of the parameterized attribute is true and

2. if not, the value of the parameterized attribute is false,

i.e. a characteristic function.

We call classes whose instances are data objects *data classes* and classes whose instances are agents *agent classes*. We call classes whose instances are participants of an action *participant classes of the action*. We deal with parameters of a parameterized attribute as data classes.

5.1.7 Business models

AA-trees model of business models is AA-trees model discussed in Section 5.1 before this one.

5.1.8 Component specifications

AA-trees model of component specifications is a special case of AA-trees model discussed in Section 5.1 before Section 5.1.7.

Interfaces and components

We model "a component" by using two agents. "The component" has "methods". We model "the methods" by using actions. The participants of the actions are the agents. The agents are an agent that calls the actions and the other is an agent that executes the actions. We call the former agent *an interface* and the latter agent *a component*.

Interface-component associations

Because "a component" is modeled by using an interface and a component, there is a relation between the interface and the component. We model the relation by using an association between the interface and the component of the action. We call the association *an interface-component association*, the side being the attribute of the interface *a component*, and the side being the attribute of the component *an interface*.

Attributes

Because "components" deal with data, we specify behavior of "the components" by using changes of data attributes of the components.

Because an interface is the interface of the corresponding component, for each data attribute of the interface, there should be a data attribute of the component such that $idatt = cdatt$ where *idatt* is the value of the data attribute of the interface and *cdatt* is the value of the data attribute of the component and vice versa. We call the constraint *interface attribute constraint*.

Actions

Because actions are assigned to components, we may call the actions *methods of the components*.

Component decomposition and interface decomposition

A component may be decomposed. When a component is decomposed, the interface should be decomposed, too. We assume that firstly a component is decomposed, then the interface is decomposed. From the reverse attribute constraint, the former agent decomposition adds an association between the interface and a child component to the AA-trees model and the latter agent decomposition adds an association between an child interfece and the child component to the AA-trees model. We regard the latter association as the interface-component association between the child interfece and the child component.

Method decomposition

A method may be decomposed. When a method is decomposed, the component is decomposed. Because each child method only changes the values of the data attributes of the component that has the child method, child methods of different components can happen in parallel if there is no synchronization constraint. So, if there is no synchronization constraint, on each child component, we can regard execution of the parent method as execution of the sequence of the child methods belonging to the parent methods.

Note that correspondences between parent methods and sequences of child methods on child components are the same as correspondences between parent actions and sequences of child actions on child components of tree architecture.

Classes

We call classes whose instances are interfaces of an action *interface classes of the action* and classes whose instances are components of the action *component classes of the action*.

5.1.9 Static constraints

There may be relations between attributes. To describe the relations, we use the following *static constraints*.

Static constraints

Consider a set of attributes, the set of owners of the attributes, and the set of actions in which some of owners participate *AOS*. *A static constraint* is a constraint on the set of attributes that are satisfied at immediately before and after each action of *AOS* has happened.

Note that because an action is decomposed, there may be an intermediate state at which the set of attributes do not satisfy the static constraint.

5.2 The Guideline How To Specify AA-trees Models by using UML Diagrams with OCL Descriptions

We specify data structures in class diagrams. We specify static structures, i.e. classes, associations, and attributes in class diagrams. We specify dynamic structures, i.e. actions and those effects in usecase diagrams. We call the former class diagrams *data class diagrams*, the latter class diagrams *basic class diagrams*, and the usecase diagrams *action usecase diagrams*.

Agents and actions may be decomposed. We specify agent decomposition in class diagrams. We call the class diagrams *decomposition class diagrams*. We specify action decomposition in sequence diagrams or statechart diagrams. We call the sequence diagrams and the statechart diagrams *decomposition sequence diagrams* and *decomposition statechart diagrams*, respectively.

5.2.1 Classes, associations, and attributes

Classes

We use squares to denote classes and call the squares *class boxes*. We assign data classes *data stereotype* and agent classes *agent stereotype*.

Example 8 *Consider the classes in Figure 2.2.* Purchaser *and* Vendor *are agent classes and* Thing*,* Int*, and* Bool *are data classes.* □

Associations

We use lines between class boxes to denote associations and we call the lines *association lines*.

Attributes and data operators

We use:

1. operator declarations drawn in the middle parts of class boxes or
2. sides of association lines

in data class diagrams to denote attributes.
We use:

1. operator declarations drawn in the middle parts of class boxes

in basic class diagrams to denote data operators.

5.2.2 Data class diagrams

In data class diagrams and complementing OCL descriptions, we specify data structres.

Data class diagrams

We specify data classes as class boxes. We specify data operators as operator declarations drawn in the middle parts of the class boxes. We use the following language "language for data operator declarations". to denote the operator declarations.

Language 1 *Let DTNM be a set of names of data classes and DTV be a set of variables assinged to the names. Let DTOP be a set of names of data operators. We call the language defined by the following context-free grammar* $G = (V_{DO}, T_{DO}, P_{DO}, LDOD)$ language for data operator declarations*:*

- $V_{DO} = \{LDOD, DTOPWV, DDLIST, DTDCL\}$
- $T_{DO} = \{(, , , :,)\} \cup DTV \cup DTNM \cup DTOP$
- $P_{DO} = \{$
  1. $LDOD \rightarrow DTOPWV \ : \ dtnm$
  2. $DTOPWV \rightarrow dtop$

3. $DTOPWV \rightarrow dtop(DDLIST)$
4. $DDLIST \rightarrow DTDCL$
5. $DDLIST \rightarrow DDLIST, DTDCL$
6. $DTDCL \rightarrow dtv : dtnm$

where dtv, dtnm, and dtop are elements of DTV, DTNM, and DTOP, respectively. □

OCL descriptions complementing data class diagrams

We use the following extension of OCL "OCL for data" to specify data structures.

Language 2 *Let DTNM be a set of names of data classes and DTV be a set of variables assinged to the names. Let DTOP be a set of names of data operators. We call the language defined by the following context-free grammar* $G = (V_D, T_D, P_D, OCLD)$ OCL for data*:*

- $V_D = \{OCLD, HEADD, DDLIST, DTDCL, INVDLIST, INVD, EQD, DTERM, DTLIST\}$
- $T_D = \{context, (, , , :,), inv:, =\} \cup DTNM \cup DTV \cup DTOP$
- $P_D = \{$
  1. $OCLD \rightarrow HEADD\ INVDLIST$
  2. $HEADD \rightarrow context\ (\ DDLIST\ )$
  3. $DDLIST \rightarrow DTDCL$
  4. $DDLIST \rightarrow DDLIST, DTDCL$
  5. $DTDCL \rightarrow dtv : dtnm$
  6. $INVDLIST \rightarrow INVD$
  7. $INVDLIST \rightarrow INVDLIST\ INVD$
  8. $INVD \rightarrow inv:\ EQD$
  9. $EQD \rightarrow DTERM = DTERM$
  10. $DTERM \rightarrow dtv$
  11. $DTERM \rightarrow dtop$
  12. $DTERM \rightarrow dtop(DTLIST)$
  13. $DTLIST \rightarrow DTERM$
  14. $DTLIST \rightarrow DTLIST, DTERM$

  }

where dtv, dtnm, and dtop are elements of DTV, DTNM, and DTOP, respectively. □

5.2.3 Basic class diagrams

In basic class diagrams and complementing OCL descriptions, we specify:

1. static structures, i.e. classes, associations, and attributes and
2. static invariants.

Basic class diagram

We specify agent classes as class boxes. We specify associations as association lines. We specify attributes as operator declarations drawn in the middle parts of the class boxes or sides of association lines.

Example 9 *The class diagram in Figure 2.2 is a basic class diagram.* □

We use the following language "language for attribute declarations". to denote the operator declarations.

Language 3 *Let DTNM be a set of names of data classes and DTV be a set of variables assigned to the names. Let AGTNM be a set of names of agent classes, AGTV be a set of variables assigned to the names. Let DATT be a set of names of data attributes and AATT be a set of names of agent attributes. We call the language defined by the following context-free grammar* $G = (V_{\text{ATTR}}, T_{\text{ATTR}}, P_{\text{ATTR}}, LAD)$ language for data operator declarations*:*

- $V_{\text{ATTR}} = \{LAD, DATTWVD, AATTWVD, ARGLIST, DDLIST, DTDCL, AGTDCL\}$
- $T_{\text{ATTR}} = \{(, , , :,)\} \cup DTV \cup DTNM \cup AGTV \cup AGTNM \cup DATT \cup AATT$
- $P_{\text{ATTR}} = \{$
  1. $LAD \rightarrow DATTWVD \ : \ dtnm$
  2. $LAD \rightarrow AATTWVD \ : \ agtnm$
  3. $DATTWVD \rightarrow datt$
  4. $DATTWVD \rightarrow datt(ARGLIST)$
  5. $AATTWVD \rightarrow aatt$
  6. $AATTWVD \rightarrow aatt(DDLIST)$
  7. $ARGLIST \rightarrow DDLIST$
  8. $ARGLIST \rightarrow DDLIST, AGTDCL$
  9. $DDLIST \rightarrow DTDCL$
  10. $DDLIST \rightarrow DDLIST, DTDCL$
  11. $DTDCL \rightarrow dtv : dtnm$
  12. $AGTDCL \rightarrow agtv : agtnm$

  }

where dtv, dtnm, agtv, agtnm, datt, and aatt are elements of DTV, DTNM, AGTV, AGTNM, DATT, and AATT, respectively. □

OCL descriptions complementing basic class diagrams

We specify static invariants in OCL descriptions complementing basic static diagrams. Static invariants are invariants on static structures. We use the following extension of OCL "OCL for static invariants" to specify the static invariants.

Language 4 *Let DTNM be a set of names of data classes, DTV be a set of variables assigned to the names. Let AGTNM be a set of names of agent classes, AGTV be a set of variables assigned to the names. Let DTOP be a set of names of data operators, DATT be a set of names of data attributes, and AATT be a set of names of agent attributes. We call the language defined by the following context-free grammar* $G = (V_{SI}, T_{SI}, P_{SI}, OCLSI)$ OCL for static invariants*:*

- $V_{SI} = \{OCLSI, HEADSI, ADLIST, AGTDCL, VARLIST, CDLIST, DDLIST, DTDCL, INVSILIST, INVSI, LFSI, LLFSI, EQSILIST, EQSI, EDTERM, EDTLIST, EATERM, AATTWT, DATTWT, EDARGLIST\}$
- $T_{SI} = \{context, (, , , :,), \epsilon, var:, inv:, and, =, .\} \cup DTV \cup DTNM \cup AGTV \cup AGTNM \cup DTOP \cup DATT \cup AATT$
- $P_{SI} = \{$
  1. $OCLSI \rightarrow HEADSI\ VARLIST\ INVSILIST$
  2. $HEADSI \rightarrow context\ (\ ADLIST\ )$
  3. $ADLIST \rightarrow AGTDCL$
  4. $ADLIST \rightarrow ADLIST, AGTDCL$
  5. $AGTDCL \rightarrow agtv : agtnm$
  6. $VARLIST \rightarrow \epsilon$
  7. $VARLIST \rightarrow var:\ CDLIST$
  8. $CDLIST \rightarrow DDLIST$
  9. $CDLIST \rightarrow ADLIST$
  10. $CDLIST \rightarrow DDLIST\ ADLIST$
  11. $DDLIST \rightarrow DTDCL$
  12. $DDLIST \rightarrow DDLIST, DTDCL$
  13. $DTDCL \rightarrow dtv : dtnm$
  14. $INVSILIST \rightarrow INVSI$
  15. $INVSILIST \rightarrow INVSILIST\ INVSI$
  16. $INVSI \rightarrow inv:\ LFSI$
  17. $LFSI \rightarrow LLFSI \Rightarrow EQSI$
  18. $LLFSI \rightarrow \epsilon$
  19. $LLFSI \rightarrow EQSILIST$
  20. $EQSILIST \rightarrow EQSI$
  21. $EQSILIST \rightarrow EQSILIST\ and\ EQSI$

22. $EQSI \rightarrow EDTERM = EDTERM$
23. $EQSI \rightarrow EATERM = EATERM$
24. $EDTERM \rightarrow dtv$
25. $EDTERM \rightarrow dtop$
26. $EDTERM \rightarrow dtop(EDTLIST)$
27. $EDTERM \rightarrow EATERM\ .\ DATTWT$
28. $EDTLIST \rightarrow EDTERM$
29. $EDTLIST \rightarrow EDTLIST, EDTERM$
30. $EATERM \rightarrow agtv$
31. $EATERM \rightarrow EATERM\ .\ AATTWT$
32. $AATTWT \rightarrow aatt(EDTLIST)$
33. $DATTWT \rightarrow datt(EDARGLIST)$
34. $EDARGLIST \rightarrow EDTLIST$
35. $EDARGLIST \rightarrow EDTLIST, EATERM$

}

where dtv, dtnm, agtv, agtnm, dtop, datt, and aatt are elements of DTV, DTNM, AGTV, AGTNM, DTOP, DATT, and AATT, respectively. □

Example 10 *The OCL description in Example 2 is an OCL description specifying static invariants.* □

5.2.4 Action usecase diagrams

In action usecase diagrams and complementing OCL descriptions, we specify dynamic structures, i.e. actions and those effects.

Action usecase diagrams

We specify an action as an usecase and the participants of the action as actors.

Example 11 *The usecase diagram in Figure 2.1 is an action usecase diagram.* □

OCL descriptions complementing action usecase diagrams

We specify behavior of the action in the OCL descriptions complementing the action usecase diagram.

For each attribute of participant classes whose value is changed by the action, we specify the effect of the action on its value as an OCL description. We use the following extension of OCL "OCL for actions" to specify the effects.

Language 5 *Let DTNM be a set of names of data classes, DTV be a set of variables assigned to the names. Let AGTNM be a set of names of agent classes, AGTV be a set of variables assigned to the names. Let DTOP be a set of names of data operators, DATT be a set of names of data attributes, and AATT be a set of names of agent attributes. Let ACTNM be a set of names of actions. We call the language defined by the following context-free grammar* $G = (V_A, T_A, P_A, OCLA)$ OCL for actions*:*

- $V_A = \{OCLA, HEADA, ADLIST, AGTDCL, DDLIST, DTDCL, PREDCL, PRELIST, EQPRE, DATTWA, DATTWV, DARGLIST, DTLIST, DTERM, AATTWA, AATTWV, POSTDCL, POSTLIST, EQPOST, DLTPOST, EDTERMPOST, EDTLISTPOST, ALTPOST, ARTPOST\}$
- $T_A = \{action, (, , , :,), \epsilon, pre:, post:, and, =, ., @pre\} \cup DTV \cup DTNM \cup AGTV \cup AGTNM \cup DTOP \cup DATT \cup AATT \cup ACTNM$
- $P_A = \{$
  1. $OCLA \rightarrow HEADA\ PREDCL\ POSTDCL$
  2. $HEADA \rightarrow action\ (\ ADLIST\ )\ ::\ actnm\ (\ DDLIST\ )$
  3. $ADLIST \rightarrow AGTDCL$
  4. $ADLIST \rightarrow ADLIST, AGTDCL$
  5. $AGTDCL \rightarrow agtv : agtnm$
  6. $DDLIST \rightarrow DTDCL$
  7. $DDLIST \rightarrow DDLIST, DTDCL$
  8. $DTDCL \rightarrow dtv : dtnm$
  9. $PREDCL \rightarrow \epsilon$
  10. $PREDCL \rightarrow pre:\ PRELIST$
  11. $PRELIST \rightarrow EQPRE$
  12. $PRELIST \rightarrow PRELIST\ and\ EQPRE$
  13. $EQPRE \rightarrow DATTWA = DTERM$
  14. $EQPRE \rightarrow AATTWA = agtv$
  15. $DATTWA \rightarrow agtv\ .\ DATTWV$
  16. $DATTWV \rightarrow datt(DARGLIST)$
  17. $DARGLIST \rightarrow DTLIST$
  18. $DARGLIST \rightarrow DTLIST, agtv$
  19. $DTLIST \rightarrow DTERM$
  20. $DTLIST \rightarrow DTLIST, DTERM$
  21. $DTERM \rightarrow dtv$
  22. $DTERM \rightarrow dtop$
  23. $DTERM \rightarrow dtop(DTLIST)$
  24. $AATTWA \rightarrow agtv\ .\ AATTWV$
  25. $AATTWV \rightarrow aatt(DTLIST)$
  26. $POSTDCL \rightarrow post:\ POSTLIST$
  27. $POSTLIST \rightarrow EQPOST$
  28. $POSTLIST \rightarrow POSTLIST\ and\ EQPOST$
  29. $EQPOST \rightarrow DLTPOST = EDTERMPOST$

30. $EQPOST \rightarrow ALTPOST = ARTPOST$
31. $DLTPOST \rightarrow agtv\ .\ DATTWV$
32. $EDTERMPOST \rightarrow dtv$
33. $EDTERMPOST \rightarrow dtop$
34. $EDTERMPOST \rightarrow agtv\ .\ DATTWV@pre$
35. $EDTERMPOST \rightarrow dtop(EDTLISTPOST)$
36. $EDTLISTPOST \rightarrow EDTERMPOST$
37. $EDTLISTPOST \rightarrow EDTLISTPOST, EDTERMPOST$
38. $ALTPOST \rightarrow agtv\ .\ AATTWV$
39. $ARTPOST \rightarrow agtv$
40. $ARTPOST \rightarrow agtv\ .\ AATTWV@pre$

}

where dtv, dtnm, agtv, agtnm, dtop, datt, aatt, and actnm are elements of DTV, DTNM, AGTV, AGTNM, DTOP, DATT, AATT, and ACTNM, respectively. □

5.2.5 Decomposition class diagrams

In decomposition class diagrams and complementing OCL descriptions, we specify agent decomposition.

Decomposition class diagrams

We specify projection-lift associations as lines with diamonds that are connected to parent agents.

Example 12 *The class diagram in Figure 2.3 is a decomposition class diagram.* □

OCL descriptions complementing decomposition class diagrams

In OCL descriptions complementing the decomposition class diagrams, we specify data attribute constraint and agent attribute constraint. We use the following extension of OCL "OCL for constraints" to specify those constraints.

Language 6 *Let DTNM be a set of names of data classes, DTV be a set of variables assigned to the names. Let AGTNM be a set of names of agent classes, AGTV be a set of variables assigned to the names. Let DTOP be a set of names of data operators, DATT be a set of names of data attributes, and AATT be a set of names of agent attributes. Let PROJ be a set of names of projections. We call the language defined by the following context-free grammar* $G = (V_C, T_C, P_C, OCLC)$ OCL for constraints*:*

- $V_C = \{OCLC, HEADC, ADLIST, AGTDCL, VARLIST, CDLIST, DDLIST, DTDCL, INVCLIST, INVC, DATTC, DATTWA, DATTWV, EDTERMC, EDTLISTC, AATTC, AATTWA, AATTWV, DARGLIST, DTLIST, DTERM, CAATTWA\}$

- $T_C = \{context, (, , , :,), \epsilon, var:, inv:, =, .\} \cup DTV \cup DTNM \cup AGTV \cup AGTNM \cup DTOP \cup DATT \cup AATT \cup PROJ$
- $P_C = \{$
  1. $OCLC \rightarrow HEADC\ VARLIST\ INVCLIST$
  2. $HEADC \rightarrow context\ (\ ADLIST\ )$
  3. $ADLIST \rightarrow AGTDCL$
  4. $ADLIST \rightarrow ADLIST, AGTDCL$
  5. $AGTDCL \rightarrow agtv : agtnm$
  6. $VARLIST \rightarrow \epsilon$
  7. $VARLIST \rightarrow var:\ CDLIST$
  8. $CDLIST \rightarrow DDLIST$
  9. $CDLIST \rightarrow ADLIST$
  10. $CDLIST \rightarrow DDLIST\ ADLIST$
  11. $DDLIST \rightarrow DTDCL$
  12. $DDLIST \rightarrow DDLIST, DTDCL$
  13. $DTDCL \rightarrow dtv : dtnm$
  14. $INVCLIST \rightarrow INVC$
  15. $INVCLIST \rightarrow INVCLIST\ INVC$
  16. $INVC \rightarrow inv:\ DATTC$
  17. $INVC \rightarrow inv:\ AATTC$
  18. $DATTC \rightarrow DATTWA = EDTERMC$
  19. $DATTWA \rightarrow agtv\ .\ DATTWV$
  20. $DATTWV \rightarrow datt(DARGLIST)$
  21. $EDTERMC \rightarrow dtv$
  22. $EDTERMC \rightarrow dtop$
  23. $EDTERMC \rightarrow agtv\ .\ proj\ .\ DATTWV$
  24. $EDTERMC \rightarrow dtop(EDTLISTC)$
  25. $EDTLISTC \rightarrow EDTERMC$
  26. $EDTLISTC \rightarrow EDTLISTC, EDTERMC$
  27. $AATTC \rightarrow AATTWA = CAATTWA$
  28. $AATTWA \rightarrow agtv\ .\ AATTWV$
  29. $AATTWV \rightarrow aatt(DTLIST)$
  30. $DARGLIST \rightarrow DTLIST$
  31. $DARGLIST \rightarrow DTLIST, agtv$
  32. $DTLIST \rightarrow DTERM$

33. $DTLIST \rightarrow DTLIST, DTERM$

34. $DTERM \rightarrow dtv$

35. $DTERM \rightarrow dtop$

36. $DTERM \rightarrow dtop(DTLIST)$

37. $CAATTWA \rightarrow agtv\ .\ proj\ .\ AATTWV$

}

where dtv, dtnm, agtv, agtnm, dtop, datt, aatt, and proj are elements of DTV, DTNM, AGTV, AGTNM, DTOP, DATT, AATT, and PROJ, respectively. □

5.2.6 Decomposition sequence diagrams

In decomposition sequence diagrams or decomposition statechart diagrams, we specify action decomposition.

Decomposition sequence diagrams

In decomposition sequence diagrams or decomposition statechart diagrams, we specify sequences of child actions whose effects are the same as effect of a parent action.

Example 13 *The sequence diagram in the bottom of Figure 2.5 is a decomposition sequence diagram.* □

5.2.7 Decomposition statechart diagrams

In decomposition sequence diagrams or decomposition statechart diagrams, we specify action decomposition.

Decomposition statechart diagrams

In decomposition sequence diagrams or decomposition statechart diagrams, we specify sequences of child actions whose effects are the same as effect of a parent action.

Example 14 *The statechart diagram in Figure 2.4 is a decomposition statechart diagram.* □

Chapter 6

A Verification Method of Equational Specification with $\neq$

There are some studies about verification of equational specification that has conditional equations, for example [16, 23, 37]. The conditions of those studies have the forms:

$(u_1 = u'_1)$ and $\cdots$ and $(u_k = u'_k)$.

When we specify behavior of software, we usually want to use the conditions whose forms are:

$(u_1 = u'_1)$ and $\cdots$ and $(u_k = u'_k)$ and $(v_1 \neq v'_1)$ and $\cdots$ and $(v_l \neq v'_l)$.

In fact, CafeOBJ [10] supports `=/=` that returns the result whether normal forms of both sides are the same. `=/=` is similar to $\neq$, but, because a term may have many normal forms, `=/=` sometimes return an unexpected result.

We call equational specification that has conditional equations whose conditions are

$(u_1 = u'_1)$ and $\cdots$ and $(u_k = u'_k)$ and $(v_1 \neq v'_1)$ and $\cdots$ and $(v_l \neq v'_l)$

equational specification with $\neq$. In this chapter, we discuss a complete deduction system of equational specification with $\neq$ and an implementation of it called *double term rewriting system with condition.*

6.1 Equational Specification with $\neq$

Firstly, we define a conditional Σ-equation with $\neq$.

Definition 38 *A* conditional Σ-equation with $\neq$ *consists of a ground signature X disjoint from Σ, a set of pairs (u_i, u'_i) $(i = [1, \ldots, k])$ of $\Sigma(X)$-terms, a set of pairs (v_i, v'_i) $(i = [1, \ldots, l])$ of $\Sigma(X)$-terms, and a pair (t, t') of $\Sigma(X)$-terms; we write such a conditional equation in the form:*

$(\forall X) t = t'$ *if* $((u_1 = u'_1)$ *and* $\cdots$ *and* $(u_k = u'_k)$ *and* $(v_1 \neq v'_1)$ *and* $\cdots$ *and* $(v_l \neq v'_l))$.

We call the part $((u_1 = u'_1)$ *and* $\cdots$ *and* $(u_k = u'_k))$ the equality condition part of the conditional Σ-equation *and use $C_=$ to denote the equality condition part. We call the part* $((v_1 \neq v'_1)$ *and* $\cdots$ *and* $(v_l \neq v'_l))$ the inequality condition part of the conditional Σ-equation *and use $C_\neq$ to denote the inequality condition part.* $\square$

Definition 39 *A Σ-algebra M* satisfies *a conditional Σ-equation with $\neq$:*

$(\forall X) t = t'$ *if* $(C_=$ *and* $C_\neq)$

iff for any assignment $as : X \to M$, if $as(u_i) = as(u'_i)$ for each $(u_i = u'_i) \in C_=$ and $as(v_i) \neq as(v'_i)$ for each $(v_i = v'_i) \in C_\neq$, then $as(t) = as(t')$ in M. In this case we write:

$M \models_\Sigma (\forall X)t = t'$ *if* $(C_= \text{ and } C_{\neq})$.
A Σ*-algebra* M satisfies *a set* E *of conditional* Σ*-equations with* $\neq$ *iff it satisfies each* $ceq \in E$*, and in this case we write:*
$M \models_\Sigma E$. □

Then, we define an equational specification with $\neq$.

Definition 40 An equational specification with $\neq$ *is a pair* (Σ, E)*, consisting of a signature* Σ *and a set* E *of conditional* Σ*-equations with* $\neq$. □

The model of the equational specification with $\neq$ (Σ, E) is the following (Σ, E)-models.

Definition 41 *Given an equational specification with* $\neq$ (Σ, E)*, a* (Σ, E)-model M *is a* Σ*-algebra such that:*
$M \models_\Sigma E$. □

We deal with conditional equations with $\neq$ as equations by using the technique introduced in Lemma 6. To do so, we introduce the following Σ-inequation and extend equational specification with $\neq$ to the following equational and inequational specification.

Definition 42 *A* Σ-inequation *consists of a ground signature* X *disjoint from* Σ *and a pair* (t, t') *of* $\Sigma(X)$*-terms; we write such an inequation in the form:*
$(\forall X)t \neq t'$ □

Definition 43 *A* Σ*-algebra* M satisfies *a* Σ*-inequation:*
$(\forall X)t \neq t'$
iff for any assignment $as : X \to M$*,* $as(t) \neq as(t')$ *in* M*. In this case we write:*
$M \models_\Sigma (\forall X)t \neq t'$.
A Σ*-algebra* M satisfies *a set* I *of* Σ*-inequations iff it satisfies each* $ieq \in I$*, and in this case we write:*
$M \models_\Sigma I$. □

The definition of an equational and inequational specification is as follows.

Definition 44 An equational and inequational specification *is a 3-tuple* (Σ, E, I)*, consisting of a signature* Σ*, a set* E *of conditional* Σ*-equations with* $\neq$*, and a set* I *of* Σ*-inequations.* □

The model of the equational and inequational specification (Σ, E, I) is the following (Σ, E, I)-models.

Definition 45 *Given an equational and inequational specification* (Σ, E, I)*, a* (Σ, E, I)-model M *is a* Σ*-algebra such that:*
$M \models_\Sigma E$ *and* $M \models_\Sigma I$. □

The properties eq of the equational and inequational specification (Σ, E, I) is specified by using $E \cup I \models_\Sigma$ eq.

Definition 46 *Given an equational and inequational specification* (Σ, E, I) *and a conditional* Σ*-equation with* $\neq$ *or a* Σ*-inequation eq. If for each* (Σ, E, I)-model M*,*
$M \models_\Sigma eq$*,*
in this case, we write:
$E \cup I \models_\Sigma eq$. □

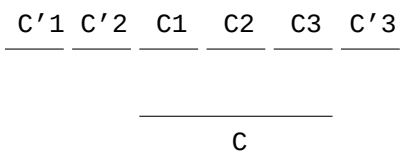

Figure 6.1: The comparison between cases

Because E may contradict I, there may be no (Σ, E, I)-model. The following proposition provides the condition whether there is a (Σ, E, I)-model.

Proposition 7 *Given an equational and inequational specification* (Σ, E, I).
There is a (Σ, E, I)*-model iff* $E \models_\Sigma I$.

Proof : ($\Rightarrow$) Consider a Σ-inequation ieq $\in I$. We assume that $E \not\models_\Sigma$ ieq. By the assumption, for each (Σ, E)-model M, $M \not\models_\Sigma$ ieq. So, M is not a (Σ, E, I)-model, i.e., there is no (Σ, E, I)-model. By contraposition, ($\Rightarrow$) is showed.
($\Leftarrow$) Because $E \models_\Sigma I$, each (Σ, E)-model is a (Σ, E, I)-model. So, ($\Leftarrow$) is showed. □

Corollary 8 *Given an equational and inequational specification* (Σ, E, I).
There is a (Σ, E, I)*-model iff any* (Σ, E)*-model is a* (Σ, E, I)*-model.* □

6.2 A Deduction System of Equational and Inequational Specification

The conditional equation with $\neq$ version of Lemma 6 is as follows.

Proposition 9 *Given an equational and inequational specification* (Σ, E, I) *and a conditional* Σ*-equation with* $\neq$
$(\forall X)t = t'$ *if* $(C_=$ *and* $C_\neq)$,
then
$E \cup I \models_\Sigma (\forall X)t = t'$ *if* $(C_=$ *and* $C_\neq)$ *iff* $E \cup I \cup C_= \cup C_\neq \models_{\Sigma(X)} (\forall \emptyset)t = t'$. □

Proof : Each condition is equivalent to the condition:
for each (Σ, E, I)-model M and for each assignment as $: X \to M$, if

1. for each $(u_i = u'_i) \in C_=$, $\mathrm{as}(u_i) = \mathrm{as}(u'_i)$ and
2. for each $(v_i \neq v'_i) \in C_\neq$, $\mathrm{as}(v_i) \neq \mathrm{as}(v'_i)$, then

$\mathrm{as}(t) = \mathrm{as}(t')$ □

The characteristics of equational specification with $\neq$ are as follows:

1. a case can be decomposed, for example, "$a(x) = 0$" is decomposed into "$(a(x) = 0)$ and $(b(x) = 0)$" and "$(a(x) = 0)$ and $(b(x) \neq 0)$", and
2. the case that can not happen exists, for example, "$(a(x) = 0)$ and $(a(x) \neq 0)$".

So, *case composition rule* and *no model rule* in Definition 50 are necessary. To deal with case composition rule, we need the procedure that decides whether the case corresponds to C can be decomposed into the cases correspond to C_j $(j \in J)$. Note that the fact that the case corresponding to C includes the case corresponding to C_j is equivalent of the fact that:

for each $(\Sigma, E \cup C_{=,j}, I \cup C_{\neq,j})$-model M, $M \models_\Sigma C$.

Note that the fact that the case corresponding to C and the case corresponding to C'_k are mutually exclusive is equivalent of the fact that there is no $(\Sigma, E \cup C_= \cup C'_{=,k}, I \cup C_\neq \cup C'_{\neq,k})$-model, i.e.,

$E \cup C_= \cup C'_{=,k} \not\models_\Sigma I \cup C_\neq \cup C'_{\neq,k}$ (by Proposition 7).

So, if the case without the case corresponds to C is the same as the sum of the cases correspond to C'_k $(k \in K)$, we can decide whether the case corresponds to C can be decomposed into the cases correspond to C_j $(j \in J)$ by using the above facts (Fig. 6.1).

Definition 47 *Let (Σ, E, I) be an equational and inequational specification and DRules be a set of deduction rules that defines the conditional Σ-equations with $\neq$ that are deducible from $E \cup I$. We say that (Σ, E, I) is* splitable on DRules *iff*

for each set of conditional Σ-equations with $\neq$ ceq_j $(j \in J)$

$(\forall X)t = t'$ *if* C_j

that are deducible by using DRules, there is the set of conditions $\{C'_k\}_{k \in K}$ such that:

1. *for each $j \in J$, for each $k \in K$, for each (Σ, E, I)-model M and for each assignment $as : X \to M$,*

   $as(C_j \cup C'_k) = false.$

2. *for each (Σ, E, I)-model M and for each assignment $as : X \to M$, there is $j \in J$ or $k \in K$ such that:*

   $as(C_j) = true$ *or* $as(C'_k) = true.$

We call $\{C'_k\}_{k \in K}$ the complement set of $\{C_j\}_{j \in J}$. □

If the deduction system using DRules is complete,

1. for each $(\Sigma, E \cup C_{=,j}, I \cup C_{\neq,j})$-model M, $M \models_\Sigma C$ and
2. $E \cup C_= \cup C'_{=,k} \not\models_\Sigma I \cup C_\neq \cup C'_{\neq,k}$

are the same as

1. $E \cup C_{=,j} \vdash C$ or $E \cup C_{=,j} \not\vdash I \cup C_{\neq,j}$ and
2. $E \cup C_= \cup C'_{=,k} \not\vdash I \cup C_\neq \cup C'_{\neq,k}$, respectively.

Note that (1) if there is a $(\Sigma, E \cup C_{=,j}, I \cup C_{\neq,j})$-model, by Corollary 8, the set of all the $(\Sigma, E \cup C_{=,j}, I \cup C_{\neq,j})$-models coincides with the set of all the $(\Sigma, E \cup C_{=,j})$-models and (2) if there is no $(\Sigma, E \cup C_{=,j}, I \cup C_{\neq,j})$-model, for any C, $E \cup I \cup C_j \models_\Sigma C$ holds.

Definition 48 *Let (Σ, E, I) be an equational and inequational specification and DRules be a set of deduction rules that defines the conditional Σ-equations with $\neq$ that are deducible from $E \cup I$ such that (Σ, E, I) is splitable on DRules. Let ceq_j $(j \in J)$ be conditional Σ-equations with $\neq$*

$(\forall X)t = t'$ *if* C_j

that are deducible by using DRules and $\{C'_k\}_{k\in K}$ *be the complement set of* $\{C_j\}_{j\in J}$*. Let ceq be a conditional* Σ*-equation with* $\neq$

$(\forall X)t = t'$ *if* C*.*

We call $\{C_j\}_{j\in J}$ a complement condition set *and* C a combined condition of $\{C_j\}_{j\in J}$ *if*

1. *for each* $j \in J$, $E \cup C_{=,j} \vdash C$ *or* $E \cup C_{=,j} \nvdash I \cup C_{\neq,j}$ *and*
2. *for each* $k \in K$, $E \cup C_{=} \cup C'_{=,k} \nvdash I \cup C_{\neq} \cup C'_{\neq,k}$

where $A \vdash B$ *means that each element of* B *is deducible from* A *and* $A \nvdash B$ *means that there is an element of* B *that is not deducible from* A*.* □

A deduction system may not terminate. So, to deal with inequations, the following undeducibility decision problem must be decidable.

Definition 49 *Let* (Σ, E, I) *be an equational and inequational specification and DRules be a set of deduction rules that defines the conditional* Σ*-equations with* $\neq$ *that are deducible from* $E \cup I$*. We call the problem that for a given conditional* Σ*-equation with* $\neq$*, we decide whether it is not deducible from* $E \cup I$ *by using DRules* undeducibility decision problem. □

A complete deduction system of equational and inequational specification is as follows.

Definition 50 *Let* (Σ, E, I) *be an equational and inequational specification such that:*

1. (Σ, E, I) *is splitable on the set of the following deduction rules and*
2. *undeducibility decision problem is decidable on* (Σ, E, I) *and specifications occurring in the following deduction.*

The following rules of deduction *define the conditional* Σ*-equations with* $\neq$ *and the* Σ*-inequations that are* deducible (from $E \cup I$)*:*

1. (Assumption) *Each conditional* Σ*-equation with* $\neq$ *in* E *is deducible.*
2. (Reflexivity) *Each* Σ*-equation of the form*

   $(\forall X)t = t$

   is deducible.
3. (Symmetry) *If*

   $(\forall X)t = t'$

   is deducible, then so is

   $(\forall X)t' = t$*.*
4. (Transitivity) *If the* Σ*-equations*

   $(\forall X)t = t'$, $(\forall X)t' = t''$

   are deducible, then so is

   $(\forall X)t = t''$*.*

5. (Congruence) *If* $\theta, \theta' : X \to T_\Sigma(Y)$ *are substitutions such that for each* $x \in X$, *the* Σ*-equation*

$$(\forall Y)\theta(x) = \theta'(x)$$

is deducible then given $t \in T_{\Sigma(X)}$, *the* Σ*-equation*

$$(\forall Y)\theta(t) = \theta'(t)$$

is also deducible.

6. (Substitutivity) *If*

$$(\forall X)t = t' \text{ if } (C_= \text{ and } C_\neq)$$

is in E, *and if* $\theta : X \to T_\Sigma(Y)$ *is a substitution such that for each* $(u_i = u_i') \in C_=$, *the* Σ*-equation*

$$(\forall X)\theta(u_i) = \theta(u_i')$$

is deducible and for each $(v_i \neq v_i') \in C_\neq$, *the* Σ*-equation*

$$(\forall X)\theta(v_i) \neq \theta(v_i')$$

is deducible, then

$$(\forall X)\theta(t) = \theta(t')$$

is deducible.

7. (Condition) *Consider the* Σ*-equation*

$$(\forall X)t = t' \text{ if } C.$$

If

$$(\forall X)t = t'$$

is deducible from $E \cup I \cup C$, *then*

$$(\forall X)t = t' \text{ if } C$$

is deducible.

8. (Case composition) *If the conditional* Σ*-equations with* $\neq$

$$(\forall X)t = t' \text{ if } C_j \quad (j \in [1, \ldots, k])$$

are deducible and $\{C_j\}_{j\in[1,\ldots,k]}$ *is a complement condition set, then*

$$(\forall X)t = t' \text{ if } C$$

is deducible where C *is a combined condition of* $\{C_j\}_{j\in[1,\ldots,k]}$.

9. (No model) *Consider the conditional* Σ*-equation with* $\neq$

$$(\forall X)t = t' \text{ if } (C_= \text{ and } C_\neq).$$

Let $\{ieq_j\}_{j\in J} = I \cup C_\neq$. *If there is* $j \in J$ *such that* ieq_j *is not deducible from* $E \cup C_=$, *then*

$$(\forall X)t = t' \text{ if } (C_= \text{ and } C_\neq)$$

is deducible.

10. (Inequality) *If*

$(\forall X)t = t'$

is not deducible, then

$(\forall X)t \neq t'$

is deducible.

When a conditional Σ-equation with $\neq$ ceq is deducible from $E \cup I$, we write: $E \cup I \vdash_\Sigma ceq$. $\square$

The completeness theorem of the deduction system is as follows.

Theorem 10 *Let (Σ, E, I) be an equational and inequational specification such that:*

1. *(Σ, E, I) is splitable on $\vdash_\Sigma$ and*
2. *undeducibility decision problem is decidable on (Σ, E, I) and specifications occurring in the deduction of Definition 50.*

Given a conditional Σ-equation with $\neq$ or a Σ-inequation eq, then
$E \cup I \vdash_\Sigma eq$ *iff* $E \cup I \models_\Sigma eq$.

Proof : If we show the case that eq is a Σ-equation, by Proposition 9, the theorem is showed. So, from now on, we show the case that eq is a Σ-equation.
(1) *The case that there is no (Σ, E, I)-model.*
Consider a Σ-equation eq. By the assumption of the case, $E \cup I \models_\Sigma$ eq. By Proposition 7, $E \not\models_\Sigma I$. So, by Rule 9, $E \cup I \vdash_\Sigma$ eq. Therefore, the theorem is showed.
(2) *The case that there is a (Σ, E, I)-model.*
($\Rightarrow$) It is straightforward from Definition 50.
($\Leftarrow$) We show the case by following the proof of completeness theorem of equational specification in [16]. The structure of this proof is as follows: we construct a Σ-algebra M such that if M satisfies eq then eq is deducible from $E \cup I$; then we show that M is a (Σ, E, I)-model.

First, we show that the following property of terms $t, t' \in T_\Sigma(X)$ defines a Σ-congruence on $T_\Sigma(X)$:

(D) $E \cup I \vdash_\Sigma (\forall X)t = t'$.

Let us denote this relation $\sim_{E(X)}$. Then Rules 2-4 say that $\sim_{E(X)}$ is an equivalence relation on $T_\Sigma(X)$. By applying Rule 5 to terms t of the form $\sigma(x_1, \ldots, x_k)$ for $\sigma \in \Sigma$, we see that $\sim_{E(X)}$ is a congruence.

Now we can form the quotient of $T_\Sigma(X)$ by $\sim_{E(X)}$, which we denote by M. Then by the construction of M, for each $t, t' \in T_\Sigma(X)$ we have

($*$) $[t] = [t']$ in M iff (D) holds,

where $[t]$ denotes the $\sim_{E(X)}$-equivalence class of t.

We next show the key property of M, that

($**$) $M \models_\Sigma (\forall X)t = t'$ implies that (D) holds.

Because $M \models_\Sigma (\forall X)t = t'$, we can use the inclusion $i_X : X \to M$ sending x to $[x]$ as an assignment to get that $[t] = [t']$ in M; then (D) holds by ($*$).

We now show that M is a (Σ, E)-model. Let $(\forall Y)t = t'$ if C be a conditional Σ-equation with $\neq$ and $\theta : Y \to M$ be an assignment such that $\theta(u_i) = \theta(u'_i)$ for each

$(u_i = u_i')$ in $C_=$ and $\theta(v_j) \neq \theta(v_j')$ for each $(v_j \neq v_j')$ in $C_{\neq}$. Then for each $y \in Y$ we can choose a representative $t_y \in T_\Sigma(X)$ such that $\theta(y) = [t_y]$ in M. Now let $\phi : Y \to T_\Sigma(X)$ be the substitution sending y to t_y. Then $\theta(y) = [\phi(y)]$ for each $y \in Y$, and therefore $\theta(t) = [\phi(t)]$ in M for any $t \in T_\Sigma(Y)$. Therefore, $[\phi(u_i)] = [\phi(u_i')]$ and $[\phi(v_j)] \neq [\phi(v_j')]$ holds in M, and by the property $(*)$, $E \cup I \models_\Sigma \phi(u_i) = \phi(u_i')$ and $E \cup I \models_\Sigma \phi(v_j) \neq \phi(v_j')$. Therefore by Rule 6, $E \cup I \models_\Sigma \phi(t) = \phi(t')$, and hence by $(*)$, $\theta(t) = \theta(t')$ holds in M, and thus the conditional Σ-equation with $\neq$ $(\forall Y)t = t'$ if C holds in M. So, M is a (Σ, E)-model.

Finally, we show that M is a (Σ, E, I)-model. By the assumption of the case, there is a (Σ, E, I)-model, and hence by Corollary 8, the (Σ, E)-model M is a (Σ, E, I)-model. □

6.3 Double Term Rewriting System with Condition

The condition that we can deal with equational and inequational specifications as term rewriting systems is the following rewritable.

Definition 51 *Let (Σ, E, I) be an equational and inequational specification and $>$ be a reduction ordering. Let $\{eq_{\mathrm{lt},i}\}_{i \in I_{\mathrm{lt}}}$ be all the elements of E whose left hand sides are lt. Let $C_{\mathrm{lt},i}$ be the condition of $eq_{\mathrm{lt},i}$. We say that the equational and inequational specification (Σ, E, I) is* rewritable *if it satisfies the following conditions:*

1. *for each (Σ, E)-model M, for each assignment $as : X \to M$, for each lt, there is exactly one i that $as(C_{\mathrm{lt},i})$ is true,*
2. *for each Σ-equation in E*

   $(\forall X) lt = rt$ *if* $(u_1 = u_1'$ *and* $u_k = u_k')$ *and* $(v_1 \neq v_1'$ *and* $v_l \neq v_l')$

   for each $i \in [1, \ldots, k]$ and for each $j \in [1, \ldots, l]$,

   (a) $var(lt) \supset var(rt)$,

   (b) $var(lt) \supset var(u_i)$ *and* $var(lt) \supset var(u_i')$,

   (c) $var(lt) \supset var(v_j)$ *and* $var(lt) \supset var(v_j')$,

   (d) $lt > rt$, *and*

   (e) $u_i > u_i'$ *and* $v_i > v_i'$, *and*
3. *for each Σ-inequation in I*

   $(\forall X) lt \neq rt$

   $var(lt) \supset var(rt)$. □

The definition of double term rewriting system with condition that is an implementation of the complete deduction system of equational and inequational specification is as follows.

Definition 52 *Let (Σ, E, I) be an equational and inequational specification and $>$ be a reduction ordering such that (Σ, E, I) is rewritable. We call a 3-tuple (t, t', C) where t, t' are $\Sigma(X)$-terms and C be a set of pairs of $\Sigma(X)$-terms* a term pair with condition. The one-step rewriting relation $\to_{\Sigma,E,I}$ *is defined for sets of term pairs with condition TPS, TPS' as follows: TPS $\to_{\Sigma,E,I}$ TPS' iff*

1. *there is a term pair with condition* (t, t', C) *in TPS and we use* $\{(t, t', C)\} \cup TPS_{\text{others}}$ *to denote TPS,*

2. *there exists: a set of conditional* Σ*-equations with* $\neq$ $\{eq_{\text{lt},i}\}_{i \in I_{lt}}$ *whose forms are* $(\forall X) lt = rt_i$ *if* C_i*;* $ct \in T_\Sigma(\{z\} \cup Y)$ *having exactly one occurrence of the varialbe* z*; and a substitution* $sb : X \to T_\Sigma(Y)$ *such that:*

   $t = ct(z \leftarrow sb(lt))$ *or*
   $t' = ct(z \leftarrow sb(lt))$*, and*

3. *TPS' is*

   $\{(t_i, t', C \cup sb(C_i))\}_{i \in I'_{lt}} \cup TPS_{\text{others}}$ *or*
   $\{(t, t_i, C \cup sb(C_i))\}_{i \in I'_{lt}} \cup TPS_{\text{others}}$

   where

   (a) I'_{lt} *is the subset of* I_{lt} *that for each* $j \in I_{lt} \backslash I'_{lt}$,
   $(t, t_j, C \cup sb(C_j))$ *is an invalid term pair with condition in Definition 53.*

   (b) $t_i = ct(z \leftarrow sb(rt_i))$*, respectively.*

The term rewriting relation *is the transitive reflexive closure of the one-step rewriting relation* $\to^*_{\Sigma,E,I}$*, for which we write* $TPS \to^*_{\Sigma,E,I} TPS'$*. Let TPSS be the set of sets of term pairs with condition such that each elenemt is*

1. TPS_1 *that* $\{(t, t', \emptyset)\} \to^*_{\Sigma,E,I} TPS_1$ *or*

2. TPS_2 *that* $\{tp\} \to^*_{\Sigma,E,I} TPS_2$

where tp is an element of TPS_1*. We call the abstract reduction system* $(TPSS, \to^*_{\Sigma,E,I})$ the double term rewriting system with condition of (Σ, E, I).

Let $\sim$ *be the maximum equivalence relation on TPSS that*
$\{(t, t', C_m)\}_{m \in M} \cup TPS_{\text{others}} \sim \{(t, t', C_n)\}_{n \in N} \cup TPS_{\text{others}}$ *if*
for each (Σ, E)*-model* M*, for each assignment* $as : X \to M$*,*
$as(\cup_{m \in M} C_m) = as(\cup_{n \in N} C_n)$.
The one-step rewriting relation $\Rightarrow_{\Sigma,E,I}$ *is defined for* $QTP, QTP' \in QTPS$ *as follows:*
$QTP \Rightarrow_{\Sigma,E,I} QTP'$ *if* $TPS \to_{\Sigma,E,I} TPS'$
where TPS and TPS' are representatives of QTP and QTP', respectively. The term rewriting relation *is the transitive reflexive closure of the one-step rewriting relation* $\Rightarrow_{\Sigma,E,I}$*, for which we write* $QTP \Rightarrow^*_{\Sigma,E,I} QTP'$ *and say that* QTP rewrites to QTP' (under (Σ, E, I)). *We call the abstract reduction system* $(QTPS, \Rightarrow^*_{\Sigma,E,I})$ the quotient double term rewriting system with condition of (Σ, E, I). □

Definition 53 *Let* (Σ, E, I) *be an equational and inequational specification and* $>$ *be a reduction ordering such that* (Σ, E, I) *and the following* $(\Sigma, E \cup E_=, I)$ *are rewritable. Let* $(TPSS, \to^*_{\Sigma,E,I})$ *be the double term rewriting system with condition of* (Σ, E, I) *and TPS be in TPSS. Let* (t, t', C) *be a term pair with condition in TPS where C is* $((u_1 = u'_1)$ *and* $(u_k = u'_k))$ *and* $((v_1 \neq v'_1)$ *and* $(v_l \neq v'_l))$*. Let* $\{(u_{m,n}, u'_{m,n}, C_{m,n})\}_{n \in N_m}$ *be a normal form of* $\{(u_m, u'_m, \emptyset)\}$ *on* $(TPSS, \to^*_{\Sigma,E,I})$*. Let* $E_= = \{ceq_{m,n}\}_{m \in [1,\ldots,k], n \in N_m}$ *be the set of conditional equation with* $\neq$ *that each* $ceq_{m,n}$ *is*

1. ϵ *if* $u_{m,n} = u'_{m,n}$,

2. $(\forall X)u_{m,n} = u'_{m,n}$ *if* $C_{m,n}$ *if* $u_{m,n} > u'_{m,n}$, *or*

3. $(\forall X)u'_{m,n} = u_{m,n}$ *if* $C_{m,n}$ *if* $u'_{m,n} > u_{m,n}$.

We call the equational and inequational specification $(\Sigma, E \cup E_=, I)$ the decision specification of (t, t', C), *the double term rewriting system with condition of* $(\Sigma, E \cup E_=, I)$ the decision term rewriting system of (t, t', C), *and the quotient double term rewriting system with condition of* $(\Sigma, E \cup E_=, I)$ the quotient decision term rewriting system of (t, t', C). *We say that* (t, t', C) *is* an invalid term pair with condition *if for each* $m \in [1, \ldots, l]$,

$\{(v_p, v'_p, \emptyset)\} \rightarrow^*_{\Sigma, E \cup E_=, I} \{(v_{p,q}, v_{p,q}, C_{p,q})\}_{q \in Q_p}$

where $\{(v_{p,q}, v_{p,q}, C_{p,q})\}_{q \in Q_p}$ *is a normal form of* $\{(v_p, v'_p, \emptyset)\}$ *on* $(TPSS', \rightarrow^*_{\Sigma, E \cup E_=, I})$. *We call the problem that for a given term pair with condition* (t, t', C) *where* $(t, t', C) \in TPS$ *and* $TPS \in TPSS$, *we decide whether* (t, t', C) *is an invalid term pair with condition* the invalidity decision problem of (t, t', C). $\square$

Definition 54 *Let* (Σ, E, I) *be an equational and inequational specification and* $>$ *be a reduction ordering such that* (Σ, E, I) *and all* the decision specifications *are rewritable. Let* $(TPSS, \rightarrow^*_{\Sigma, E, I})$ *be the double term rewriting system with condition of* (Σ, E, I). *Let* $\sim$ *be the maximum equivalence relation on TPSS that*

$\{(t, t', C_m)\}_{m \in M} \cup \{(v_k, v'_k, \hat{C}_k)\}_{k \in K} \sim \{(t, t', \bar{C}_n)\}_{n \in N} \cup \{(v_k, v'_k, \hat{C}_k)\}_{k \in K}$ *if*

1. *for each* $m \in M$, *for each* $k \in K$, *for each* $h \in H$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(t_h, t'_h, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{m,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$,

2. *for each* $m \in M$, *for each* $k \in K$, *for each* $j \in J_m$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(u_{m,j}, u'_{m,j}, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{m,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$,

3. *for each* $m \in M$, *for each* $k \in K$, *for each* $j \in J_k$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(\bar{u}_{k,j}, \bar{u}'_{k,j}, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{m,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$,

4. *for each* $n \in N$, *for each* $k \in K$, *for each* $h \in H$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(t_h, t'_h, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{n,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$,

5. *for each* $n \in N$, *for each* $k \in K$, *for each* $j \in J_n$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(u_{n,j}, u'_{n,j}, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{n,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$, *and*

6. *for each* $n \in N$, *for each* $k \in K$, *for each* $j \in J_k$,
   there is $\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}$ *such that*
   $[\{(\bar{u}_{k,j}, \bar{u}'_{k,j}, \emptyset)\}] \Rightarrow^*_{\Sigma(X), E \cup C_{n,=} \cup \hat{C}_{k,=}, \emptyset} [\{(n_l, n'_l, \tilde{C}_l)\}_{l \in L}]$

where

1. I *is* $\{(\forall X) t_h \neq t'_h\}_{h \in H}$,
2. C_m *is* $(\cup_{i \in I_m}(u_{m,i} = u'_{m,i})) \cup (\cup_{j \in J_m}(v_{m,j} \neq v'_{m,j}))$,
3. $\bar{C}_n$ *is* $(\cup_{i \in I_n}(\bar{u}_{n,i} = \bar{u}'_{n,i})) \cup (\cup_{j \in J_n}(\bar{v}_{n,j} \neq \bar{v}'_{n,j}))$, *and*
4. $\hat{C}_k$ *is* $(\cup_{i \in I_k}(\hat{u}_{k,i} = \hat{u}'_{k,i})) \cup (\cup_{j \in J_k}(\hat{v}_{k,j} \neq \hat{v}'_{k,j}))$. □

The condition that double term rewriting system with condition is an implementation of the complete deduction system is defined by using the following decreasing and complete.

Definition 55 *Let* (Σ, E, I) *be an equational and inequational specification and* $>$ *be a reduction ordering such that* (Σ, E, I) *is rewritable. We say that the equational and inequational specification* (Σ, E, I) *is* decreasing *if it satisfies the following conditions:*

for each Σ*-equation in* E
$(\forall X) lt = rt$ *if* $(u_1 = u'_1$ *and* $u_k = u'_k)$ *and* $(v_1 \neq v'_1$ *and* $v_l \neq v'_l)$
for each $i \in [1, \ldots, k]$ *and for each* $j \in [1, \ldots, l]$,
$lt > u_i$ *and* $lt > v_j$. □

Definition 56 *Let* (Σ, E, I) *be an equational and inequational specification and* $>$ *be a reduction ordering such that* (Σ, E, I) *is rewritable. We say that* (Σ, E, I) *is* complete *if it satisfies the following conditions:*

1. (Σ, E, I) *is decreasing and*
2. *the quotient double term rewriting system with condition* $(QTPS, \Rightarrow^*_{\Sigma,E,I})$ *is confluent.* □

Proposition 11 *Given an equational and inequational specification* (Σ, E, I) *and a reduction ordering* $>$ *such that:*

1. (Σ, E, I) *is rewritable and*
2. *the quotient double term rewriting system with condition* $(QTPS, \Rightarrow^*_{\Sigma,E,I})$ *is confluent,*

then
$(QTPS, \Rightarrow^*_{\Sigma,E,I})$ *is complete.*

Proof : Because (Σ, E, I) is rewritable, (TPSS, $\rightarrow^*_{\Sigma,E,I}$) is terminating. Therefore, (QTPS, $\Rightarrow^*_{\Sigma,E,I}$) is terminating. From the assumption, (QTPS, $\Rightarrow^*_{\Sigma,E,I}$) is confluent. Therefore, (QTPS, $\Rightarrow^*_{\Sigma,E,I}$) is complete. □

Proposition 12 *Given an equational and inequational specification* (Σ, E, I), *a reduction ordering* $>$, *and a term pair with condition* (t, t', C) *such that:*

1. (t, t', C) *is in* TPS *where*

   (a) TPS $\in$ TPSS *and*

   (b) $(\mathrm{TPSS}, \rightarrow^*_{\Sigma,E,I})$ *is the double term rewriting system with condition of* (Σ, E, I),

2. (Σ, E, I) *is rewritable and decreasing, and*

3. *the decision specifications occurring in the invalidity decision problem are rewritable,*

then

to solve the invalidity decision problem, at most finite invalidity decision problems must be solved.

Proof : Let $(t_i, t'_i, \cup_{j\in[1,\ldots,i]} C_j)$ be the term pairs with condition such that:

1. $\{(t_0, t'_0, C_0)\} \rightarrow^*_{\Sigma,E,I} \{(t_i, t'_i, C_i)\} \cup \mathrm{TPS}_{\mathrm{others},i} \rightarrow^*_{\Sigma,E,I} \{(t, t', C)\} \cup \mathrm{TPS}_{\mathrm{others}}$,

2. C_0 is $\emptyset$, and

3. for each i, there exists a conditional Σ-equation with $\neq$

   $(\forall X)\mathrm{lt} = \mathrm{rt}$ if $C_{\mathrm{axm}} \in E$ such that:

   (a) $t_i = \mathrm{ct}_i(z \leftarrow \mathrm{sb}(\mathrm{lt}))$, $t_{i+1} = \mathrm{ct}_i(z \leftarrow \mathrm{sb}(\mathrm{rt}))$, and $C_{i+1} = \mathrm{sb}(C_{\mathrm{axm}})$ or

   (b) $t'_i = \mathrm{ct}_i(z \leftarrow \mathrm{sb}(\mathrm{lt}))$, $t'_{i+1} = \mathrm{ct}_i(z \leftarrow \mathrm{sb}(\mathrm{rt}))$, and $C_{i+1} = \mathrm{sb}(C_{\mathrm{axm}})$.

Let $((u_1 = u'_1$ and $u_k = u'_k)$ and $(v_1 \neq v'_1$ and $v_l \neq v'_l))$ be C. Because (Σ, E, I) is rewritable and decreasing,

$(C1)$ $t_0 > \mathrm{ct}_{u_m}[u_m]$ and $t_0 > \mathrm{ct}_{v_p}[v_p]$.

Consider an invalidity decision problem of $(\hat{t}, \hat{t}', \hat{C})$ that occurs in the invalidity decision problem. We assume that the problem is caused by the construction of a decision specification $(\Sigma, E \cup \bar{E}_=, I)$. Now let $(\Sigma, E \cup \bar{E}_= \cup \hat{E}_=, I)$ be the decision specification of $(\hat{t}, \hat{t}', \hat{C})$ and let $((\hat{u}_1 = \hat{u}'_1$ and $\hat{u}_k = \hat{u}'_k)$ and $(\hat{v}_1 \neq \hat{v}'_1$ and $\hat{v}_l \neq \hat{v}'_l))$ be $\hat{C}$. New invalidity decision problems may have to be solved in the rewriting process of

1. $(\hat{u}_m, \hat{u}'_m, \emptyset)$ and

2. $(\hat{v}_p, \hat{v}'_p, \emptyset)$.

Consider the rewriting process of $(\hat{u}_m, \hat{u}'_m, \emptyset)$. Let $\{(\hat{u}_{m,n}, \hat{u}'_{m,n}, \hat{C}_{m,n})\}_{n\in N_m}$ be a set of term pairs with condition that $\{(\hat{u}_m, \hat{u}'_m, \emptyset)\} \rightarrow^*_{\Sigma,E\cup\bar{E}_=,I} \{(\hat{u}_{m,n}, \hat{u}'_{m,n}, \hat{C}_{m,n})\}_{n\in N_m}$. Note that the invalidity decision problem of $(\hat{u}_{m,n}, \hat{u}'_{m,n}, \hat{C}_{m,n})$ is a new invalidity decision problem that must be solved. Because (Σ, E, I) is rewritable and decreasing and $(\Sigma, E\cup\bar{E}_=, I)$ is rewritable,

$(C2)$ $\hat{u}_m > \mathrm{ct}_{\hat{u}_{m,n}}[\hat{u}_{m,n}]$.

Consider the rewriting process of $(\hat{v}_p, \hat{v}'_p, \emptyset)$. Let $\{(\hat{v}_{p,q}, \hat{v}'_{p,q}, \hat{C}_{p,q})\}_{n\in N_p}$ be a set of term pairs with condition that $\{(\hat{v}_p, \hat{v}'_p, \emptyset)\} \rightarrow^*_{\Sigma,E\cup\bar{E}=\cup\hat{E}_=,I} \{(\hat{v}_{p,q}, \hat{v}'_{p,q}, \hat{C}_{p,q})\}_{n\in N_p}$. Note that the invalidity decision problem of $(\hat{v}_{p,q}, \hat{v}'_{p,q}, \hat{C}_{p,q})$ is a new invalidity decision problem that must be solved. Because (Σ, E, I) is rewritable and decreasing and $(\Sigma, E \cup \bar{E}_= \cup \hat{E}_=, I)$ is rewritable,

$(C3)$ $\hat{v}_p > \mathrm{ct}_{\hat{v}_{p,q}}[\hat{v}_{p,q}]$.

By (C1), (C2), (C3), and induction,

1. $t_0 > \mathrm{ct}_{u_m}[u_m] > \cdots > \mathrm{ct}_{u_m}[\cdots[\hat{u}_m]\cdots] > \mathrm{ct}_{u_m}[\cdots[\mathrm{ct}_{\hat{u}_{m,n}}[\hat{u}_{m,n}]]\cdots]$ and
2. $t_0 > \mathrm{ct}_{v_p}[v_p] > \cdots > \mathrm{ct}_{v_p}[\cdots[\hat{v}_p]\cdots] > \mathrm{ct}_{v_p}[\cdots[\mathrm{ct}_{\hat{v}_{p,q}}[\hat{v}_{p,q}]]\cdots]$.

Because $>$ is Noetherian, to solve the invalidity decision problem, finite layers of invalidity decision problems must be solved. Moreover, because $>$ is Noetherian, for each layer, there are at most finite invalidity decision problems that must be solved. Therefore, to solve the invalidity decision problem, at most finite invalidity decision problems must be solved. □

Proposition 13 *Given an equational and inequational specification (Σ, E, I) and a reduction ordering $>$ such that:*

1. *(Σ, E, I) is rewritable and complete and*
2. *all the decision specifications are rewritable,*

then
all the quotient decision term rewriting systems are complete.

Proof : Because $>$ is Noetherian, all the quotient decision term rewriting systems are terminating. From Definition 53, the addition of $E_=$ in the definition does not destroy confluent property. And (Σ, E, I) is complete, therefore, all the quotient decision term rewriting systems are confluent. Therefore, all the quotient decision term rewriting systems are complete. □

Corollary 14 *Given an equational and inequational specification (Σ, E, I), a reduction ordering $>$, and a term pair with condition (t, t', C) such that:*

1. *(t, t', C) is in TPS where*
   (a) *TPS $\in$ TPSS and*
   (b) *$(TPSS, \rightarrow^*_{\Sigma,E,I})$ is the double term rewriting system with condition of (Σ, E, I),*
2. *(Σ, E, I) is rewritable and decreasing, and*
3. *the decision specifications occurring in the invalidity decision problem are rewritable,*

then
the invalidity decision problem of (t, t', C) is decidable. □

From now on, we show that double term rewriting system with condition is an implementation of the complete deduction system.

Proposition 15 *Given an equational and inequational specification (Σ, E, I) and a reduction ordering $>$ such that:*

1. *(Σ, E, I) is rewritable and*
2. *all the decision specifications are rewritable,*

then
(Σ, E, I) is splitable on $\vdash_\Sigma$.

Proof : Consider conditional Σ-equations with $\neq \mathrm{ceq}_j$ $(j \in J)$
$(\forall X)t = t'$ if C_j that $E \cup I \vdash_\Sigma \mathrm{ceq}_j$.
Because (Σ, E, I) and all the decision specifications are rewritable, there is a set of conditions $\cup_{k \in K} \hat{C}_k$ such that:

1. $J \subset K$,
2. $\hat{C}_j$ is C_j for each $j \in J$, and
3. for each (Σ, E, I)-model M, for each assignment as $: X \to M$, there is exactly one i that $\mathrm{as}(\hat{C}_i)$ is true.

From 3 of the properties of $\hat{C}_k$, the following relations hold:

1. for each $j \in J$, for each $k \in K \backslash J$, for each (Σ, E, I)-model M and for each assignment as $: X \to M$,
   $\mathrm{as}(\hat{C}_j \cup \hat{C}_k) = \mathrm{false}$.
2. for each (Σ, E, I)-model M and for each assignment as $: X \to M$, there is $j \in J$ or $k \in K \backslash J$ such that:
   $\mathrm{as}(\hat{C}_j) = \mathrm{true}$ or $\mathrm{as}(\hat{C}_k) = \mathrm{true}$.

Therefore, $\{\hat{C}_k\}_{k \in K \backslash J}$ is the complement set of $\{C_j\}_{j \in J}$. Therefore, (Σ, E, I) is splitable on $\vdash_\Sigma$. □

Proposition 16 *Let (Σ, E, I) be an equational and inequational specification (Σ, E, I) and $>$ be a reduction ordering such that:*

1. *(Σ, E, I) is rewritable and complete and*
2. *all the decision specifications are rewritable.*

Let $\{(t, t', C)\}$ and $\{(n_j, n'_j, C_j)\}_{j \in J}$ be sets of term pairs with condition on (Σ, E, I) or the decision specifications such that $[\{(n_j, n'_j, C_j)\}_{j \in J}]$ is the normal form of $[\{(t, t', C)\}]$ on the quotient double term rewriting system with condition. Then,
$E \cup I \vdash_\Sigma (\forall X)t = t'$ *if* C *if*
$\forall j \in J \ . \ n_j = n'_j$ *and*
$E \cup I \vdash_\Sigma (\forall X)t \neq t'$ *if*
$C = \emptyset$ *and* $\exists i \in J \ . \ n_i \neq n'_i$.

Proof : Each single step of rewriting and the decisions is an application of the deduction rules in Definition 50. Induction now extends the result from $\Rightarrow_{\Sigma,E,I}$ to the relation $\Rightarrow^*_{\Sigma,E,I}$. So, the proposition holds. □

Corollary 17 *Let (Σ, E, I) be an equational and inequational specification (Σ, E, I) and $>$ be a reduction ordering such that:*

1. *(Σ, E, I) is rewritable and complete and*
2. *all the decision specifications are rewritable.*

undeducibility decision problem is decidable on (Σ, E, I) *and specifications occurring in the deduction of Definition 50.* □

Proposition 18 *Given an equational and inequational specification* (Σ, E, I)*, a reduction ordering* $>$*, a* Σ*-term* t*, and a set of term pairs with condition* $\{(n_j, n'_j, C_j)\}_{j\in J}$ *such that:*

1. (Σ, E, I) *is rewritable and complete,*
2. *all the decision specifications are rewritable, and*
3. $[\{(n_j, n'_j, C_j)\}_{j\in J}]$ *is the normal form of* $[\{(t, t, \emptyset)\}]$ *on the quotient double term rewriting system with condition*

then

1. *for each* $j \in J$, $n_j = n'_j$,
2. *the normal form of* $[\{(t, n_j, C_j)\}_{j\in J}]$ *is* $[\{(n_j, n_j, C_j)\}_{j\in J}]$, *and*
3. *the normal form of* $[\{(n_j, t, C_j)\}_{j\in J}]$ *is* $[\{(n_j, n_j, C_j)\}_{j\in J}]$.

Proof : From Definition 52, the first terms and the second terms of the term pairs with condition are rewritten independently. So, the proposition holds. □

Corollary 19 *Given an equational and inequational specification* (Σ, E, I)*, a reduction ordering* $>$*, a* Σ*-term* t*, and a set of term pairs with condition* $\{(n_j, n'_j, C_j)\}_{j\in J}$ *such that:*

1. (Σ, E, I) *is rewritable and complete,*
2. *all the decision specifications are rewritable, and*
3. $[\{(n_j, n'_j, C_j)\}_{j\in J}]$ *is the normal form of* $[\{(t, t, \emptyset)\}]$ *on the quotient double term rewriting system with condition*

then

1. $E \cup I \vdash_\Sigma (\forall X)t = n_j$ *if* C_j *for each* $j \in J$ *and*
2. $E \cup I \vdash_\Sigma (\forall X)n_j = t$ *if* C_j *for each* $j \in J$. □

Proposition 20 *Given an equational and inequational specification* (Σ, E, I) *and a reduction ordering* $>$ *such that:*

1. (Σ, E, I) *is rewritable and complete and*
2. *all the decision specifications are rewritable*

then
the normal form of $[\{(t, t', \emptyset)\}]$ *on the quotient double term rewriting system has the form* $[\{n_i, n_i, C_i\}_{i\in I}]$ *iff* $E \cup I \vdash_\Sigma (\forall X)t = t'$

Proof : $\Rightarrow$ It is proved in Proposition 16.
$\Leftarrow$ By Corollary 19,

1. $E \cup I \vdash_\Sigma (\forall X)\bar{n}_j = t$ if $\bar{C}_j$ $(j \in J)$ and

2. $E \cup I \vdash_\Sigma (\forall X)t' = \hat{n}_k$ if $\hat{C}_k$.

Therefore $E \cup I \vdash_\Sigma (\forall X)\bar{n}_j = \hat{n}_k$ if $\bar{C}_j \cup \hat{C}_k$. Therefore, $[\{(t, t', \emptyset)\}] \Rightarrow_{\Sigma,E,I} [\{(\bar{n}_j, \bar{n}_j, \bar{C}_j \cup \hat{C}_k\}_{j\in J,k\in K}]$. So, the proposition is showed. □

Theorem 21 *Given an equational and inequational specification (Σ, E, I) and a reduction ordering $>$ such that:*

1. *(Σ, E, I) is rewritable and complete and*

2. *all the decision specifications are rewritable*

then

the normal form of $[\{(t, t', \emptyset)\}]$ on the quotient double term rewriting system has the form $[\{n_i, n_i, C_i\}_{i\in I}]$ iff $E \cup I \models_\Sigma (\forall X)t = t'$

Proof : By Corollary 17, undeducibility decision problem is decidable on (Σ, E, I) and all the decision specifications. So, by Theorem 10 and Proposition 20, the theorem is proved. □

Chapter 7

LFMB

In LFMB, we formalize AA-trees model of component specifications by using projection-style behavioral specifications. Projection-style behavioral specification is a kind of behavioral specification.

We specify behavior of components, behavior of software, and how to combine components to construct the software by using projection-style behavioral specifications. As consistency verification, we verify whether the combination of the components satisfies the behavior of the software by using the projection-style behavioral specifications.

The target problem of LFMB is the above verification and the simple logic of LFMB is behavioral logic.

In Chapter 7, firstly, we discuss projection-style behavioral specification. Secondly, we formalize AA-trees model of component specifications by using projection-style behavioral specifications. As we discussed in Chapter 5, an AA-trees model is specified by using UML with OCL. Then, we discuss a translation method from UML diagrams with OCL descriptions into projection-style behavioral specifications. Finally, we discuss consistency verification methods of the UML diagrams by using the projection-style behavioral specifications.

7.1 A Formalization of Tree Architecture by using Projection-style Behavioral Specification

Projection-style behavioral specification is a special class of algebraic behavioral specification [5, 10, 15]. We developed projection-style behavioral specification to formalize event model of tree architecture. We use projection-style behavioral specification for formalizing AA-trees model of component specifications. The idea of it dates back to [21, 26, 27].

Projection-style behavioral specification is component specification. As we discussed in Definition 23, we call the data part of a projection-style behavioral specification *a data specification*.

7.1.1 Data structures

In data specification, we specify data structures.

7.1.2 Event model of component

We specify an event model of a component by using a component specification, which is an algebraic behavioral specification.

Observations and actions

We assign observations and actions of the event model to observations and actions of the algebraic behavioral specification.

Effect axioms

We specify effects of an action as changes between observations' values immediately before and after the action has happened.

Definition 57 *Let* $(V, \Psi, \{h\}, \Sigma)$ *be a hidden signature,* Σ_{obs} *be a set of observations, and* Σ_{act} *be a set of actions such that* $\Sigma = \Psi \cup \Sigma_{\text{obs}} \cup \Sigma_{\text{act}}$. *Consider an action* acti *and an observation* obse. *We call a* Σ*-equation* an effect axiom of acti by obse *if it has the following form:*

1. $(\forall X) obse(DS, acti(DS', S)) = F[obse(DS, S)]$.

We use $\text{effaxm}_{\text{obse,acti}}$ *to denote the effect axiom.* □

Event model of component

We specify an event model of a component by using the following primitive component specification.

Definition 58 *Let* $(V, \Psi, \{h\}, \Sigma)$ *be a hidden signature,* Σ_{obs} *be a set of observations, and* Σ_{act} *be a set of actions such that* $\Sigma = \Psi \cup \Sigma_{\text{obs}} \cup \Sigma_{\text{act}}$. *Let* E_{effaxm} *be* $\{\text{effaxm}_{\text{obse,acti}} \mid \forall obse \in \Sigma_{\text{obs}}, \forall acti \in \Sigma_{\text{act}}\}$. *We call the algebraic behavioral specification* $(V, \Psi, E_\Psi, \{h\}, \Sigma, E_{\text{effaxm}})$ a primitive component specification.

For the primitive component specification, we call h the root sort of the primitive component specification. □

Note that the root sort is the set of states of the component.

Example 15 *Consider PUT-A of Example 3. The primitive component specification of PUT group component, like PUT-A is as follows:*

```
mod* PUT { pr(BOOL+MACHINE+FILE) *[ Put ]*
 bop getremote  : Put -> Machine
 bop isinlocal  : File Put -> Bool
 bop isinremote : File Machine Put -> Bool
 bop setremote  : Machine Put -> Put
 bop put        : File Put -> Put
 var P : Put      vars I J : File
 var M : Machine
 eq isinlocal(I, put(J,P))=isinlocal(I, P) .
 ceq isinremote(I, M, put(J, P)) = t
     if (I == J) and (getremote(P) == M) .
```

[The remaining codes are omitted.]

In CafeOBJ , bop, var(s), *and* (c)eq *declare observations and actions, variables, and (conditional) equations, respectively.* Put *surrounded by* *[*and*]* *is a hidden sort (type), that is the set of PUT group component's states.* getremote, isinlocal, *and* isinremote *are observations.* setremote *and* put *are actions. The first equation specifies the effect of put on states through isinlocal, i.e. put does not add or does not delete files on the local machine. The second equation specifies the effect of put on states through isinremote, i.e. put transfers the specified file to the target remote machine.* □

7.1.3 Composite component

A composite component is constructed from components and a connector. The connector delegates observations and actions of the composite component to those of the constructing components. We use the following projections to denote the delegation.

Definition 59 *Let* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$ *be an algebraic behavioral specification such that* $H_i \cap H_j = \emptyset$ *if* $i \neq j$. *Let* h_i *be in* H_i. *Let* $\Sigma_{\mathrm{obs},i}$ *and* $\Sigma_{\mathrm{act},i}$ *be sets of observations and actions in* Σ_i, *respectively. Let* $(V, \Psi, \{h\}, \Sigma)$ *be a hidden signature such that* $h \notin H_i$ *for each* i. *Let* (Ψ, E_Ψ) *be a data specification. We call the operator whose rank is* (h, h_i) a projection from h to h_i. *We use* $proj_{h,h_i}$ *to denote the projection.*

Let $\hat{\Sigma}$ *be* $\Sigma \cup (\cup_{i \in I} \Sigma_i)$. *Let* Σ_{obs} *and* Σ_{act} *be the sets of all the observations and all the actions of* Σ, *respectively. Consider an observation* obse *in* Σ_{obs}. *We call a* $\hat{\Sigma}$*-equation* a projection axiom of *obse if it has the following form:*

1. $(\forall X) obse(DS, S) = F[obse_{c1}(DS_1, proj_{h,h_{c1}}(S)), \ldots, obse_{ck}(DS_k, proj_{h,h_{ck}}(S))]$

where $c1, \ldots, ck \in I$ *and* $obse_{c1}, \ldots, obse_{ck}$ *are observations in* Σ_{ck}, *respectively. We use* $obspj_{\mathrm{obse}}$ *to denote the projection axiom. Consider an action* acti *in* Σ_{act}. *We call a* $\hat{\Sigma}$*-equation* a projection axiom of *acti* to h_i *if it has the following form:*

1. $(\forall X) proj_{h,h_i}(acti(DS, S)) = acti_k(DS_k, \cdots acti_1(DS_1, proj_{h,h_i}(S)) \cdots)$

where $acti_1, \ldots, acti_k$ *are actions in* Σ_i. *We use* $actpj_{\mathrm{acti},h_i}$ *to denote the projection axiom.*

Let Σ_{proj} *be* $\{proj_{h,h_i} \mid \forall i \in I\}$. *Let* $obspj_{\mathrm{obse}}$ *be a projection axiom of* obse *for each* obse *in* Σ_{obs}. *Let* $actpj_{\mathrm{acti},h_i}$ *be a projection axiom of* acti *to* h_i *for each* acti *in* Σ_{act} *and for each* i *in* I. *Let* E_{obspj} *be* $\{obspj_{\mathrm{obse}} \mid \forall obse \in \Sigma_{\mathrm{obs}}\}$ *and* E_{actpj} *be* $\{actpj_{\mathrm{acti},h_i} \mid \forall acti \in \Sigma_{\mathrm{act}}, \forall i \in I\}$. *We call a 8-tuple* $(V, \Psi, E_\Psi, h, \Sigma, \{h_i\}_{i \in I}, \Sigma_{\mathrm{proj}}, E_{\mathrm{obspj}} \cup E_{\mathrm{actpj}})$ a connector of $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$. □

Consider an action of a composite component. To get the effects of the action, sequential execution of actions of the constructing components may be necessary. We use the following sequentializing operators to denote the sequential execution.

Definition 60 *Let* $(V, \Psi, \{h\}, \Sigma)$ *be a hidden signature and* Σ_{act} *be the set of all the actions of* Σ. *Consider an action* acti *whose rank is* $(dseq\ s, s)$. *Let* seqop_i $(i \in I)$ *be operators whose ranks are* $(dseq\ s, s)$ *and* $\hat{\Sigma}$ *be* $\Sigma \cup (\cup_{i \in I} \{seqop_{\mathrm{acti},i}\})$. *We call each* $seqop_{\mathrm{acti},i}$ a sequentializing operator of acti *and call acti* a sequentialized action *if there is a* $\hat{\Sigma}$*-equation that has the following form:*

1. $(\forall X)\, acti(DS, S) = seqop_{\mathrm{acti},k}(DS, \cdots seqop_{\mathrm{acti},1}(DS, S)\cdots)$.

We call the $\hat{\Sigma}$*-equation* a sequentializing axiom of acti *and use* $seqaxm_{\mathrm{acti}}$ *to denote it. We use* $\Sigma_{\mathrm{seqop,acti}}$ *to denote the set of all the sequentializing operators* $\{seqop_i\}_{i\in I}$. □

We extend the definition of *connectors* (Definition 59) for dealing with sequential execution of actions of the constructing components.

Definition 61 *Let* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ *and so on be those in Definition 59. Let* Σ_{sact} *be a set of sequentialized actions in* Σ_{act} *and* Σ_{seqop} *be* $\cup_{\mathrm{acti}\in\Sigma_{\mathrm{sact}}}\Sigma_{\mathrm{seqop,acti}}$. *Let* E_{seqaxm} *be* $\{seqaxm_{\mathrm{acti}}\}_{\mathrm{acti}\in\Sigma_{\mathrm{sact}}}$. *Let* $\tilde{\Sigma}_{\mathrm{act}}$ *be* $(\Sigma_{\mathrm{act}}\backslash\Sigma_{\mathrm{sact}}) \cup \Sigma_{\mathrm{seqop}}$ *and* $\tilde{E}_{\mathrm{actpj}}$ *be* E_{actpj} *defined by using* $\tilde{\Sigma}_{\mathrm{act}}$ *instead of* Σ_{act}. *Let* $\tilde{\Sigma}$ *be* $((\Sigma\backslash\Sigma_{act}) \cup \tilde{\Sigma}_{act}) \cup (\cup_{i\in I}\Sigma_i)$. *We call a 8-tuple* $(V, \Psi, E_\Psi, h, \tilde{\Sigma}, \{h_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E_{\mathrm{obspj}}\cup\tilde{E}_{\mathrm{actpj}}\cup E_{\mathrm{seqaxm}})$ a connector of $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$. □

To specify composite components, we use the following combinations.

Definition 62 *Let* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$ *be an algebraic behavioral specification such that* $H_i \cap H_j = \emptyset$ *if* $i \neq j$. *Let* $(V, \Psi, E_\Psi, h, \Sigma, \{h_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E)$ *be a connector of* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$. *Let* $\hat{V}$ *be* $V \cup (\cup_{i\in I}V_i)$, $\hat{\Psi}$ *be* $\Psi \cup (\cup_{i\in I}\Psi_i)$, $\hat{E_\Psi}$ *be* $E_\Psi \cup (\cup_{i\in I}E_{\Psi,i})$, $\hat{H}$ *be* $\{h\} \cup (\cup_{i\in I}H_i)$, *and* $\hat{\Sigma}$ *be* $\Sigma \cup \Sigma_{\mathrm{proj}} \cup (\cup_{i\in I}\Sigma_i)$, $\hat{E}$ *be* $E \cup (\cup_{i\in I}E_i)$. *We call the algebraic behavioral specification* $(\hat{V}, \hat{\Psi}, \hat{E_\Psi}, \hat{H}, \hat{\Sigma}, \hat{E})$ a combination of $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$ by $(V, \Psi, E_\Psi, h, \Sigma, \{h_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E)$. *For the combination, we call* h the root sort of the combination. □

We specify composite components by using the following component specifications.

Definition 63 Component specifications *are inductively defined as follows:*

1. *let* $(V_i, \Psi_i, E_{\Psi,i}, \{h_i\}, \Sigma_i, E_i)$ $(i \in I)$ *be primitive component specifications such that* $h_i \neq h_j$ *if* $i \neq j$ *and let* $(V, \Psi, E_\Psi, h, \Sigma, \{h_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E)$ *be a connector of* $(V_i, \Psi_i, E_{\Psi,i}, \{h_i\}, \Sigma_i, E_i)$ $(i \in I)$, *then the combination* $(\hat{V}, \hat{\Psi}, \hat{E_\Psi}, \hat{H}, \hat{\Sigma}, \hat{E})$ *is a component specification and*

2. *let* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$ *be component specifications such that* $H_i \cap H_j = \emptyset$ *if* $i \neq j$ *and let* $(V, \Psi, E_\Psi, h, \Sigma, \{h_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E)$ *be a connector of* $(V_i, \Psi_i, E_{\Psi,i}, H_i, \Sigma_i, E_i)$ $(i \in I)$, *then the combination* $(\hat{V}, \hat{\Psi}, \hat{E_\Psi}, \hat{H}, \hat{\Sigma}, \hat{E})$ *is a component specification.* □

Note that each h_i is the set of states of a constructing component and h is the set of states of a composite component.

Example 16 *Consider PUT of Example 4. The primitive component specification of PUT group component that specifies how to combine FTP group component and INFO group component is as follows:*

```
mod* PUT { pr(BOOL+MACHINE+FILE)
           pr(FTP+INFO)  *[ Put ]*
 bop getremote  : Put -> Machine
 bop isinlocal  : File Put -> Bool
 bop isinremote : File Machine Put -> Bool
```

```
bop setremote  : Machine Put -> Put
bop put        : File Put -> Put
op ftp         : Put -> Ftp
op info        : Put -> Info
eq getremote(P) = getmachine(info(P)) .
eq ftp(put(I, P))
    = put(I, getmachine(info(P)),
             name(getmachine(info(P))),
             passwd(getmachine(info(P))),
             ftp(P)) .
eq info(put(I, P)) = info(P) .
```

[The remaining codes are omitted.]

pr(FTP+INFO) *declares that primitive component specifications of FTP group component and INFO group component are imported.* ftp *and* info *are projections to the states of FTP group component and of INFO group component, respectively. The first equation specifies that an observation* getremote *corresponds to an observation* getmachine *of INFO group component. The second and the third equations specify that an action* put *corresponds to an action* put *on FTP group component and it does not influence the state of INFO group component.* □

Definition 64 *We call component specifications* projection-style behavioral specifications. □

7.1.4 Conditional component specification

A component specification can have conditional equations. By replacing equations $(\forall X)\mathrm{lt} = \mathrm{rt}$ in Definition 63 with the set of conditional equations $(\forall X)\mathrm{lt} = \mathrm{rt}_i$ if C_i that each pair $(\mathrm{lt}, \mathrm{rt}_i)$ satisfies the constraint of the pair $(\mathrm{lt}, \mathrm{rt})$, we construct a conditional component specification.

Definition 65 *Consider an action acti and an observation obse. We call the following set of conditional Σ-equation with $\neq$ $\{ceq_i\}_{i\in I}$* the effect axiom set of acti by obse*:*

1. *each ceq_i has the form:*

   $(\forall X)\, obse(DS, acti(DS', S)) = F_i[obse(DS, S)]$ *if* C_i

   where F_i is $\bar{\Sigma}(X)$-term and $\bar{\Sigma}$ is $\Sigma \backslash \Sigma_{act}$,

2. *each C_i has the form:*

   $(u_{i,1} = u'_{i,1})$ *and* $\cdots$ *and* $(u_{i,k} = u'_{i,k})$ *and* $(u_{i,k+1} \neq u'_{i,k+1})$ *and* $\cdots$ *and* $(u_{i,l} \neq u'_{i,l})$,

3. *each $u_{i,j}$ is $\bar{\Sigma}(X)$-term and each $u'_{i,j}$ is $\Psi(X)$-term, and*

4. *for each Σ-algebra M, for each assignment $as : X \to M$, there is exactly one i that $as(C_i)$ is true.* □

Definition 66 *Consider an observation obse whose sort is h and the set of projections* $\{proj_{h,h_i}\}_{i\in I}$. *We call the following set of conditional* Σ*-equation with* $\neq$ the projection axiom set of obse*:*

1. *each* ceq_i *has the form:*

   $(\forall X)\, obse(DS, S) = F_i[\cdots, obse_{i,j}(DS_{i,j}, proj_{h,h_{c_{i,j}}}(S)), \cdots]$ *if* C_i

   where $F_i[\cdots]$ *is* $\tilde{\Sigma}(X)$*-term and* $\tilde{\Sigma}$ *is* $(\cup_{i\in I}\Sigma_{\mathrm{obs},h_i}) \cup \Sigma_{\mathrm{proj}} \cup \Psi$,

2. *each* C_i *has the form:*

   $(u_{i,1} = u'_{i,1})$ *and* $\cdots$ *and* $(u_{i,k} = u'_{i,k})$ *and* $(u_{i,k+1} \neq u'_{i,k+1})$ *and* $\cdots$ *and* $(u_{i,l} \neq u'_{i,l})$,

3. *each* $u_{i,j}$ *is* $\tilde{\Sigma}(X)$*-term and each* $u'_{i,j}$ *is* $\Psi(X)$*-term, and*

4. *for each* Σ*-algebra* M, *for each assignment* $as : X \to M$, *there is exactly one* i *that* $as(C_i)$ *is true.* $\square$

Definition 67 *Consider an action acti whose sort is h and a projection* $proj_{h,h'}$. *We call the following set of conditional* Σ*-equation with* $\neq$ the projection axiom set of acti to h'*:*

1. *each* ceq_i *has the form:*

   $proj_{h,h'}(acti(DS, S)) > acti_{i,k_i}(DS_{i,k_i}, \cdots acti_{i,1}(DS_{i,1}, proj_{h,h'}(S)) \cdots)$,

2. *each* C_i *has the form:*

   $(u_{i,1} = u'_{i,1})$ *and* $\cdots$ *and* $(u_{i,k} = u'_{i,k})$ *and* $(u_{i,k+1} \neq u'_{i,k+1})$ *and* $\cdots$ *and* $(u_{i,l} \neq u'_{i,l})$,

3. *each* $u_{i,j}$ *is* $\tilde{\Sigma}(X)$*-term and each* $u'_{i,j}$ *is* $\Psi(X)$*-term, and*

4. *for each* Σ*-algebra* M, *for each assignment* $as : X \to M$, *there is exactly one* i *that* $as(C_i)$ *is true.* $\square$

In the verification about component specifications that include conditional equations with $\neq$, the following descent ordering is useful.

Definition 68 *Let* (V, Ψ, H, Σ) *be a hidden signature and* $>_{\mathrm{data}}$ *be a reduction ordering on* $\Psi(X)$. *We call the following ordering* $>$ the descent ordering of Σ (with $>_{\mathrm{data}}$)

1. *for each observation obse and for each action acti,*

   for each $\bar{\Sigma}(X)$*-term* F,

   $obse(DS, acti(DS', S)) > F[obse(DS, S)]$.

   where $\bar{\Sigma}$ *is* $\Sigma \backslash \Sigma_{act}$,

2. *for each observation obse,*

   for each set of projections $\{proj_{h,h_i}\}_{i\in I}$, *for each set of observations* $\{obs_{h_i}\}_{i\in I}$ *that the sorts of* obs_{h_i} *are* h_i, *and for each* $\tilde{\Sigma}(X)$*-term* F,

   $obse(DS, S) > F[\cdots, obse_j(DS_j, proj_{h,h_{c_j}}(S)), \cdots]$

   where h is the sort of obse and $\tilde{\Sigma}$ *is* $(\cup_{i\in I}\Sigma_{\mathrm{obs},h_i}) \cup \Sigma_{\mathrm{proj}} \cup \Psi$,

3. *for each action acti and for each projection* $proj_{h,h'}$,

   for each finite action sequence of sort h',

   $proj_{h,h'}(acti(DS, S)) > acti_k(DS_k, \cdots acti_1(DS_1, proj_{h,h'}(S)) \cdots)$

   where h *is the sort of acti,*

4. *for each sequentializing operator seqop and for each projection* $proj_{h,h'}$,

   for each finite action sequence of sort h',

   $proj_{h,h'}(seqop(DS, S)) > acti_k(DS_k, \cdots acti_1(DS_1, proj_{h,h'}(S)) \cdots)$

   where h *is the sort of seqop,*

5. *for each sequentialized action acti,*

   $acti(DS, S) > seqop_{\mathrm{acti},k}(DS, \cdots seqop_{\mathrm{acti},1}(DS, S) \cdots)$,

6. *for each action acti and for each projection* $proj_{h,h'}$,

   for each set of projections $\{proj_{h,h_i}\}_{i \in I}$, *for each set of observations* $\{obs_{h_i}\}_{i \in I}$ *that the sorts of* obs_{h_i} *are* h_i, *and for each* $\tilde{\Sigma}(X)$*-term* F,

   $proj_{h,h'}(acti(DS, S)) > F[\cdots, obse_i(DS_i, proj_{h,h_i}(S)), \cdots]$

   where h *is the sort of acti,*

7. *for each sequentializing operator seqop and for each projection* $proj_{h,h'}$,

   for each set of projections $\{proj_{h,h_i}\}_{i \in I}$, *for each set of observations* $\{obs_{h_i}\}_{i \in I}$ *that the sorts of* obs_{h_i} *are* h_i, *and for each* $\tilde{\Sigma}(X)$*-term* F,

   $proj_{h,h'}(seqop(DS, S)) > F[\cdots, obse_i(DS_i, proj_{h,h_i}(S)), \cdots]$

   where h *is the sort of seqop,*

8. *for each observation obse,*

   for each $\Psi(X)$*-term* d,

   $obse(DS, S) > d$.

9. *for each pair of* $\Psi(X)$*-terms* (d, d'),

   $d > d' \quad if \quad d >_{\mathrm{data}} d'$,

10. *for each pair of* $\Sigma(X)$*-terms* (t, t'),

    for each $\Sigma(X)$*-term* F,

    $F[t] > F[t'] \quad if \quad t > t'$, *and*

11. *for each pair of* $\Sigma(X)$*-terms* (t, t'),

    for each substitution sb,

    $sb(t) > sb(t') \quad if \quad t > t'$. □

Proposition 22 *Let* (V, Ψ, H, Σ) *be a hidden signature and* $>_{\mathrm{data}}$ *be a reduction ordering on* $\Psi(X)$. *The descent ordering of* Σ *(with* $>_{\mathrm{data}}$*) is a reduction ordering.* □

Theorem 23 *Consider a component specification that includes conditional Σ-equations with $\neq (\Sigma, E)$ and a descent ordering of $\Sigma >$, then*

1. *(Σ, E) is rewritable and complete and*

2. *all the decision specifications are rewritable.* □

7.2 A Formalization of AA-trees Model of Component Specifications by using Projection-style Behavioral Specifications

7.2.1 A formalization of AA-trees model of component specifications by using projection-style behavioral specifications

Data classes

A data class is a set of data objects. A visible sort of a projection-style behavioral specification corresponds to a set of data. So, we assign a data class to a visible sort. We call the sort *a data sort*.

Data operators

Because data operators are functions on data classes, we assign data operators *operators* of data sorts.

Component classes and states

A hidden sort of a projection-style behavioral specification corresponds to a set of states of a component. So, we assign the set of states of the component to a hidden sort.

Attributes

An attribute of an component is used for observing the state of the component. So, we assign the attribute to an observation.

Methods

A method of an component is used for changing the state of the component. So, we assign the method to an action.

Projection-lift associations

A projection-lift association is used for describing a parent-child relation. So, we assign the projection-lift association to a projection.

7.3 Translation from UML diagrams into Projection-style Behavioral Specifications

We translate data class diagrams into a data specification, translate basic class diagrams and action usecse diagrams into a primitive component specification, and translate (a) basic class diagrams, (b) decomposition class diagrams, and (c) decomposition sequence diagrams or decomposition statechart diagrams into a component specification.

7.3.1 Data class diagrams

We translate data class diagrams into a data specification (Ψ, E_Ψ).

Data class diagrams

We translate operator declarations drawn in the middle parts of class boxes into Ψ.

Translation 1
Input: *An operator declaration drawn in the middle part of a class box, i.e. a word of language for data operator declarations discussed in Section 5.2*
Output: *An operator declaration of a data specification*

The translation function F is defined as follows:

1. $F(LDOD) =$ `op` $F(DTOPWV)$ `->` $dtnm$ (Production rule (1))
2. $F(DTOPWV) =$ $dtop$ `:` (Production rule (2))
3. $F(DTOPWV) =$ $dtop$ `:` $F(DDLIST)$ (Production rule (3))
4. $F(DDLIST) =$ $F(DTDCL)$ (Production rule (4))
5. $F(DDLIST) =$ $F(DDLIST)$ $F(DTDCL)$ (Production rule (5))
6. $F(DTDCL) =$ $dtnm$ (Production rule (6)) □

OCL descriptions complementing data class diagrams

We translate invariant declarations of OCL descriptions complementing data class diagrams into E_Ψ.

Translation 2
Input: *An invariant declaration, i.e. a word of* EQD *of OCL for data discussed in Section 5.2*
Output: *An equation declaration of a data specification*

The translation function F is defined as follows:

1. $F(EQD) =$ `eq` $F(DTERM)$ `=` $F(DTERM)$ `.` (Production rule (9))
2. $F(DTERM) =$ dtv (Production rule (10))

3. $F(DTERM) =$ *dtop* (Production rule (11))

4. $F(DTERM) =$ *dtop*($F(DTLIST)$) (Production rule (12))

5. $F(DTLIST) = F(DTERM)$ (Production rule (13))

6. $F(DTLIST) = F(DTLIST)$`,`$F(DTERM)$ (Production rule (14)) □

7.3.2 Basic class diagrams

We translate basic class diagrams into a part of a primitive component specification $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$ and a part of a component specification $(V, \Psi, E_\Psi, H, \Sigma, E)$.

Basic class diagram

We translate operator declarations drawn in the middle parts of class boxes into $\Sigma \backslash \Psi$.

Translation 3
Input: *An operator declaration drawn in the middle part of a class box, i.e. a word of language for attribute declarations discussed in Section 5.2*
Output: *An operator declaration of a primitive component specification*

The translation function F is defined as follows:

1. $F(LAD) =$ `bop` $F(DATTWVD)$ `->` *dtnm* (Production rule (1))

2. $F(DATTWVD) =$ *datt* `:` `State` (Production rule (3))

3. $F(DATTWVD) =$ *datt* `:` $F(ARGLIST)$ `State` (Production rule (4))

4. $F(ARGLIST) = F(DDLIST)$ (Production rule (7))

5. $F(DDLIST) = F(DTDCL)$ (Production rule (9))

6. $F(DDLIST) = F(DDLIST)\ F(DTDCL)$ (Production rule (10))

7. $F(DTDCL) =$ *dtnm* (Production rule (11)) □

7.3.3 Action usecase diagrams

We translate action usecase diagrams into a part of a primitive component specification $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$.

OCL descriptions complementing action usecase diagrams

We translate pre declarations and post declarations of OCL descriptions complementing action usecase diagrams into $E \backslash E_\Psi$.

Translation 4
Input: *A pre declaration about data classes, i.e. a word of* EQPRE *of OCL for actions that produces "DATTWA = DTERM" discussed in Section 5.2*
Output: *A conditional part of an equation declaration of a primitive component specification*

Let fspre be the translation function from AGTV *into the set of variables corresponding to the pre states of the agents. The translation function F is defined as follows:*

1. $F(EQPRE) = F(DATTWA)$ `==` $F(DTERM)$ (Production rule (13))
2. $F(DATTWA) = F(DATTWV)$`,` *fspre*`(`*agtv*`)` (Production rule (15))
3. $F(DATTWV) =$ *datt*`(` $F(DARGLIST)$ (Production rule (16))
4. $F(DARGLIST) = F(DTLIST)$ (Production rule (17))
5. $F(DTLIST) = F(DTERM)$ (Production rule (19))
6. $F(DTLIST) = F(DTLIST)$`,` $F(DTERM)$ (Production rule (20))
7. $F(DTERM) =$ *dtv* (Production rule (21))
8. $F(DTERM) =$ *dtop* (Production rule (22))
9. $F(DTERM) =$ *dtop*`(` $F(DTLIST)$ `)` (Production rule (23)) □

Translation 5
Input: *A post declaration about data classes, i.e. a word of* EQPOST *of OCL for actions that produces "DLTPOST = EDTERMPOST" discussed in Section 5.2*
Output: *A conditional part of an equation declaration of a primitive component specification*

Let fspre and fspost be the translation functions from AGTV *into the set of variables corresponding to the pre states of the agents and the set of terms corresponding to the post states of the agents, respectively. The translation function F′ is defined as follows:*

1. $F'(EQPOST) = F'(DLTPOST)$ `=` $F'(EDTERMPOST)$ (Production rule (29))
2. $F'(DLTPOST) = F'(DATTWV)$`,` *fspost*`(`*agtv*`)` (Production rule (31))
3. $F'(DATTWV) =$ *datt*`(` $F'(DARGLIST)$ (Production rule (16))
4. $F'(DARGLIST) = F'(DTLIST)$ (Production rule (17))
5. $F'(DTLIST) = F'(DTERM)$ (Production rule (19))
6. $F'(DTLIST) = F'(DTLIST)$`,` $F'(DTERM)$ (Production rule (20))
7. $F'(DTERM) =$ *dtv* (Production rule (21))
8. $F'(DTERM) =$ *dtop* (Production rule (22))

9. $F'(DTERM) =$ *dtop*($F'(DTLIST)$) (Production rule (23))

10. $F'(EDTERMPOST) =$ *dtv* (Production rule (32))

11. $F'(EDTERMPOST) =$ *dtop* (Production rule (33))

12. $F'(EDTERMPOST) =$ $F'(DATTWV)$, *fspre*(*agtv*)) (Production rule (34))

13. $F'(EDTERMPOST) =$ *dtop*($F'(EDTLISTPOST)$) (Production rule (35))

14. $F'(EDTLISTPOST) =$ $F'(EDTERMPOST)$ (Production rule (36))

15. $F'(EDTLISTPOST) =$ $F'(EDTLISTPOST)$, $F'(EDTERMPOST)$ (Production rule (37))

□

Translation 6
Input: *OCL descriptions complementing an action usecase diagram*
Output: *Equation declarations of a primitive component specification*

Let F and F' be the translation functions discussed in Translation 4 and Translation 5, respectively. Let $\{EQPRE_i\}_{i\in[1,\ldots,k]}$ and $\{EQPOST_j\}_{j\in[1,\ldots,l]}$ be the sets of EQPRE words and EQPOST words occurring in the OCL descriptions, respectively.

The output is as follows:
for each $j \in [1, \ldots, l]$,

1. `eq` $F'(EQPOST_j)$. *if $k = 0$ or*

2. `ceq` $F'(EQPOST_j)$ `if` $F(EQPOST_1)$ `and` ... `and` $F(EQPOST_k)$. *if $k > 0$.* □

7.3.4 Decomposition class diagrams

We translate decomposition class diagrams into a part of a component specification $(V, \Psi, E_\Psi, H, \Sigma, E)$.

OCL descriptions complementing decomposition class diagrams

We translate invariant declarations of OCL descriptions complementing decomposition class diagrams into a part of $E \backslash E_\Psi$.

Translation 7
Input: *An invariant declaration about data classes, i.e. a word of* DATTC *of OCL for constraints discussed in Section 5.2*
Output: *An equation declaration of a component specification*

Let fspost be the translation function from AGTV *into the set of terms corresponding to the post states of the agents. The translation function F is defined as follows:*

1. $F(DATTC) =$ `eq` $F(DATTWA)$ `=` $F(EDTERMC)$. (Production rule (18))

2. $F(DATTWA) =$ $F(DATTWV)$, *fspost*(*agtv*)) (Production rule (19))

3. $F(DATTWV) =$ *datt*($F(DARGLIST)$ (Production rule (20))
4. $F(EDTERMC) =$ *dtv* (Production rule (21))
5. $F(EDTERMC) =$ *dtop* (Production rule (22))
6. $F(EDTERMC) =$ $F(DATTWV)$, *proj*(*fspost*(*agtv*))) (Production rule (23))
7. $F(EDTERMC) =$ *dtop*($F(EDTLISTC)$) (Production rule (24))
8. $F(EDTLISTC) =$ $F(EDTERMC)$ (Production rule (25))
9. $F(EDTLISTC) =$ $F(EDTLISTC)$, $F(EDTERMC)$ (Production rule (26))
10. $F(DARGLIST) =$ $F(DTLIST)$ (Production rule (30))
11. $F(DTLIST) =$ $F(DTERM)$ (Production rule (32))
12. $F(DTLIST) =$ $F(DTLIST)$, $F(DTERM)$ (Production rule (33))
13. $F(DTERM) =$ *dtv* (Production rule (34))
14. $F(DTERM) =$ *dtop* (Production rule (35))
15. $F(DTERM) =$ *dtop*($F(DTLIST)$) (Production rule (36)) □

7.3.5 Decomposition sequence diagrams

We translate decomposition sequence diagrams into a part of a component specification $(V, \Psi, E_\Psi, H, \Sigma, E)$.

Decomposition sequence diagrams

We translate decomposition sequence diagrams into a part of $E \backslash E_\Psi$.

Translation 8
Input: *A decomposition sequence diagram*
Output: *Equation declarations of a component specification*

Let pmth *be the parent method,* ccmp *be an child component,* proj *be the projection from the parent component to* ccmp*, and* cmths *be the sequence of the child methods of* ccmp *belonging to* pmth*. The result of the translation is defined as follows:*

1. *For each component, the result is* eq *proj*(*pmth*(*S*)) = *cmths*(*proj*(*S*)) . □

7.3.6 Decomposition statechart diagrams

We translate decomposition statechart diagrams into a part of a component specification $(V, \Psi, E_\Psi, H, \Sigma, E)$.

Decomposition statechart diagrams

We translate decomposition statechart diagrams into a part of $E\backslash E_\Psi$.

Translation 9
Input: *A decomposition sequence diagram*
Output: *Equation declarations of a component specification*

Let pmth *be the parent method,* ccmp *be an child component,* proj *be the projection from the parent component to* ccmp*, and* cmths *be the sequence of the child methods of* ccmp *belonging to* pmth*. The result of the translation is defined as follows:*

1. *For each component, the result is* `eq` $proj(pmth(S))$ `=` $cmths(proj(S))$. □

7.3.7 Data specifications

Consider a data specification (Ψ, E_Ψ). As we discussed in Section 7.3.1, we get Ψ by translating from data class diagrams and E_Ψ by translating from OCL descriptions complementing data class diagrams.

7.3.8 Primitive component specifications

Consider a primitive component specification $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$. As we discussed in Section 7.3.2. we get $\Sigma\backslash\Psi$ by translating from basic class diagrams and as we discussed in Section 7.3.3, we get $E\backslash E_\Psi$ by translating from OCL descriptions complementing action usecase diagrams.

7.3.9 Component specifications

Consider a component specification $(V, \Psi, E_\Psi, H, \Sigma, E)$. As we discussed in Section 7.3.2. we get $\Sigma\backslash\Psi$ by translating from decomposition class diagrams and as we discussed in Section 7.3.4, Section 7.3.5, and Section 7.3.6, we get $E\backslash E_\Psi$ by translating from (1) OCL descriptions complementing decomposition class diagrams and (2) decomposition sequence diagrams or decomposition statechart diagrams.

7.4 Consistency Verification of UML diagrams

As consistency verification, we verify whether the combination of the components satisfies the behavior of the software by using the projection-style behavioral specifications. We call the verification *refinement verification.*

7.4.1 Refinement verification

For algebraic behavioral specification [5, 10, 15], there is no deduction system that can verify whether any equations satisfy any algebraic behavioral specifications [7]. But, for projection-style behavioral specification, term rewriting based reducers can deduce those (Theorem 24).

The formal definition of refinement is as follows:

Definition 69 *Let* $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$ *be a primitive component specification and* $(V', \Psi', E_{\Psi'}, H', \Sigma', E')$ *be a component specification such that (1)* $h \in H'$ *and (2)* $\Sigma = \Sigma'|_h$ *where* $\Sigma'|_h$ *is the set of* Σ'*-operators whose arities include* h *without projections. We call* $(V', \Psi', E_{\Psi'}, H', \Sigma', E')$ is a refinement of $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$ *if for each* $eq \in E$, $E' \models_{\Sigma'} eq$ *holds.* □

Note that $\Sigma'|_h$ is the set of observations and actions of the composite component.

Theorem 24 *Let* $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$ *be a primitive component specification such that* (Ψ, E_Ψ) *is a complete TRS and* $(V', \Psi', E_{\Psi'}, H', \Sigma', E')$ *be a component specification such that (1)*$V \subset V'$, $\Psi \subset \Psi'$, *and* $E_\Psi \subset E_{\Psi'}$, *(2)* $h \in H'$, *and (3)* $\Sigma = \Sigma'|_h$ *where* $\Sigma'|_h$ *is the set of* Σ'*-operators whose arities include* h *without projections.* $(V', \Psi', E_{\Psi'}, H', \Sigma', E')$ *is a refinement of* $(V, \Psi, E_\Psi, \{h\}, \Sigma, E)$ *iff for each* $(\forall X)v = v' \in E$, $norm_{E'}(v) = norm_{E'}(v')$
□

In data specifications (Ψ, E_Ψ), we specify data structures. Usually, data structures are simple, for example the set of machine names. So, we can usually write (Ψ, E_Ψ) as a complete TRS.

Term rewriting based reducers can automatically check whether $norm_{E'}(v) = norm_{E'}(v')$ holds. So, refinement verification can be automatically executed.

Example 17 *For* CafeOBJ *, there is a term rewriting based reducer* the CafeOBJ verification system*. It supports* red *command that compares the normal forms of the both sides. The* CafeOBJ *verification system executes verification scripts like the following:*

```
open .
red isinlocal(I,put(J,P)) == isinlocal(I,P) .
close
```

Then, it returns the verification result, i.e. true or false. open *command reads a specification, on which term rewriting is executed.*

Consider refinement verification that each axiom of the component specification in Example 15 holds on the component specification in Example 16. For the script corresponding to each axiom, it returns true. So, refinement verification succeeds. □

Chapter 8

LFME

In LFME, we formalize AA-trees model by using equational specifications.

We specify a business model, behavior of components, behavior of software, and how to combine components to construct the software by using equational specifications. As consistency verification, we verify properties of the business model and whether the combination of the components satisfies the behavior of the software by using the equational specifications.

The target problem of LFME is the above verification and the simple logic of LFME is equational logic.

Firstly, we formalize AA-trees model by using equational specifications. As we discussed in Chapter 5, an AA-trees model is specified by using UML with OCL. Then, we discuss a translation method from UML diagrams with OCL descriptions into equational specifications. Finally, we discuss consistency verification methods of the UML diagrams by using the equational specifications.

8.1 A Formalization of AA-trees Model by using Equational Specifications

8.1.1 Static structure

Classes

A class is a set of objects. A sort of an equational specification corresponds to a set of something. So, we assign a class to a sort. We call the sorts assigned to agent classes *agent sorts* and the sorts assigned to data classes *data sorts*.

The set of states

We assign the set of states to a sort. We call the sort the *state sort*.

Attributes

An attribute is a function of a class. So, we assign an attribute to an operator whose arities include the sort corresponding to the class. Because attribute's values are changed by actions, the operator assigned to the attribute must have the state sort in its arities.

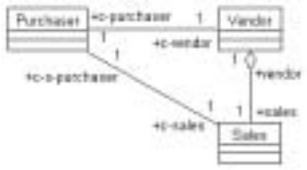

Figure 8.1: Decomposition of an association

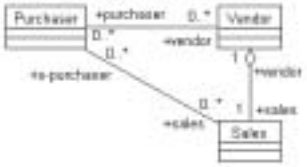

Figure 8.2: Decomposition of an association

Data operators

Because data operators are functions on data classes, we assign data operators *operators* of data sorts.

Definition 70 *Let AG and DT be sets of sorts such that $AG \cap DT = \emptyset$ and s be a sort. Let Σ_{AG} and Σ_{DT} be sets of operators whose ranks are (1) $(d_1 \cdots d_n\ c\ s, b)$ or $(d_1 \cdots d_n\ c'\ c\ s, b)$ and (2) $(d_1 \cdots d_n, d)$ where $d_i, d \in DT$, $b \in DT \cup AG$, $n \geq 0$, and $c, c' \in AG$, respectively. We call the $(AG \cup DT \cup \{s\})$-sorted signature $(\Sigma_{\mathrm{AG}} \cup \Sigma_{\mathrm{DT}})$* a static signature. *We call elements of AG* agent sorts*, elements of DT* data sorts*, and s* the state sort. *We call elements of Σ_{AG}* attributes *and elements of Σ_{DT}* data operators. *Especially, we call attributes* attributes of c. □

Associations

For the case that those attributes' multiplicity is "1", we specify the association between attr and attr' as equations whose meanings are that attr and attr' are the reverse of attr' and attr, respectively. For the case that those attributes' multiplicity is "$0 \ldots n$", we specify the association as an equation whose meaning is that both characteristic functions corresponding to those attributes return the same value.

Definition 71 *Let attr and attr' be attributes. We call a set of Σ-equations whose forms are one of the following forms* the association axiom set of attr-attr' association*:*

1. $(\forall X)\, attr'(DS_n, attr(DS_n, C, S), S) = C$ *and*
   $(\forall X)\, attr(DS_n, attr'(DS_n, C', S), S) = C'$ *or*
2. $(\forall X)\, attr'(DS_n, C, C', S) = attr(DS_n, C', C, S)$

where DS_n is a sequence of variables of data sorts whose length is $n(\geq 0)$, C and C' are variables of agent sorts, and S is a variable of the state sort and we call the pair $(attr, attr')$ attr-attr' association*. We call attr and attr'* the reverse attributes of attr' and attr*, respectively.* □

Example 18 *Consider* c-purchaser - c-vendor *association in Figure 8.1 whose multiplicity is "1". The association is specified as follows:*

```
var V : Vendor  var P : Purchaser  var S : State
eq c-vendor(c-purchaser(V, S), S) = V .
eq c-purchaser(c-vendor(P, S), S) = P .
```

□

Example 19 *Consider* purchaser-vendor *association in Figure 8.2 whose multiplicity is "*$0 \ldots n$*". The association is specified as follows:*

```
var V : Vendor  var P : Purchaser  var S : State
eq vendor(V, P, S) = purchaser(P, V, S) .
```

□

Data Structures

We specify data structures as equations of data sorts.

Definition 72 *Let* $(\Sigma_{\mathrm{DT}} \cup \Sigma_{\mathrm{AG}})$ *be a static signature. We call a set of* Σ_{DT}*-equations* a data axiom set. □

Basic Static Structures

A basic static structure is a static structure constructed from classes, attributes, associations, and data structures. We specify the basic static structure as an equational specification. We call the equational specification *a basic static specification.*

Definition 73 *Let* Σ *be a static signature and* E_{data} *be a data axiom set. Let ASS be a set of associations whose attributes are in* Σ *and* E_{ass} *be the sum of the association axiom set of all the elements of ASS. We call* $(\Sigma, E_{\mathrm{ass}} \cup E_{\mathrm{data}})$ a basic static specification. □

A basic static specification has the following property.

Proposition 25 *Let* (Σ, E) *be a basic static specification. If* $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ *is complete,* $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ *is complete, too.*

Proof : Firstly, we prove that $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is Noetherian. Consider a reduction sequence $a_0 \rightarrow_E a_1 \rightarrow_E \cdots$. Let n be the occurrence number of the attributes having associations in a_0. From Definition 71, the sequence has at most n reductions by rewrite rules of E_{ass}. Note that the sequence is divided into (1) at most n reduction sequences by using rewrite rules of E_{ass} and (2) at most $n+1$ reduction sequences on $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$. Because $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ is Noetherian, any reduction sequences on $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ are finite. So, the sequence is finite. Therefore, $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is Noetherian. Secondly, we prove that $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is local confluent. Consider a critical pair $[P, Q]$ caused by rewrite rules of E_{data}. Because $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ is complete and $E_{\mathrm{data}} \subset E$, $\mathrm{norm}_E(P) = \mathrm{norm}_E(Q)$. Consider a critical pair $[P, Q]$ caused by rewrite rules of E_{ass}. Because these rewrite rules are *1* of Definition 71, $P = \mathrm{attr'}(\mathrm{DS}_n, C', S), Q = C(= \mathrm{attr'}(\mathrm{DS}_n, C', S))$ where the minimal unifier is $\{C \leftarrow \mathrm{attr'}(\mathrm{DS}_n, C', S)\}$, i.e. $P = Q$. From Definition 71, there is no critical pair caused by a rewrite rule of E_{data} and that of E_{ass}. By Knuth-Bendix lemma, $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is local confluent. Finally, by Newman lemma, $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is confluent. So, $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ is complete.

Projection-lift associations

Definition 74 *Let Σ be a static signature and $p \in AG$. Let $AG' = \{c_i\}_{i \in I}$ be sets of sorts such that $AG \cap AG' = \emptyset$ and $DT \cap (AG \cup AG') = \emptyset$. Let $proj_i$ and $lift_i$ be attributes whose ranks are $(p\ s, c_i)$ and $(c_i\ s, p)$, respectively. Let E_{pjlt} be the sum of all the association axiom sets of $proj_i$-$lift_i$ associations. We call 3-tuple $(AG', \{proj_i, lift_i\}_{i \in I}, E_{\mathrm{pjlt}})$* the sort decomposition of *p, each $proj_i$* a projection*, each $lift_i$* a lift*, and each $proj_i$-$lift_i$ association* a projection-lift association. □

Example 20 *Consider the sort decomposition of* Vendor *in Figure 2.3. AG' is the set of* sales*,* accounts*, and* distribution*. There are three projection-lift associations that those projections are* sales*,* accounts*, and* distribution *and those lifts are* vendor*s. The operator declarations corresponding to the projections are as follows:*

```
op sales : Vendor State -> Sales
op accounts : Vendor State -> Accounts
op distribution : Vendor State -> Distribution
```

The operator declarations corresponding to the lifts are as follows:

```
op vendor : Sales State -> Vendor
op vendor : Accounts State -> Vendor
op vendor : Distribution State -> Vendor
```

Especially, consider the association whose projection is sales*. The association is specified as follows:*

```
var V : Vendor  var SA : Sales  var S : State
eq vendor(sales(V, S), S) = V .
eq sales(vendor(SA, S), S) = SA .
```

□

Definition 75 *Let Σ be a static signature, $p \in AG$, and $(AG', \{proj_i, lift_i\}_{i \in I}, E_{\mathrm{pjlt}})$ be the sort decomposition of p. Let LC be the selection of AG that each $lc \in LC$ does not have projections. We call an element of LC* a leaf sort. □

Data attribute constraint and agent attribute constraint

Definition 76 *Let Σ be a static signature, $p \in AG$, and $(AG', \{proj_i, lift_i\}_{i \in I}, E_{\mathrm{pjlt}})$ be the sort decomposition of p. Let $attr_p$ be an attribute of p. We call a Σ-equation whose form is one of the following forms* a projection axiom of attr_p*:*

1. $(\forall X)\, attr_p(DS, P, S) = F(attr_{c1}(DS_1, proj_1(P, S), S), \ldots, attr_{ck}(DS_k, proj_k(P, S), S))$ *if the sort of $attr_p$ is a data sort,*
2. $(\forall X)\, attr_p(DS, P, S) = attr_c(DS_c, proj(P, S), S)$ *if the sort of $attr_p$ is an agent sort, and*
3. $(\forall X)\, attr_p(DS, P', P, S) = attr_c(DS_c, P', proj(P, S), S)$

where F is a term. We use $pjaxm_{\mathrm{attr}_p}$ to denote a projection axiom of $attr_p$. □

Note that *1* and *2* are the cases that multiplicity of attr_p is "1" and *3* is the case that multiplicity of attr_p is "$0 \ldots n$".

Example 21 *Consider the relation between* c-purchaser *and* c-s-purchaser *in Figure 8.1. The relation is specified as follows:*

```
eq c-purchaser(V,S) = c-s-purchaser(sales(V,S),S) .
```

□

Example 22 *Consider the relation between* purchaser *and* s-purchaser *in Figure 8.2. The relation is specified as follows:*

```
eq purchaser(P,V,S) = s-purchaser(P,sales(V,S),S) .
```

□

Reverse attribute constraint

Definition 77 *Let* Σ *be a static signature,* $p \in AG$*, and* $(AG', \{proj_i, lift_i\}_{i \in I}, E_{\text{pjlt}})$ *be the sort decomposition of* p*. Let* $reva_p$ *be a reverse attribute of* p*. We call a* Σ*-equation whose form is one of the following forms* a projection axiom of reva_p*:*

1. $(\forall X) reva_p(DS, P', S) = lift(reva_c(DS_c, P', S), S)$ *and*
2. $(\forall X) reva_p(DS, P', P, S) = reva_c(DS_c, P', proj(P, S), S)$.

We use $pjaxm_{\text{reva}_p}$ *to denote a projection axiom of* $reva_p$*.* □

Note that *1* is the case that multiplicity of reva_p is "1" and *2* is the case that multiplicity of attr_p is "$0 \ldots n$".

Example 23 *Consider the relation between* c-vendor *and* c-sales*. The relation is specified as follows:*

```
eq c-vendor(P,S) = vendor(c-sales(P,S),S) .
```

Note that vendor *is a lift.* □

Example 24 *Consider the relation between* vendor *and* sales*. The relation is specified as follows:*

```
eq vendor(V,P,S) = sales(sales(V,S),P,S) .
```

Note that the second sales *is a projection.* □

Agent decomposition

We deal with agent decomposition as an extension of an equational specification. Consider attr_p-reva_p association in Figure 5.4. Agent decomposition causes projection axioms of attr_p and reva_p. The projection axioms causes attr_c-reva_c association. Reversely, the association axiom set of attr_p-reva_p association is deduced from the projection axioms and the association axiom set of attr_c-reva_c association. Briefly speaking, we model agent decomposition as the addition of projection axioms and the association axioms of attr_c-reva_c associations.

Definition 78 *Let Σ be a static signature and E be a set of Σ-equations. Let p be in AG and $(AG', \Sigma_{\text{pjlt}}, E_{\text{pjlt}})$ be the sort decomposition of p. Let DT' be a set of sorts such that $DT \cap DT' = \emptyset$ and $(DT \cup DT') \cap (AG \cup AG') = \emptyset$. Let $\Sigma_p = \{attr_{p,i}\}_{i \in I}$ be the selection of Σ that (1) each $attr_{p,i}$ is an attribute of p and (2) the sort of each $attr_{p,i}$ is a data sort or a leaf sort. Let $\bar{\Sigma}_p = \{reva_{p,j}\}_{j \in J}$ be in Σ such that (1) J is the selection of $i \in I$ that each $attr_{p,i}$ has a reverse attribute and (2) each $reva_{p,j}$ is the reverse attribute of $attr_{p,j}$. Let $\hat{\Sigma}_{\text{AG'}} = \{attr_{c,i}\}_{i \in I}$ be sets of attributes where each $attr_{c,i}$ is $attr_c$ in Definition 76 obtained by using $attr_{p,i}$ instead of $attr_p$. Let $\Sigma_{\text{AG'}}$ be sets of attributes of sorts in AG' such that $\hat{\Sigma}_{\text{AG'}} \subset \Sigma_{\text{AG'}}$. Let $\Sigma_{\text{DT'}}$ be a set of operators on $DT \cup DT'$. Let $E_{\text{DT'}}$ be a set of $\Sigma_{\text{DT} \cup \text{DT'}}$-equations. Let $E_{\text{pjaxm}} = \{pjaxm_{\text{attr}_{p,i}}\}_{i \in I}$ and $\bar{E}_{\text{pjaxm}} = \{pjaxm_{\text{reva}_{p,j}}\}_{j \in J}$. Let $\tilde{\Sigma}_{\text{AG'}} = \{reva_{c,j}\}_{j \in J}$ be sets of attributes where each $reva_{c,j}$ is $reva_c$ in Definition 77 obtained by using $reva_{p,j}$ instead of $reva_p$. Let $\bar{\Sigma}_{\text{AG'}}$ be sets of reverse attributes of sorts in AG' such that $\tilde{\Sigma}_{\text{AG'}} \subset \bar{\Sigma}_{\text{AG'}}$. Let CHASS be a set of associations such that (1) at least one element of each association in CHASS is an attribute of a sort in AG' and (2) for each $j \in J$, $(attr_{c,j}, reva_{c,j}) \in$ CHASS. Let E_{chass} be the sum of the association axiom sets of all the elements of CHASS. We call the addition of $\Sigma_{\text{DT'}} \cup \Sigma_{\text{pjlt}} \cup \Sigma_{\text{AG'}} \cup \bar{\Sigma}_{\text{AG'}}$ and $E_{\text{DT'}} \cup E_{\text{pjlt}} \cup E_{\text{pjaxm}} \cup \bar{E}_{\text{pjaxm}} \cup E_{\text{chass}}$ to Σ and E* agent decomposition of p on (Σ, E). □

From Definition 78, the following proposition holds.

Proposition 26 *Let E_{paass} be the sum of the association axiom sets of $attr_{p,j}$-$reva_{p,j}$ associations for $j \in J$.*

$(E_{\text{pjlt}} \cup E_{\text{pjaxm}} \cup \bar{E}_{\text{pjaxm}} \cup E_{\text{chass}}) \vdash_{\Sigma_p \cup \bar{\Sigma}_p \cup \Sigma_{\text{pjlt}} \cup \Sigma_{\text{AG'}} \cup \bar{\Sigma}_{\text{AG'}}} E_{\text{paass}}$

Proof : Consider an axiom in E_{paass} whose form is

$(\forall X)\text{reva}_{p,j}(DS_{p,j}, \text{attr}_{p,j}(DS_{p,j}, P, S), S) = P.$

From Definition 78, there are:

1. $(\forall X)\text{attr}_{p,j}(DS_{p,j}, P, S) = \text{attr}_{c,j}(DS_{c,j}, \text{proj}_j(P, S), S)$ in E_{pjaxm},
2. $(\forall X)\text{reva}_{p,j}(DS_{p,j}, P', S) = \text{lift}_j(\text{reva}_{c,j}(DS_{c,j}, P', S), S)$ in $\bar{E}_{\text{pjaxm}}$,
3. $(\forall X)\text{reva}_{c,j}(DS_{c,j}, \text{attr}_{c,j}(DS_{c,j}, C, S), S) = C$ in E_{chass}, and
4. $(\forall X)\text{lift}_j(\text{proj}_j(P, S), S)$ in E_{pjlt}.

By using these equations as rewrite rules,

$\text{reva}_{p,j}(DS_{p,j}, \text{attr}_{p,j}(DS_{p,j}, P, S), S)$
$\rightarrow \text{reva}_{p,j}(DS_{p,j}, \text{attr}_{c,j}(DS_{c,j}, \text{proj}_j(P, S), S))$
$\rightarrow \text{lift}_j(\text{reva}_{c,j}(DS_{c,j}, \text{attr}_{c,j}(DS_{c,j}, \text{proj}_j(P, S), S), S)$

$\rightarrow \mathrm{lift}_j(\mathrm{proj}_j(P,S),S)$
$\rightarrow P$
So, $(E_{\mathrm{pjlt}} \cup E_{\mathrm{pjaxm}} \cup \bar{E}_{\mathrm{pjaxm}} \cup E_{\mathrm{chass}}) \vdash (\forall X)\mathrm{reva}_{p,j}(DS_{p,j}, \mathrm{attr}_{p,j}(DS_{p,j}, P, S), S) = P$.
For axioms having the other forms, a similar discussion holds.

Static Structures

Definition 79 A static specification *is inductively defined as follows:*

1. *a basic static specification is a static specification and*
2. *let* (Σ, E) *be a static specification, then* $(\Sigma \cup \Sigma_{\mathrm{DT'}} \cup \Sigma_{\mathrm{pjlt}} \cup \Sigma_{\mathrm{AG'}} \cup \bar{\Sigma}_{\mathrm{AG'}}, E \cup E_{\mathrm{DT'}} \cup E_{\mathrm{pjlt}} \cup E_{\mathrm{pjaxm}} \cup \bar{E}_{\mathrm{pjaxm}} \cup E_{\mathrm{chass}})$ *obtained by agent decomposition of* p *on* (Σ, E) *is a static specification.* □

Theorem 27 *Let* (Σ, E) *be a static specification. If* $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ *is complete,* $\langle T_{\Sigma}, \rightarrow_{E} \rangle$ *is complete, too.*

Proof : It is proved by using the technique used in the proof of Proposition 25.

8.1.2 Dynamic structure

An action

An action is a function on the state sort, data sorts corresponding to the arguments, and agent sorts corresponding to the participants.

Definition 80 *Let* (Σ, E) *be a static specification. Let* Σ_{act} *be a set of operators whose ranks are* $(d_1 \cdots d_m\ p_1 \cdots p_n\ s, s)$ *where* $p_i \in AG$*,* $m \geq 0$*, and* $n \geq 1$*. We call* $(\Sigma \cup \Sigma_{\mathrm{act}})$ a dynamic structure on Σ*, elements of* Σ_{act} actions*. Consider an action. We call each* p_i a participant sort of the action *and the set of* $p_i (i \in [1, \ldots, n])$ the participant sort set of the action*. Let LC be the set of all the leaf sorts. We call an action that each* p_i *is in LC* a leaf action. □

Example 25 *Consider* buy *action in Example 1. The operator corresponding to* buy *action is as follows:*

```
op buy : Thing Purchaser Vendor State -> State
```
□

Definition 81 *Let act be an action and p be a participant sort of the action. Let attr be an attribute of p such that (1) it is not a projection and (2) it has a reverse attribute. Let* p' *be the sort of attr. We call* p' a co-participant sort of the action *if* p' *is a leaf sort and the set of all the co-participant sorts of the action* the co-participant sort set of the action. □

Example 26 *Consider the static structure specified by the class diagrams of Figure 2.2, Figure 2.3, and Figure 8.2. Let notifyorder is an action whose form is:*

```
notifyorder : Order Sales Distribution Accounts State -> State
```

where Order *is a data sort.* Sales, Distribution, *and* Accounts *are participant sorts. Because there is* s-purchaser-sales *association,* Purchaser *is a co-participant sort.* □

Effect axioms

We specify the effects of an action as changes between attributes' values immediately before and after the action has happened.

Definition 82 *Let (Σ, E) be a static specification. Let s be the state sort. Let p and d be agent sorts. If there is a set of projections $\{proj_i\}_{i\in[1,\ldots,k]}$ such that:*
$d = proj_k(\cdots proj_1(p, s)\cdots, s)$,
we call d a decendant sort of p. □

Definition 83 *Let (Σ, E) be a static specification and $\Sigma' = \Sigma \cup \Sigma_{\mathrm{act}}$ be a dynamic structure on Σ. Let act be an action and attr be an attribute. Consider a Σ'-equation whose form is the following form:*

1. $(\forall X)attr(DS, C, act(DS', CS, S)) = F[attr(DS, C, S)]$.

We call the Σ'-equation an effect axiom of act by attr *if F satisfies the following constraint:*

1. *F is $\circ$ if:*
   (a) *attr is a projection or a lift or*
   (b) *C is a leaf sort without a participant sort, a descendant sort of a participant sort, and a co-participant sort and*
2. *F is F' if:*
   (a) *attr has a reverse attribute reva where*
   $(\forall X)reva(DS, C', act(DS', CS, S)) = F'[reva(DS, C', S)]$
   is the effect axiom of act by reva,
   (b) *attr is $reva_c$ in 1 of Definition 77 where*
   $F'[\circ] = proj(F''[lift(\circ, S)], S)$ *where*
   $(\forall X)reva_p(DS, C', act(DS', CS, S)) = F''[reva_p(DS, C', S)]$
   is the effect axiom of act by $reva_p$, or
   (c) *attr is $reva_c$ in 2 of Definition 77 where*
   $(\forall X)reva_p(DS, C', C, act(DS', CS, S)) = F'[reva_p(DS, C', C, S)]$
   is the effect axiom of act by $reva_p$.

Let acti and attr be an action and an attribute, respectively. We use $effaxm_{\mathrm{acti,attr}}$ to denote an effect axiom of acti by attr. □

Example 27 *Consider the static structure in Example 26. The effect axioms of* notify-order *by* s-purchaser *and* sales *are as follows:*

```
eq s-purchaser(P,SA,notifyorder(O,SA,D,A,S)) = s-purchaser(P,SA,S) .
eq sales(SA,P,notifyorder(O,SA,D,A,S)) = sales(SA,P,S) .
```
□

Definition 84 *Let Σ be a static structure and $\Sigma' = \Sigma \cup \Sigma_{\mathrm{act}}$ be a dynamic structure on Σ. Let acti be an action and attr be an attribute. We call attr* a nondeterministic attribute of acti *if attr is an attribute of a decendant sort of a participant sort of acti. We use NDA_{acti} to denote the set of all the nondeterministic attributes of acti.* □

Definition 85 *Let* (Σ_i, E_i) $(i \geq 0)$ *be a static specification and* p_i *be sorts such that* (Σ_0, E_0) *is a basic static specification and each* (Σ_{k+1}, E_{k+1}) *is obtained from* (Σ_k, E_k) *by agent decomposition of* p_i*. Let* $\Sigma_{\mathrm{pjlt},i}$*,* $\Sigma_{\mathrm{AG'},i}$*, and* $\bar{\Sigma}_{\mathrm{AG'},i}$ *be* Σ_{pjlt}*,* $\Sigma_{\mathrm{AG'}}$*, and* $\bar{\Sigma}_{\mathrm{AG'}}$ *of agent decomposition of* p_i*, respectively. Let* $\Sigma_{\mathrm{act},i}$ *be a dynamic structure on* Σ_i *such that* $\Sigma_{\mathrm{act},0}$ *is a set of leaf actions on* Σ_0 *and each* $\Sigma_{\mathrm{act},k+1} \backslash \Sigma_{\mathrm{act},k}$ *is a set of leaf actions on* Σ_{k+1}*. Let* $E_{\mathrm{pjlt},i}$ *be* $\{effaxm_{\mathrm{acti,pjlt}} \mid \forall acti \in \Sigma_{\mathrm{act},i}, \forall pjlt \in \Sigma_{\mathrm{pjlt},i}\}$*,* $E_{\mathrm{AG'},i}$ *be* $\{effaxm_{\mathrm{acti,catt}} \mid \forall acti \in \Sigma_{\mathrm{act},i} \ . \ \forall catt \in \Sigma_{\mathrm{AG'},i} \backslash NDA_{\mathrm{acti}}\}$*, and* $\bar{E}_{\mathrm{AG'},i}$ *be* $\{effaxm_{\mathrm{acti,crev}} \mid \forall acti \in \Sigma_{\mathrm{act},i} \ . \ \forall crev \in \bar{\Sigma}_{\mathrm{AG'},i} \backslash NDA_{\mathrm{acti}}\}$*. Let* $E_{\mathrm{act},i}$ *be* $\{effaxm_{\mathrm{acti,attr}} \mid \forall acti \in \Sigma_{\mathrm{act},0}, \forall attr \in \Sigma_0\}$ *if* $i = 0$ *and* $E_{\mathrm{act},k} \cup E_{\mathrm{pjlt},k+1} \cup E_{\mathrm{AG'},k+1} \cup \bar{E}_{\mathrm{AG'},k+1}$ *if* $i = k + 1$*. We call* $(\Sigma_i \cup \Sigma_{\mathrm{act},i}, E \cup E_{\mathrm{act},i})$ a dynamic specification on (Σ, E)*. We call the sequence* $(\Sigma_0 \cup \Sigma_{\mathrm{act},0}, E_0 \cup E_{\mathrm{act},0}) \rightarrow \cdots \rightarrow (\Sigma_i \cup \Sigma_{\mathrm{act},i}, E_i \cup E_{\mathrm{act},i}) \rightarrow \cdots$ a pseudo-zooming-in sequence. □

Theorem 28 *Let* $(\hat{\Sigma}, \hat{E})$ *be a dynamic specification on* (Σ, E)*. If* $\langle T_{\Sigma_{DT}}, \rightarrow_{E_{data}} \rangle$ *is complete,* $\langle T_{\hat{\Sigma}}, \rightarrow_{\hat{E}} \rangle$ *is complete, too.*

Proof : It is proved by using the technique used in the proof of Proposition 25.

Action decomposition

Definition 86 *Let* $(\hat{\Sigma}, \hat{E}) = (\Sigma \cup \Sigma_{\mathrm{act}}, E \cup E_{\mathrm{act}})$ *be a dynamic specification on* (Σ, E)*. Let act and* $act_i (i \in [0, \ldots, n])$ *be in* Σ_{act}*. Let acts(S) be the sequence of* act_i*s with variables* $act_0(DS_0, CS_0, \cdots act_n(DS_n, CS_n, S) \cdots)$*. If for each effect axiom:*
$(\forall X)\, attr(DS, C, act(DS', CS, S)) = F(attr(DS, C, S))$ *of* E_{act}*,*
$(\forall X)\, attr(DS, C, acts(S)) = F(attr(DS, C, S))$ *holds,*
we call acts an action decomposition sequence of act. □

Let acts1 and acts2 be action decomposition sequences of act. Let (Σ', E') be a static specification obtained by agent decomposition on (Σ, E). Note that the behavior of acts1 and acts2 are the same at the level of (Σ, E), but the behavior of those may not be the same at the level of (Σ', E').

Definition 87 *Let* $(\Sigma_0 \cup \Sigma_{\mathrm{act},0}, E_0 \cup E_{\mathrm{act},0}) \rightarrow \cdots \rightarrow (\Sigma_i \cup \Sigma_{\mathrm{act},i}, E_i \cup E_{\mathrm{act},i}) \rightarrow \cdots$ *be a pseudo-zooming-in sequence. We call* $ADD_i = \{act, \{acts_{\mathrm{act},j}\}_{j \in J_{\mathrm{act}}}\}_{\mathrm{act} \in \Sigma_{\mathrm{act},i}}$ *where* $acts_{\mathrm{act},i}$ *are sequences of actions in* $\Sigma_{\mathrm{act},i+1}$ *and* $I_{\mathrm{act}} \neq \emptyset$ an action decomposition definition of $\Sigma_{\mathrm{act},i}$*.* □

Definition 88 *Let* $(\Sigma_0 \cup \Sigma_{\mathrm{act},0}, E_0 \cup E_{\mathrm{act},0}) \rightarrow \cdots \rightarrow (\Sigma_i \cup \Sigma_{\mathrm{act},i}, E_i \cup E_{\mathrm{act},i}) \rightarrow \cdots$ *be a pseudo-zooming-in sequence and* ADD_i *be action decomposition definitions of* $\Sigma_{\mathrm{act},i}$*. We call the pseudo-zooming-in sequence* a zooming-in sequence satisfying $\{\mathrm{ADD}_i\}$ *if for each i, for each* $act \in \Sigma_{\mathrm{act},i}$*, for each* $j \in J_{\mathrm{act}}$*,* $acts_{\mathrm{act},j}$ *is an action decomposition sequence of act.* □

Parallel executions of child actions

Some of child actions can happen in parallel. We deal with parallel executions of *act1* and *act2* by using the interleave model that the effects of the sequence *act1,act2* is the same as the effects of the sequence *act2,act1* (Figure 8.3).

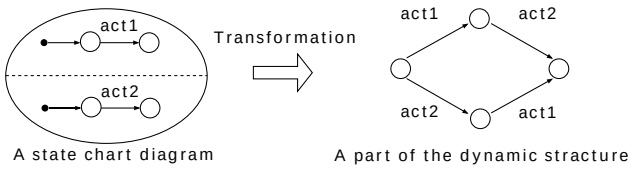

Figure 8.3: The interleave model

Definition 89 *Let act_1 and act_2 be actions. Let PC_{act_1} and PC_{act_2} be the participant sort sets of act_1 and act_2, respectively. Let PA_{act_1} and PA_{act_2} be the set of attributes of elements of PC_{act_1} and PC_{act_2} without projections and lifts, respectively. Let $acts_a(S)$ and $acts_b(S)$ be the sequences of act_is with variables $act_0(DS_0, CS_0, act_1(DS_1, CS_1, S))$ and $act_1(DS_1, CS_1, act_0(DS_0, CS_0, S))$, respectively. If for each element attr of $PA_{act_1} \cup PA_{act_2}$ $(\forall X) attr(DS, acts_a(S)) = attr(DS, acts_b(S))$ holds, we call the pair (act_1, act_2)* a parallel execution pair. □

Static invariants

Definition 90 *Let $(\hat{\Sigma}, \hat{E}) = (\Sigma \cup \Sigma_{\mathrm{act}}, E \cup (\cup_{\mathrm{act} \in \Sigma_{\mathrm{act}}} E_{\mathrm{act}}))$ be a dynamic specification on (Σ, E). Let $eq_i (i \in [1, \ldots, k], k \geq 0)$ and eq be Σ-equations. If (1) there exists i that $\hat{E} \nvdash_{\hat{\Sigma}} eq_i$ or (2) for all i, $\hat{E} \vdash_{\hat{\Sigma}} eq_i$ and $\hat{E} \vdash_{\hat{\Sigma}} eq$, we call the logical formula "$eq_1 \wedge \cdots \wedge eq_k \Rightarrow eq$"* a static invariant on $(\hat{\Sigma}, \hat{E})$. □

8.1.3 Conditional static specification and conditional dynamic specification

A static specification can have conditional equations. By replacing equations in Definition 79 whose forms are $(\forall X)\mathrm{lt} = \mathrm{rt}$ with the following branch conditional equation set of lt that each pair $(\mathrm{lt}, \mathrm{rt}_i)$ satisfies the constraint of the pair $(\mathrm{lt}, \mathrm{rt})$, we construct a conditional static specification.

Definition 91 *Let $(\Sigma_{\mathrm{DT}}, E_{\mathrm{data}})$ be a specification and Σ be a signature such that $\Sigma_{\mathrm{DT}} \subset \Sigma$. Let lt be a $\Sigma(X)$-term. We call the following set of conditional equations $\{ceq_i\}_{i \in I}$* the branch conditional equation set of lt*:*

1. *each ceq_i has the form:*

   $(\forall X) lt = rt_i$ *if* C_i

   where C_i is

   $(u_{i,1} = u'_{i,1})$ *and* $\cdots$ *and* $(u_{i,k} = u'_{i,k})$ *and* $(u_{i,k+1} \neq u'_{i,k+1})$ *and* $\cdots$ *and* $(u_{i,l} \neq u'_{i,l})$,

2. *each $u_{i,j}$ and each $u'_{i,j}$ are a $\Sigma(X)$-term and a $\Sigma_{\mathrm{DT}}(X)$-term, respectively,*

3. *for each $(\Sigma, \{ceq_i\}_{i \in I})$-model M, for each assignment $as : X \to M$, there is exactly one i that $as(C_i)$ is true.* □

A dynamic specification can have conditional equations. By replacing equations in Definition 85 whose forms are $(\forall X)\mathrm{lt} = \mathrm{rt}$ with the following branch conditional equation set of lt that each pair $(\mathrm{lt}, \mathrm{rt}_i)$ satisfies the constraint of the pair $(\mathrm{lt}, \mathrm{rt})$, we construct a conditional dynamic specification.

Definition 92 *Let $(\Sigma_{\mathrm{DT}}, E_{\mathrm{data}})$ be a specification and (Σ, E) be a static signature such that $\Sigma_{\mathrm{DT}} \subset \Sigma$ and $E_{\mathrm{data}} \subset E$. Let Σ_{act} be a set of actions and $\hat{\Sigma}$ be $\Sigma \cup \Sigma_{\mathrm{act}}$. Let lt be a $\hat{\Sigma}(X)$-term. We call the following set of conditional equations $\{ceq_i\}_{i \in I}$* the branch conditional equation set of lt*:*

1. *each ceq_i has the form:*

   $(\forall X) lt = rt_i$ *if* C_i

   where C_i is

   $(u_{i,1} = u'_{i,1})$ *and* $\cdots$ *and* $(u_{i,k} = u'_{i,k})$ *and* $(u_{i,k+1} \neq u'_{i,k+1})$ *and* $\cdots$ *and* $(u_{i,l} \neq u'_{i,l})$,

2. *each $u_{i,j}$ and each $u'_{i,j}$ are a $\Sigma(X)$-term and a $\Sigma_{\mathrm{DT}}(X)$-term, respectively,*

3. *for each $(\Sigma, \{ceq_i\}_{i \in I})$-model M, for each assignment $as : X \to M$, there is exactly one i that $as(C_i)$ is true.* □

8.1.4 Business models

The formalization of AA-trees model of business models is the formalization of AA-trees model discussed in Section 8.1 before this one.

8.1.5 Component specifications

The formalization of AA-trees model of component specifications is a special case of the formalization of AA-trees model discussed in Section 8.1 before Section 8.1.4.

Interfaces, components, and interface-component associations

In component specifications, an action has exactly two participants, i.e. *the interface of the action* and *the component of the action*. There is exactly one association between the interface and the component of the action. We call the association *an interface-component association*. Because an interface is an interface of the corresponding component, for each data attribute of the interface, there should be a data attribute of the component such that *idatt* = *cdatt* where *idatt* is the value of the data attribute of the interface and *cdatt* is the value of the data attribute of the component and vice versa. We call the constraint *interface attribute constraint*. Because actions are assigned to components, we may call the actions *methods of the components*.

Definition 93 *Let DT be a set of sorts and s be a sort. Let itf and cmp be sorts such that itf $\notin$ DT and cmp $\notin$ DT. Let AG be $\{itf, cmp\}$. Let cmpatr and itfatr be operators whose ranks are $(itf\ s, cmp)$ and $(cmp\ s, itf)$, respectively. Let $dtatr_{\mathrm{itf},i}$ and $dtatr_{\mathrm{cmp},i}$ $(i \in I)$ be operators whose ranks are $(d_{i,1} \cdots d_{i,n_i}\ itf\ s, d_{i,0})$ and $(d_{i,1} \cdots d_{i,n_i}\ cmp\ s, d_{i,0})$ where $d_{i,j}$ are in DT, respectively. Let $\Sigma_{\mathrm{AG,itf}}$ and $\Sigma_{\mathrm{AG,cmp}}$ be $\{cmpatr\} \cup \{dtatr_{\mathrm{itf},i}\}_{i \in I}$ and $\{itfatr\} \cup$*

$\{dtatr_{\mathrm{cmp},i}\}_{i\in I}$, *respectively. Let* Σ_{DT} *be a set of operators whose ranks are* $(d_1 \cdots d_n, d_0)$ *where* d_i *is in DT. We call the* $(AG \cup DT \cup \{s\})$*-sorted signature* $(\Sigma_{\mathrm{AG,itf}} \cup \Sigma_{\mathrm{AG,cmp}} \cup \Sigma_{\mathrm{DT}})$ a primitive component static signature. *We call itf* an interface sort, *cmp* a component sort, *cmpatr* − *itfatr association* an interface-component association, *cmpatr* the component attribute of the interface sort, *and itfatr* the interface attribute of the component sort. *We call each pair* $(dtatr_{\mathrm{itf},i}, dtatr_{\mathrm{cmp},i})$ an attribute pair. □

Note that *primitive component static signatures* are *static signatures*.

Definition 94 *Let* $(\Sigma_{\mathrm{AG,itf}} \cup \Sigma_{\mathrm{AG,cmp}} \cup \Sigma_{\mathrm{DT}})$ *be a primitive component static signature and* E_{DT} *be a data axiom set. Let* $E_{\mathrm{ass,itfcmp}}$ *be the association axiom set of cmpatr*− *itfatr association. We call* $(\Sigma_{\mathrm{AG,itf}} \cup \Sigma_{\mathrm{AG,cmp}} \cup \Sigma_{\mathrm{DT}}, E_{\mathrm{ass,itfcmp}} \cup E_{\mathrm{DT}})$ a primitive component static specification. □

Note that *primitive component static specifications* are *basic static specifications*.

Component decomposition and interface decomposition

A component may be decomposed. When a component is decomposed, the interface should be decomposed, too. We assume that firstly a component is decomposed, then the interface is decomposed.

Definition 95 *Let* itf_p, cmp_p, itf_{c_i}, *and* cmp_{c_i} $(i \in I)$ *be sorts. Let* AG_p *and* AG_{c_i} *be* $\{itf_p, cmp_p\}$ *and* $\{itf_{c_i}, cmp_{c_i}\}$, *respectively. Let* $(\Sigma_p, E_p) = (\Sigma_{\mathrm{AG}_p,\mathrm{itf}} \cup \Sigma_{\mathrm{AG}_p,\mathrm{cmp}} \cup \Sigma_{\mathrm{DT}_p}, E_{\mathrm{ass,itfcmp}_p} \cup E_{\mathrm{DT}_p})$ *and* $(\Sigma_{c_i}, E_{c_i}) = (\Sigma_{\mathrm{AG}_{c_i},\mathrm{itf}} \cup \Sigma_{\mathrm{AG}_{c_i},\mathrm{cmp}} \cup \Sigma_{\mathrm{DT}_{c_i}}, E_{\mathrm{ass,itfcmp}_{c_i}} \cup E_{\mathrm{DT}_{c_i}})$ *be primitive component static specifications. Let* (Σ, E) *be a static specification such that* $\Sigma_p \subset \Sigma$, $E_p \subset E$, *and* $(\cup_{i\in I}\Sigma_{c_i}) \cap \Sigma = \emptyset$. *Let* $cmpatr_p$ *and* $itfatr_p$ *be the component attribute and the interface attribute of* $(\Sigma_{\mathrm{AG}_p,\mathrm{itf}} \cup \Sigma_{\mathrm{AG}_p,\mathrm{cmp}} \cup \Sigma_{\mathrm{DT}_p}, E_{\mathrm{ass,itfcmp}_p} \cup E_{\mathrm{DT}_p})$. *Let* $(\{cmp_{c_i}\}_{i\in I}, \{proj_{\mathrm{cmp}_{c_i}}, lift_{\mathrm{cmp}_{c_i}}\}_{i\in I}, E_{\mathrm{pjlt,cmp}})$ *be the sort decomposition of* cmp_p. *Let* $k \in I$, *main be* c_k, *and cmain be* cmp_{c_k}. *Let* $ccmpatr_p$ *and* $citfatr_{\mathrm{main}}$ *be operators whose ranks are* $(itf_p\ s, cmain)$ *and* $(cmain\ s, itf_p)$, *respectively. Let* $pjaxm_{\mathrm{cmpatr}_p}$ *and* $pjaxm_{\mathrm{itfatr}_p}$ *be projection axioms of* $cmpatr_p$ *and* $itfatr_p$ *whose forms are:*

$(\forall X)\, cmpatr_p(I, S) = lift_{\mathrm{cmain}}(ccmpatr_p(I, S), S)$ *and*

$(\forall X)\, itfatr_p(C, S) = citfatr_{\mathrm{main}}(proj_{\mathrm{cmain}}(C, S), S)$, *respectively.*

Let $E_{\mathrm{pjaxm,dtatr,cmp}_p}$ *be the set of all the data attributes of* cmp_p. *Let* $E_{\mathrm{chass,itfcmp}_p}$ *be the association axiom set of* $ccmpatr_p$*-*$citfatr_{\mathrm{main}}$ *association. Let* $\Sigma_{\mathrm{cmpdec}_p}$ *be* $(\cup_{i\in I}\Sigma_{\mathrm{DT}_{c_i}} \backslash \Sigma_{\mathrm{DT}_p}) \cup \{proj_{\mathrm{cmp}_{c_i}}, lift_{\mathrm{cmp}_{c_i}}\}_{i\in I} \cup ((\cup_{i\in I}(\Sigma_{\mathrm{AG}_{c_i},\mathrm{cmp}} \backslash \{itfatr_{c_i}\}) \cup \{citfatr_{\mathrm{main}}\}) \cup \{ccmpatr_p\}$ *and* E_{cmpdec_p} *be* $(\cup_{i\in I} E_{\mathrm{DT}_{c_i}} \backslash E_{\mathrm{DT}_p}) \cup E_{\mathrm{pjlt,cmp}} \cup (\{pjaxm_{\mathrm{itfatr}_p}\} \cup E_{\mathrm{pjaxm,dtatr,cmp}_p}) \cup \{pjaxm_{\mathrm{cmpatr}_p}\} \cup E_{\mathrm{chass,itfcmp}_p}$. *We call the addition of* $\Sigma_{\mathrm{cmpdec}_p}$ *and* E_{cmpdec_p} *to* Σ *and* E component decomposition of cmp_p.

Let $(\{itf_{c_i}\}_{i\in I}, \{proj_{\mathrm{itf}_{c_i}}, lift_{\mathrm{itf}_{c_i}}\}_{i\in I}, E_{\mathrm{pjlt,itf}})$ *be the sort decomposition of* itf_p. *Let imain be* itf_{c_k}. *Let* $pjaxm_{\mathrm{ccmpatr}_p}$ *and* $pjaxm_{\mathrm{citfatr}_{\mathrm{main}}}$ *be projection axioms of* $ccmpatr_p$ *and* $citfatr_{\mathrm{main}}$ *whose forms are:*

$(\forall X)\, ccmpatr_p(I, S) = cmpatr_{\mathrm{main}}(proj_{\mathrm{imain}}(I, S), S)$ *and*

$(\forall X)\, citfatr_{\mathrm{main}}(C', S) = lift_{\mathrm{imain}}(itfatr_{\mathrm{main}}(C', S), S)$, *respectively.*

Let $E_{\mathrm{pjaxm,dtatr,itf}_p}$ *be the set of all the data attributes of* itf_p. *Let* $\Sigma_{\mathrm{itfdec}_p}$ *be* $\{proj_{\mathrm{itf}_{c_i}}, lift_{\mathrm{itf}_{c_i}}\}_{i\in I} \cup (\cup_{i\in I}\Sigma_{\mathrm{AG}_{c_i},\mathrm{itf}}) \cup \{itfatr_{c_i}\}_{i\in I}$ *and* E_{itfdec_p} *be* $E_{\mathrm{pjlt,itf}} \cup (\{pjaxm_{\mathrm{citfatr}_{\mathrm{main}}}\} \cup E_{\mathrm{pjaxm,dtatr,itf}_p}) \cup \{pjaxm_{\mathrm{ccmpatr}_p}\} \cup E_{\mathrm{chass,itfcmp}_p}$. *We call the addition of* $\Sigma_{\mathrm{itfdec}_p}$ *and* E_{itfdec_p} *to* $\Sigma \cup \Sigma_{\mathrm{cmpdec}_p}$ *and* $E \cup E_{\mathrm{cmpdec}_p}$ interface decomposition of itf_p. □

Definition 96 A component static specification *is inductively defined as follows:*

1. *a primitive component static specification is a component static specification and*
2. *let* (Σ, E) *be a component static specification,* itf_p *and* cmp_p *be leaf sorts such that there is the* $cmpatr_p$*-*$itfatr_p$ *association,* $(\hat{\Sigma}, \hat{E})$ *be the static specification obtained from* (Σ, E) *by component decomposition of* cmp_p*, and* $(\tilde{\Sigma}, \tilde{E})$ *be the static specification obtained from* $(\hat{\Sigma}, \hat{E})$ *by interface decomposition of* itf_p*, then* $(\tilde{\Sigma}, \tilde{E})$ *is a component static specification.* □

Note that *component static specifications* are *static specifications.*

Definition 97 *Let* (Σ, E) *be a component static specification. We call* Σ a component static signature. □

Methods and interface attribute constraint

Definition 98 *Let* Σ *be a component static signature. Let* Σ_{mtd} *be a set of operators whose ranks are* $(d_1 \cdots d_m\ itf_i\ cmp_i\ s, s)$ *where (1)* $m \geq 0$ *and (2)* itf_i *and* cmp_i *are an interface sort and a component sort, respectively, such that there is the interface-component association between* itf_i *and* cmp_i*. We call* $(\Sigma \cup \Sigma_{\mathrm{mtd}})$ a component dynamic structure on Σ *and each element of* Σ_{mtd} a method of cmp_i. □

Note that *component dynamic structures* are *dynamic structures* and *methods* are *actions.*

Definition 99 *We call methods that are leaf actions* leaf methods. □

We define *interface attribute constraint* as follows:

Definition 100 *Let* (Σ, E) *be a component static specification,* $\Sigma \cup \Sigma_{mtd}$ *be a component dynamic structure on* Σ*, and* $(\hat{\Sigma}, \hat{E})$ *be a dynamic specification on* (Σ, E)*. We call the following constraint on* $\hat{E}$ interface attribute constraint*:*

1. *for each action* act *and for each attribute pair* $(dtatr_{\mathrm{itf}}, dtatr_{\mathrm{cmp}})$*,* $F_{\mathrm{itf}} = F_{\mathrm{cmp}}$ *where*
   - *(a)* $(\forall X)\, dtatr_{\mathrm{itf}}(DS, I, act(DS', I, C, S)) = F_{\mathrm{itf}}[dtatr_{\mathrm{itf}}(DS, I, S)]$ *is the effect axiom of* act *by* $dtatr_{\mathrm{itf}}$ *and*
   - *(b)* $(\forall X)\, dtatr_{\mathrm{cmp}}(DS, C, act(DS', I, C, S)) = F_{\mathrm{cmp}}[dtatr_{\mathrm{cmp}}(DS, C, S)]$ *is the effect axiom of* act *by* $dtatr_{\mathrm{cmp}}$*.*

We call $(\hat{\Sigma}, \hat{E})$ a component dynamic specification on (Σ, E) *if* $\hat{E}$ *satisfies interface attribute constraint.* □

8.2 Translation from UML diagrams into Equational Specifications

We translate data class diagrams, basic class diagrams, and decomposition class diagrams into a static specification and translate (1) action usecase diagrams and (2) decomposition sequence diagrams or decomposition statechart diagrams into a dynamic specification.

We translate OCL descriptions complementing basic class diagrams into static invariants.

8.2.1 Data class diagrams

We translate data class diagrams into a data part of a static specification.

Data class diagrams

We translate operator declarations drawn in the middle parts of class boxes into Σ_{DT}.

Translation 10
Input: *An operator declaration drawn in the middle part of a class box, i.e. a word of language for data operator declarations discussed in Section 5.2*
Output: *An operator declaration of a data specification*

The translation function F is defined as follows:

1. $F(LDOD) =$ `op` $F(DTOPWV)$ `->` $dtnm$ (Production rule (1))
2. $F(DTOPWV) =$ $dtop$ `:` (Production rule (2))
3. $F(DTOPWV) =$ $dtop$ `:` $F(DDLIST)$ (Production rule (3))
4. $F(DDLIST) =$ $F(DTDCL)$ (Production rule (4))
5. $F(DDLIST) =$ $F(DDLIST)$ $F(DTDCL)$ (Production rule (5))
6. $F(DTDCL) =$ $dtnm$ (Production rule (6)) □

OCL descriptions complementing data class diagrams

We translate invariant declarations of OCL descriptions complementing data class diagrams into the data axiom set E_{data} of the static specification.

Translation 11
Input: *An invariant declaration, i.e. a word of* EQD *of OCL for data discussed in Section 5.2*
Output: *An equation declaration of a data specification*

The translation function F is defined as follows:

1. $F(EQD) =$ `eq` $F(DTERM)$ `=` $F(DTERM)$ `.` (Production rule (9))
2. $F(DTERM) =$ dtv (Production rule (10))
3. $F(DTERM) =$ $dtop$ (Production rule (11))
4. $F(DTERM) =$ $dtop$`(` $F(DTLIST)$ `)` (Production rule (12))
5. $F(DTLIST) =$ $F(DTERM)$ (Production rule (13))
6. $F(DTLIST) =$ $F(DTLIST)$`,`$F(DTERM)$ (Production rule (14)) □

8.2.2 Basic class diagrams

We translate basic class diagrams into a part of a static specification.

Basic class diagram

We translate operator declarations drawn in the middle parts of class boxes and association lines into Σ_{AG}.

Translation 12
Input: *An operator declaration drawn in the middle part of a class box, i.e. a word of language for attribute declarations discussed in Section 5.2*
Output: *An operator declaration of a component specification*

Let agt *be the name of the class. Let* state *be the state sort. The translation function F is defined as follows:*

1. $F(LAD) =$ `op` $F(DATTWVD)$ `->` *dtnm* (Production rule (1))
2. $F(LAD) =$ `op` $F(AATTWVD)$ `->` *agtnm* (Production rule (2))
3. $F(DATTWVD) =$ *datt* `:` *agt state* (Production rule (3))
4. $F(DATTWVD) =$ *datt* `:` $F(ARGLIST)$ *agt state* (Production rule (4))
5. $F(AATTWVD) =$ *aatt* `:` *agt state* (Production rule (5))
6. $F(AATTWVD) =$ *aatt* `:` $F(DDLIST)$ *agt state* (Production rule (6))
7. $F(ARGLIST) = F(DDLIST)$ (Production rule (7))
8. $F(ARGLIST) = F(DDLIST)\ F(AGTDCL)$ (Production rule (8))
9. $F(DDLIST) = F(DTDCL)$ (Production rule (9))
10. $F(DDLIST) = F(DDLIST)\ F(DTDCL)$ (Production rule (10))
11. $F(DTDCL) =$ *dtnm* (Production rule (11))
12. $F(AGTDCL) =$ *agtnm* (Production rule (12)) □

Translation 13
Input: *A side of an association*
Output: *An operator declaration of a component specification*

Let attr *be the name of the side,* mul *be the multiplicity of the association, and* agt *be the agent connecting to the side. Let* state *be the state sort. The result of translation F is defined as follows:*

1. *If* $\text{mul} = 1$, $F =$ `op` *attr* `:` *state* `->` *agt*
2. *If* $\text{mul} = 0 \ldots n$, $F =$ `op` *attr* `:` *agt state* `->` `Bool` □

OCL descriptions complementing basic class diagrams

We translate invariant declarations of OCL descriptions complementing basic class diagrams into static invariants. We translate invariant declarations of OCL descriptions complementing action usecase diagrams into static invariants.

Translation 14
Input: *A invariant declaration, i.e. a word of* LSFI *of OCL for static invariants discussed in Section 5.2*
Output: *A static invariant*

Let statePre be the variable corresponding to the pre state. The translation function F is defined as follows:

1. $F(LFSI) = F(LLFSI) \Rightarrow F(EQSI)$ (Production rule (17))
2. $F(LLFSI) = \epsilon$ (Production rule (18))
3. $F(LLFSI) = F(EQSILIST)$ (Production rule (19))
4. $F(EQSILIST) = F(EQSI)$ (Production rule (20))
5. $F(EQSILIST) = F(EQSILIST)$ `and` $F(EQSI)$ (Production rule (21))
6. $F(EQSI) =$ `eq` $F(EDTERM)$ `=` $F(EDTERM)$ `.` (Production rule (22))
7. $F(EQSI) =$ `eq` $F(EATERM)$ `=` $F(EATERM)$ `.` (Production rule (23))
8. $F(EDTERM) = dtv$ (Production rule (24))
9. $F(EDTERM) = dtop$ (Production rule (25))
10. $F(EDTERM) = dtop($ `(` $F(EDTLIST)$ `)` (Production rule (26))
11. $F(EDTERM) = F(DATTWT)$, $F(EATERM)$, *preState*) (Production rule (28))
12. $F(EDTLIST) = F(EDTERM)$ (Production rule (28))
13. $F(EDTLIST) = F(EDTLIST)$, $F(EDTERM)$ (Production rule (29))
14. $F(EATERM) = agtv$ (Production rule (30))
15. $F(EATERM) = F(AATTWT)$, $F(EATERM)$, *preState*) (Production rule (31))
16. $F(AATTWT) = aatt($ `(` $F(EDTLIST)$ (Production rule (32))
17. $F(DATTWT) = datt($ `(` $F(EDARGLIST)$ (Production rule (33))
18. $F(EDARGLIST) = F(EDTLIST)$ (Production rule (34))
19. $F(EDARGLIST) = F(EDTLIST)$, $F(EATERM)$ (Production rule (35)) □

8.2.3 Action usecase diagrams

We translate action usecase diagrams into a part of a dynamic specification.

OCL descriptions complementing action usecase diagrams

We translate pre declarations and post declarations of OCL descriptions complementing action usecase diagrams into E_{act}.

Translation 15
Input: *A pre declaration, i.e. a word of* EQPRE *of OCL for actions discussed in Section 5.2*
Output: *A conditional part of an equation declaration of a component specification*

Let statePre be the variable corresponding to the pre state. The translation function F is defined as follows:

1. $F(EQPRE) = F(DATTWA)$ == $F(DTERM)$ (Production rule (13))
2. $F(EQPRE) = F(AATTWA)$ == $agtv$ (Production rule (14))
3. $F(DATTWA) = F(DATTWV), agtv, statePre)$ (Production rule (15))
4. $F(DATTWV) = datt($ $F(DARGLIST)$ (Production rule (16))
5. $F(DARGLIST) = F(DTLIST)$ (Production rule (17))
6. $F(DARGLIST) = F(DTLIST)$, $agtv$ (Production rule (18))
7. $F(DTLIST) = F(DTERM)$ (Production rule (19))
8. $F(DTLIST) = F(DTLIST)$, $F(DTERM)$ (Production rule (20))
9. $F(DTERM) = dtv$ (Production rule (21))
10. $F(DTERM) = dtop$ (Production rule (22))
11. $F(DTERM) = dtop($ $F(DTLIST)$) (Production rule (23))
12. $F(AATTWA) = F(AATTWV), agtv, statePre)$ (Production rule (24))
13. $F(AATTWV) = aatt($ $F(DTLIST)$ (Production rule (25)) □

Translation 16
Input: *A post declaration, i.e. a word of* EQPOST *of OCL for actions discussed in Section 5.2*
Output: *A conditional part of an equation declaration of a component specification*

Let statePre be the variable corresponding to the pre state and statePost be the term corresponding to the post state. The translation function F' is defined as follows:

1. $F'(EQPOST) = F'(DLTPOST)$ = $F'(EDTERMPOST)$ (Production rule (29))
2. $F'(EQPOST) = F'(ALTPOST)$ = $F'(ARTPOST)$ (Production rule (30))
3. $F'(DLTPOST) = F'(DATTWV)$, $agtv$, $statePost)$ (Production rule (31))
4. $F'(DATTWV) = datt($ $F'(DARGLIST)$ (Production rule (16))

5. $F'(DARGLIST) = F'(DTLIST)$ (Production rule (17))

6. $F'(DARGLIST) = F'(DTLIST)$, *agtv* (Production rule (18))

7. $F'(DTLIST) = F'(DTERM)$ (Production rule (19))

8. $F'(DTLIST) = F'(DTLIST)$, $F'(DTERM)$ (Production rule (20))

9. $F'(DTERM) =$ *dtv* (Production rule (21))

10. $F'(DTERM) =$ *dtop* (Production rule (22))

11. $F'(DTERM) =$ *dtop*($F'(DTLIST)$) (Production rule (23))

12. $F'(EDTERMPOST) =$ *dtv* (Production rule (32))

13. $F'(EDTERMPOST) =$ *dtop* (Production rule (33))

14. $F'(EDTERMPOST) = F'(DATTWV)$, *agtv*, *statePre*) (Production rule (34))

15. $F'(EDTERMPOST) =$ *dtop*($F'(EDTLISTPOST)$) (Production rule (35))

16. $F'(EDTLISTPOST) = F'(EDTERMPOST)$ (Production rule (36))

17. $F'(EDTLISTPOST) = F'(EDTLISTPOST), F'(EDTERMPOST)$ (Production rule (37))

18. $F'(ALTPOST) = F'(AATTWV)$, *agtv*, *statePost*) (Production rule (38))

19. $F'(AATTWV) =$ *aatt*($F'(DTLIST)$ (Production rule (25))

20. $F'(ARTPOST) =$ *agtv* (Production rule (39))

21. $F'(ARTPOST) = F'(AATTWV)$, *agtv*, *statePre*) (Production rule (40)) □

Translation 17
Input: *OCL descriptions complementing an action usecase diagram*
Output: *Equation declarations of a component specification*

Let F and F' be the translation functions discussed in Translation 15 and Translation 16, respectively. Let $\{EQPRE_i\}_{i\in[1,\ldots,k]}$ and $\{EQPOST_j\}_{j\in[1,\ldots,l]}$ be the sets of EQPRE words and EQPOST words occurring in the OCL descriptions, respectively.

The output is as follows:
for each $j \in [1, \ldots, l]$,

1. `eq` $F'(EQPOST_j)$. *if $k = 0$ or*

2. `ceq` $F'(EQPOST_j)$ `if` $F(EQPOST_1)$ `and` ... `and` $F(EQPOST_k)$. *if $k > 0$.* □

8.2.4 Decomposition class diagrams

We translate decomposition class diagrams into a part of a static specification.

OCL descriptions complementing decomposition class diagrams

We translate invariant declarations of OCL descriptions complementing decomposition class diagrams into $E_{\text{pjaxm}} \cup \bar{E}_{\text{pjaxm}}$.

Translation 18
Input: *An invariant declaration, i.e. a word of* DATTC *or* AATTC *of OCL for constraints discussed in Section 5.2*
Output: *An equation declaration of a connector specification*

Let statePre be the variable corresponding to the pre state. The translation function F is defined as follows:

1. $F(DATTC) =$ `eq` $F(DATTWA)$ `=` $F(EDTERMC)$ `.` (Production rule (18))
2. $F(DATTWA) =$ $F(DATTWV)$`,` *agtv*`,` *statePre*`)` (Production rule (19))
3. $F(DATTWV) =$ *datt*`(` $F(DARGLIST)$ (Production rule (20))
4. $F(EDTERMC) =$ *dtv* (Production rule (21))
5. $F(EDTERMC) =$ *dtop* (Production rule (22))
6. $F(EDTERMC) =$ $F(DATTWV)$`,` *proj*`(` *agtv*`,` *statePre* `)`, *statePre*`)` (Production rule (23))
7. $F(EDTERMC) =$ *dtop*`(` $F(EDTLISTC)$ `)` (Production rule (24))
8. $F(EDTLISTC) =$ $F(EDTERMC)$ (Production rule (25))
9. $F(EDTLISTC) =$ $F(EDTLISTC)$`,` $F(EDTERMC)$ (Production rule (26))
10. $F(AATTC) =$ `eq` $F(AATTWA)$ `=` $F(CAATTWA)$ `.` (Production rule (27))
11. $F(AATTWA) =$ $F(AATTWV)$`,` *agtv*`,` *statePre*`)` (Production rule (28))
12. $F(AATTWV) =$ *aatt*`(` $F(DTLIST)$ (Production rule (29))
13. $F(DARGLIST) =$ $F(DTLIST)$ (Production rule (30))
14. $F(DARGLIST) =$ $F(DTLIST)$`,` *argv* (Production rule (31))
15. $F(DTLIST) =$ $F(DTERM)$ (Production rule (32))
16. $F(DTLIST) =$ $F(DTLIST)$`,` $F(DTERM)$ (Production rule (33))
17. $F(DTERM) =$ *dtv* (Production rule (34))
18. $F(DTERM) =$ *dtop* (Production rule (35))
19. $F(DTERM) =$ *dtop*`(` $F(DTLIST)$ `)` (Production rule (36))
20. $F(CAATTWA) =$ $F(AATTWV)$`,` *proj*`(` *agtv*`,` *statePre* `)`, *statePre*`)` (Production rule (37))

□

8.2.5 Static specifications

As we discussed in Section 8.2.1, Section 8.2.2, and Section 8.2.4, we get a static specification by translating from data class diagrams, basic class diagrams, and decomposition class diagrams.

8.2.6 Dynamic specifications

As we discussed in Section 8.2.3, we get a dynamic specification by translating from action usecase diagrams.

8.2.7 Static invariants

As we discussed in Section 8.2.2, we get static invariants by translating from OCL descriptions complementing basic class diagrams.

8.3 Consistency Verification of UML diagrams

The UML diagrams are specified by a number of software engineers. So, there may be inconsistencies in the UML diagrams. The main causes of the inconsistencies are descriptions of action decomposition and static invariants.

8.3.1 Refinement verification

Action decomposition is specified in decomposition sequence diagrams and decomposition statechart diagrams. The effects of actions are specified in action usecase diagrams. To check the consistency, i.e. refinement, we verify whether effects of a parent action is deduced from effects of the child actions. We use the following property in the verification.

Theorem 29 *Let* $(\hat{\Sigma}, \hat{E}) = (\Sigma \cup \Sigma_{\mathrm{act}}, E \cup (\cup_{\mathrm{act} \in \Sigma_{\mathrm{act}}} E_{\mathrm{act}}))$ *be a dynamic specification on* (Σ, E)*. Let act and* $act_i (i \in [0, \ldots, n])$ *be in* Σ_{act}*. Let acts(S) be the sequence of* $act_i s$ *with variables* $act_0(DS_0, CS_0, \cdots act_n(DS_n, CS_n, S) \cdots)$*. If for each effect axiom of* E_{act} $(\forall X)\, attr(DS, C, act(DS', CS, S)) = F(attr(DS, C, S))$*,*
the normal forms of $attr(DS, C, acts(S))$ *and* $F(attr(DS, C, S))$ *on* $\langle T_{\hat{\Sigma}}, \rightarrow_{\hat{E}} \rangle$ *are the same, acts is an action decomposition sequence of act.*

Proof : It is straightforward from Definition 86 and Theorem 28. Note that this verification process can be executed by using term rewriting based reducers like the CafeOBJ verification system.

Example 28 *Consider a dynamic specification specified by the action usecase diagram in Example 1 and so on. Consider action decomposition specified by the decomposition sequence diagram in the bottom of Figure 2.5. Let* buy-seq *be the sequence of* makeorder*,* notifyorder*,* deliver*, and* pay*. Consider the first effect axiom of* buy *action in Example 1. Corresponding comparison of the normal forms are executed on the* CafeOBJ *verification system as follows:*

```
red p-balance(p,buy-seq(t,p,v,s)) = p-balance(p,s) - price(t) .
```

The comparison returns true. So, a part of the verification succeeds. □

If some child actions are executed in parallel, we must verify whether we can deal with the child actions as the interleave model. The verification is that for each attribute attr we verify whether:

1. $\mathrm{attr}(\mathrm{BS}, \mathrm{act2}(\mathrm{DS}_2, \mathrm{CS}_2, \mathrm{act1}(\mathrm{DS}_1, \mathrm{CS}_1, S)))$
   $= \mathrm{attr}(\mathrm{BS}, \mathrm{act1}(\mathrm{DS}_1, \mathrm{CS}_1, \mathrm{act2}(\mathrm{DS}_2, \mathrm{CS}_2, S)))$

The verification is executed by comparing normal forms of both sides of the equations, too. Note that if $\mathrm{CS}_1 \cap \mathrm{CS}_2 = \emptyset$, the equation holds. In a low level of software specification, we assign actions to methods of components. We assume that methods of a component can not execute in parallel. So, many of the case $\mathrm{CS}_1 \cap \mathrm{CS}_2 = \emptyset$.

Example 29 *Consider a software specification corresponding to the business model in Figure 2.4. We assume that* deliver *is assigned to a method of* Distribution *component and* pay *is assigned to a method of* Accounts *component. So, CS of* deliver *(CS_1) is* $\{Distribution\}$ *and CS of* pay *(CS_2) is* $\{Accounts\}$*, i.e.* $CS_1 \cap CS_2 = \emptyset$*. So, we do not need to verify whether the equation holds.* □

8.3.2 Verification of satisfaction of static invariants

A static invariant "$\mathrm{eq}_1 \wedge \cdots \wedge \mathrm{eq}_k \Rightarrow \mathrm{eq}$" is specified in basic class diagrams.

From Theorem 28, the following property holds.

Theorem 30 *Let* $(\hat{\Sigma}, \hat{E}) = (\Sigma \cup \Sigma_{\mathrm{act}}, E \cup (\cup_{\mathrm{act} \in \Sigma_{\mathrm{act}}} E_{\mathrm{act}}))$ *be a dynamic specification on* (Σ, E)*. Let* $eq_i (i \in [1, \ldots, k], k \geq 0)$ *and eq be* Σ*-equations. If the following property hold, "*$eq_1 \wedge \cdots \wedge eq_k \Rightarrow eq$*" is a static invariant on* $(\hat{\Sigma}, \hat{E})$*:*

1. *the normal forms of both sides of each* eq_i *and eq on* $\langle T_{\hat{\Sigma}}, \rightarrow_{\hat{E}} \rangle$ *are the same or*
2. *there is i such that the normal forms of both sides of* eq_i *on* $\langle T_{\hat{\Sigma}}, \rightarrow_{\hat{E}} \rangle$ *are different.*

□

By using Theorem 30, we verify satisfaction of static invariants. Note that the verification process can be executed by using term rewriting based reducers.

Example 30 *Consider the static invariant and* buy *action. The verification of the logical formula at the state immediately before* buy *action has happened is as follows:*

```
red vendor(v,p,bs) = true .
red p-possess(t,p,bs) = true .
red v-possess(t,v,bs) = false .
```

The second comparison returns false. So, the logical formula holds at the state. By iterating the same comparisons for each state for each preserving action, the static invariant is verified. □

Chapter 9

A Comparison between LFMB and LFME

Because we formalize the same AA-trees model in LFMB and LFME, for the same verification about the model, the results of LFMB and LFME must be the same. The common verification of LFMB and LFME is refinement verification. So, firstly, we discuss a correspondence between refinement verification in LFMB and that in LFME.

Because the verification results of LFMB and LFME are the same, we predicted that the logic of projection-style behavioral specification was equational logic. The prediction is true. Then, we discuss that the logic of projection-style behavioral specification.

9.1 Refinement Verification

The correspondences between the formalizations of AA-trees model in LFMB and LFME are as follows.

A primitive component specification corresponds to a component static specification.

Definition 101 *Let* $(V, \Psi, E_\Psi, \{p\}, \Sigma, E)$ *be a primitive component specification. Let* $\{obse_j\}_{j\in J}$ *be the set of all the observations in* Σ *such that the rank of each* $obse_j$ *is* $(d_{1,j} \cdots d_{n_j,j}\ p, d_{0,j})$*. Let* cmp_p*,* itf_p*, and* s *be sorts. Let itfatr and cmpatr be operators whose ranks are* $(cmp_p\ s, itf_p)$ *and* $(itf_p\ s, cmp_p)$*, respectively. Let* $\{obse_{\mathrm{cmp},j}\}_{j\in J}$ *and* $\{obse_{\mathrm{itf},j}\}_{j\in J}$ *be sets of operators such that the ranks of each* $obse_{\mathrm{cmp},j}$ *and each* $obse_{\mathrm{itf},i}$ *are* $(d_{1,j} \cdots d_{n_j,j}\ cmp_p\ s, d_{0,j})$ *and* $(d_{1,j} \cdots d_{n_j,j}\ itf_p\ s, d_{0,j})$*, respectively. Let* $\Sigma_{\mathrm{AG,cmp}}$ *and* $\Sigma_{\mathrm{AG,itf}}$ *be* $\{itfatr\} \cup \{obse_{\mathrm{cmp},j}\}_{j\in J}$ *and* $\{cmpatr\} \cup \{obse_{\mathrm{itf},j}\}_{j\in J}$*, respectively. Let* $E_{\mathrm{ass,itfcmp}}$ *be the association axiom set of cmpatr-itfatr association. We call* $(\Sigma_{\mathrm{AG,cmp}} \cup \Sigma_{\mathrm{AG,itf}} \cup \Psi, E_{\mathrm{ass,itfcmp}} \cup E_\Psi)$ the primitive component static specification *of* $(V, \Psi, E_\Psi, \{p\}, \Sigma, E)$*.* □

A connector corresponds to a component decomposition and an interface decomposition.

Definition 102 *Let* $(V, \Psi, E_\Psi, \{p\}, \Sigma, E)$ *and* $(V_i, \Psi_i, E_{\Psi,i}, \{c_i\}, \Sigma_i, E_i)$ $(i \in I)$ *be primitive component specifications such that* $c_i \neq c_j$ *if* $i \neq j$ *and* $p \neq c_i$*,* $(V, \Psi, E_\Psi, p, \Sigma, \{c_i\}_{i\in I}, \Sigma_{\mathrm{proj}}, E)$ *be a connector of* $(V_i, \Psi_i, E_{\Psi,i}, \{c_i\}, \Sigma_i, E_i)$ $(i \in I)$*, and* $(\hat{V}, \hat{\Psi}, \hat{E}_\Psi, \hat{H}, \hat{\Sigma}, \hat{E})$ *be a component specification such that (1)* $(\hat{V}, \hat{\Psi}, \hat{E}_\Psi, \hat{H}, \hat{\Sigma}, \hat{E})$ *is a refinement of* $(V, \Psi, E_\Psi, \{p\}, \Sigma, E)$ *and (2)* $(\hat{V}, \hat{\Psi}, \hat{E}_\Psi, \hat{H}, \hat{\Sigma}, \hat{E})$ *is the combination of* $(V_i, \Psi_i, E_{\Psi,i}, \{c_i\}, \Sigma_i, E_i)$ $(i \in I)$*. Let*

$\{obse_j\}_{j\in J}$ *be the set of all the observation in* Σ *and* $obspj_{\mathrm{obse}_j}$ *be the projection axiom of* $obse_j$ *in* E *whose forms are:*

$(\forall X)obse_j(DS,S) = F[obse_{c1,j_{c1}}(DS_1, proj_{h,c_{c1}}(S)),\ldots,obse_{ck,j_{ck}}(DS_k, proj_{h,c_{ck}}(S))]$.

Let $(\Sigma_{\mathrm{AG}_p,\mathrm{itf}}\cup\Sigma_{\mathrm{AG}_p,\mathrm{cmp}}\cup\Psi, E_{\mathrm{ass,itfcmp}_p}\cup E_\Psi)$ *and* $(\Sigma_{\mathrm{AG}_{c_i},\mathrm{itf}}\cup\Sigma_{\mathrm{AG}_{c_i},\mathrm{cmp}}\cup\Psi_i, E_{\mathrm{ass,itfcmp}_{c_i}}\cup E_{\Psi,i})$ *be the primitive component static specifications of* $(V,\Psi,E_\Psi,\{p\},\Sigma,E)$ *and* $(V_i,\Psi_i,E_{\Psi,i},\{c_i\},\Sigma_i,E_i)$, *respectively. Let* $(\{cmp_{c_i}\}_{i\in I},\{proj_{\mathrm{cmp}_{c_i}}, lift_{\mathrm{cmp}_{c_i}}\}_{i\in I}, E_{\mathrm{pjlt,cmp}})$ *be the sort decomposition of* cmp_p. *Let* $pjaxm_{\mathrm{obse}_{\mathrm{cmp}_p,j}}$ *be the projection axiom of* $obse_{\mathrm{cmp}_p,j}$ *whose forms are:*

$(\forall X)obse_{\mathrm{cmp}_p,j}(DS,C,S)$

$= F[obse_{\mathrm{cmp}_{c1},j_{c1}}(DS_1, proj_{\mathrm{cmp}_{c_{c1}}}(C,S)),\ldots,obse_{\mathrm{cmp}_{ck},j_{ck}}(DS_k, proj_{\mathrm{cmp}_{c_{c1}}}(C,S))]$

where F *is* F *of* $obspj_{\mathrm{obse}_j}$. *Let main be an element of* $\{c_i\}_{i\in I}$. *Let* $\Sigma_{\mathrm{cmpdec}_p}$ *be that in Definition 95 where* $\Sigma_{\mathrm{DT}_{c_i}}$ *and* Σ_{DT_p} *are* Ψ_i *and* Ψ, *respectively. Let* E_{cmpdec_p} *be that in Definition 95 where* $E_{\mathrm{DT}_{c_i}}$ *and* E_{DT_p} *are* $E_{\Psi,i}$ *and* E_Ψ, *respectively and* $E_{\mathrm{pjaxm,dtatr,cmp}_p}$ *is* $\{pjaxm_{\mathrm{obse}_{\mathrm{cmp}_p,j}}\}_{j\in J}$.

Let $(\{itf_{c_i}\}_{i\in I},\{proj_{\mathrm{itf}_{c_i}}, lift_{\mathrm{itf}_{c_i}}\}_{i\in I}, E_{\mathrm{pjlt,itf}})$ *be the sort decomposition of* itf_p. *Let* $pjaxm_{\mathrm{obse}_{\mathrm{itf}_p,j}}$ *be the projection axiom of* $obse_{\mathrm{itf}_p,j}$ *whose forms are:*

$(\forall X)obse_{\mathrm{itf}_p,j}(DS,C,S)$

$= F[obse_{\mathrm{itf}_{c1},j_{c1}}(DS_1, proj_{\mathrm{itf}_{c_{c1}}}(C,S)),\ldots,obse_{\mathrm{itf}_{ck},j_{ck}}(DS_k, proj_{\mathrm{itf}_{c_{c1}}}(C,S))]$

where F *is* F *of* $obspj_{\mathrm{obse}_j}$. *Let* $\Sigma_{\mathrm{itfdec}_p}$ *be that in Definition 95. Let* E_{itfdec_p} *be that in Definition 95 where* $E_{\mathrm{pjaxm,dtatr,itf}_p}$ *is* $\{pjaxm_{\mathrm{obse}_{\mathrm{itf}_p,j}}\}_{j\in J}$.

We call the 6-tuple $(cmp_p,\Sigma_{\mathrm{cmpdec}_p},E_{\mathrm{cmpdec}_p},itf_p,\Sigma_{\mathrm{itfdec}_p},E_{\mathrm{itfdec}_p})$ component-interface decomposition caused by the connector $(V,\Psi,E_\Psi,p,\Sigma,\{c_i\}_{i\in I},\Sigma_{\mathrm{proj}},E)$ with main. □

By combining primitive component static specifications in Definition 101 and connectors in Definition 102, we can get a component static specification.

Effect axioms in a component specification correspond to effect axioms in a dynamic specification.

Definition 103 *Let* $(V,\Psi,E_\Psi,\{h\},\Sigma,E_{\mathrm{effect}})$ *be a component specification. Let* $\{obse_i\}_{i\in I}$ *be the set of all the observations in* Σ *such that the rank of each* $obse_i$ *is* $(d_{1,i}\cdots d_{n_i,i}\ h, d_{0,i})$. *Let* $\{acti_j\}_{j\in J}$ *be the set of all the actions in* Σ *such that the rank of each* $acti_j$ *is* $(d'_{1,j}\cdots d'_{n_j,j}\ h, h)$. *Let* $\{eq_{i,j}\}_{i\in I,j\in J}$ *be* E_{effect} *such that the form of* $eq_{i,j}$ *is:*

$(\forall X)obse_i(DS, acti_j(DS', H)) = F_{i,j}[obse_i(DS,H)]$.

Let cmp_h, if_h, *and* s *be sorts. Let* $\{obse_{\mathrm{cmp},i}\}_{i\in I}$ *and* $\{obse_{\mathrm{if},i}\}_{i\in I}$ *be the set of operators that the ranks of each* $obse_{\mathrm{cmp},i}$ *and each* $obse_{\mathrm{if},i}$ *are* $(d_{1,i}\cdots d_{n_i,i}\ cmp_h\ s, d_{0,i})$ *and* $(d_{1,i}\cdots d_{n_i,i}\ if_h\ s, d_{0,i})$, *respectively. Let* $\{acti_{\mathrm{meth},j}\}_{j\in J}$ *be the set of operators that the rank of each* $acti_{\mathrm{meth},j}$ *is* $(d'_{1,j}\cdots d'_{n_j,j}\ if_h\ cmp_h\ s, s)$. *Let* $eq_{\mathrm{cmp},i,j}$ *and* $eq_{\mathrm{if},i,j}$ $(i\in I, j\in J)$ *be equations whose forms are:*

$(\forall X)obse_{\mathrm{cmp},i}(DS, cmp_h, acti_{\mathrm{meth},j}(DS', if_h, cmp_h, S)) = F_{i,j}[obse_{\mathrm{cmp},i}(DS, cmp_h, S)]$ *and*

$(\forall X)obse_{\mathrm{if},i}(DS, if_h, acti_{\mathrm{meth},j}(DS', if_h, cmp_h, S)) = F_{i,j}[obse_{\mathrm{if},i}(DS, if_h, S)]$, *respectively.*

We call $\{eq_{\mathrm{cmp},i,j}\}_{i\in I}\cup\{eq_{\mathrm{if},i,j}\}_{i\in I}$ the basic effect axiom set of $\mathrm{acti}_{\mathrm{meth},j}$. □

The dynamic specification is obtained from the component static specification and the basic effect axiom sets by adding effect axioms by observations without $\mathrm{obse}_{\mathrm{cmp}}$ and $\mathrm{obse}_{\mathrm{if}}$ whose forms are $(\forall X)\mathrm{attr}(DS,C,\mathrm{act}(DS', CS, S)) = \mathrm{attr}(DS,C,S)$.

Because there are correspondences between reduction sequences used in refinement verification of LFMB and those of LFME, the results of LFMB and LFME are the same.

9.2 Logic of Projection-style Behavioral Specification

The logic of behavioral specification is *behavioral logic* and the logic of equational specification is *equational logic*. The verification using behavioral logic is more complex than the verification using equational logic. The latter only needs equational deduction. But, the former may need context induction [19], coinduction[5, 15], test set coinduction[25], and so on.

As we discussed in Section 7.4.1, refinement verification in LFMB only needs equational deduction. So, we predicted that the logic of projection-style behavioral specification was equational logic. The prediction is true. Firstly, we discuss that the semantic of projection-style behavioral specification $(V, \Psi, E_\Psi, H, \Sigma, E)$ is the same as the semantics of equational specifications (Σ, E), i.e. the set of all the hidden $(V, \Psi, E_\Psi, H, \Sigma, E)$-algebras are the same as the set of all the (Σ, E)-models.

Theorem 31 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be a projection-style behavioral specification. A hidden $(V, \Psi, E_\Psi, H, \Sigma, E)$-algebra is a (Σ, E)-model and vice versa.*

Proof : From Definition 64, all axioms of E are equations. So, (Σ, E) is an equational specification. From Definition 18 and Definition 28, a hidden $(V, \Psi, E_\Psi, H, \Sigma, E)$-algebra is a (Σ, E)-model and vice versa. The verification about projection-style behavioral specifications is iterations of the satisfaction verification of Σ-equations and Σ-behavioral equations.

The satisfaction relation of behavioral logic is an extension of that of equational logic. In behavioral logic, the satisfaction relation about Σ-behavioral equations is added (Definition 26). Because the satisfaction relation about Σ-equations in behavioral logic is the same as that in equational logic, the satisfaction verification of Σ-equations is executed by using equational deduction.

Then, we discuss the satisfaction verification of Σ-behavioral equations on projection-style behavioral specifications.

Theorem 32 *Let $(V, \Psi, E_\Psi, H, \Sigma, E)$ be a projection-style behavioral specification and Obs be the set of observations in Σ.*

$M \models_\Sigma (\forall X) t \sim t'$ *iff* *for each* $obs \in Obs$, $M \models_\Sigma (\forall X) obs[t] = obs[t']$.

Proof : It is straightforward by using the technique of test set coinduction [25]. Because the number of Obs is finite, from Theorem 32, the satisfaction verification of Σ-behavioral equations on projection-style behavioral specifications is executed by using equational deduction.

So, we conclude that the logic of projection-style behavioral specification is equational logic.

Chapter 10

Connector Generation

10.1 JavaBeans Implementation of Tree Architecture

JavaBeans has the following interfaces:

1. *events* used for reporting change of the states of JavaBeans,
2. *properties* used for observing the states, and
3. *methods* used for calling inner functions of JavaBeans.

We implement a component of tree architecture by using the following two kinds of JavaBeans, *a function bean* and *an interface bean* (Fig. 10.1):

1. for each observation or action of the component, the interface bean has a corresponding input and output interface, like comboboxes for selecting values of arguments, an execution button, and a label for displaying an observational result (Fig. 10.2),
2. for each observation or action, the function bean has a corresponding *press event*,
3. for each observation, the interface bean has a corresponding *obs event*,
4. for each observation, the function bean has a corresponding *obs property* that returns the observational result,
5. for each observation or action, (1) the interface bean has a corresponding *press event occurring routine* that occurs the press event when the execution button is pressed and (2) the function bean has a corresponding *press event process routine* that executes the procedure corresponding to the press event, and
6. for each observation, (1) the function bean has a corresponding *obs event occurring routine* that is called from the press event process routine when the observational result is generated and occurs the obs event and (2) the interface bean has a corresponding *obs event process routine* that displays the observational result on the output interface.

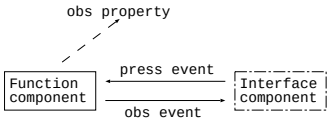

Figure 10.1: A function bean and an interface bean

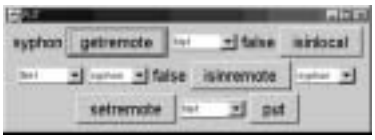

Figure 10.2: Input and output interfaces of an interface bean

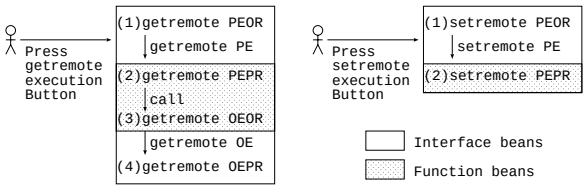

Figure 10.3: Process flows of getremote and setremote of PUT group component

Example 31 *Fig. 10.2 shows input and output interfaces of the interface bean of* PUT *group component. Comboboxes displaying* file1 *or* syphon *are used for selecting arguments.* getremote *button,* isinlocal *button,* isinremote *button,* setremote *button, and* put *button are execution buttons. Labels displaying* syphon *or* false *are used for displaying the observational results when the corresponding execution buttons are pressed. Fig. 10.3 shows process flows of* getremote *and* setremote *of* PUT *group component. PEOR, PE, PEPR, OEOR, OE, and OEPR are abbreviations of a press event occurring routine, a press event, a press event process routine, an obs event occurring routine, an obs event, and an obs event process routine, respectively.*

Because a composite component is a component, a composite component is implemented by using a function bean and an interface bean. The function bean of the composite component is implemented as follows:

1. for each observation, the obs property returns the value of the corresponding obs property of the constructing component,
2. for each observation, the obs event occurring routine occurs the obs event with the value of the corresponding obs property of the constructing component, and
3. for each action, for each constructing component, the press event occurring routine occurs the corresponding press event if it exists and as the result of the occurrence, the corresponding press event process routine is started.

Note that these correspondences are described in the connector specification.

Example 32 *Fig. 10.4 shows process flows of* getremote *and* put *of* PUT *composite component in Example 16.*

10.2 Automated Connector Generation

As we discussed in Section 10.1, a component of tree architecture is implemented by using two kinds of JavaBeans, a function bean and an interface bean. The function bean

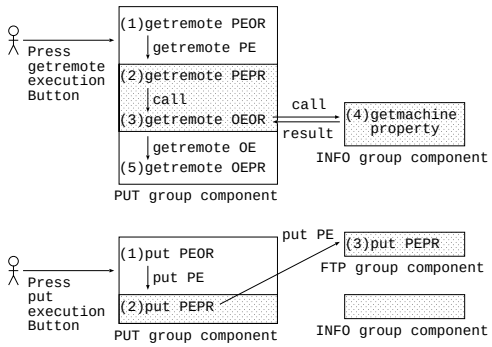

Figure 10.4: Process flows of getremote and put of PUT composite component

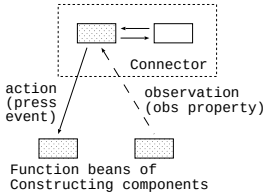

Figure 10.5: The structure of the connector implementation

of the composite component is implemented by using only information described in the connector specification.

But to implement the interface bean, some information is necessary. The input and output interface needs information about what input interfaces for setting arguments are necessary and what an output interface for displaying the observational result is necessary. Moreover, the obs event process routine needs information about how to display the observational result.

The support tool of CBDL using LFMB prepares default interfaces and a default obs event process routine. For setting arguments, textfields are used. For displaying the observational result, a label is used. The default obs event process routine display the observational result on the label.

The connector is the function bean and the interface bean of the composite component (Fig. 10.5). The support tool automatically generates the connector, i.e. these beans by using information described in the connector specification, default interfaces, and a default routine.

As an optional function, the support tool supports a function that changes an input and output interface and an obs event process routine. In fact, the input interfaces of Fig. 10.2 are comboboxes changed by using this function.

10.3 Servlets with JavaBeans Implementation of Tree Architecture

Consider a component of tree architecture and its JavaBeans implementation, i.e. the interface bean and the function bean for the component. As we discussed in Section 10.1, the roles of the interface bean are:

1. the user interface of the component,
2. the sender of *press events*, and
3. the receiver of *obs events*.

We can implement the roles by using a Servlet. We call the Servlet *interface servlet*. By replacing the interface bean with *the interface servlet*, we get another implementation of the component.

Chapter 11

Support Tools

We developed support tools of CBDL.

Firstly, we developed a support tool of CBDL using LFMB[1]. The input of the tool is projection-style behavioral specifications written by using CafeOBJ and components that are JavaBeans. The tool stores components in the inner component library, verifies refinement, generates connectors whose correctness is guaranteed by the refinement verification and which are JavaBeans, and generates component-software by combining the connectors and components of the component library.

Then, we developed support tools of CBDL using LFMB and LFME[2]. Based on the research of the above tool, we design the tool. The tools are a support tool for stand-alone type component-based software (abb. the tool for SA) and a support tool for client-server type component-based software (abb. the tool for CS). In the tool for SA, components and connectors are JavaBeans. In the tool for CS, components and connectors are Servlets and JavaBeans. The input of the tools are (1) UML diagrams with OCL descriptions whose forms are XML files and (2) components. The tools store components in the inner component library, verify consistency of the UML diagrams, generate connectors whose correctness is guaranteed by the verification, and generate component-software by combining the connectors and components of the component library.

11.1 A Support Tool of CBDL using LFMB

The input of the tool is (a) a requirement specification of target software, (b) a refined specification specifying how to combine components, and (c) components (Fig. 11.1). (a) and (b) are projection-style behavioral specifications written by using CafeOBJ . (c) is JavaBeans. The output of the tool is JavaBeans that is obtained by combining (c) and generated connectors (Fig. 11.1). The tool guarantees high reliability of the output by verifying refinement and generating the connectors.

[1]The tool was developed in the project "A Highly Reliable Java Code Generator Using Component Specifications".

[2]The tool was developed in the project "A Management System of Component-based Software Using Internet".

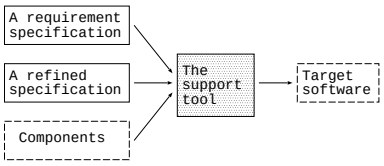

Figure 11.1: Input and output of the support tool

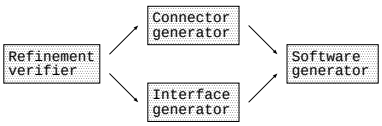

Figure 11.2: The structure of the support tool

Figure 11.3: An outlook of the support tool

11.1.1 The structure of the support tool

The tool is constructed from refinement verifier, connector generator, interface generator, and software generator (Fig. 11.2).

Refinement verifier generates verification scripts and then, sends those scripts to CafeOBJ verification system and gets the results.

Connector generator generates the function bean of the composite component by using the method discussed in Section 10.2.

Interface generator generates the interface bean of the composite component by using the method discussed in Section 10.2.

Software generator generates the target software by combining (1) the function bean and the interface bean of the composite component and (2) the function beans of the constructing components.

11.1.2 The functions of the support tool

Fig. 11.3 is an outlook of the support tool. When the tool starts, the textarea shows parameters of the tool, like TmpDir and SpecDir (Fig. 11.3).

TmpDir directory is the directory in which JavaBeans are stored. So, in TmpDir directory, components of the component library are stored. Moreover, the output of the tool is stored in TmpDir directory.

SpecDir directory is the directory in which CafeOBJ specifications are stored. So, component specifications and connector specifications are selected from these CafeOBJ specifications.

When *Add Spec* button is pressed, a dialog is displayed. By using this dialog, component specifications and connector specifications are selected. At the first time, a component specification is selected as a requirement specification. At the second time and after, a connector specification is selected as a refined specification and refinement verification is executed. If refinement verification fails, the textarea shows unsatisfied equations.

When the textarea shows unsatisfied equations, *Add Spec* button is changed to *Eq Verify* button. When *Eq Verify* button is pressed, the tool enters manual verification mode for an unsatisfied equation. After writing a verification script for the unsatisfied equation on the textarea and then pressing *Verify* button, *refinement verifier* executes verification whether this script succeeds. By iterating this process for all the unsatisfied equations, the tool executes refinement verification. When all the unsatisfied equations are verified, *Eq Verify* button is changed to *Add Spec* button. By using manual verification mode, we found the idea of the automated refinement verification.

Constructing components of composite components may be composite components. This means composite components may have hierarchical structures. The tool supports stepwise refinement to deal with the hierarchical structures. A component specification imported to a connector specification may have a corresponding connector specification. The process of stepwise refinement is as follows: In a stage, the former connector specification is input and refinement verification is executed by using *Add Spec* button. In the next stage, the latter connector specification is input and refinement verification between the component specification and the latter connector specification is executed by using *Add Spec* button.

When *Add Comp* button is pressed, a dialog is displayed. By using this dialog, the correspondences between (1) the component specifications input by using the dialog of Add Spec button and (2) components in the component library are input. The software generator uses these correspondences.

When *Gene Comp* button is pressed, the connector generator and the interface generator generate the function beans and the interface beans, respectively.

When *Gene App* button is pressed, the software generator generates the target software.

If *Chg Data Comp* button is pressed before Gene Comp button is pressed, a dialog is displayed. By using this dialog, input and output interfaces and obs event process routines are changed as we discussed in Section 10.2.

11.2 Support Tools of CBDL using LFMB and LFME

The tools are a support tool for stand-alone type component-based software (abb. the tool for SA) and a support tool for client-server type component-based software (abb. the tool for CS). In the tool for SA, components and connectors are JavaBeans. In the tool for CS, components and connectors are Servlets and JavaBeans. The input of the tools are (1) UML diagrams with OCL descriptions whose forms are XML files and (2) components. The tools store components in the inner component library, verify consistency of the UML diagrams, generate connectors whose correctness is guaranteed by the verification, and generate component-software by combining the connectors and components of the component library.

We developed the tools as client-server type component-based software. So, component developers and component users can access to the tools through Internet at any place at any time.

11.2.1 The structure of the support tools

The tools are constructed from

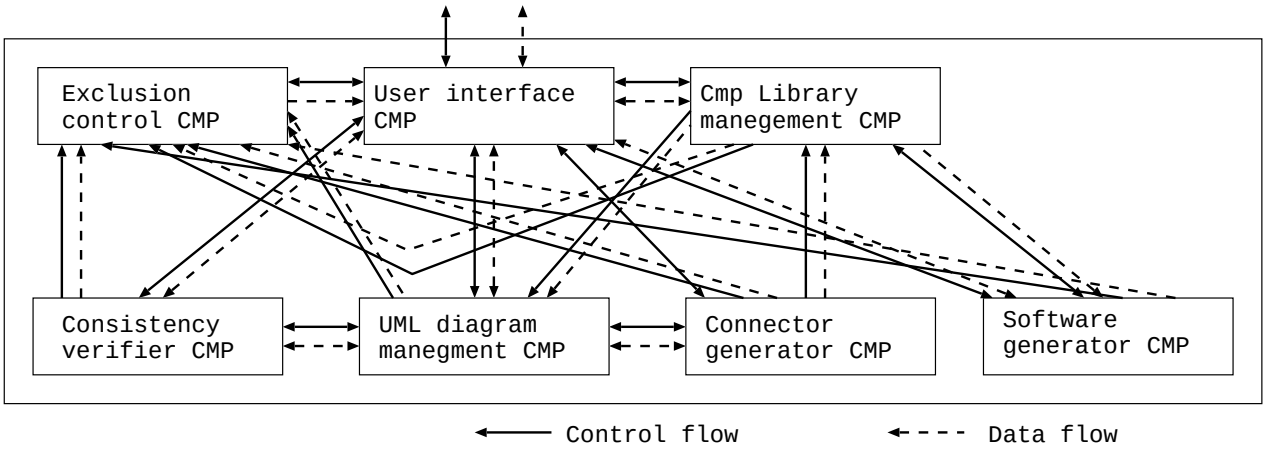

Figure 11.4: The structure of the support tools

1. UML diagram management component,
2. consistency verifier component,
3. connector generator component,
4. component library management component,
5. software generator component,
6. exclusion control component, and
7. user interface component (Figure. 11.4).

The tool for SA and the tool for CS are the same without connector generator component and software generator component.

11.2.2 The functions of the support tools

The functions of the support tools are as follows.

UML diagram management component

UML diagram management component manages UML diagrams of the target domain. It provides the following functions:

1. the function for storing a UML diagram,
2. the function for deleting a stored UML diagram,
3. the function for searching a selected UML diagram,
4. the function for making the list of the stored UML diagrams,

5. the function for making the list of the component names occurring in the stored UML diagrams,

6. the function for generating the information about unverified properties,

7. the function for changing the information about verified properties,

8. the function for generating the information about ungenerated connectors, and

9. the function for changing the information about generated connectors.

Consistency verifier component

Consistency verifier component verifies the consistency of the stored UML diagrams. To verify the consistency, firstly, it translates UML diagrams into CafeOBJ specifications. Then, it generates verification scripts for the consistency verification. Finally, by sending the CafeOBJ specifications and the verification scripts to the CafeOBJ verification system, it verifies the consistency. It provides the following functions:

1. the function for translating the stored UML diagrams into CafeOBJ specifications,

2. the function for generating the verification scripts of refinement verification about business model,

3. the function for generating the verification scripts of refinement verification about component specification,

4. the function for generating the verification scripts of verification of satisfaction of static invariants,

5. the function for manipulating the CafeOBJ verification system, and

6. the function for supporting the verification that uses the verification scripts written by users.

Connector generator component

Connector generator component for the tool for SA generates function beans and interface beans corresponding to connectors from the verified CafeOBJ verification. It provides the following functions:

1. the function for generating function beans and

2. the function for generating interface beans.

Connector generator component for the tool for CS generates function beans and interface servlets corresponding to connectors from the verified CafeOBJ verification. It provides the following functions:

1. the function for generating function beans and

2. the function for generating interface servlets.

Component library management component

Component library management component manages

1. *components* that provides basic functionalities of the target domain and
2. *connectors* that are generated by connector generator component.

It provides the following functions:

1. the function for storing a component,
2. the function for deleting a stored component,
3. the function for searching a selected component,
4. the function for making the list of the stored components' names,
5. the function for storing a connector,
6. the function for deleting a stored connector,
7. the function for searching a selected connector, and
8. the function for making the list of the stored connectors' names.

Software generator component

Software generator component, firstly, generates the target software by combining components and connectors that are stored in component library management component, then, collect the binary files of the software as the archive, finally, send the archive to users. It provides the following functions:

1. the function for generating the target software,
2. the function for collecting the binary files of the software as the archive, and
3. the function for sending the archive.

Exclusion control component

Exclusion control component controls concurrent accesses to stored UML diagrams from some users. It provides the following functions:

1. the function for checking whether the selected UML diagrams can be changeable and
2. the function for changing the management information of the UML diagrams.

It decides whether the selected UML diagrams can be changeable based on the management information.

User interface component

User interface component mediates between users and the other components. It provides the following functions:

1. the function for selecting an user interface screen depending on a situation and
2. the function for controlling the other components to execute a user request.

Chapter 12

Case Studies

We did case studies:

1. of the domain of file transfer programs and
2. of the domain of online bookstores.

12.1 The Domain of File Transfer Programs

We did a case study of the domain of file transfer programs by using the support tool of CBDL using LFMB.

The software family of the domain includes *PUT-A* that transfers A's files on the local machine to a remote machine and *GET-B* that transfers B's files on a remote machine to the local machine. The component library is divided into *FTP* group that transfers files and *INFO* group that manages personal information, like user names and passwords (Fig. 12.1). *FTPftp* and *FTPcopy* provide file transfer functions using FTP protocol and using copy command of OS, respectively. *FTPftp* and *FTPcopy* belong to *FTP* group. *INFO-A* and *INFO-B* provide management functions of A's personal information and B's personal information, respectively. *INFO-A* and *INFO-B* belong to *INFO* group. *PUT-A* is constructed from *FTPftp*, *INFO-A*, and the connector of *PUT*. *GET-B* is constructed from *FTPftp*, *INFO-B*, and the connector of *GET*.

We developed *INFO-A* and *INFO-B* from scratch. But, we developed *FTPftp* and *FTPcopy* by using the free software providing functions about FTP protocol and copy command of OS, respectively. By developing "the lappers" and by combining them with the existing software, we developed *FTPftp* and *FTPcopy*.

We developed *PUT-A* as follows. Firstly, we prepared the component library with CafeOBJ specifications. Secondly, we input

1. the primitive component specification that specifies behavior of *PUT* group and
2. the component specification that specifies how to combine *FTP* group and *INFO* group to develop the *PUT* group

into the tool. Then, we selected *FTPftp* and *INFO-A* on the tool. Finally, the tool verified refinement and generated *PUT-A* whose correctness is assured by the refinement verification (Fig. 12.2).

Through this case study, we show that:

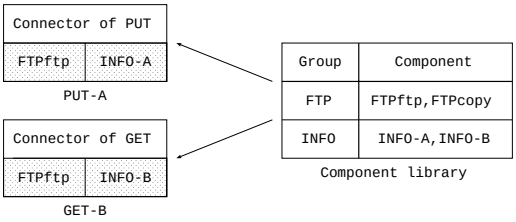

Figure 12.1: The software family and the component library

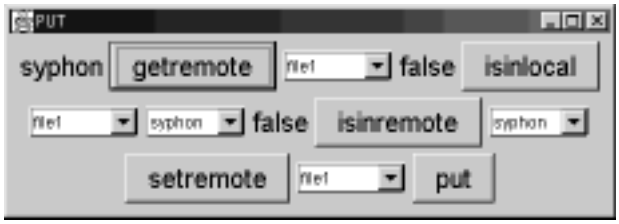

Figure 12.2: The outlook of PUT-A component

1. for the small domain, the support tool is useful for generating components whose correctness is assured and

2. we can develop components of the component library by using existing software.

12.2 The Domain of Online Bookstores

We are doing a case study of the domain of online bookstores by using the tool for CS, i.e. a support tool of CBDL using LFMB and LFME.

Firstly we analyzed the behavior of the existing online bookstores. Secondly, we specified the extracted business processes in UML diagrams. Then, we decided primitive components and specified behavior of the component in UML diagrams. Note that the components are contents of the component library. Finally, we specified behavior of an online bookstore system in UML diagrams. In the process, we noticed that the technique for dealing with large quantities of UML diagrams is necessary and the technique of frameworks is a good candidate of the solution. Because the frameworks correspond to parameterized specifications of algebraic specifications, the consistency verification in the UML diagrams can be executed efficiently by using the frameworks.

Chapter 13

Conclusion

In the thesis, we discussed:

1. lightweight formal methods for component-based software *LFMB* and *LFME*,
2. the component-based software development *CBDL* using LFMB or LFME, and
3. the support tools of CBDL.

By using CBDL, the obstacles of component reuse that are:

1. a lack of a consensus about component usage between component developers and software developers, i.e. component users and
2. an architectural mismatch

are eliminated. In CBDL, the former obstacle is eliminated by specifying business models and component specifications by using UML diagrams with OCL descriptions and verifying consistency in the UML diagrams. Because LFMB and LFME include automated verification methods of the consistency, component developers and software developers who are not familiar with formal methods can get the benefit of the verification by using the support tools of CBDL. In CBDL, the latter obstacle is eliminated by selecting *tree architecture* that we developed.

We can regard the UML diagrams as specifications specified by *a language for programming-in-the-large*. So, in CBDL, moreover, we generate component-based software from the UML diagrams by combining the connectors specified in the UML diagrams and components of a component library. Note that the above consistency verification guarantees the correctness of the connectors.

The support tools of CBDL are designed as client-server type software. So, component developers and software developers can access to the tools through Internet at any place at any time.

To summarize, by using the support tools of CBDL, we can increase component reuse and correctness of component-based software.

We compare *refinement verification* of LFMB and LFME. Because the verification results are the same, we predicted that the logic of LFMB was equational logic. The prediction is true. So, we conclude that consistency verification in the UML diagrams is a problem that only needs equational logic.

Through case studies, we noticed that the technique for dealing with large quantities of UML diagrams is necessary and the technique of frameworks is a good candidate of the solution. Because the frameworks correspond to parameterized specifications of algebraic specifications, the consistency verification in the UML diagrams can be executed efficiently by using the frameworks. So, our future work is:

1. the study for getting knowhow for using the frameworks and
2. improvements on the support tool for supporting techniques related to the frameworks.

Bibliography

[1] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[2] Leonor Barroca, Jon Hall, and Patrick Hall, editors. *Software architectures : advances and applications*. Springer-Verlag, 1999.

[3] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transaction on Software Engineering and Methodology*, 1(4):355–398, 1992.

[4] Juan C. Bicarregui, John S. Fitzgerald, Pater A. Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. Springer-Verlag, 1994.

[5] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165:3–55, 1996.

[6] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The unified modeling language user guide*. Addison-Wesley, 1999.

[7] Samuel Buss and Grigore Roşu. Incompleteness of behavioral logics. In *Proceedings of CMCS'2000*, volume 33 of *ENTCS*. Elsevier Science, 2000.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. Components and generative programming (in ESEC/FSE'99). *Software Engineering Notes*, 24(6):2–19, 1999.

[9] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on software engineering*, 2(2):80–86, 1976.

[10] Răzvan Diaconescu and Kokichi Futatsugi. CafeOBJ *Report*. AMAST Series in Computing 6. World Scientific, 1998.

[11] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6(1):74–96, 2000.

[12] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks in UML*. Addison-Wesley, 1998.

[13] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1994.

[14] Joseph Goguen. *Theorem Proving and Algebra*. MIT Press, to appear.

[15] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245:55–101, 2000.

[16] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[17] Joseph Goguen, Timothy Winkler, José Mesegure, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, SRI International, Computer Science Laboratory, 1993.

[18] The RAISE Method Group. *The RAISE Development Method*. Prentice Hall, 1995.

[19] Rolf Hennicker. Context induction: a proof principle for behavioural abstractions. In *Design and Implementation of Symbolic Computation Systems. International Symposium DISCO 1990*, number 429 in LNCS, pages 101–110. Springer-Verlag, 1990.

[20] Rolf Hennicker and Michel Bidoit. Observational logic. In *Algebraic Methodology and Software Technology (AMAST'98)*, number 1548 in LNCS, pages 263–277. Springer-Verlag, 1999.

[21] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu, Kokichi Futatsugi, and Dorel Lucanu. Concurrent object composition in CafeOBJ. Technical Report IS-RR-98-0009S, JAIST, 1998.

[22] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, 2000.

[23] Stéphane Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33:175–193, 1984.

[24] K. Lano. *The B Language and Method: A Guide to Practical Formal Development*. Springer-Verlag, 1996.

[25] Michihiro Matsumoto and Kokichi Futatsugi. Test set coinduction — toward automated verification of behavioural properties —. In *Proceedings of Second International Workshop on Rewriting Logic and It's applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

[26] Michihiro Matsumoto and Kokichi Futatsugi. Object composition and refinement by using non-observable projection operators: A case study of the automated teller machine system. In *OBJ/CafeOBJ/Maude at Formal Methods '99*, pages 133–157. THETA, 1999.

[27] Michihiro Matsumoto and Kokichi Futatsugi. Simply observable behavioral specification. In *Proceedings of Asia-Pacific Software Engineering Conference'99*, pages 460–467. IEEE, 1999.

[28] Michihiro Matsumoto and Kokichi Futatsugi. Highly reliable component-based software development by using algebraic behavioral specification. In *Proceedings of Third IEEE International Conference on Formal Engineering Methods*, pages 35–43. IEEE, 2000.

[29] Michihiro Matsumoto and Kokichi Futatsugi. The support tool for highly reliable component-based software development. In *Proceedings of Asia-Pacific Software Engineering Conference'2000*, pages 172–179. IEEE, 2000.

[30] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development (in OOPSLA'98). *ACM SIGPLAN Notices*, 33(10):97–116, 1998.

[31] David L. Parnas. On the design and development of program families. *IEEE Transactions on software engineering*, 2(1):1–9, 1976.

[32] Ruben Prieto-Diaz and James M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6:307–334, 1986.

[33] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming'98*, number 1445 in LNCS, pages 550–570. Springer-Verlag, 1998.

[34] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[35] Clemens Szyperski. *Component software*. Addison-Wesley, 1997.

[36] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 676–788. Elsevier Science, 1990.

[37] Toshiyuki Yamada, Jurgen Avenhaus, Carlos Loría-Sáenz, and Aart Middeldorp. Logicality of conditional rewrite systems. *Theoretical Computer Science*, 236:209–232, 2000.

Publications

[1] Takashi Nagaya, Michihiro Matsumoto, Kazuhiro Ogata and Kokichi Futatsugi: "How to Give Local Strategies to Function Symbols for Equality of Two Implementations of the E-strategy with and without Evaluated Flags", Proceedings of Asian Symposium on Computer Mathematics (ASCM'98), p71-81, Lanzhou University Press, 1998.

[2] Michihiro Matsumoto and Kokichi Futatsugi: "Test Set Coinduction - Toward Automated Verification of Behavioural Properties -", Proceedings of Second International Workshop on Rewriting Logic and It's applications (WRLA'98), Electronic Notes in Theoretical Computer Science, Vol. 15, Elsevier Science, 1998.

[3] Michihiro Matsumoto and Kokichi Futatsugi: "Object-Oriented Algebraic Specification in Hidden Sorted Algebra", Proceedings of Foundation of Software Engineering'98 (FOSE'98), p157-162, Kindaikagakusya, 1998 (In Japanese).

[4] Michihiro Matsumoto and Kokichi Futatsugi: "Verification Methods Based on Behavioral Semantics", Computer Software, Vol.16, No.2, p47-50, 1999 (In Japanese).

[5] Michihiro Matsumoto and Kokichi Futatsugi: "Object Composition and Refinement by using Non-Observable Projection Operators: A Case Study of the Automated Teller Machine system", OBJ/CafeOBJ/Maude at Formal Methods'99, p133-157, THETA, 1999.

[6] Michihiro Matsumoto and Kokichi Futatsugi: "Specifications of Object Hierarchical Structures and Refinement by using Behavioral Semantics", Proceedings of Foundation of Software Engineering'99 (FOSE'99), p132-139, Kindaikagakusya, 1999 (In Japanese).

[7] Michihiro Matsumoto and Kokichi Futatsugi: "Simply Observable Behavioral Specification", Proceedings of 6th Asia-Pacific Software Engineering Conference (APSEC'99), p460-467, IEEE, 1999.

[8] Michihiro Matsumoto and Kokichi Futatsugi: "Highly Reliable Component-Based Software Development by using Algebraic Behavioral Specification", Proceedings of 3rd IEEE International Conference on Formal Engineering Methods (ICFEM'2000), p35-43, IEEE, 2000.

[9] Michihiro Matsumoto and Kokichi Futatsugi: "Highly Reliable Component-based Software Development by using Projection-style Behavioral Specification", Proceedings of Foundation of Software Engineering'2000 (FOSE'2000), p229-236, Kindaikagakusya, 2000 (In Japanese).

[10] Michihiro Matsumoto and Kokichi Futatsugi: "The Support Tool for Highly Reliable Component-Based Software Development", Proceedings of 7th Asia-Pacific Software Engineering Conference (APSEC'2000), p172-179, IEEE, 2000.

[11] Michihiro Matsumoto and Kokichi Futatsugi: "The Tool that Supports Highly Reliable Component-Based Software Development", The Transactions of the IEICE D-I, Vol.J84-D-I, No.6, p736-744, 2001 (In Japanese).

[12] Michihiro Matsumoto, Yoshihito Katayama, Takanori Nakama, Yoshiharu Hashimoto, and Kokichi Futatsugi: "A Lightweight Formal Method for the Catalysis Approach", Proceedings of Foundation of Software Engineering'2001 (FOSE'2001), p159-162, Kindaikagakusya, 2001 (In Japanese).

[13] Michihiro Matsumoto and Kokichi Futatsugi: "Verification of behavioral equations by using Test Set Coinduction", Computer Software, Vol.19, No.1, p10-21, 2002 (In Japanese).

Projects

The projects that the author was the project leader are as follows:

1. Michihiro Matsumoto, Shusaku Iida, and Kokichi Futatsugi, "A Tool That Supports Code Generation of Highly Reliable Java Codes using Component Specifications", Support program for young software researchers 99-004, Information-technology Promotion Agency (IPA) and Research Institute of Software Engineering (RISE).

2. Michihiro Matsumoto, Yoshihito Katayama, Takanori Nakama, Yoshiharu Hashimoto, and Kokichi Futatsugi, "A Management System of Component-based Software Using Internet", Support program for young software researchers 01-006, Information-technology Promotion Agency (IPA) and Research Institute of Software Engineering (RISE).