Title	振舞仕様の検証方法に関する研究
Author(s)	松本，充広
Citation	
Issue Date	1998-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1144
Rights	
Description	Supervisor:二木 厚吉，情報科学研究科，修士

Verification Methods for Behavioural Specifications

By MICHIHIRO MATSUMOTO

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

Written under the direction of
Professor KOKICHI FUTATSUGI

February 13, 1998

Abstract

The purposes of our research are to clear problems of previous verification methods for behavioural specifications and to propose improved verification methods. We selected behavioural semantics (hidden algebras) as the foundations of our research. Behavioural specifications are specifications whose semantics are behavioural semantics. As to verification methods for behavioural specifications, there are coinduction and induction over length of contexts. To use coinduction, users must find a hidden congruence. Until now, this hidden congruence should be given by hand. Note that relations which can be defined on verification systems are relations defined by syntax — we call these relations syntactically definable hidden congruences. Firstly, we show that the only useful syntactically definable hidden congruence for verifications is behavioural equivalence. Behavioural equivalence is the conjunction over all visible contexts. Consequently, a selection of hidden congruences corresponds to a selection of the set of visible contexts which construct behavioural equivalence. We provide the algorithm which generates a simple form by eliminating redundant visible contexts. That is *GSB*-algorithm (test set coinduction). By analysing the structure of the set of all visible contexts, we show the sufficient condition that *GSB*-algorithm can eliminate all redundant visible contexts. Until now, coinduction was regarded more efficient than induction over length of contexts. By analysing the structure of the set of all visible contexts, we show the case that coinduction (test set coinduction) coincides with induction over length of contexts. The main application of these verification methods is stepwise refinement of behavioural specifications as restriction of possible implementations. As to the above research, there are researches by Dr.Goguen and Dr.Malcolm. But, these are not satisfactory. Firstly, they give the original specification (for example, a stack). Then, they construct it from primitive modules (for example, an array and a pointer) in the refined specification. Finally, they prove that the composed module (for example, a stack constructed from the array and the pointer) satisfy the original specification. In the last process, they treat the composed module as data. But, in behavioural specification, specifications of concurrent systems must be treated as black boxes. Concretely, the problem is that there are states of the composed module which do not correspond to states of primitive modules. We provide projection operators which specify correspondences between states of composed module and states of primitive modules. We provide the method which construct a composed module from primitive modules using these projection operators. We call the specifications which are written under the above method object-oriented specifications. Specifying concurrent systems by using object-oriented specifications, we solved the above problem. Moreover, we provide the method to verify stepwise refinement of object-oriented specifications.

Contents

Chapter 1

Introduction

The purposes of our research are to clear problems of previous verification methods for behavioural specifications and to propose improved verification methods. We want to decrease costs of developments of concurrent systems. As a candidate of solutions, we selected verification methods for behavioural specifications.

Concurrent systems are constructed from many objects that communicate with other objects. Because possible states and transitions are huge, numbers of the necessary tests to ensure reliability are also huge. Therefore, costs of these tests is high. On the other hand, logical verifications can find bugs of the logical level and costs of logical verifications is lower than those of the tests. Consequently, we can expect the costs of developments to decrease by exchanging the tests for a combination of tests and logical verifications.

From this expectation, logical verification methods have been studied in process algebras [Hoa85, Mil89, BW90]. We think abstract data types (abbreviate *ADT*) have key roles when we verify data flows over concurrent systems. But, most of process algebras can not deal with *ADT*. In behavioural semantics (hidden algebras) [GM97], concurrent systems are treated as black boxes. So, behavioural semantics (hidden algebras) can be seen as a generalization of process algebras which can deal with *ADT*. So, we can adapt the techniques provided in process algebras for behavioural semantics (hidden algebras) and we can deal with *ADT* in behavioural semantics (hidden algebras). Therefore, we selected behavioural semantics (hidden algebras) as the foundations of our research. Behavioural specifications are specifications whose semantics are behavioural semantics.

In behavioural specification, we specify interactions between a concurrent system and a user. Operations which observe the states (of the concurrent system) are called attributes and operations which change the states are called methods. Attributes and methods are called behavioural operators. We can only recognize the current state by observing states changed by methods through attributes. So, we can regard method sequences with an attribute as observation tools. these observation tools are called visible contexts. Behavioural equivalence $\equiv$ between states s, s' are defined as follows:

$(s \equiv s') = \wedge_{ct \in VisCt}(ct[s] == ct[s'])$

where $VisCt$ is the set of all visible contexts. In behavioural specifications, we verify behavioural properties that are behavioural equivalence relations between states of concurrent systems.

As to verification methods for behavioural specifications, there are coinduction and induction over length of contexts [GM97]. The main application of these verification methods is stepwise refinement of behavioural specifications as restriction of possible implementations [GM97, MG96].

Coinduction is a verification method based on the following fact:

behavioural equivalence is the largest hidden congruence,

where a hidden congruence is a congruence such that: identity relation on data values. Consider to verify a behavioural property $s \equiv s'$. The algorithm of coinduction is as follows:

1. find a candidate R of hidden congruences,
2. check whether R is a hidden congruence, and
3. verify whether $s \equiv s'$ holds, by proving $s\ R\ s'$.

So, to use coinduction, users must find a hidden congruence. Until now, this hidden congruence should be given by hand [GM97, BH94, BH96].

Note that relations which can be defined on verification systems are relations defined by syntax — we call these relations **syntactically definable hidden congruences**. Firstly, we show that the only useful syntactically definable hidden congruence for verifications is behavioural equivalence. Therefore, R should be behavioural equivalence. Behavioural equivalence is the conjunction over all visible contexts. Consequently, a selection of hidden congruences corresponds to a selection of the set of visible contexts which construct behavioural equivalence. We let R denote the form of behavioural equivalence defined by syntax — conjunction over visible contexts — and we let $\#(R)$ denote the numbers of these visible contexts. We regard a verification method which use R as an **efficient method** if $\#(R)$ is small. We regard R as a **simple form** if $\#(R)$ is small. So, to verify behavioural properties efficiently, we need a simple form of behavioural equivalence. By eliminating redundant visible contexts, we get this simple form. We provide the algorithm which generates this simple form. That is ***GSB*-algorithm (test set coinduction)**.

Consider to verify a behavioural property $s \equiv s'$. The algorithm of test set coinduction is as follows:

1. generate a simple form R of behavioural equivalence (by *GSB*-algorithm), and
2. verify whether $s \equiv s'$ holds, by proving $s\ R\ s'$.

By analysing the structure of the set of all visible contexts, we show the sufficient condition that *GSB*-algorithm can eliminate all redundant visible contexts.

Until now, coinduction was regarded more efficient than induction over length of contexts [GM97, BH94, BH96]. By analysing the structure of the set of all visible contexts, we show the case that coinduction (test set coinduction) coincides with induction over length of contexts.

As to research of stepwise refinements of behavioural specifications as restriction of possible implementations, there are researches by Dr.Goguen and Dr.Malcolm [GM97, MG96]. But, these are not satisfactory. Firstly, they give the original specification (for example, a stack). Then, they construct it from primitive modules (for example, an array and a pointer) in the refined specification. Finally, they prove that the composed module (for example, a stack constructed from the array and the pointer) satisfy the original specification. In the last process, they treat the composed module as data. But, in behavioural specification, specifications of concurrent systems must be treated as black boxes. Concretely, the problem is that there are states of the composed module which do not correspond to states of primitive modules.

We provide **projection operators** which specify correspondences between states of composed module and states of primitive modules. We provide the method which construct a composed module from primitive modules using these projection operators. We call the specifications which are written under the above method **object-oriented specifications**. Specifying concurrent systems by using object-oriented specifications, we solved the above problem. Moreover, we provide the method to verify stepwise refinement of object-oriented specifications.

As to the previous version of projection operators — we call these operators **pseudo-projection operators** in this paper —, there is a co-operative research with Mr.Iida, Dr.Diaconescu, and Dr.Lucanu. We only wrote our contribution in this paper. In this co-operative research, we specify dynamic systems using pseudo-projection operators. By changing contents of *ObjId* dynamically, we can specify dynamic systems.

The difference between projection operators and pseudo-projection operators is that projection operators are ordinary operators but pseudo-projection operators are behavioural operators. Consider to construct a stack from an array and a pointer. If we use pseudo-projection operators, we get just an array with a pointer. We can observe all contents of the array through visible contexts. On the other hand, if we use projection operator, we get a stack. We can only observe contents under a pointer. By using projection operator, we can restrict the set of visible contexts. To compose modules, this kind of restriction is necessary.

This paper is structured as follows. Chapter 2 comprises some preliminary definitions and results. Chapter 3 presents syntactically definable hidden congruence. Chapter 4 presents test set coinduction. Chapter 5 presents extensions of test set coinduction. Chapter 6 presents object composition. Chapter 7 presents stepwise refinement. Chapter 8 presents related works. Finally, Chapter 9 summarizes the results of this paper.

Chapter 2

Preliminaries

In this chapter, we introduce some preliminary definitions and results. Most of those are originally from [Gog, GM97]. But, we slightly customized some of those. See remarks.

2.1 Algebraic Specification

2.1.1 Signature, Algebra, and Term

Signature

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

Definition 2 (from [Gog]) *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 (from [Gog]) *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 (from [Gog]) *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 (from [Gog]) *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 (from [Gog]) *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)**. □

2.1.2 Homomorphism, Equation and Satisfaction

Homomorphism

Definition 8 (from [Gog]) *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 (from [Gog]) *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.* $\square$

Equation

Definition 10 (from [Gog]) 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.* $\square$

Definition 11 (from [Gog]) *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* $w \neq []$. $\square$

Definition 12 *We call* $\Sigma(X)$*-terms* **Σ-terms with variables***. We call* Σ*-terms* **ground Σ-terms***.* $\square$

Definition 13 (from [Gog]) *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:*
$(\forall X) t = t'$. $\square$

Definition 14 (from [Gog]) *A* **conditional Σ-equation** *consists of a ground signature* X *disjoint from* Σ*, a set* C *of pairs 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* C. $\square$

Satisfaction

Fact 1 (from [Gog]) *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*.* $\square$

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

Definition 15 (from [Gog]) *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* $\overline{sb_2}\ sb_1 : X \to T_\Sigma(Z)$*.* $\square$

Definition 16 (from [Gog]) *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 17 (from [Gog]) *A* Σ*-algebra* M **satisfies** *a conditional* Σ*-equation* $(\forall X)t = t'$ *if* C *iff for any assignment* $as : X \to M$*, if* $as(t_i) = as(t'_i)$ *for each* $\langle t_i, t'_i\rangle \in C$*, 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:*
$M \models_\Sigma E$. □

Fact 2 (from [Gog]) *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.

2.1.3 Specification and Model

Specification

Definition 18 (from [Gog]) *A* **specification** *is a pair* (Σ, E)*, consisting of a signature* Σ *and a set* E *of conditional* Σ*-equations.* □

Model

Definition 19 *Given a specification* (Σ, E)*, a* (Σ, E)**-model** M *is a* Σ*-algebra such that:*
$M \models E$. □

Definition 20 (from [Gog]) *Let* E *be a set of conditional* Σ*-equations, and let* eq *be a* Σ*-equation. Then*
$E \models eq$
iff for all (Σ, E)*-models* M*,* $M \models eq$*.* □

Term Model

Definition 21 (from [Gog]) *Given a* Σ*-algebra* M*, a* Σ**-congruence relation** *on* M *is an* S*-sorted equivalence relation* $\equiv = \{\equiv_s \mid s \in S\}$ *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 $e_i, e'_i \in M_{s_i}$*.* □

Fact 3 (from [Gog]) *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 $e_i \in M_{s_i}$.* $\Box$

Corollary 1 *Given a 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))**. $\Box$

Specification Equivalence

Before we can give the definition of specification equivalence, we need some more notations. Each Σ-algebra has an interpretation of each operation symbol $\sigma \in \Sigma$ as an actual operation; we show how this extends to an interpretation for Σ-terms with variables.

Definition 22 (from [Gog]) *We let $\#(S)$ denote the cardinality of a set S.* $\Box$

Definition 23 (from [Gog]) *Given $w = s_1 \cdots s_l \in S^*$, we let $X^{\langle w \rangle}$ denote a S-sorted ground signature disjoint from Σ such that $\#(X_s^{\langle w \rangle}) = \#\{i \mid s_i = s\}$* $\Box$

One way to construct such a signature is to let $|X^{\langle w \rangle}| = \{x_1, \ldots, x_l\}$ where $l = \#(w)$, and then let $X_s^{\langle w \rangle} = \{x_i \mid s_i = s\}$. For example, if $S = \{a, b, c\}$ and $w = abbac$, then $X^{\langle w \rangle}$ has $X_a^{\langle w \rangle} = \{x_1, x_4\}$, $X_b^{\langle w \rangle} = \{x_2, x_3\}$, and $X_c^{\langle w \rangle} = \{x_5\}$.

Definition 24 (from [Gog]) *Given a signature Σ, the signature of all* **derived Σ-operations** *is the S-sorted signature $Der(\Sigma)$ with:*
$Der(\Sigma)_{w,s} = T_\Sigma(X^{\langle w \rangle})_s$ for each $w \in S^*$ and each $s \in S$.
Any $t \in Der(\Sigma)_{w,s}$ defines an actual operation $M_t : M_w \to M_s$ as follows:
given $a \in M_w$, there is a naturally corresponding S-indexed map $as : X^{\langle w \rangle} \to M$, which lets us view M as a $\Sigma(X^{\langle w \rangle})$-algebra; hence there is a unique $\Sigma(X^{\langle w \rangle})$-homomorphism $\overline{as} : T_{\Sigma(X^{\langle w \rangle})} \to M$ which lets us define $M_t(a)$ to be $\overline{as}(t)$.
This is called the **derived operation** *defined by t. In this way, we can view any Σ-algebra M as a $Der(\Sigma)$-algebra, also denoted M.* $\Box$

Definition 25 (from [Gog]) *Given signatures Σ and Σ' with sort sets S and S' respectively, then a* **signature morphism** *$\varphi : \Sigma \to \Sigma'$ consists of a map $f : S \to S'$ and an S-indexed map g with components $g_{w,s} : \Sigma_{w,s} \to \Sigma'_{f(w),f(s)}$ where f is extended to lists by $f([]) = []$ and $f(s_1 \cdots s_l) = f(s_1) \cdots f(s_l)$. Given $s \in S$, we may write $\varphi(s)$ instead of $f(s)$, and given $\sigma \in \Sigma_{w,s}$, we may write $\varphi(\sigma)$ instead of $g(\sigma)$.* $\Box$

Definition 26 (from [Gog]) *Given a signature morphism $\varphi : \Sigma \to \Sigma'$ and a Σ'-algebra M, we get a Σ-algebra, called the* **reduct** *of M under φ and denoted φM, as follows:*

1. *given $s \in S$, let $(\varphi M)_s = M_{\varphi(s)}$;*

2. *given $\sigma \in \Sigma_{w,s}$, let $(\varphi M)_\sigma = M_{\varphi(\sigma)} : M_{\varphi(w)} \to M_{\varphi(s)}$.*

In particular, given a signature morphism $\varphi : \Sigma \to Der(\Sigma')$ and a Σ'-algebra M, we can view M as $Der(\Sigma')$-algebra by Definition 24, and then get a Σ-algebra denoted φM from the construction above. $\square$

Definition 27 (from [Gog]) *An* **interpretation** *of specifications $\varphi : (\Sigma, E) \to (\Sigma', E')$ is a signature morphism $\varphi : \Sigma \to Der(\Sigma')$ such that for each Σ'-algebra M':*
$M' \models_{\Sigma'} E'$ *implies* $\varphi M' \models_{\Sigma} E$. $\square$

Definition 28 (from [Gog]) *Two specifications (Σ, E) and (Σ', E') are* **specification equi-**
valent *iff there exists two interpretations $\varphi : (\Sigma, E) \to (\Sigma', E')$ and $\psi : (\Sigma', E') \to (\Sigma, E)$ such that:*
$\varphi(\psi M) = M$, *and*
$\psi(\varphi M') = M'$
for each (Σ, E)-model M and each (Σ', E')-model M'. $\square$

2.1.4 Loose Semantics and Initial Semantics

Loose Semantics

Definition 29 (from [Gog]) *Given a specification (Σ, E), we call semantics which is given by all (Σ, E)-models* **loose semantics**. $\square$

Definition 30 *Given a specification (Σ, E). Let eq be an equation such that: $E \models eq$. We call eq* **properties of** (Σ, E). $\square$

Initial Semantics

Fact 4 (from [Gog]) *Given a specification (Σ, E), then the term model $T_{\Sigma,E}$ is an* **initial** (Σ, E)**-model**, *in the sense that for each Σ-algebra M and each assignment $as : X \to M$, there is a unique Σ-homomorphism $\overline{as} : T_{\Sigma,E}(X) \to M$ such that $\overline{as}(x) = as(x)$ for each $x \in X$.* $\square$

Definition 31 (from [Gog]) *Given a specification (Σ, E), and a Σ-equation eq, we say that (Σ, E)* **initially satisfies** *eq iff $T_{\Sigma,E} \models_{\Sigma} eq$; in this case we write $E \models\!\!\!\cong_{\Sigma} eq$, and we may omit the subscript Σ when there is no confusion.* $\square$

Definition 32 (from [Gog]) *Given a specification (Σ, E), we call semantics which is given by the term model* **initial semantics**. $\square$

Definition 33 *Given a specification (Σ, E). Let eq be an equation such that: $E \models\!\!\!\cong eq$. We call eq* **inductive properties of** (Σ, E). $\square$

2.2 Behavioural Semantics (Hidden Algebra)

We regard systems as black boxes. Hidden algebra captures the fundamental distinction between observational (visible) data values and internal states of a black box by modeling the former with visible sorts and the latter with hidden sorts. These are treated in this chapter.

2.2.1 Visible Data Values

Definition 34 *Given a specification (Ψ, E_Ψ) where Ψ is a V-sorted signature and each $(T_{\Psi,E_\Psi})_v$ is non-empty for each $v \in V$, then a* **data algebra** *D is T_{Ψ,E_Ψ}. We call V* **visible sorts**, *we call (Ψ, E_Ψ) a* **data specification**, *and we call (V, Ψ, D) the* **visible data universe**. □

Remark 1 *The above definition of data algebra is slightly different from the original one [GM97]. The original one is as follows:*

let D be a fixed data algebra, with Ψ its signature and V its sort set, such that each D_v with $v \in V$ is non-empty and for each $d \in D_v$ there is some $\psi \in \Psi_{[],v}$ such that ψ is interpreted as d in D.

To specify visible data values, we usually need functions of visible data values. Therefore, we have customized the definition. □

2.2.2 Hidden Signature and Hidden Algebra

Hidden Signature

Definition 35 (from [GM97]) *A* **hidden signature (over (V, Ψ, D))** *is a pair (H, Σ), where H is a set of* **hidden sorts** *disjoint from V, Σ is an $S = (H \cup V)$-sorted signature with $\Psi \subseteq \Sigma$ such that:*

1. *each $\sigma \in \Sigma_{w,s}$ with $w \in V^*$ and $s \in V$ lies in $\Psi_{w,s}$, and*
2. *for each $\sigma \in \Sigma_{w,s}$ at most one hidden sort occurs in w.*

We may abbreviate (H, Σ) to just Σ. If $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a **method** *if the sort of σ is this hidden sort, and an* **attribute** *if $s \in V$. If $w \in V^*$ and $s \in H$, then $\sigma \in \Sigma_{w,s}$ is called a* **hidden constant**. *We call methods and attributes* **behavioural operators**. □

From now on, we may call ordinary constants **visible constants**.

Remark 2 *The above definition of a method is slightly different from the original one [GM97]. The original one is as follows:*

if $w \in S^*$ contains a hidden sort, then $\sigma \in \Sigma_{w,s}$ is called a method if $s \in H$.

Firstly, we treat methods as operators which change states of a black box. Secondly, we distinguish methods from (pseudo-)projection operators which are introduced in Chapter 6. Each of these new operators contains a hidden sort s in it's arity and it's sort is a hidden sort s' where $s' \neq s$. Therefore, we have customized the definition. □

Hidden Algebra

Definition 36 (from [GM97]) *Given a hidden signature* Σ*, a* **hidden** Σ**-algebra** A *is a* Σ*-algebra* A *such that* $A\restriction_{\Psi} = T_{\Psi,E_{\Psi}}$*.* □

2.2.3 Context and Behavioural Satisfaction

Context

Definition 37 *Given a hidden signature* Σ *and a hidden sort* h*, then a* Σ**-context** *of sort* h *is a* Σ*-term having a single occurrence of a new variable symbol* $\Box_h$ *of sort* h*. We call* $\Box_h$ *a* **hole of sort** h*. A* Σ*-context* ct *is called a* **visible** Σ**-context** *if the sort of* ct *is a visible sort, and a* **hidden** Σ**-context** *if the sort of* ct *is a hidden sort. A* Σ*-context is* **appropriate** *for a term* t *iff the sort of* t *matches that of* $\Box_h$*. Write* $ct[t]$ *for the result of substituting* t *for* $\Box_h$ *in the context* ct*.* $(VisCt_{\Sigma})_{h,v}$ *denotes the set of all visible* Σ*-contexts whose sorts of the holes are* h *and whose sorts are* $v \in V$*.* □

Definition 38 (from [GM97]) *A* $\Psi(X)$*-term is* **local** *iff it is a constant or a variable (i.e., is in* D *or in* X*); a* $\Sigma(X)$*-term that is not a* $\Psi(X)$*-term is* **local** *iff all visible proper subterms are local.* □

Definition 39 *We call local* Σ*-contexts* **observational** Σ**-contexts**. □

Remark 3 *As we will discuss in Chapter 4, we regard observational* Σ*-contexts as observational tools of black boxes. So, we call these visible* Σ*-contexts observational* Σ*-contexts, instead of local* Σ*-contexts. On the other hand, we regard visible* Σ*-contexts without observational* Σ*-contexts as manipulation tools of observational values.* □

Property 2 *An observational* Σ*-context can be regarded as a sequence of behavioral operators.* [1] □

Definition 40 *Given an observational* Σ*-context* oc*, we regard* oc *as a sequence of behavioural operators. We call the length of this sequence the* **length of** oc*.* □

Definition 41 *We call hidden* Σ*-contexts which can be regarded as sequences of methods* **method** Σ**-contexts**. □

We generally omit the subscript Σ when there is no confusion.

[1] For example, $get(put(B,\Box))$ can be regarded as the sequence of prefix behavioural operators $get\ put(B)$ where get and put are behavioural operators. As to prefix operators, see Section 2.3.

Behavioural Satisfaction

Definition 42 (from [GM97]) *A hidden* Σ*-algebra* M **behaviourally satisfies** *a conditional* Σ*-equation* $(\forall X)t = t'$ *if* C *iff for any assignment* $as : X \to M$*, we have* $as(ct[t]) = as(ct[t'])$ *for each appropriate context* ct *whenever* $as(ct_j[t_i]) = as(ct_j[t'_i])$ *for each* $\langle t_i, t'_i \rangle$
$\in C$ *and each appropriate context* ct_j*. In this case we write:*
$M \models\!\!\equiv_\Sigma (\forall X)t = t'$ *if* C.
A Σ*-algebra* M **behaviourally satisfies** *a set* E *of conditional* Σ*-equations iff it satisfies each* $bceq \in E$*, and in this case we write:*
$M \models\!\!\equiv_\Sigma E$. $\Box$

We generally omit the subscript Σ when there is no confusion.

2.2.4 Behavioural Specification and Hidden Model

Behavioural Specification

Definition 43 (from [GM97]) *A* **behavioural specification** *is a triple* (H, Σ, E)*, where* (H, Σ) *is a hidden signature and* E *is a set of conditional* Σ*-equations that does not include any conditional* Ψ*-equations.* $\Box$

Hidden Model

Definition 44 (from [GM97]) *Given a specification* (H, Σ, E)*, a hidden* (Σ, E)**-model** M *is a hidden* Σ*-algebra such that:*
$M \models\!\!\equiv E$. $\Box$

Specification Equivalence

Definition 45 *Given hidden signature (over* (V, Ψ, D)*)* (H, Σ) *and* (H', Σ')*, a* **hidden signature morphism** $\varphi : \Sigma \to \Sigma'$ *is a signature morphism* $\varphi : \Sigma \to \Sigma'$ *such that:*

1. $\varphi(v) = v$ *for each* $v \in V$ *and* $\varphi(\sigma) = \sigma$ *for each* $\sigma \in \Psi$,
2. $\varphi(H) \subseteq H'$. $\Box$

Definition 46 *An* **interpretation** *of behavioural specifications (over* (V, Ψ, D)*)* $\varphi : (H, \Sigma, E) \to (H', \Sigma', E')$ *is a hidden signature morphism* $\varphi : \Sigma \to Der(\Sigma')$ *such that for each* Σ'*-algebra* M'*:*
$M' \models\!\!\equiv_{\Sigma'} E'$ *implies* $\varphi M' \models\!\!\equiv_\Sigma E$. $\Box$

Definition 47 *Two behavioural specifications* (H, Σ, E) *and* (H', Σ', E') *are* **specification equivalent** *iff there exists two interpretations* $\varphi : (H, \Sigma, E) \to (H', \Sigma', E')$ *and* $\psi : (H', \Sigma', E') \to (H, \Sigma, E)$ *such that:*

1. $\varphi(\psi(h)) = h$, *and*
   $\psi(\varphi(h')) = h'$

   for each $h \in H$ *and each* $h' \in H'$, *and*

2. $\varphi(\psi M) = M$, *and*
   $\psi(\varphi M') = M'$

   for each hidden (Σ, E)*-model* M *and each hidden* (Σ', E')*-model* M'. $\square$

2.2.5 Behavioural Equivalence

Definition 48 (from [GM97]) *Given a hidden signature* Σ, *a hidden subsignature* $\Phi \subseteq \Sigma$, *and a hidden* Σ*-algebra* M. *Then* **behavioural Φ-equivalence on** M, *denoted* $\equiv_\Phi$ *is defined as follows, for* $s, s' \in M_{st}$

$s(\equiv_\Phi)_{st} s'$ *iff* $s = s'$

when $st \in V$, *and*

$s(\equiv_\Phi)_{st} s'$ *iff* $M_{ct}(s) = M_{ct}(s')$ *for each* $ct \in (VisCt_\Phi)_{st,v}$ *and each* $v \in V$

when $st \in H$ *where* M_{ct} *denotes the function interpreting the context* ct *as an operation on* M. *When* $\Phi = \Sigma$, *we may call* $\equiv_\Phi$ *just* **behavioural equivalence** *and denote it* $\equiv$.
$\square$

2.2.6 Finality

Definition 49 *Given a hidden signature* Σ *without hidden constants and a behavioural specification* (H, Σ, E), *then* $F_{\Sigma,E}$ *denotes the following hidden* (Σ, E)*-model:*

1. *the carrier are given by the following formula:*

   $(F_{\Sigma,E})_h = \Pi_{v \in V}((VisCt_\Sigma)_{h,v} \to D_v)$,

   the product of the sets of functions taking contexts to data values (appropriate sort),

2. *an attribute* $\sigma \in \Sigma_{hw,v}$ *are interpreted as follows:*

   let $s \in (F_{\Sigma,E})_h$ *and let* $d \in D_v$; *then we define* $(F_{\Sigma,E})_\sigma = s_v(\sigma(\square_h, d))$; *i.e.,* s_v *is a function taking contexts in* $(VisCt_\Sigma)_{h,v}$ *to data values in* D_v, *so applying it to the context* $\sigma(\square_h, d)$ *gives the data value resulting from that observation,*

3. *a behavioural operator without attributes* $\sigma \in \Sigma_{hw,h'}$ *are interpreted as follows:*

   let $s \in (F_{\Sigma,E})_h$ *and let* $d \in D_w$; *For* $v \in V$ *and* $ct \in (VisCt_\Sigma)_{h',v}$, *we define*

   $((F_{\Sigma,E})_\sigma(s, d))_v(ct) = s_v(ct[\sigma(\square_h, d)])$;

   i.e.,with a slight abuse of notation, given an state s, *the result we get from looking at* $\sigma(s, d)$ *in a context* ct *it the same as the result that* s *gives in the context* $c[\sigma(\square_h, d)]$. $\square$

Remark 4 *In [GM97], they define F_Σ for a hidden signature Σ, instead of $F_{\Sigma,E}$. But, we need $F_{\Sigma,E}$ for a behavioural specification (H, Σ, E). The only difference between $F_{\Sigma,E}$ and F_Σ is that there are relations between observational results through visible Σ-contexts in $F_{\Sigma,E}$. The effect of this is only the restriction of the sets of carriers. So, in $F_{\Sigma,E}$, the same discussion of F_Σ holds. From now on, we use $F_{\Sigma,E}$ instead of F_Σ in references from [GM97].* □

Fact 5 (from [GM97]) *Given a hidden signature Σ without hidden constants and a behavioural specification (H, Σ, E). Then, $F_{\Sigma,E}$ is the* **final hidden** (Σ, E)**-model***, in the sense that for each hidden (Σ, E)-model M, there is a unique Σ-homomorphism $M \to F_{\Sigma,E}$.* □

Definition 50 (from [GM97]) *Given a hidden signature Σ and a behavioural specification (H, Σ, E), let $\Sigma^\diamond$ denote Σ with all hidden constants removed. Given a hidden (Σ, E)-model M, let $M^\diamond$ denote M viewed as a hidden $(\Sigma^\diamond, E)$-model.* □

Fact 6 (from [GM97]) *Two elements of a hidden (Σ, E)-model M are behaviourally equivalent iff they map to the same element under the unique $\Sigma^\diamond$-homomorphism $M^\diamond \to F_{\Sigma^\diamond,E}$ to the final hidden $(\Sigma^\diamond, E)$-model $F_{\Sigma^\diamond,E}$.* □

2.2.7 Behavioural Semantics

Definition 51 *Given a specification (H, Σ, E), we call semantics which is given by all hidden (Σ, E)-models* **behavioural semantics***.* □

Definition 52 *Given a behavioural specification (H, Σ, E). Let beq be a behavioural equation. Then*

$E \models\equiv beq$

iff for all hidden (Σ, E)-models M, $M \models\equiv beq$. □

Definition 53 *Given a behavioural specification (H, Σ, E). Let beq be a behavioural equation such that: $E \models\equiv beq$. We call beq* **behavioural properties of** (H, Σ, E)*.* □

2.3 The Specification Language CafeOBJ

In this paper, we specify specifications by using specification language CafeOBJ. Of course, our theory can be applied to other specifications written by other specification languages.

CafeOBJ [FS95, SF95, DF96, Fut97, DF98] is a multi-paradigm algebraic specification language which is a successor of OBJ[FGJM85, GWM+93]. CafeOBJ is based on the combination of several logics consisting of many sorted algebra, **order sorted algebra** [GD94, GM92], hidden algebra and **rewriting logic**[Mes92, Mes93]. This combination is handled by **institutions** [GB92, BD92].

We use only many sorted algebra and hidden algebra. In this section, we describe the syntax of CafeOBJ which we use later and examples written by CafeOBJ.

2.3.1 Loose Semantics

Example 1 *GROUP*
Let *GRP* be the following specification:

```
mod* GRP {
  [ Grp ]
  op e : -> Grp
  op inv_ : Grp -> Grp
  op _*_ : Grp Grp -> Grp

  vars X Y Z : Grp
  eq X * e = X .
  eq X * (inv X) = e .
  eq (X * Y) * Z = X * (Y * Z) .
}
```

GRP is a specification of all groups, for example, a module, a ring, and so on. It specify the common property of all groups. In CafeOBJ , specifications are divided into modules which are declared by `mod*` or `mod*` (see Example 2). `mod*` declares that the semantics of *GRP* module is the semantics given by many models. For *GRP* module, this means that the semantics of *GRP* module is loose semantics. As to another case, see Example 3. *Grp* which is surrounded by `[` and `]` is a visible sort, which denote data types. `op`, `eq`, and `vars` declare a operator, a equation, and variables, respectively. In CafeOBJ , we can use prefix operators (for example, `inv`) and mixfix operators (for example, `*`). □

We usually use prefix operators. We use prefix operators in the body of this paper, too. So, $(f\ \ g)[s]$ is equal to $f(g(s))$.

2.3.2 Initial Semantics

Example 2 *NAT*
Let *NAT* be the following specification:

```
mod! NAT {
  [ Nat ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
}
```

NAT is a specification of the Peano notation of natural numbers. `mod!` declares that the semantics of *GRP* module is initial semantics. □

2.3.3 Behavioural Semantics

Example 3 *SFLAG*
Let *DATA* be the following specification:

```
mod! DATA {
  [ Nat < Int ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op s_ : Int -> Int
  op p_ : Int -> Int
  op _+_ : Int Int -> Int

  [ DBool ]
  op t : -> DBool
  op f : -> DBool
  op not_ : DBool -> DBool

  vars I1 I2 : Int
  eq s p I1 = I1 .
  eq p s I1 = I1 .
  eq I1 + 0 = I1 .
  eq I1 + s I2 = s(I1 + I2) .
  eq I1 + p I2 = p(I1 + I2) .

  var B : DBool
  eq not t = f .
  eq not f = t .
  eq not not B = B .
}
```

DATA is a specification of natural numbers (*Nat* sort), integers (*Int* sort), and Bool values (*DBool* sort). `<` in `[ Nat < Int ]` declares that *Nat* sort is a part of *Int* sort. In CafeOBJ , we can use a partial ordering on a set of sorts. But, in this paper, we only use the ordering in *DATA* module. So, we do not describe about the ordering in detail. As to the ordering, see [GD94, GM92]. `var` declares a variable. Instead of `vars`, we can use `var`.

Let *SFLAG* be the following specification:

```
mod* SFLAG {
  pr(DATA)

  *[ Flag ]*
  bop up?_ : Flag -> DBool
  bop up_ : Flag -> Flag
  bop dn_ : Flag -> Flag

  var B : DBool
  var F : Flag
  eq up? up F = t .
  eq up? dn F = f .
```

}

SFLAG is a specification of a flag which can be either up or down. States of this flag can only be observed through the attribute *up?*. `pr(DATA)` declare that *DATA* module is imported to *SFLAG* without destroying the semantics. This means that $M\restriction_{DATA} = T_{DATA}$ for each hidden model M of *SFLAG* module where T_{DATA} denotes the term model of *DATA* module. Note that the semantics of *DATA* module is initial semantics. `mod*` and `pr(DATA)` declare that the semantics of *SFLAG* is behavioural semantics. `bop` declares a behavioural operator.

There are many models of *SFLAG* module. One of these is Boolean cell model C, and another is History model H. Here, $C_{Flag} = C_{DBool}$, $up\ F = t$, $dn\ F = f$, and $up?\ F = F$. On the other hand, $H_{Flag} = \{up, dn\}^*$. H_{Flag} constructed from complete histories of interactions, so that the action of a method is merely to concatenate its name to the front of a list of method names. Here, $up\ F = up^\frown F$, $dn\ F = dn^\frown F$, $up?\ up^\frown F = t$, and $up?\ dn^\frown F = f$ where $^\frown$ denotes the concatenation operation. Note that C and H are not isomorphic. □

Example 4 *HSS*
Let *HSS* be the following specification:

```
mod* HSS {
  pr(DATA)

  *[ Hss ]*
  bop get_ : Hss -> DBool
  bop put : DBool Hss -> Hss
  bop rest_ : Hss -> Hss

  var B : DBool
  var S : Hss
  eq get put(B, S) = B .
  beq rest put(B, S) = S .
}
```

HSS is a specification of a black box version of a stack. *get*, *put*, and *rest* correspond to *top*, *push*, and *pop*, respectively. `beq` declares a behavioural equation. □

2.4 Deduction

2.4.1 Equational Deduction

To prove properties, we use equational deduction.

Definition 54 (from [Gog]) *Given a specification* (Σ, E)*, the following* **rules of deduction** *define the* Σ*-equations eq that are* **deducible (from** E**)***:*

1. *Each equation of the form*

   $(\forall X)t = t$

   is deducible,

2. *If the equations*

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

   are deducible, then so is

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

   is also deducible,

3. *Given* $t_0 \in T_\Sigma(\{z\} \cup Y)$ *with exactly one occurrence of* z *and with* $z \notin Y$,

   $(\forall X)t_1 = t_2 \text{ if } C$

   is in E, *and given a substitution* $sb : X \to T_\Sigma(Y)$ *such that* $(\forall Y)sb(u) = sb(v)$ *is deducible for each pair* $\langle u, v\rangle \in C$, *then*

   $(\forall Y)t_0(z \leftarrow sb(t_1)) = t_0(z \leftarrow sb(t_2))$

   is deducible,

4. *Given* $t_0 \in T_\Sigma(\{z\} \cup Y)$ *with exactly one occurrence of* z *and with* $z \notin Y$,

   $(\forall X)t_2 = t_1 \text{ if } C$

   is in E, *and given a substitution* $sb : X \to T_\Sigma(Y)$ *such that* $(\forall Y)sb(u) = sb(v)$ *is deducible for each pair* $\langle u, v\rangle \in C$, *then*

   $(\forall Y)t_0(z \leftarrow sb(t_1)) = t_0(z \leftarrow sb(t_2))$

   is deducible,

We let $E \vdash eq$ *mean that* eq *is deducible from* E. $\square$

Fact 7 (from [Gog]) *Given a specification* (Σ, E) *and another* Σ*-equation* eq*, then* $E \vdash eq$ *iff* $E \models eq$. $\square$

2.4.2 Induction

To prove inductive properties, we use inductions. For example, structural induction, test set induction [Bou97], and so on. In this paper, we only use structural induction.

Structural Induction

Definition 55 *Let* t, t' *be* $\Sigma(\{x\})$*-terms. We call* $(\forall\{x\})t = t'$ *a* $\Sigma(\{x\})$-**sentence**.

Fact 8 (Structural Induction) (from [Gog]) *Given a specification* (Σ, E)*, let* $Q(x)$ *be a* $\Sigma(\{x\})$*-sentence where* x *is a variable of sort* s*. Then* $E \models\!\!\!\approx_\Sigma (\forall x)Q(x)$ *if*

1. $c \in \Sigma_{[],s}$ *implies* $E \models^{\cong}_{\Sigma} Q(c)$, *and*

2. $f \in \Sigma_{s_1 \cdots s_l, s}$ *for* $l > 0$ *and* $t_i \in (T_\Sigma)_{s_i}$ *for* $i = [1, \ldots, l]$ *and* $E \models^{\cong}_{\Sigma} Q(t_i)$ *when* $s_i = s$ *imply* $E \models^{\cong}_{\Sigma} Q(f(t_1, \ldots t_l))$. □

Definition 56 (from [Gog]) *Given a specification* (Σ, E), *let* $E \models^{\simeq}_{\Sigma} eq$ *mean that* eq *can be proved from* E *using the new rule given below plus the usual rules for* $\vdash_\Sigma$ *in Definition 54.*

Given $t, t' \in T_\Sigma(\{x\})$ *with* x *of sort* s, *if* $E \models^{\simeq}_{\Sigma} (\forall \emptyset) t(x \leftarrow c) = t'(x \leftarrow c)$ *for each* $c \in (T_\Sigma)_s$, *and if* $E \models^{\simeq}_{\Sigma} (\forall \emptyset) t(x \leftarrow t_i) = t'(x \leftarrow t_i)$ *for* $i = [1, \ldots l]$ *and* $f \in \Sigma_{s_1 \cdots s_l, s}$ *imply* $E \models^{\simeq}_{\Sigma} (\forall \emptyset) t(x \leftarrow f(t_1, \ldots, t_l)) = t'(x \leftarrow f(t_1, \ldots, t_l))$, *then* $E \models^{\simeq}_{\Sigma} (\forall x) t = t'$. □

Fact 9 (from [Gog]) *Given a specification* (Σ, E) *and another* Σ*-equation* eq, *then* $E \models^{\simeq} eq$ *implies* $E \models^{\cong} eq$. □

2.5 Abstract Reduction System

2.5.1 Abstract Reduction System

Definition 57 (from [Klo92]) *An* **abstract reduction system (ARS)** *is a structure* $\mathcal{A} = (A, (\rightarrow_\alpha)_{\alpha \in I})$ *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 58 (from [Klo92]) *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 59 (from [Klo92]) *Let* $\mathcal{A} = (A, (\rightarrow))$ *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 60 (from [Klo92]) *Let* $\mathcal{A} = (A, (\rightarrow))$ *be an ARS,* $\rightarrow$ *is* **terminating** *if every reduction sequence* $a_0 \rightarrow a_1 \rightarrow \cdots$ *eventually must terminate.* □

Definition 61 (from [Klo92]) *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. □

2.5.2 Term Rewriting System

Definition 62 (from [Gog]) *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 63 (from [Gog]) *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* $t_1 \to t_2$ *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 64 (from [Gog]) *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* $(\forall X)t_1 \to t_2$ *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* $sb : X \to T_\Sigma(Y)$ *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** $t_1 \to t_2$ *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)**)***.* □

2.6 The CafeOBJ verification system

There is the CafeOBJ verification system that executes specifications written by CafeOBJ by regarding (behavioural) equations as rewrite rules. In this section, we describe its function used later.

2.6.1 Reduce Command

In this paper, we assume that specifications are complete TRS. As to complete TRS, the following fact holds.

Definition 65 *We let* $==$ *denote syntactically identity.* □

Fact 10 (from [Gog]) *Given a complete* Σ*-TRS* (Σ, E)*, then*

$E \models (\forall X)t = t'$ *iff* $norm(t) == norm(t')$

where $norm(t)$ *is a normal form of* t*.* □

The CafeOBJ verification system supports *reduce* command (abbreviate *red* command) which calculates normal forms of inputs. By using *red* command, we prove properties.

Example 5 *GRP (continued)*
Consider to prove a property $e * e = e$ in GRP module.

Firstly, hit "cafeobj" from the current command line. The CafeOBJ verification system starts up.

```
[mitihiro@is27e0s04] 1 % cafeobj
-- loading standard prelude
Loading /cafe/cafeobj-1.4/prelude/std.bin
Finished loading /cafe/cafeobj-1.4/prelude/std.bin

           -- CafeOBJ system Version 1.4.0(Beta-5) --
                built: 1997 Dec 3 Wed 11:34:27 GMT
                      prelude file: std.bin
                               ***
                   1998 Feb 10 Tue 7:40:06 GMT
                         Type ? for help
                               ---
                   uses GCL (GNU Common Lisp)
            Licensed under GNU Public Library License
              Contains Enhancements by W. Schelter

CafeOBJ>
```

We assume that GRP module is written in *grp.mod*. Then, hit "in grp.mod".

```
CafeOBJ> in grp
-- processing input : ./grp.mod
-- defining module* GRP....._...* done.
GRP>
```

Now, the CafeOBJ verification systems become Σ_{GRP}-TRS.

Finally, hit "red $e * e == e$.".

```
GRP> red e * e == e .
-- reduce in GRP : e * e == e
true : Bool
(0.017 sec for parse, 2 rewrites(0.000 sec), 3 match attempts)
```

red command returns *true*. This mean that $norm(e * e) == norm(e)$. From Fact 10, $e * e = e$ is valid. □

2.6.2 Open and Close commands

When we prove some properties, we may want to extend a given specification. We can extend the specification by using open and close commands. We can add constants, variables, (behavioural) operators, and (behavioural) equations to the specification after we execute open command. This effect continues until we execute close command.

One case we want to extend the specification is rewriting of Σ-terms with variables. In Σ-TRS, Σ-terms rewrites to Σ-terms (Definition 64). To rewrite Σ-terms with variables, we need the following fact.

Fact 11 (Theorem of Constants) (from [Gog]) *Given a signature Σ, a ground signature X disjoint from Σ, a set E of Σ-equations, and $t, t' \in T_{\Sigma(X)}$, then*
$A \models_{\Sigma} (\forall X)t = t'$ *iff* $A \models_{\Sigma \cup X} (\forall \emptyset)t = t'$. □

Example 6 *NATP*
NATP module is the following module:

```
mod! NATP {
  [ 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) .
}
```

Consider to prove a property (?) $(X + Y) + Z = X + (Y + Z)$ in *NATP* module. We write proof commands in a file as follows.

```
--> (X + Y) + Z = X + (Y + Z) is not a property!
open .
ops l m n : -> Nat .

red l + (m + n) == (l + m) + n .
close
```

Note that we use constants l, m, n instead of variables and we extend the specification by adding the constants.

We assume that *NATP* module and the above proof commands are written in *natp.mod*. The execution result is as follows:

```
CafeOBJ> in natp
-- processing input : ./natp.mod
-- defining module! NATP......_..* done.
--> (X + Y) + Z = X + (Y + Z) is not a property!
-- opening module NATP.. done._*
-- reduce in % : l + (m + n) == (l + m) + n
false : Bool
(0.000 sec for parse, 1 rewrites(0.017 sec), 17 match attempts)
```

red command returns *false*. So, $(X + Y) + Z = X + (Y + Z)$ is not a property. □

2.6.3 Induction

To prove inductive properties, we may use structural induction.

Example 7 *NATP (continued)*
Consider to prove an inductive property $(X + Y) + Z = X + (Y + Z)$ in *NATP* module. Proof commands for proving the inductive property are as follows:

```
--> Prove (X + Y) + Z = X + (Y + Z)
--> Base Case)
open .
ops l m n : -> Nat .

red l + (m + 0) == (l + m) + 0 .
close
--> Induction Step)
open .
ops l m n : -> Nat .
-- induction hypothesis
eq l + (m + n) = (l + m) + n .

red l + (m + s n) == (l + m) + s n .
close
```

The execution result is as follows:

```
--> Prove (X + Y) + Z = X + (Y + Z)
--> Base Case)
-- opening module NATP.. done._*
-- reduce in % : l + (m + 0) == (l + m) + 0
true : Bool
(0.000 sec for parse, 3 rewrites(0.017 sec), 11 match attempts)
--> Induction Step)
-- opening module NATP.. done._*
-- reduce in % : l + (m + s n) == (l + m) + s n
true : Bool
(0.017 sec for parse, 5 rewrites(0.000 sec), 27 match attempts)
```

Each *red* command returns *true*. So, $(X + Y) + Z = X + (Y + Z)$ is an inductive property.
□

2.7 Induction over Length of Contexts

As we will discuss in Chapter 4, the following relation holds:
$(s \equiv_h s') = \wedge_{ct \in ObsCt_h}(ct[s] == ct[s'])$

where h is a hidden sort, s, s' are states of a black box (sort h), and $ObsCt_h$ is the set of all observational contexts of sort h.
Therefore, by using induction over length of observational contexts, we can prove behavioural properties.

Example 8 *HSS (continued)*
Consider to prove $(rest\ put(t, S)) \equiv (S)$ in *HSS*.
The process of induction over length of contexts is as follows:

```
--> Prove (rest put(t, S)) Reqv (S) .
--> Base Case)
open .
red get rest put(t, S) == get S .
close

--> Induction Step)
open .
bop c_ : Hss -> DBool .
eq c rest put(t, S) = c S .
red c rest rest put(t, S) == c rest S .
red c put(B, rest put(t, S)) == c put(B, S) .
close
```

The result is as follows:

```
--> Prove (rest put(t, S)) Reqv (S) .
-->  Base Case)
-- opening module HSS.. done.
-- reduce in % : get (rest put(t,S)) == get S
true : Bool
(0.017 sec for parse, 2 rewrites(0.000 sec), 4 match attempts)
-->  Induction Step)
-- opening module HSS.. done._*
-- reduce in % : c (rest (rest put(t,S))) == c (rest S)
true : Bool
(0.017 sec for parse, 2 rewrites(0.000 sec), 10 match attempts)
-- reduce in % : c put(B,rest put(t,S)) == c put(B,S)
true : Bool
(0.017 sec for parse, 2 rewrites(0.000 sec), 6 match attempts)
```

Because all executions of *red* commands return *true*, $(rest\ put(t, S)) \equiv (S)$ is valid. □

Because induction over length of contexts is usually inefficient, coinduction in the next chapter is usually used to prove behavioural properties.

2.8 Coinduction

Before we can give the algorithm of coinduction, we need the definition of hidden congruence.

Definition 66 (from [GM97]) *Given a hidden signature Σ, a hidden subsignature $\Phi \subseteq \Sigma$, and a hidden Σ-algebra M, a* **hidden Φ-congruence** $\simeq$ *on M is a Φ-congruence $\simeq$ which is the identity on visible sorts, i.e., such that $e \simeq_v e'$ iff $e = e'$ for each $v \in V$ and each $e, e' \in D_v$. We may call a hidden Σ-congruence just a* **hidden congruence**. □

Coinduction is a verification method based on the following fact:

Fact 12 (from [GM97]) *Let (H, Σ, E) be a behavioural specification and M is a hidden Σ-algebra, then behavioural Σ-equivalence $\equiv_\Sigma$ is the largest hidden Σ-congruence on M.* □

Algorithm 1 *Consider a verification of a behavioural property $s \equiv s'$.* **The algorithm of coinduction** *is as follows:*

1. *find a candidate R of hidden congruences,*
2. *check whether R is a hidden congruence, and*
3. *verify whether $s \equiv s'$ holds, by proving $s\ R\ s'$.* □

Example 9 *Flag*
Let $FLAG$ be the following specification:

```
mod* FLAG {
  pr(DATA)

  *[ Flag ]*
  bop up?_ : Flag -> DBool
  bop up_ : Flag -> Flag
  bop dn_ : Flag -> Flag
  bop rev_ : Flag -> Flag

  var B : DBool
  var F : Flag
  eq up? up F = t .
  eq up? dn F = f .
  eq up? rev F = not up? F .
}
```

When the CafeOBJ verification system read a specification, it automatically generates a candidate `=*=` (1) and checks whether `=*=` is a hidden congruence (2). `=*=` is a H-sorted relation such that $(s \texttt{=*=}_h s') = \wedge_{at \in Attr_h}(at[s] == at[s'])$ where $Attr_h$ is the set of all attributes of sort h for each hidden sort h. The above process is as follows:

```
CafeOBJ> in flag
-- processing input : ./flag.mod
-- reading in file  : data
-- processing input : data.mod
-- defining module! DATA........_......._...* done.
-- done reading in file: data
-- defining module* FLAG......._...*
** system already proved =*= is a congruence of FLAG done.
```

The last line shows that =*= is a hidden congruence.

Consider to prove $(rev\ up\ F) \equiv (dn\ F)$ in *FLAG*.

The process of (3) of coinduction is as follows:

```
open .
red rev up F =*= dn F .
close
```

The result is as follows:

```
--> Prove (rev up F) Reqv (dn F) .
-- opening module FLAG.. done.
-- reduce in % : rev (up F) =*= dn F
true : Bool
(0.017 sec for parse, 6 rewrites(0.000 sec), 9 match attempts)
```

Because an execution of *red* command returns *true*, $(rev\ up\ F)\ \equiv\ (dn\ F)$ is valid. □

Unfortunately, in *HSS*, =*= is not a hidden congruence. Therefore, users must find a hidden congruence.

Chapter 3

Syntactically Definable Hidden Congruence

We check 2 and 3 in Algorithm 1 by using verification systems. Note that relations which can be defined on verification systems are not only relations on hidden algebras but also relations defined by syntax.

Definition 67 *Let* (H, Σ, E) *be a behaivoural specification over* (V, Ψ, D). **Syntactically definable hidden Σ-congruences** *are hidden* Σ*-congruences which can only be defined by operators and behavioural operators in* Ψ *and* Σ. $\Box$

Before we can prove the property of syntactically definable hidden Σ-congruences, we need the following property of the final hidden (Σ, E)-model.

Definition 68 *Given a behavioural specification* (H, Σ, E). *Let* M *be a* (Σ, E)*-model. Let* SS *be a* $(V \cup H)$*-sorted set such that:*

1. $SS_{vh} = M_{vh}$ *when* $vh \in V$ *and*
2. $SS_{vh} = \{s \in M_{vh} \mid \forall\, h \in H\ .\ (\forall\, bop \in Bop_{vh,h}\ .\ bop[s] \in M_h)\}$ *when* $vh \in H$ *where* $Bop_{vh,h}$ *is the set of all behavioural operators from* vh *to* h *(if we ignore visible sorted arguments).*

We let $R_{SS} = \{(s, s) \mid s \in SS\}$. *Then, we call* R_{SS} *a* **partial identity relation**. *Especially, we call* R_M *the* **identity relation** *and we call* $R_\emptyset$ *the* **empty relation**. $\Box$

Property 3 *Given a behavioural specification* (H, Σ, E). *Hidden* Σ*-congruences on the final hidden* $(\Sigma^\diamond, E)$*-model* $F_{\Sigma^\diamond,E}$ *are partial identity relations.*

Proof : Given states $s, s' \in h$ $(h \in H)$ such that: $s \equiv s'$. From Fact 6, there is the unique $\Sigma^\diamond$-homomorphism $\varphi : F_{\Sigma^\diamond,E} \to F_{\Sigma^\diamond,E}$ such that: $\varphi(s) = \varphi(s')$. Because $F_{\Sigma^\diamond,E}$ is the final hidden $(\Sigma^\diamond, E)$-model, φ is the identity map. So $s = s'$. Therefore, behavioural Σ-equivalence coincides with the identity relation. On the other hand, from Definition 68, every partial identity relation is a hidden Σ-congruence. From Fact 12, hidden Σ-congruences on $F_{\Sigma^\diamond,E}$ are partial identity relations. $\Box$
Then, as to syntactically definable hidden Σ-congruences, the following theorem holds.

Theorem 4 *Given a behavioural specification* (H, Σ, E). *Syntactically definable hidden* Σ*-congruences without case analyses are behavioural* Σ*-equivalence, the identity relation, and the empty relation.*

Proof : Syntactically definable hidden Σ-congruences are defined on all hidden (Σ, E)-models. So, these are defined on the final hidden $(\Sigma^\diamond, E)$-model. But, on the final hidden $(\Sigma^\diamond, E)$-model, hidden Σ-congruences are behavioural Σ-equivalence and partial identity relations. Partial identity relations defined by syntax without case analysis are the identity relation and the empty relation. Consequently, syntactically definable hidden Σ-congruences are behavioural Σ-equivalence, the identity relation, and the empty relation. $\Box$

Corollary 5 *Let* R_{ID} *be the identity relation and let* $R_\emptyset$ *be the empty relation. Syntactically definable hidden* Σ*-congruences* R *are categorized as follows:*

1. $R \;=\; \equiv_\Sigma$,
2. $R \;=\; R_{ID}$,
3. $R \;=\; R_\emptyset$,
4. *let* $cond_1$, $cond_2$, *and* $cond_3$ *be conditions that:* $cond_1 \cup cond_2 \cup cond_3 = true$ *and* $cond_i \cap cond_j = \emptyset$ $(i \neq j)$, *then*

$$\begin{array}{lll} R = \equiv_\Sigma & \textit{if } cond_1, \\ R = R_{ID} & \textit{if } cond_2, \textit{ and} \\ R = R_\emptyset & \textit{if } cond_3. \; \Box \end{array}$$

From this fact, the only useful syntactically definable hidden congruence for verifications is behavioural equivalence. Behavioural equivalence is the conjunction over all visible contexts. Consequently, a selection of hidden congruences corresponds to a selection of the set of visible contexts which construct behavioural equivalence. We let R denote the form of behavioural equivalence defined by syntax — conjunction over visible contexts — and we let $\#(R)$ denote the numbers of these visible contexts. We regard a verification method with R as an **efficient method** if $\#(R)$ is small. We regard R as a **simple form** if $\#(R)$ is small. So, to verify behavioural properties efficiently, we need a simple form of behavioural equivalence. By eliminating redundant visible contexts, we get this simple form. The method which generates this simple form is *GSB*-algorithm of test set coinduction in the next chapter. As we will discuss in the next chapter, for proving $(rest\ put(t, S)) \equiv (S)$ in *HSS*, test set coinduction is more efficient than induction over length of contexts.

Chapter 4

Test Set Coinduction

In this chapter, we only treat specifications which have exactly one hidden sort. For many hidden sorted cases, we will discuss in Chapter 6.

4.1 Test Set Coinduction

Algorithm 2 *Consider a verification of a behavioural property* $s \equiv s'$*.* **The algorithm of test set coinduction** *is as follows:*

1. *generate a simple form of behavioural equivalence (by GSB-algorithm), and*
2. *verify whether* $s \equiv s'$ *holds, by proving* $s \ R \ s'$*.* □

4.1.1 Cap Elimination

Definition 69 *Let vc be a visible context and oc be an observational context which is a subterm of vc. If* $vc = cp\ oc$*, then we call cp* **cap***. We call the process which gets oc from vc* **cap elimination***.* □

Property 6 *Let vc be a visible context, cp be a cap, oc be a observational context, and* $vc = cp\ oc$*. Then given states* s*,* s'*,*
$(vc[s] == vc[s']) \wedge (oc[s] == oc[s']) = (oc[s] == oc[s'])$*.* □

We can eliminate visible contexts which have caps. Therefore, the following property holds.

Property 7 *Given states* s*,* s'*,*
$(s \ \equiv \ s') \ = \ \bigwedge_{ct \in ObsCt}(ct[s] == ct[s'])$
where ObsCt denotes the set of all observational contexts. □

From now on, we consider elimination of redundant observational contexts from the set of observational contexts.

4.1.2 Context Rewriting System

Observational contexts are one part of terms. So, context rewriting systems can be generated from specifications like term rewriting systems. But, restrictions and changes are necessary to ensure rewrite rules between observational contexts. With these changes, we apply following index elimination to both sides of (behavioural) equations in the specifications.

Definition 70 *Let behop be a behavioural operator and* $v_1 \cdots v_l\ h$ *be an arity of it, where* $v_1, \cdots, v_l$ *are visible sorts and* h *is a hidden sort. We call* $(v_1, \cdots, v_l)$ **index**. *We call values of index* **index values**. *Let* $idel(behop)$ *be behop except an arity* h. *We call the transformation from behop to* $idel(behop)$ **index elimination**. *If there is no confusion, we use behop instead of* $idel(behop)$. *Next, let vc be a visible context,* $bop_1, \cdots, bop_l$ *be behavioural operators,* $vc = bop_1\ \cdots\ bop_l$*, and* $i_1, \cdots, i_l$ *be their indexes. We call* $(i_1, \cdots, i_l)$ **index of** *vc. We define* $idel(vc)$ *as* $idel(bop_1)\ \cdots\ idel(bop_l)$. □

As we will discuss in the latter part of this chapter, the form generated by *GSB*-algorithm is the conjunction over behavioural operators for all index values. This comes from the process of *GSB*-algorithm that every behavioural operator is decided whether it can eliminate. So, information about indexes is redundant. From this fact, we deal with behavioural operators to whose index elimination was applied.

Definition 71 *A* **context rewriting system (CRS)** *is generated from* (H, Σ, E) *by following processes:*

1. *select (behavioural) equations eq which satisfy following conditions from E:*
   (a) *all visible sorted arguments in the left hand side of eq are variables,* [1]
   (b) *if eq is a equation, the left hand side of eq is an observational context,* [2]
   (c) *if eq is a behavioural equation, there exists exactly one hidden sorted variable in the both sides of eq, and*
   (d) *if eq is a equation, there exists exactly one hidden sorted variable in the right hand side of eq,* [3]
2. *apply index elimination to each eq,*
3. *apply cap elimination to the right hand side of each equation, and*
4. *regard each eq as a* Σ*-rewrite rule.*

We call rewrite rules generated from equations **visible context rewrite rules***, and rewrite rules generated from behavioural equations* **hidden context rewrite rules***. We call visible context rewrite rules and hidden context rewrite rules* **context rewrite rules***. If every eq in E satisfies the conditions (a) to (d), we say that this CRS is* **completely generated**. □

[1] This condition is necessary to apply index elimination to *eq*.

[2] We regard a hidden sorted variable as a hole.

[3] This condition is necessary to uniquely determine the result of cap elimination.

In order to denote terms which are results of applying cap elimination to visible constants, and variables which are right hand sides of behavioural equations, we introduce following notations.

Definition 72 Constant observational context φ *is an observational context such that:*
$\forall\, mt : method\ .\ (\varphi\ mt = \varphi)$.
Unit context ψ *is a context such that:*
$\forall\, ab : attribute\ .\ (ab\ \psi = ab)$ *and* $\forall\, mt : method\ .\ (mt\ \psi = mt$ *and* $\psi\ mt = mt)$. □

Note that we can regard each visible constant vc of sort v as a function of rank $\langle h, v\rangle$ which returns vc for each state s of sort h. So, the next property holds.

Property 8 $\forall\, vc : visible\ constant\ .\ (vc\ \varphi = vc)$.
Therefore, the result of applying cap elimination to every visible constant is φ. □

Example 10 *HSS (continued)*
The CRS is generated from the specification *HSS* by following processes:

1. all (behavioural) equations of E satisfy the conditions (a) to (d).

   ```
   eq get put(B, S) = B .    beq rest put(B, S) = S .
   ```

2. by applying index elimination to both sides of each (behavioural) equation, we got following relations:

   $(get\ put(S),\ B)\qquad (rest\ put(S),\ S)$

3. by applying cap elimination to the right hand side of the left relation, we got following relations:

   $(get\ put(S),\ \varphi(S))\qquad (rest\ put(S),\ S)$

4. by recognizing both relations as context rewrite rules, we got following context rewrite rules:

   $get\ put \rightarrow \varphi\qquad rest\ put \rightarrow \psi.$

Because context rewrite rules are rewrite rules between contexts, we use ψ instead of S. From the process 1, the CRS generated from *HSS* is completely generated. □

Example 11 *EXP*
Let *EXP* be the following specification:

```
mod* EXP {
  pr(DATA)

  *[ Exp ]*
  bop a1_ : Exp -> DBool
  bop a2_ : Exp -> DBool
```

```
  bop a3_ : Exp -> DBool
  bop m1_ : Exp -> Exp
  bop m2_ : Exp -> Exp
  bop m3_ : Exp -> Exp

  var S : Exp
  eq a1 m1 m1 S = t .
  eq a1 m2 S = f .
  eq a2 m1 S = t .
  eq a2 m2 S = f .
  eq a3 S = a1 S .
  beq m2 m1 S = m2 S .
  beq m3 S = m1 S .
}
```

The CRS is generated from the specification *EXP* by following processes:

1. all (behavioural) equations of E satisfy the conditions (a) to (d).

   ```
   eq a1 m1 m1 S = t .    eq a1 m2 S = f .
   eq a2 m1 S = t .    eq a2 m2 S = f .
   eq a3 S = a1 S .
   beq m2 m1 S = m2 S .    beq m3 S = m1 S .
   ```

2. by applying index elimination to both sides of each (behavioural) equation, we got following relations:

   $(a1\ m1\ m1\ S,\ t)$ $\quad$ $(a1\ m2\ S,\ f)$
   $(a2\ m1\ S,\ t)$ $\quad$ $(a2\ m2\ S,\ f)$
   $(a3\ S,\ a1\ S)$
   $(m2\ m1\ S,\ m2\ S)$ $\quad$ $(m3\ S,\ m1\ S)$

3. by applying cap elimination to the right hand side of the left relation, we got following relations:

   $(a1\ m1\ m1\ S,\ \varphi\ S)$ $\quad$ $(a1\ m2\ S,\ \varphi\ S)$
   $(a2\ m1\ S,\ \varphi\ S)$ $\quad$ $(a2\ m2\ S,\ \varphi\ S)$
   $(a3\ S,\ a1\ S)$
   $(m2\ m1\ S,\ m2\ S)$ $\quad$ $(m3\ S,\ m1\ S)$

4. by recognizing both relations as context rewrite rules, we got following context rewrite rules:

   $a1\ m1\ m1 \to \varphi$ $\quad$ $a1\ m2 \to \varphi$
   $a2\ m1 \to \varphi$ $\quad$ $a2\ m2 \to \varphi$
   $a3 \to a1$
   $m2\ m1 \to m2$ $\quad$ $m3 \to m1$

From the process 1, the CRS generated from *EXP* is completely generated. □

Property 9 *Let c_{iv}, $c'_{iv'}$ be observational contexts, id, id' be these index values (i, i' be these indexes), and $idel(c_{iv}) \rightarrow^* idel(c'_{iv'})$ by a CRS. Then given states s, s',*
$(\bigwedge_{i \in PossId}(c_i[s] == c_i[s'])) \wedge (\bigwedge_{i' \in PossId'}(c'_{i'}[s] == c'_{i'}[s']))$
$= \bigwedge_{i' \in PossId'}(c'_{i'}[s] == c'_{i'}[s'])$
where PossId (PossId') denotes the set of all index values of i (i'), respectively. □

From this fact, if a CRS is complete, then the following theorem holds.

Theorem 10 *let a CRS be complete. Then,*
$(s \ \equiv \ s') \ = \ \bigwedge_{ct \in NormCt}(ct[s] == ct[s'])$
where NormCt denotes the set of all normal forms of observational contexts without φ. □

Consequently, if a CRS is complete and completely generated, by finding the set of all normal forms of observational contexts without φ, we can get the simplest form of behavioural equivalence. From the next subsection, we describe the method how to eliminate redundant contexts — which are not normal forms — efficiently and a sufficient condition to get the set of all normal forms without φ by this method.

4.1.3 Cover Set

Definition 73 *Given a behavioural specification, let $vf_1, \cdots, vf_l$ be observational contexts, $hf_{i,1}, \cdots, hf_{i,l_i}$ be method contexts, and Ct_i be the set of all concatenation of a sequence of $\{hf_{i,1}, \cdots, hf_{i,l_i}\}^*$ [4] after vf_i, like $vf_i\ hf_{i,1}$ and $vf_i\ hf_{i,3}\ hf_{i,2}$. If $(s \ \equiv \ s') \ = \ \wedge_{i \in [1,\ldots,l]}(\bigwedge_{ct \in Ct_i}$ $(ct[s] == ct[s']))$ holds, we call $\{(vf_i, \{hf_{i,1}, \cdots, hf_{i,l_i}\})\}_{i \in [1,\ldots,l]}$ a* **cover set**, *$vf_1, \cdots, vf_l$* **visible fragments**, *and $hf_{i,1}, \cdots, hf_{i,l_i}$* **hidden fragments assigned to** *vf_i* □

Example 12 *HSS (continued)*
$\{(get, \{put, rest\})\}$ is a cover set of the specification *HSS*. □

Example 13 *EXP (continued)*
$\{(a1, \{m1, m2, m3\}), (a2, \{m1, m2, m3\}), (a3, \{m1, m2, m3\})\}$ is a cover set of the specification *EXP*. □

4.1.4 Test Set

In *GSB*-algorithm, we reduce compositions of a visible fragment and a hidden fragment, or two hidden fragments by context rewrite rules. If the maximal length of these compositions coincides with the maximal length of left hand sides of context rewrite rules, all context rewrite rules have possibilities that they may be matched to these compositions. [5] A test set is a cover set which satisfies this condition.

[4] $\{hf_{i,1}, \cdots, hf_{i,l_i}\}^*$ shows the set of all sequences of elements of $\{hf_{i,1}, \cdots, hf_{i,l_i}\}$.

[5] As we will discuss in the next subsection, there exists context rewrite rules which are not used in *GSB*-algorithm by the property of *GSB*-algorithm.

Definition 74 *Given a behavioural specification, let mlv be the maximum of the length of the left hand side of visible context rewrite rules, and let mlh be the maximum of the length of the left hand side of hidden context rewrite rules. let lhf be the maximum of* 1 *and* $[mlh/2]$. [6] *let lvf be the maximum of* 1, *lhf, and* $(mlv - lhf)$. *A* **test set** *is a cover set such that:*

1. *visible fragments are all combinations of an attribute and methods whose lengths are equal or less than lvf,*

2. *if the length of a visible fragment vf is less than lvf, there is no hidden fragments assigned to vf, and*

3. *if the length of a visible fragment vf is equal to lvf, hidden fragments assigned to vf are all combinations of methods whose lengths are equal to lhf.* □

Example 14 *HSS (continued)*
In the specification *HSS*, $(mlv = 2)$, $(mlh = 2)$, $(lhf = 1)$, and $(lvf = 1)$. Consequently, the test set is $\{(get, \{put, rest\})\}$. □

Example 15 *EXP (continued)*
In the specification *EXP*, $(mlv = 3)$, $(mlh = 2)$, $(lhf = 1)$, and $(lvf = 2)$. Consequently, the test set is $\{(a1, \emptyset), (a2, \emptyset), (a3, \emptyset), (a1\ m1, \{m1, m2, m3\}), (a1\ m2, \{m1, m2, m3\}),$
$(a1\ m3, \{m1, m2, m3\}), (a2\ m1, \{m1, m2, m3\}), (a2\ m2, \{m1, m2, m3\}),$
$(a2\ m3, \{m1, m2, m3\}), (a3\ m1, \{m1, m2, m3\}), (a3\ m2, \{m1, m2, m3\}),$
$(a3\ m3, \{m1, m2, m3\})\}$. □

4.1.5 *GSB*-algorithm

In this subsection, we describe how to generate a cover set by eliminating redundant visible and hidden fragments from a test set, and how to generate a simple form of behavioural equivalence from this cover set. vf and vf' denote visible fragments and hf, hf', and hf'' denote hidden fragments. We assume that:

Assumption 1

1. *a CRS is complete, and*

2. *for each context rewrite rule,*
   (*the length of left hand side*) $\geq$ (*the length of right hand side*). □

Moreover, we assume that index elimination was applied to each behavioural operators. Firstly, we introduce ideas which are necessary to describe *GSB*-algorithm. [7]

[6] [ ] is a function for raising to a unit. For example, $[2.5] = 3$.
[7] *GSB* is an abbreviation for "Generate a Simple form of Behavioural equivalence".

Definition 75 *If the set generated by eliminating a visible fragment and all hidden fragments assigned to it from a test set is a cover set, we call this visible fragment an* **eliminable visible fragment**. *Also, if the set generated by eliminating a hidden fragment assigned to vf is a cover set, we call this hidden fragment an* **eliminable hidden fragment assigned to** vf. *Moreover, we call eliminable visible fragments and eliminable hidden fragments* **eliminable fragments**. □

Property 11 *If one of following context rewrite rules matches to a visible fragment vf (or a hidden fragment hf), this fragment is an eliminable fragment.*
$vf \to \varphi,\ \ vf \to vf'\ (vf' \neq vf),\ \ hf \to \psi$, *and* $hf \to hf'\ (hf' \neq hf)$
We call these context rewrite rules **elimination rules**.

Proof : From Property 9. □

Definition 76 *We call eliminations of vf or hf in Property 11* **visible fragment elimination** *or* **hidden fragment elimination** *respectively.* □

Definition 77 *Let hf be a hidden fragment assigned to vf. We call the process which calculates the normal form of $vf\ hf$* **visible fragment application to** hf. □

Definition 78 *Let wl and hf be hidden fragments assigned to vf. We call the process which calculates the normal form of $wl\ hf$* **wall application to** hf **by** wl. □

Definition 79 *Let wl be a hidden fragment assigned to vf. A* **wall assigned to** vf *is defined as follows:*

1. *if the result of visible fragment application to wl is not φ, vf', or $vf'\ hf'(hf' \neq wl)$, then wl is a wall, and*

2. *if the result of wall application to wl by a wall wl' is not ψ, hf', or $hf'\ hf''\ (hf'' \neq wl)$, then wl is a wall.* □

Input of *GSB*-algorithm is a test set. Firstly, we eliminate apparent redundant visible and hidden fragments like visible fragments which change only names from other visible fragments (visible and hidden fragment elimination). Then, we divide hidden fragments between eliminable fragments and walls, for every visible fragments. The process of this division is as follows:

Let $vf\ hf_1\ \cdots\ hf_l$ be an observational context which starts from vf. We check whether the i-th hidden fragment hf can be eliminated from all observational contexts whose i-th hidden fragments are hf, inductively. This "eliminate" means that there exists a context rewrite rule such that i-th hidden fragment of the rewrite result of $vf\ hf_1\ \cdots\ hf_l(hf_i = hf)$ by it is not hf. If we can eliminate hf of the i-th hidden fragment for every i, then hf is an eliminable fragment. If not, hf is a wall. The check of base case ($i = 1$) is visible fragment application and it of inductive step is wall application. The reason of the latter is as follows:

Let $ct\ hf$ be an observational context whose $i+1$-th hidden fragment is hf. From 1 of Assumption 1, there exists ct' which is generated by a visible fragment and walls, and $ct \to^* ct'$. From 2 of Assumption 1, the length of ct' is equal or less than the length of ct. Consequently, we can assume that the i-th hidden fragment is a wall without loss of generality.

If the result of visible fragment application to hf is φ, vf', or $vf'\ hf'(hf' \neq hf)$, then hf is a candidate of eliminable fragment. If not, hf is a wall. If the result of wall application to a candidate of eliminable fragment hf by every wall is ψ, hf', or $hf'\ hf''(hf'' \neq hf)$, then hf is an eliminable fragment. If not, hf is a wall.

From Property 9, the set generated by eliminating these eliminable fragments from a test set is a cover set. Finally, we generate a simple form of behavioural equivalence from this cover set. This simple form is output of *GSB*-algorithm.

The algorithm of *GSB*-algorithm is as follows:

Algorithm 3 (***GSB*-algorithm**)

1. *generate a cover set from a test set by applying visible and hidden fragment elimination,*

2. *apply visible fragment application to every hidden fragment assigned to vf for every visible fragment vf, then, divide hidden fragments between walls and candidates of eliminable fragments, for every visible fragment vf,*

3. *apply wall application to every candidate by every (new) wall,*

4. *if new walls occur in* 3*, turn back to* 3 *and if not, remaining candidates are eliminable fragments, and*

5. *generate a cover set by eliminating these eliminable fragments from a cover set of* 1*, then, we generate a simple form of behavioural equivalence from this cover set.* □

By summarizing the above argument, we get the following theorem.

Theorem 12 *If a CRS satisfy following conditions, then we can get a simple form of behavioural equivalence by GSB-algorithm:*

1. *a CRS is complete, and*

2. *for each context rewrite rule,*
   (*the length of left hand side*) $\geq$ (*the length of right hand side*). □

Example 16 *HSS (continued)*
The CRS generated from the specification *HSS* is
$get\ put \to \varphi \qquad rest\ put \to \psi$.
The test set is $\{(get, \{put, rest\})\}$.

The process of *GSB*-algorithm is as follows:

1. There is no visible or hidden fragments which can be eliminated by visible or hidden fragment elimination.

2. $get\ put \to \varphi \qquad get\ rest \not\to$
   Therefore, *rest* is a wall and *put* is a candidate of eliminable fragment.

3. $rest\ put \to \psi$.

4. No new wall occurs in 3. Therefore, *put* is an eliminable fragment.

5. The generated cover set is $\{(get, \{rest\})\}$. Therefore, a simple form of behavioural equivalence is $\bigwedge_{i \in Nat}(get\ rest^{(i)}[s] == get\ rest^{(i)}[s'])$.

□

Example 17 *EXP (continued)*
The CRS generated from the specification *EXP* is
$a1\ m1\ m1 \to \varphi \qquad a1\ m2 \to \varphi$
$a2\ m1 \to \varphi \qquad a2\ m2 \to \varphi$
$a3 \to a1$
$m2\ m1 \to m2 \qquad m3 \to m1$
The test set is $\{(a1, \emptyset), (a2, \emptyset), (a3, \emptyset), (a1\ m1, \{m1, m2, m3\}), (a1\ m2, \{m1, m2, m3\}),$
$(a1\ m3, \{m1, m2, m3\}), (a2\ m1, \{m1, m2, m3\}), (a2\ m2, \{m1, m2, m3\}),$
$(a2\ m3, \{m1, m2, m3\}), (a3\ m1, \{m1, m2, m3\}), (a3\ m2, \{m1, m2, m3\}),$
$(a3\ m3, \{m1, m2, m3\})\}$.
The process of *GSB*-algorithm is as follows:

1. $a3 \to a1 \qquad a1\ m2 \to \varphi \qquad a2\ m1 \to \varphi \qquad a2\ m2 \to \varphi \qquad m3 \to m1$
   Therefore, $a3$, $a1\ m2$, $a2\ m1$, $a2\ m2$, $m3$ are eliminable fragments. Consequently, the generated cover set is $\{(a1, \emptyset), (a2, \emptyset), (a1\ m1, \{m1, m2\})\}$.

2. $a1\ m1\ m1 \to \varphi \qquad a1\ m1\ m2 \not\to$
   Therefore, $m2$ is a wall and $m1$ is a candidate of eliminable fragment.

3. $m2\ m1 \to m2$.

4. No new wall occurs in 3. Therefore, $m1$ is an eliminable fragment.

5. The generated cover set is $\{(a1, \emptyset), (a2, \emptyset), (a1\ m1, \{m2\})\}$. Therefore, a simple form of behavioural equivalence is $(a1[s] == a1[s']) \wedge (a2[s] == a2[s']) \wedge (\bigwedge_{i \in Nat}(a1\ m1\ m2^{(i)}[s] == a1\ m1\ m2^{(i)}[s']))$.

□

GSB-algorithm only checks whether behavioural operators are walls or eliminable fragments. Therefore, there may exist the simplest form which is different from the form generated by *GSB*-algorithm. Then, the next problem is what is a sufficient condition that the form generated by *GSB*-algorithm coincides with the simplest form of behavioural equivalence.

In order to make this problem easier, we only deal with the case that lengths of left hand sides of context rewrite rules are 1 or 2.

Theorem 13 *Let a CRS be complete, completely generated, and for each context rewrite rule,* (*the length of left hand side*) $\geq$ (*the length of right hand side*). *If this CRS does not include a context rewrite rule whose left hand side is only constructed by wall, then the form generated by GSB-algorithm coincides with the simplest form of behavioural equivalence.*

Proof : Context rewrite rules which are not used in *GSB*-algorithm are only context rewrite rules whose left hand sides are only constructed by walls, like $wl\ wl \rightarrow \psi$ where wl is a wall. Consequently, if there are not these kind of context rewrite rules in a CRS, all context rewrite rules are used in *GSB*-algorithm. From this fact, the form generated from this CRS by *GSB*-algorithm coincides with the simplest form of behavioural equivalence. □

Corollary 14 *Let a CRS be complete and completely generated. If this CRS satisfy the following condition and every visible fragment in a cover set generated by GSB-algorithm has at most one wall for each visible fragment, then the form generated by GSB-algorithm coincides with the simplest form of behavioural equivalence.*

Let ab_1, ab_2 *be attributes and* mt_1, mt_2 ($mt_2 \neq mt_1$), mt_3, mt_4 ($mt_4 \neq mt_2$) *be methods. All context rewrite rules of CRS coincides with one of the following rules:*

$$ab_1 \rightarrow \varphi,\ \ ab_1 \rightarrow ab_2,\ \ ab_1\ mt_1 \rightarrow \varphi,\ \ ab_1\ mt_1 \rightarrow ab_2,\ \ ab_1\ mt_1 \rightarrow ab_2\ mt_2,$$
$$mt_1 \rightarrow \psi,\ \ mt_1 \rightarrow mt_2,\ \ mt_1\ mt_2 \rightarrow \psi,\ \ mt_1\ mt_2 \rightarrow mt_3,\ \ mt_1\ mt_2 \rightarrow mt_3\ mt_4.$$

Proof : From the assumption, in the test set, visible fragments are attributes and hidden fragments are methods. Therefore, this CRS does not have context rewrite rules whose left hand sides are only constructed by walls. Consequently, the form generated by *GSB*-algorithm coincides with the simplest form of behavioural equivalence. □

Example 18 *HSS (continued)*
The CRS generated from the specification *HSS* satisfy the condition of Corollary 14. So, $\bigwedge_{i \in Nat}(get\ rest^{(i)}[s] == get\ rest^{(i)}[s'])$ is the simplest form of behavioural equivalence. □

If there exists two walls for a visible fragment and there are not context rewrite rules whose left hand sides are only constructed by walls, then, for proving $s \equiv s'$, observations through infinite number of observational contexts are necessary. Consequently, for proving $s \equiv s'$, induction over length of contexts are necessary.

Example 19 *WLL2*
Let *WLL2* be the following specification:

```
mod* WLL2 {
  pr(DATA)

  *[ Wll2 ]*
  bop a_ : Wll2 -> DBool
  bop m1_ : Wll2 -> Wll2
  bop m2_ : Wll2 -> Wll2
  bop m3_ : Wll2 -> Wll2

  var S : Wll2
  eq a m3 S = t .
  beq m1 m3 S = m1 S .
  beq m2 m3 S = m2 S .
}
```

The cover set generated by *GSB*-algorithm is {(a, {m1, m2})}. So, the CRS generated from the specification *WLL2* satisfy the condition of Theorem 13. Therefore, the form generated from this cover set is the simplest form of behavioural equivalence. Consequently, for proving behavioural properties, induction over length of contexts are necessary. But, $m3$ is an eliminable fragment. So, we should only deal with $m1$ and $m2$ in induction step.

Consider to prove $(m3\ m3\ S) \equiv (m3\ S)$ in *WLL2*.

The process is as follows:

```
--> Prove (m3 m3 S) Reqv (m3 S) .
--> Base Case)
open .
red a m3 m3 S == a m3 S .
close

--> Induction Step)
open .
bop c_ : Wll2 -> DBool .
eq c m3 m3 S = c m3 S .
red c m1 m3 m3 S == c m1 m3 S .
red c m2 m3 m3 S == c m2 m3 S .
close
```

The result is as follows:

```
--> Prove (m3 m3 S) Reqv (m3 S) .
-->  Base Case)
-- opening module WLL2.. done.
-- reduce in % : a (m3 (m3 S)) == a (m3 S)
true : Bool
(0.017 sec for parse, 3 rewrites(0.017 sec), 3 match attempts)
```

```
-->  Induction Step)
-- opening module WLL2.. done._*
-- reduce in % : c (m1 (m3 (m3 S))) == c (m1 (m3 S))
true : Bool
(0.167 sec for parse, 4 rewrites(0.000 sec), 14 match attempts)
-- reduce in % : c (m2 (m3 (m3 S))) == c (m2 (m3 S))
true : Bool
(0.017 sec for parse, 4 rewrites(0.000 sec), 14 match attempts)
```

Because all executions of *red* commands return *true*, $(m3\ m3\ S) \equiv (m3\ S)$ is valid. □

4.2 An Application of Test Set Coinduction

Firstly, we introduce the technique which makes verifications easier.

Definition 80 *Consider a specification* $(\{State\}, \Sigma, E)$ *whose cover set generated by GSB-algorithm has at most one wall for every visible fragment. We call transformation from walls* wl *to following behavioural operators* $wl*$ **wall transformation***.*

```
op wl* : State Nat -> State
eq wl*(S, 0) = S .
eq wl*(S, s N) = wl*(wl S, N) .
```
□

Wall transformation has the following property:

Property 15 *Consider a specification* $(\{State\}, \Sigma, E)$ *whose cover set generated by GSB-algorithm has at most one wall for every visible fragment. Let* $(\{State\}, \Sigma', E')$ *be* $(\{State\}, \Sigma, E)$*, plus every* $wl*$ *and it's equations. Then,* $(\{State\}, \Sigma, E)$ *and* $(\{State\}, \Sigma', E')$ *are specification equivalence.*

Proof : $wl*$ is exactly $wl^{(i)}$. Therefore, the set of all hidden Σ-algebra which satisfy E coincides with the set of all hidden Σ'-algebra which satisfy E'. Consequently, $(\{State\}, \Sigma, E)$ and $(\{State\}, \Sigma', E')$ are specification equivalence. □

Corollary 16 *Consider a specification* $(\{State\}, \Sigma, E)$ *whose cover set generated by GSB-algorithm has at most one wall for every visible fragment. By wall transformation, the cover set generated by GSB-algorithm is transformed into a cover set.* □

Then, a verification by test set coinduction is as follows:

Example 20 *HSS (continued)*
Consider to verify $rest\ put(t,\ S) \equiv S$ in a specification *HSS*. As we described in Example 16, the cover set generated by *GSB*-algorithm is $\{(get, \{rest\})\}$. This satisfies the assumption of Corollary 16. Therefore, $\{(get, \{rest*\})\}$ is a cover set, too. Consequently, $\bigwedge_{i \in Nat}(get\ rest*(s,\ i) == get\ rest*(s',\ i))$ is also behavioural equivalence. Then, on the CafeOBJ verification system, we should show that:

```
red get rest*(rest put(t, S), N) == get rest*(S, N) .
```

We show it using case analysis that: $N = 0$ or $N = s\ n$:

```
--> Prove (rest put(t, S)) Reqv (S) .
open .
op rest* : Hss Nat -> Hss .
var S : Hss .
var N : Nat .
eq rest*(S, 0) = S .
eq rest*(S, s N) = rest*(rest S, N) .

op n : -> Nat .
op h : -> Hss .
red get rest put(t, h) == get h .
red get rest*(rest put(t, h), s n) == get rest*(h, s n) .
close
```

The result is as follows:

```
--> Prove (rest put(t, S)) Reqv (S) .
-- opening module HSS.. done.__*
-- reduce in % : get (rest put(t,h)) == get h
true : Bool
(0.017 sec for parse, 2 rewrites(0.000 sec), 4 match attempts)
-- reduce in % : get rest*(rest put(t,h),s n) == get rest*(h,s n)

true : Bool
(0.017 sec for parse, 4 rewrites(0.000 sec), 18 match attempts)
```

Because each execution of *red* command returns *true*, $rest\ put(t,\ S) \equiv S$ is valid. □

Chapter 5

Extension of Test Set Coinduction

In this chapter, we discuss an extension of test set coinduction. From the conditions of Definition 71, we will eliminate *(d)*. Also test set coinduction will be extended for handling conditional (behavioural) equations.

5.1 Extended Context Rewriting System

Firstly, context rewriting systems are extended from reductions between observational contexts to reductions between sets of observational contexts. We will extend the definition of extended context rewrite rules step by step.

Definition 81 *Given a (behavioural) equation eq which satisfy the conditions of Definition 71, let* $(\forall X) lhs \rightarrow rhs$ *be the context rewrite rule generated from eq. The* **extended context rewrite rule** *generated from eq is* $(\forall X)(\{lhs\} \rightarrow \{rhs\})$*. We may omit* $(\forall X)$ *when it is not needed.* □

Definition 82 *Left hand sides of extended context rewrite rules must be sets which have exactly one element.* □

Definition 83 *Given a behavioural specification* (H, Σ, E)*, let* E' *be the set of (behavioural) equations which can generate extended context rewrite rules, and let* R *be the set of extended context rewrite rules generated from* E'*. We call* R *the* **extended context rewriting system (ECRS)** *generated from* (H, Σ, E)*. We call extended context rewrite rules generated from equations* **visible extended context rewrite rules***. We call extended context rewrite rules generated from behavioural equations* **hidden extended context rewrite rules***.* □

Definition 84 *Given an ECRS* R*, the* **one-step rewriting relation** *is defined for sets of observational contexts* $OcSet = \{oc_1, \ldots, oc_l\}$ *and OcSet' as follows:*

OcSet $\Rightarrow$ *OcSet' iff there exists: a rule* $(\forall X)(\{lc\} \rightarrow \{rc_1, \ldots, rc_m\})$ *in* R*; an observational context* $ot \in T_\Sigma(\{\Box\} \cup Y)$ *if the above rule is a hidden extended context rewrite rule; a substitution* $sb : X \rightarrow T_\Sigma(Y)$*; and an index* n *such that:*

$oc_n = ot[sb(lc)]$ (or $oc_n = sb(lc)$) and
OcSet' $= \{oc_i \mid i \neq n \ \wedge \ i \in [1, \ldots l]\} \cup \{oc'_i \mid oc'_i = ot[sb(rc_i)]$ for $i \in [1, \ldots, m]\}$
(or OcSet' $= \{oc_i \mid i \neq n \ \wedge \ i \in [1, \ldots l]\} \cup \{oc'_i \mid oc'_i = sb(rc_i)$ for $i \in [1, \ldots, m]\}$),
respectively.

The **extended context rewriting relation** *is the transitive reflexive closure of one-step rewriting relation, for which we write OcSet $\Rightarrow^*$ OcSet' and say that OcSet* **rewrites to** *OcSet'* **(under** R**)**. □

We will change the definitions of test sets, elimination rules, visible fragment application, wall application, and wall, corresponding to the above changes.

Definition 85 *Given a behavioural specification, let mlv be the maximum of the length of the element of the left hand side of visible context rewrite rules, and let mlh be the maximum of the length of the element of the left hand side of hidden context rewrite rules. let lhf be the maximum of 1 and $\lceil mlh/2 \rceil$. let lvf be the maximum of 1, lhf, and (mlv − lhf). A* **test set** *is a cover set such that:*

1. *visible fragments are all combinations of an attribute and methods whose lengths are equal or less than lvf,*

2. *if the length of a visible fragment vf is less than lvf, there is no hidden fragments assigned to vf, and*

3. *if the length of a visible fragment vf is equal to lvf, hidden fragments assigned to vf are all combinations of methods whose lengths are equal to lhf.* □

Definition 86 *Let vf, $vf'_i(vf'_i \neq vf)$ be visible fragments and let hf, $hf'_i(hf'_i \neq hf)$ be hidden fragments. We call the following extended context rewrite rules* **elimination rules**.
$\{vf\} \rightarrow \{cvf'_1, \ldots, cvf'_l\}$ *where* $(cvf'_i = \varphi)$ *or* $(cvf'_i = vf'_i)$, *and*
$\{hf\} \rightarrow \{chf'_1, \ldots, chf'_l\}$ *where* $(chf'_i = \psi)$ *or* $(chf'_i = hf'_i)$. □

Definition 87 *Let hf be a hidden fragment assigned to vf. We call the process which calculates the normal form of $\{vf \ \ hf\}$* **visible fragment application to** *hf*. □

Definition 88 *Let wl and hf be hidden fragments assigned to vf. We call the process which calculates the normal form of $\{wl \ \ hf\}$* **wall application to** *hf* **by** *wl*. □

Definition 89 *Let wl be a hidden fragment assigned to vf. A* **wall assigned to** *vf is defined as follows:*

1. *if the result of visible fragment application to wl includes an observational context except φ, vf', or $vf' \ hf'(hf' \neq wl)$, then wl is a wall, and*

2. *if the result of wall application to wl by a wall wl' includes a hidden context except ψ, hf', or $hf' \ hf''$ $(hf'' \neq wl)$, then wl is a wall.* □

In ECRS, we deal with $\{oc\}$ (or $\{mc\}$), instead of observational contexts oc (or method contexts mc), respectively. So, we introduce the following definition.

Definition 90 *Let omc be observational contexts or method contexts. We call $\{omc\}$ the* **corresponding set of** *omc*. □

5.2 Elimination of condition *(d)*

In this section, we will eliminate the condition *(d)* from Definition 71.

Let eq be an equation whose right hand side has more than one hidden variables. By regarding these hidden variables as holes, eq has more than one observational contexts.

Definition 91 *Let eq be an equation that:*

1. *its left hand side is an observational context, and*
2. *its right hand side has more than one hidden variables.*

We can write eq in the form:
$oc = f(oc_1, \ldots, oc_l)$ *where* $oc, oc_1, \ldots, oc_l$ *are observational contexts, and* f *is a term which does not include hidden sorted variables.*
The **extended context rewrite rule** *is* $\{idel(oc)\} \rightarrow \{idel(oc_1), \ldots, idel(oc_l)\}$. □

Property 17 *Let* $oc = f(oc_1, \ldots, oc_l)$ *be an equation where* $oc, oc_1, \ldots, oc_l$ *are observational contexts, and* f *is a term which does not include hidden sorted variables. This means that this equation satisfies the following conditions:*

1. *its left hand side is an observational context, and*
2. *its right hand side has more than one hidden variables.*

Let $(oc\ mc)_{id}$ *be an observational context where mc is a method context and id is the index value of this observational context (let i be this index). Let* $(oc_j\ mc)_{id_j}$ *be an observational context such that* $(oc\ mc)_{id} = f((oc_1\ mc)_{id_1}, \ldots, (oc_l\ mc)_{id_l})$, *where* id_j *is the index value of this observational context (let* i_j *be this index) for each* j*. Therefore,* $\{idel((oc\ mc)_{id})\} \rightarrow \{idel((oc_1\ mc)_{id_1}), \ldots, idel((oc_l\ mc)_{id_l})\}$*. Then given states* s*,* s'*,*
$(\bigwedge_{i \in PossId}((oc\ mc)_i[s] == (oc\ mc)_i[s']))$
$\wedge\ (\bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{i_j \in PossId_j}((oc_j\ mc)_{i_j}[s] == (oc_j\ mc)_{i_j}[s'])))$
$= \bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{i_j \in PossId_j}((oc_j\ mc)_{i_j}[s] == (oc_j\ mc)_{i_j}[s']))$.
where PossId ($PossId_j$) *denotes the set of all index values of* i (i_j)*, respectively.* □

By using Property 9 and Property 17 instead of Property 9, we get Theorem 10 of ECRS, Property 11 of ECRS and Theorem 12 of ECRS, as follows:

Theorem 18 *let an ECRS be complete. Then,*
$(s \ \equiv\ s') \ =\ \bigwedge_{C \in NormCt}(C[s] == C[s'])$
where NormCt denotes the set of elements of all normal forms of the corresponding sets of all observational contexts without φ. □

Property 19 *If one of elimination rules matches to the corresponding set of a visible fragment vf (or a hidden fragment hf), then this fragment is an eliminable fragment.* □

Theorem 20 *If an ECRS satisfy the following conditions, then we can get a simple form of behavioural equivalence by GSB-algorithm:*

1. *an ECRS is complete, and*

2. *for each context rewrite rule,*
   (*the length of the element of left hand side*) $\geq$
   (*the length of each element of right hand side*). □

Example 21 *ELM3*
Let *ELM3* be the following specification:

```
mod* ELM3 {
  pr(DATA)

  *[ Elm3 ]*
  bop a_  : Elm3 -> Nat
  bop a1_ : Elm3 -> Nat
  bop a2_ : Elm3 -> Nat

  var S : Elm3
  eq a S = a1 S + a2 S .
}
```

The ECRS generated from *ELM3* is
$\{a\} \rightarrow \{a1, a2\}$.
So, the test set is
$\{(a, \emptyset), (a_1, \emptyset), (a_2, \emptyset)\}$.
The process of *GSB*-algorithm is as follows:

1. $\{a\} \rightarrow \{a1, a2\}$
   Therefore, a is an eliminable fragment. Consequently, the generated cover set is $\{(a_1, \emptyset), (a_2, \emptyset)\}$.

2. There is no hidden fragment.

3. There is no hidden fragment.

4. There is no hidden fragment.

5. The generated cover set is $\{(a_1, \emptyset), (a_2, \emptyset)\}$. Therefore, a simple form of behavioural equivalence is $(a_1[s] == a_1[s']) \wedge (a_2[s] == a_2[s'])$.

□

5.3 Conditional Extended Context Rewrite Rule

In this section, we will extend test set coinduction for handling conditional (behavioural) equations.

Recall that condition *(d)* of Definition 71 is as follows:
if eq is a equation, there exists exactly one hidden sorted variable in the right hand side of eq.

Definition 92 *Let*

$lc = rc_1$ *if* cd_1

$\vdots$

$lc = rc_l$ *if* cd_l

be conditional (behavioural) equations eq_i $(i \in [1, \ldots, l])$ *such that:*

1. *all visible sorted arguments in the left hand side of* eq_i *are variables,*
2. *if each* eq_i *is an equation,* lc *is an observational context,*
3. *if each* eq_i *is a behavioural equation, there exists exactly one hidden sorted variable in* lc *and* rc_i*, and*
4. *each* cd_i *is a conjunction over the following forms:*

   $cvv_{i,j} == vcv_{i,j}$ *or* $coc_{i,j} == vcc_{i,j}$

   where $cvv_{i,j}$ *is a visible sorted variable which occurs in* lc*,* $coc_{i,j}$ *is an observational context, and* $vcv_{i,j}$*,* $vcc_{i,j}$ *are visible sorted constants, such that both sides of the above relations have the same sorts.*
5. $cd_1 \vee \cdots \vee cd_l = true$.

If these equations are equations which satisfy condition (d) of Definition 71 or these are behavioural equations, then the **extended context rewrite rule** *is* $\{idel(lc)\} \rightarrow \{idel(rc_1), \ldots, idel(rc_l)\}$.
If one of these equations is an equation which does not satisfy condition (d), then the **extended context rewrite rule** *is* $\{idel(lc)\} \rightarrow \{idel(rc_{1,1}), \ldots, idel(rc_{l,m_l})\}$ *where* $rc_i = f_i(rc_{i,1}, \ldots, rc_{i,m_i})$. $\Box$

Property 21 *Let*

$lc = rc_1$ *if* cd_1

$\vdots$

$lc = rc_l$ *if* cd_l

be conditional behavioural equations beq_i $(i \in [1, \ldots, l])$ *such that:*

1. *all visible sorted arguments in the left hand side of* beq_i *are variables,*
2. *there exists exactly one hidden sorted variable in* lc *and* rc_i*, and*
3. *each* cd_i *is a conjunction over the following forms:*

   $cvv_{i,j} == vcv_{i,j}$ *or* $coc_{i,j} == vcc_{i,j}$

   where $cvv_{i,j}$ *is a visible sorted variable which occurs in* lc*,* $coc_{i,j}$ *is an observational context, and* $vcv_{i,j}$*,* $vcc_{i,j}$ *are visible sorted constants, such that both sides of the above relations have the same sorts.*

4. $cd_1 \vee \cdots \vee cd_l = true$.

Let $(oc\ lc\ mc)_{id}$ *be an observational context where* oc *is an observational context,* mc *is a method context, and* id *is the index value of this observational context (let* i *be this index). Given a state* s*, let* $(oc\ rc_{j_s}\ mc)_{id_{j_s}}$ *be an observational context where* id_{j_s} *is the index value of this observational context (let* i_{j_s} *be this index) such that* $(oc\ lc\ mc)_{id} = (oc\ rc_{j_s}\ mc)_{id_{j_s}}$*. Therefore,* $\{idel((oc\ lc\ mc)_{id})\} \rightarrow \{idel(oc\ rc_1\ mc), \ldots, idel(oc\ rc_l\ mc)\}$*. Then given states* s*,* s'*,*

$$\begin{aligned}
&(\textstyle\bigwedge_{i \in PossId}((oc\ lc\ mc)_i[s] == (oc\ lc\ mc)_i[s']))\\
&\wedge (\textstyle\bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{i_j \in PossId_j}((oc\ rc_j\ mc)_{i_j}[s] == (oc\ rc_j\ mc)_{i_j}[s'])))\\
&\wedge (\textstyle\bigwedge_{i \in [1,\ldots,l]}(\bigwedge_{j \in [1,\ldots,l_i]}(coc_{i,j}[s] == coc_{i,j}[s'])))\\
&= (\textstyle\bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{i_j \in PossId_j}((oc\ rc_j\ mc)_{i_j}[s] == (oc\ rc_j\ mc)_{i_j}[s'])))\\
&\wedge (\textstyle\bigwedge_{i \in [1,\ldots,l]}(\bigwedge_{j \in [1,\ldots,l_i]}(coc_{i,j}[s] == coc_{i,j}[s']))).
\end{aligned}$$

where $PossId$ *(*$PossId_j$*) denotes the set of all index values of* i *(*i_j*), respectively.* □

Property 22 *Let*

$$\begin{aligned}
&lc = rc_1 \ \ if \ \ cd_1\\
&\qquad \vdots\\
&lc = rc_l \ \ if \ \ cd_l
\end{aligned}$$

be conditional equations eq_i $(i \in [1, \ldots, l])$ *such that:*

1. *all visible sorted arguments in the left hand side of* eq_i *are variables,*

2. lc *is an observational context,*

3. *each* cd_i *is a conjunction over the following forms:*

   $$cvv_{i,j} == vcv_{i,j} \ \ or \ \ coc_{i,j} == vcc_{i,j}$$

   where $cvv_{i,j}$ *is a visible sorted variable which occurs in* lc*,* $coc_{i,j}$ *is an observational context, and* $vcv_{i,j}$*,* $vcc_{i,j}$ *are visible sorted constants, such that both sides of the above relations have the same sorts.*

4. $cd_1 \vee \cdots \vee cd_l = true$*, and*

Let $(lc\ mc)_{id}$ *be an observational context where* mc *is a method context and* id *is the index value of this observational context (let* i *be this index). Given a state* s*, let* $(rc_{j_s}\ mc)_{id_{j_s}}$ *be an observational context where* id_{j_s} *is the index value of this observational context (let* i_{j_s} *be this index) such that* $(lc\ mc)_{id} = (rc_{j_s}\ mc)_{id_{j_s}}$*, and* $rc_{j_s} = f_{j_s}(rc_{j_s,1}, \ldots, rc_{j_s,m_{j_s}})$*. Therefore,* $\{idel((lc\ mc)_{id})\} \rightarrow \{idel(rc_{1,1}\ mc), \ldots, idel(rc_{l,m_l}\ mc)\}$*. Then given states* s*,* s'*,*

$$\begin{aligned}
&(\textstyle\bigwedge_{i \in PossId}((lc\ mc)_i[s] == (lc\ mc)_i[s']))\\
&\wedge (\textstyle\bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{k \in [1,\ldots,m_j]}(\bigwedge_{i_{j,k} \in PossId_{j,k}}((rc_{j,k}\ mc)_{i_{j,k}}[s] == (rc_{j,k}\ mc)_{i_{j,k}}[s']))))\\
&\wedge (\textstyle\bigwedge_{i \in [1,\ldots,l]}(\bigwedge_{j \in [1,\ldots,l_i]}(coc_{i,j}[s] == coc_{i,j}[s'])))\\
&= (\textstyle\bigwedge_{j \in [1,\ldots,l]}(\bigwedge_{k \in [1,\ldots,m_j]}(\bigwedge_{i_{j,k} \in PossId_{j,k}}((rc_{j,k}\ mc)_{i_{j,k}}[s] == (rc_{j,k}\ mc)_{i_{j,k}}[s']))))\\
&\wedge (\textstyle\bigwedge_{i \in [1,\ldots,l]}(\bigwedge_{j \in [1,\ldots,l_i]}(coc_{i,j}[s] == coc_{i,j}[s']))).
\end{aligned}$$

where $PossId$ *(*$PossId_{j,k}$*) denotes the set of all index values of* i *(*$i_{j,k}$*), respectively.* □

By using Property 9, Property 17, Property 21, and Property 22, instead of Property 9, we get Theorem 10 of ECRS, Property 11 of ECRS and Theorem 12 of ECRS, as follows:

Theorem 23 *let an ECRS be complete. Then,*
$(s \equiv s') = \bigwedge_{C \in NormCt}(C[s] == C[s'])$
where NormCt denotes the set of elements of all normal forms of the corresponding sets of all observational contexts without φ. □

Property 24 *If one of elimination rules matches to the corresponding set of a visible fragment vf (or a hidden fragment hf), then this fragment is an eliminable fragment.* □

Theorem 25 *If an ECRS satisfy the following conditions, then we can get a simple form of behavioural equivalence by GSB-algorithm:*

1. *an ECRS is complete, and*
2. *for each context rewrite rule,*
   (*the length of the element of left hand side*) $\geq$
   (*the length of each element of right hand side*). □

Example 22 *ELM4*
Let *ELM4* be the following specification:

```
mod* ELM4 {
  pr(DATA)

  *[ Elm4 ]*
  bop a : DBool Elm4 -> Nat
  bop a1_ : Elm4 -> Nat
  bop a2_ : Elm4 -> Nat

  var B : DBool
  var S : Elm4
  ceq a(B, S) = a1 S if B == t .
  ceq a(B, S) = a2 S if B == f .
}
```

The ECRS generated from *ELM4* is
$\{a\} \rightarrow \{a1, a2\}$.
□

Example 23 *COND*
Let *COND* be the following specification:

```
mod* COND {
  pr(DATA)

  *[ Cond ]*
  op cnd_ : Cond -> DBool
  op a_ : Cond -> Nat
  op a1_ : Cond -> Nat
  op a2_ : Cond -> Nat

  var B : DBool
  var S : Cond
  ceq a S = a1 S if cnd S == t .
  ceq a S = a2 S if cnd S == f .
}
```

The ECRS generated from *COND* is
$\{a\} \to \{a1, a2\}$.
□

Chapter 6

Object Composition

In this chapter, we introduce an object-oriented approach into behavioural specifications. We deal with the following behavioural specifications:

1. there is exactly one module which is declared by `mod!`, and

2. there are some modules which have exactly one hidden sort, have declarations related to this hidden sort, and are declared by `mod*`.

The former module specifies data structures. The latter modules correspond to classes.

Consider to specify a system by an object-oriented approach. Firstly, we divide this system to many primitive components and specify these components. Then, we specify a composition of components. This specification is the specification of the component composed by the above components. By iterating to specify compositions of components, we can specify this system. The above components are objects. We can regard a object (component) as a black box. So, the specification of this object (component) is the module corresponding to the class of this object (component). The hidden sort of this class includes the set of states of this black box.

6.1 Object Composition

Firstly, we give the formal definitions of the above behavioural specifications.

Definition 93 *A* **data module** *is a module constructed from the following declarations:*

1. *declarations of visible sorts,*

2. *declarations of operators of these sorts, and*

3. *declarations of conditional equations related to these operators.* □

Definition 94 class modules *are modules constructed from the following declarations:*

1. *importation declarations of a data module and other class modules,*

2. a declaration of exactly one hidden sort,

3. declarations of operators of this sort,

4. declarations of behavioural operators of this sort, and

5. declarations of conditional (behavioural) equations related to these operators. □

Given a class module C, we call this hidden sort the **sort of** C *and we call this data module the* **data module of** C. *We let Φ_C denote (behavioural) operators of C, let AM_C denote attributes and methods of C, let E_C denote conditional (behavioural) equations of C, and let D_C denote the data module of C.*

Definition 95 **Object-oriented specifications** *are behavioural specifications which can be regarded as class modules. We let (H_C, Σ_C, AE_C) denote a class module C when we regard a class module as a behavioural specification.* □

In this chapter, we deal with object-oriented specifications.

Secondly, we give the formal definition of objects. There may be many objects for one class. The sets of states of these objects are included in the sort of this class. So, we need to make a distinction between sets of states of these objects. Recall that we treat methods as operators which change states of a black box. So, if we regard methods as connections between states of the same object, the sets of states of objects are connected components in the sort of this class.

Definition 96 *Given an object-oriented specification (H, Σ, E), a class C, and hidden Σ-algebra M, let h_C be the sort of C and regard interpretations of methods of C on M as connections between an element of M_{h_C} corresponding to its arity and an element of M_{h_C} corresponding to its sort. We call connected components of M_{h_C}* **objects of class** C **on** M. □

In the processes to specify systems, we use two kinds of class modules. One is the class modules which specify primitive components. Another is the class modules which specify compositions of components. We specify these compositions by correspondences between behavioural operators of composed objects and those of composing objects. We use pseudo-projection operators to specify these correspondences. Finally, we give the definitions of primitive modules, pseudo-projection operators, and pseudo-composition modules.

Definition 97 *A* **primitive module** *is a class module whose importation declaration is only importation declaration of a data module.* □

Before we can give the definition of pseudo-projection operators, we need the following notation.

Definition 98 *Given an object-oriented specification (H, Σ, E) and a class C, let h_C be the sort of C. We call observational (method) Σ-contexts of sort h_C* **observational (method) C-contexts**, *respectively.* □

The definition of pseudo-projection operators for the cases that the correspondences can be specified by (behavioural) equations is as follows.

Definition 99 *Let O be a composed object, C be the class of O, and h be the sort of C. Let O_i be a composing object, C_i be the class of O_i, and h_i be the sort of C_i for each $i \in ObjId$ where $ObjId$ is a set of all identifiers of composing objects. We call behavioural operators $\pi_i : h \rightarrow h_i$ which satisfy the following conditions* **pseudo-projection operators of** *C:*

1. *given an attribute ab of C, there exists composing objects $O_{i_1}, \ldots, O_{i_l}$, observational C_{i_j}-contexts $ct_{i_j} : h_{i_j} \rightarrow v_{i_j}$, and a operator $f : v_{i_1} \cdots v_{i_l} \rightarrow v$ such that:*

   $ab = f(ct_{i_1}\ \pi_{i_1}, \ldots, ct_{i_l}\ \pi_{i_l})$, [1]

2. *given a method mt of C and a composing object O_i, there exists a method C_i-context ms_i such that:*

   $\pi_i\ mt = ms_i\ \pi_i$, *and*

3. *given a hidden constant hc of C and a composing object O_i, there exists a hidden constant hc_i of C_i such that:*

   $\pi_i\ hc = hc_i$.

We call the above (behavioural) equations **composition definitions of** *C.* $\Box$

The definition of conditional composition definitions is as follows.

Definition 100 *We call conditional (behavioural) equations*

$$\begin{array}{l} lhs = rhs_1 \ \text{if} \ cd_1 \\ \qquad \vdots \\ lhs = rhs_l \ \text{if} \ cd_l \end{array}$$

which satisfy the following conditions **conditional composition definitions of** *C:*

1. *each (behavioural) equation is an ordinary composition definition of C whenever condition is true,*

2. *let ct_j be an observational C_j-context, let s be a state of the composed object O, and let D be a visible sorted term which does not have hidden sorted variables and hidden constants as subterms, then cd_i is a finite conjunction of the forms:*

   $ct_j\ \pi_j[s] == D$, *and*

3. $cd_1 \vee \cdots \vee cd_l = true$. $\Box$

Definition 101 *A* **pseudo-composition module** *C is a class module C such that:*

1. *importation declarations are an importation declaration of a data module and importation declarations of other class modules (which correspond to composing objects),*

[1] Precisely, $ab(s) = f(ct_{i_1}\ \pi_{i_1}(s), \ldots, ct_{i_l}\ \pi_{i_l}(s))$.

2. *declarations of behavioural operators are declarations of attributes, methods, hidden constants, and pseudo-projection operators of C, and*

3. *declarations of conditional (behavioural) equations are declarations of (conditional) composition definitions.* □

Example 24 *DATA, CELL* and *PCARR*
DATA module is a data module.

```
mod! DATA {
  [ Nat < Int ]
  op 0 : -> Nat
  op s_ : Nat -> Nat
  op s_ : Int -> Int
  op p_ : Int -> Int
  op _+_ : Int Int -> Int

  [ DBool ]
  op t : -> DBool
  op f : -> DBool
  op not_ : DBool -> DBool

  vars I1 I2 : Int
  eq s p I1 = I1 .
  eq p s I1 = I1 .
  eq I1 + 0 = I1 .
  eq I1 + s I2 = s(I1 + I2) .
  eq I1 + p I2 = p(I1 + I2) .

  var B : DBool
  eq not t = f .
  eq not f = t .
  eq not not B = B .
}
```

CELL module is a primitive module.

```
mod* CELL {
  pr(DATA)

  *[ Cell ]*
  bop view_ : Cell -> DBool
  bop set : DBool Cell -> Cell

  var B : DBool
```

```
  var C : Cell
  eq view set(B, C) = B .
}
```

PCARR module is a pseudo-composition module which constructed from DATA module and CELL module. *ObjId* is *Int* and $\pi_i(\Box)$ is $cell(i, \Box)$.

```
mod* PCARR {
  pr(DATA)
  pr(CELL)

  *[ CArr ]*
  bop get : Int CArr -> DBool
  bop put : DBool Int CArr -> CArr

-- pseudo-projection operator
  bop cell : Int CArr -> Cell

  vars I J : Int
  var B : DBool
  var CA : CArr

-- conditional composition definitions
  eq get(I, CA) = view cell(I, CA) .
  ceq cell(I, put(B, J, CA)) = set(B, cell(I, CA))
      if I == J .
  ceq cell(I, put(B, J, CA)) = cell(I, CA)
      if I =/= J .
}
```

$\Box$

From now on, we use the following definitions.

Definition 102 *Given an object-oriented specification (H, Σ, E) and a class C, let h_C be the sort of C. We call behavioural Σ-equivalence of sort h_C* **behavioural C-equivalence***.*
$\Box$

For behavioural C-equivalence of a composed object of class C and behavioural C_i-equivalences of composing objects of class C_i, the next theorem holds.

Theorem 26 *Let O be a composed object, let C be the class of O, and let $\equiv$ be behavioural C-equivalence. Let O_i be a composing object, C_i be the class of O_i, and $\equiv_i$ be behavioural C_i-equivalence for each $i \in$ ObjId where ObjId is a set of all identifiers of composing objects. Let π_i be a pseudo-projection operator of C for each $i \in$ ObjId. Then, given states s, s' of O,*
$(s \equiv s') = \wedge_{i \in ObjId}(\pi_i(s) \equiv_i \pi_i(s'))$.

Proof : Let h be a sort of class C and h_i be a sort of class C_i.

Behavioural C-equivalence is a conjunction over observational C-contexts — sequences of behavioural operators. We categorize observational C-contexts as follows:

1. $ab\ mt_1 \cdots mt_l\ (l \geq 0)$,
2. $ab_i\ mt_{i,1} \cdots mt_{i,l_i}\ \pi_i\ mt_1 \cdots mt_l\ (l_i \geq 0 \quad l \geq 1)$, and
3. $ab_i\ mt_{i,1} \cdots mt_{i,l_i}\ \pi_i\ (l_i \geq 0)$,

where ab is an attribute of C, $mt_1, \ldots, mt_l$ are methods of C, ab_i is an attribute of C_i, $mt_{i,1}, \ldots, mt_{i,l}$ are methods of C_i.

Therefore, to prove this theorem, we should show that behavioural C-equivalence is the conjunction over all observational C-contexts of 3.

A. The cases that there is no conditional composition definition of C
Firstly, we will eliminate observational C-contexts of 1. From Definition 99, for each attribute ab of C, there exists an observational C_{i_j}-context $ct_{i_j} : h_{i_j} \to v_{i_j}$ for $j \in [1, \ldots, m]$, and a operator $f : v_{i_1} \cdots v_{i_m} \to v$ such that $ab = f(ct_{i_1}\ \pi_{i_1}, \ldots, ct_{i_m}\ \pi_{i_m})$. We let $cpm_{i_j} = ct_{i_j}\ \pi_{i_j}\ mt_1 \cdots mt_l$, then cpm_{i_j} is an observational C-context of 2 and $ab\ mt_1 \cdots mt_m = f(cpm_{i_1}, \cdots, cpm_{i_m})$. Therefore, given states s, s' of O,
$(ab\ mt_1 \cdots mt_l[s] == ab\ mt_1 \cdots mt_l[s']) \wedge (\bigwedge_{j \in [1,\cdots,m]}(cpm_{i_j}[s] == cpm_{i_j}[s']))$
$= (\bigwedge_{j \in [1,\cdots,m]}(cpm_{i_j}[s] == cpm_{i_j}[s']))$.
From this fact, behavioural C-equivalence is the conjunction over all observational C-contexts of 2 and 3.

Secondly, we will eliminate observational C-contexts of 2. From Definition 99, for each method mt of C, there exists a method C_i-context $ms_{i,j}$ such that $\pi_i\ mt_j = ms_{i,j}\ \pi_i$. Then, $ab_i\ mt_{i,1} \cdots mt_{i,l_i}\ \pi_i\ mt_1 \cdots mt_l = ab_i\ mt_{i,1} \cdots mt_{i,l_i}\ ms_{i,1} \cdots ms_{i,l}\ \pi_i$. We let lhs denote the left hand side of the above equation, and let rhs denote the right hand side of it. Note that rhs is an observation C-context of 3. Given states s, s' of O,
$(lhs[s] == lhs[s']) \wedge (rhs[s] == rhs[s']) = (rhs[s] == rhs[s'])$.
From this fact, behavioural C-equivalence is the conjunction over all observational C-contexts of 3.

B. The cases that there are conditional composition definitions of C
Firstly, we will eliminate observational C-contexts of 1. We assume that:
$ab\ mt_1 \cdots mt_l = f(cpm_{i_{1,1}}, \cdots, cpm_{i_{1,m_1}})\ \mathit{if}\ cd_1$,
$\vdots$
$ab\ mt_1 \cdots mt_l = f(cpm_{i_{n,1}}, \cdots, cpm_{i_{n,m_n}})\ \mathit{if}\ cd_n$, and
$cd_1 \vee \cdots \vee cd_n = true$ (conditional composition definitions).
Let $ct_{j,1}\ \pi_{i'_{j,1}}, \cdots, ct_{j,n_j}\ \pi_{i'_{j,n_j}}$ be observational C-contexts which occur in cd_j. We can select cd_j, depending on the observational value through $\bigwedge_{j \in [1,\cdots,n]}(\bigwedge_{k \in [1,\cdots,n_j]} ct_{j,i'_{j,k}}\ \pi_{i'_{j,k}})$, hereafter denoted obs. Therefore, given states s, s' of O,

$(ab\ mt_1 \cdots mt_l[s] == ab\ mt_1 \cdots mt_l[s'])$
$\wedge (\bigwedge_{j \in [1,\cdots,n]} (\bigwedge_{k \in [1,\cdots,m_j]} (cpm_{i_{j,k}}[s] == cpm_{i_{j,k}}[s']))) \wedge (obs[s] == obs[s'])$
$= (\bigwedge_{j \in [1,\cdots,n]} (\bigwedge_{k \in [1,\cdots,m_j]} (cpm_{i_{j,k}}[s] == cpm_{i_{j,k}}[s']))) \wedge (obs[s] == obs[s'])$.
Note that $cpm_{i_{j,k}}$ is an observational C-context of 2 and obs is a conjunction over observational C-contexts of 3. Therefore, behavioural C-equivalence is the conjunction over all observational C-contexts of 2 and 3.

Secondly, we will eliminate observational C-contexts of 2. We assume that:
$\pi_i\ mt_j = ms_{i,j,1} \pi_i\ \mathit{if}\ \ cd_{j,1}$,
$\vdots$
$\pi_i\ mt_j = ms_{i,j,m_j} \pi_i\ \mathit{if}\ \ cd_{j,m_j}$, and
$cd_{j,1} \vee \cdots \vee cd_{j,m_j} = \mathit{true}$ (conditional composition definitions).
Let $ct_{j,j',1}\ \pi_{i''_{j,j',1}}, \cdots, ct_{j,j',n_{j,j'}}\ \pi_{i''_{j,j',n_{j,j'}}}$ be observational C-contexts which occur in $cd_{j,j'}$.
We can select $cd_{1,j'_1}, \ldots, cd_{l,j'_l}$, depending on the observational value through $\bigwedge_{j \in [1,\cdots,m]} (\bigwedge_{j' \in [1,\cdots,m_j]} (\bigwedge_{j'' \in [1,\cdots,n_{j,j'}]} ct_{j,j',j''}\ \ \pi_{i''_{j,j',j''}}))$, hereafter denoted obs. We let $lhs = ab_i\ mt_{i,1} \cdots mt_{i,l_i}\ \pi_i\ mt_1 \cdots mt_l$. Regard conditional composition definitions as conditional rewrite rules. Then, let $rhs_{j'_1,\cdots,j'_l}$ be normal forms of lhs under the condition $cd_{1,j'_1} \wedge \ldots \wedge cd_{l,j'_l} = \mathit{true}$. Let $CondId$ be the set of all identifiers of rhs_I. Then, given states s, s' of O,
$(lhs[s] == lhs[s']) \wedge (\bigwedge_{j \in CondId} (rhs_j[s] == rhs_j[s'])) \wedge (obs[s] == obs[s'])$
$= (\bigwedge_{j \in CondId} (rhs_j[s] == rhs_j[s'])) \wedge (obs[s] == obs[s'])$.
Note that rhs_j is an observational C-context of 3 and obs is a conjunction over observational C-contexts of 3. From this fact, behavioural C-equivalence is the conjunction over all observational C-contexts of 3. □

$\equiv_i$ is generated by *GSB*-algorithm. The idea of the proof of Theorem 26 is the same with that of Theorem 25 (Theorem 12). Both ideas are elimination of redundant contexts. So, Theorem 26 can be seen as the generalization of *GSB*-algorithm of test set coinduction.

Example 25 *DATA, CELL* and *PCARR (continued)*
A simple form $\equiv_{CELL}$ of behavioural *CELL*-equivalence generated by *GSB*-algorithm is that
$(s\ \ \equiv_{CELL}\ \ s') = (\mathit{view}\ s == \mathit{view}\ s')$.
So, from Theorem 26, a simple form $\equiv_{PCARR}$ of behavioural *PCARR*-equivalence is that
$(s\ \ \equiv_{PCARR}\ \ s') = (\bigwedge_{i\ \in Int} (\mathit{view}\ \ \mathit{cell}(i, s) == \mathit{view}\ \ \mathit{cell}(i, s')))$.
□

6.2 Composition of Objects and Data

Let O be a composed object, C be a class of O, and h be a sort of C. Let v be a visible sort and I_v be an identity of v — for all terms t of sort v, $I_v\ t = t$. Let at be an attribute of C whose sort is v. We can regard at as $I_v\ at$. So, by regarding I_v as an "attribute" of v, we can regard v as an "object", and at as a "pseudo-projection operator" of C.

We extend the definition of pseudo-projection operators to data.

Definition 103 *Let O be a composed object, C be the class of O, and h be the sort of C. Let O_i be a composing object, C_i be the class of O_i, and h_i be the sort of C_i for each $i \in ObjId$ where $ObjId$ is a set of all identifiers of composing objects. Let v_j be a visible sort for each $j \in DId$ where DId is a set of identifiers of visible sorts. We call behavioural operators $\pi_i : h \to h_i$ which satisfy conditional composition definitions and behavioural operators $\pi_j : h \to v_j$* **pseudo-projection operators of** C. $\Box$

From now on, we select a method or a pseudo-projection operator for a meaning of a behavioural operator whose rank is $\langle h, v_j \rangle$, depending on the purpose.

We extend Theorem 26 to this pseudo-projection operator.

Corollary 27 *Let O be a composed object, let C be the class of O, and let $\equiv$ be behavioural C-equivalence. Let O_i be a composing object, C_i be the class of O_i, and $\equiv_i$ be behavioural C_i-equivalence for each $i \in ObjId$ where $ObjId$ is a set of all identifiers of composing objects. Let v_j be a visible sort for each $j \in DId$ where DId is a set of identifiers of visible sorts. Let π_i and π_j be pseudo-projection operators of C for each $i \in ObjId$ and each $j \in DId$. Then, given states s, s' of O,*
$(s \equiv s') = (\bigwedge_{i \in ObjId}(\pi_i(s) \equiv_i \pi_i(s'))) \wedge (\bigwedge_{j \in DId}(\pi_j(s) == \pi_j(s')))$. $\Box$

Example 26 *PAPHSS*
PAPHSS class is composed from *ARR* class and *Nat* sort as follows.

```
mod* ARR {
  pr(DATA)

  *[ Arr ]*
  bop get : Int Arr -> DBool
  bop put : DBool Int Arr -> Arr

  vars I J : Int
  var B : DBool
  var A : Arr
  ceq get(I, put(B, J, A)) = B
      if I == J .
  ceq get(I, put(B, J, A)) = get(I, A)
      if I =/= J .
}

mod* PAPHSS {
  pr(DATA)
  pr(ARR)

  *[ APHss ]*
  bop get_ : APHss -> DBool
  bop put : DBool APHss -> APHss
```

```
  bop rest_ : APHss -> APHss

-- pseudo-projection operators
  bop arr_ : APHss -> Arr
  bop ptr_ : APHss -> Int

  var AP : APHss
  var B : DBool

-- composition definitions
  eq get AP = get(ptr AP, arr AP) .
  eq ptr put(B, AP) = s ptr AP .
  eq arr put(B, AP) = put(B, s ptr AP, arr AP) .
  eq ptr rest AP = p ptr AP .
  eq arr rest AP = arr AP .
}
```

A simple form $\equiv_{ARR}$ of behavioural *ARR*-equivalence generated by *GSB*-algorithm is that
$(s \ \equiv_{ARR} \ s') = (\bigwedge_{i \ \in Int}(get(i,s) == get(i,s')))$.
So, from Corollary 27, a simple form $\equiv_{PAPHSS}$ of behavioural *PAPHSS*-equivalence is that
$(s \ \equiv_{PAPHSS} \ s') = (\bigwedge_{i \ \in Int}(get(i, arr \ s) == get(i, arr \ s'))) \wedge (ptr \ s == ptr \ s')$.
□

6.3 Projection Operator

Consider to construct *HSS* class from *ARR* class and a pointer (*Nat* sort). If we use pseudo-projection operators, the class of the composed object is not *HSS* (see Example 26). Because, in *PAPHSS*, we can observe the contents of cells upper than a pointer, but, in *HSS*, we can not observe these contents. Moreover, the value of a pointer is not necessary in *HSS*. In *PAPHSS*, there are observational contexts in which pseudo-projection operators occur. This means that there are observational contexts without observational contexts which constructed from attributes and methods of *HSS*. To eliminate these observational contexts, we introduce projection operators that are pseudo-projection operators but ordinary operators.

The formal definition of projection operators is as follows.

Definition 104 *Let O be a composed object, C be the class of O, and h be the sort of C. Let O_i be a composing object, C_i be the class of O_i, and h_i be the sort of C_i for each $i \in ObjId$ where $ObjId$ is a set of all identifiers of composing objects. Let v_j be a visible sort for each $j \in DId$ where DId is a set of identifiers of visible sorts. We call ordinary operators $\pi_i : h \to h_i$ which satisfy conditional composition definitions and behavioural operators $\pi_j : h \to v_j$* **projection operators of** C. □

Also, we define composition modules as follows.

Definition 105 *A* **composition module** *C is a class module C such that:*

1. *importation declarations are an importation declaration of a data module and importation declarations of other class modules (which correspond to composing objects),*
2. *declarations of operators are declarations of projection operators of C, and*
3. *declarations of behavioural operators are declarations of attributes, methods, and hidden constants of C, and*
4. *declarations of conditional (behavioural) equations are declarations of (conditional) composition definitions.* □

Example 27 *APHSS*
APHSS module is a composition module which constructed from DATA module and ARR module.

```
mod* ARR {
  pr(DATA)

  *[ Arr ]*
  bop get : Int Arr -> DBool
  bop put : DBool Int Arr -> Arr

  vars I J : Int
  var B : DBool
  var A : Arr
  ceq get(I, put(B, J, A)) = B
      if I == J .
  ceq get(I, put(B, J, A)) = get(I, A)
      if I =/= J .
}

mod* APHSS {
  pr(DATA)
  pr(ARR)

  *[ APHss ]*
  bop get_ : APHss -> DBool
  bop put : DBool APHss -> APHss
  bop rest_ : APHss -> APHss

-- projection operators
  op arr_ : APHss -> Arr
  op ptr_ : APHss -> Int
```

```
  var AP : APHss
  var B : DBool

-- composition definitions
  eq get AP = get(ptr AP, arr AP) .
  eq ptr put(B, AP) = s ptr AP .
  eq arr put(B, AP) = put(B, s ptr AP, arr AP) .
  eq ptr rest AP = p ptr AP .
  eq arr rest AP = arr AP .
}
```

□

Chapter 7

Stepwise Refinement

In this chapter, we introduce stepwise refinements into object-oriented specifications.

Consider to specify a system under stepwise refinements. Firstly, we specify an abstract level specification. Then, we specify a more concrete level specification. All models of the latter specification must satisfy all conditional (behavioural) equations of the former one. Specifying more concrete level specifications again and again, a refined specification reach the level that we want to specify the system.

In object-oriented specifications, the above refinement process corresponds to exchanging a primitive module for a composition module that:

1. it is constructed from primitive modules, in the sense that importation declarations of class modules are declarations of these primitive modules, and

2. all its models satisfy all conditional (behavioural) equations of the original primitive module.

7.1 Stepwise Refinement

The formal definition of refinements is as follows.

Definition 106 *Given behavioural specifications (H, Σ, E) and (H', Σ', E'), A hidden signature morphism $\varphi : \Sigma \to \Sigma'$ is a* **refinement** *$\varphi : (H, \Sigma, E) \to (H', \Sigma', E')$ iff $\varphi M' \models\!\!\equiv_\Sigma E$ for each hidden (Σ', E')-model M'.* $\Box$

Remark 5 *In the definition of refinements in [GM97], a hidden signature map (which is a hidden signature morphism that preserves hidden sorts) are used instead of a hidden signature morphism. But, in the refinement process, only correspondence of hidden sorts between H and H' is necessary. Consequently, we used a hidden signature morphism instead of a hidden signature map in the above definition.* $\Box$

As to refinements, the following property holds.

Property 28 *Let $\varphi : (\Sigma, E) \to (\Sigma', E')$ and $\varphi' : (\Sigma', E') \to (\Sigma'', E'')$ be refinements. The composition $\varphi'\ \varphi : (\Sigma, E) \to (\Sigma'', E'')$ is a refinement, too.* $\Box$

In this section, we describe stepwise refinements between object-oriented specifications.

Definition 107 *Let PM be a primitive module and CM be a composition module such that:*

1. $D_{PM} = D_{CM}$ *and*

2. $AM_{PM} = AM_{CM}$ *(regarding* $h_{PM} = h_{CM}$ *where* h_{PM} (h_{CM}) *is the sort of PM (CM), respectively).*

From the above property, $\Sigma_{PM} \subseteq \Sigma_{CM}$*. So, the inclusion* $i : \Sigma_{PM} \to \Sigma_{CM}$ *is a hidden signature morphism. We call CM a* **corresponding composition module of** *PM iff* $i : \Sigma_{PM} \to \Sigma_{CM}$ *is a refinement* $i : (H_{PM}, \Sigma_{PM}, AE_{PM}) \to (H_{CM}, \Sigma_{CM}, AE_{CM})$*.* □

Theorem 29 *Let PM be a primitive module and let* (H, Σ, E) *be an object-oriented specification which include PM. Let CM be a corresponding composition module of PM. By exchanging PM for CM, we get an object-oriented specification from* (H, Σ, E)*. We let* (H', Σ', E') *denote this object-oriented specification. By regarding* $h_{PM} = h_{CM}$ *(where* h_{PM} (h_{CM}) *is the sort of PM (CM), respectively),* $\Sigma \subseteq \Sigma'$*. Then, the inclusion* $i : \Sigma \to \Sigma'$ *is a refinement* $i : (H, \Sigma, E) \to (H', \Sigma', E')$*.*

Proof : Let E_{ps} be the set of conditional (behavioural) equations such that $E_{ps} \cup E_{PM} = E$ and $E_{ps} \cap E_{PM} = \emptyset$. So, $E_{ps} \cup E_{CM} = E'$ and $E_{ps} \cap E_{CM} = \emptyset$. Given a (Σ', E')-model M. Because $i : \Sigma \to \Sigma'$ is an inclusion, $iM \models\!\equiv E_{ps}$. On the other hand, from Definition 107, $iM \models\!\equiv E_{PM}$. Therefore, $iM \models\!\equiv E$. So, $i : \Sigma \to \Sigma'$ is a refinement $i : (H, \Sigma, E) \to (H', \Sigma', E')$. □

By exchanging primitive modules for corresponding composition modules again and again, a refined specification reach the level that we want to specify the system.

Example 28 *HSS, APHSS,* and*, CAPHSS*
Recall that *HSS* is the following module:

```
mod* HSS {
  pr(DATA)

  *[ Hss ]*
  bop get_ : Hss -> DBool
  bop put : DBool Hss -> Hss
  bop rest_ : Hss -> Hss

  var B : DBool
  var S : Hss
  eq get put(B, S) = B .
  beq rest put(B, S) = S .
}
```

Recall that *ARR* and *APHSS* are the following modules.

```
mod* ARR {
  pr(DATA)

  *[ Arr ]*
  bop get : Int Arr -> DBool
  bop put : DBool Int Arr -> Arr

  vars I J : Int
  var B : DBool
  var A : Arr
  ceq get(I, put(B, J, A)) = B
      if I == J .
  ceq get(I, put(B, J, A)) = get(I, A)
      if I =/= J .
}

mod* APHSS {
  pr(DATA)
  pr(ARR)

  *[ APHss ]*
  bop get_ : APHss -> DBool
  bop put : DBool APHss -> APHss
  bop rest_ : APHss -> APHss

-- projection operators
  op arr_ : APHss -> Arr
  op ptr_ : APHss -> Int

  var S : APHss
  var B : DBool

-- composition definitions
  eq get S = get(ptr S, arr S) .
  eq ptr put(B, S) = s ptr S .
  eq arr put(B, S) = put(B, s ptr S, arr S) .
  eq ptr rest S = p ptr S .
  eq arr rest S = arr S .
}
```

Note that *HSS* module is a primitive module. Firstly, we prove that the inclusion $\varphi_1 : \Sigma_{Hss} \to \Sigma_{APHss}$ is an refinement, by showing that *APHSS* module is a corresponding composition module of *HSS*. As discussed in Example 20, a simple form $\equiv_{HSS}$ of behavioural *HSS*-equivalence is that: $\bigwedge_{i \in Nat}(get\ rest{*}(s,\ i) == get\ rest{*}(s',\ i))$. So, this process is as follows:

```
--> Verifying refinement from HSS to APHSS
open .
op rest* : APHss Nat -> APHss .
op p* : Int Nat -> Int .

var S : APHss .
var I : Int .
var N : Nat .
eq rest*(S, 0) = S .
eq rest*(S, s N) = rest*(rest S, N) .
eq p*(I, 0) = I .
eq p*(I, s N) = p*(p I, N) .
eq ptr rest*(S, N) = p*(ptr S, N) .
eq arr rest*(S, N) = arr S .

op b : -> DBool .
op n : -> Nat .
op h : -> APHss .

--> eq get put(B, S) = B .
red get put(b, h) == b .
--> beq rest put(B, S) = S .
red get rest put(b, h) == get h .
red get rest*(rest put(b, h), s n) == get rest*(h, s n) .
close
```

The result is as follows:

```
--> Verifying refinement from HSS to APHSS
-- opening module APHSS.. done._
--> eq get put(B, S) = B ._*
-- reduce in % : get put(b,h) == b
true : Bool
(0.000 sec for parse, 7 rewrites(0.000 sec), 23 match attempts)
--> beq rest put(B, S) = S .
-- reduce in % : get (rest put(b,h)) == get h
true : Bool
(0.017 sec for parse, 10 rewrites(0.000 sec), 44 match attempts)
-- reduce in % : get rest*(rest put(b,h),s n) == get rest*(h,s n)
true : Bool
(0.000 sec for parse, 20 rewrites(0.017 sec), 86 match attempts)
```

Because each execution of *red* command returns *true*, *APHSS* module is a corresponding composition module of *HSS*. So, the inclusion $\varphi_1 : \Sigma_{Hss} \rightarrow \Sigma_{APHss}$ is an refinement.

CELL module, *CARR* module, and *CAPHSS* module are the following modules:

```
mod* CELL {
  pr(DATA)

  *[ Cell ]*
  bop view_ : Cell -> DBool
  bop set : DBool Cell -> Cell

  var B : DBool
  var C : Cell
  eq view set(B, C) = B .
}

mod* CARR {
  pr(DATA)
  pr(CELL)

  *[ CArr ]*
  bop get : Int CArr -> DBool
  bop put : DBool Int CArr -> CArr

-- projection operator
  op cell : Int CArr -> Cell

  vars I J : Int
  var B : DBool
  var A : CArr

-- conditional composition definitions
  eq get(I, A) = view cell(I, A) .
  ceq cell(I, put(B, J, A)) = set(B, cell(I, A))
      if I == J .
  ceq cell(I, put(B, J, A)) = cell(I, A)
      if I =/= J .
}

mod* CAPHSS {
  pr(DATA)
  pr(CARR)

  *[ CAPHss ]*
  bop get_ : CAPHss -> DBool
  bop put : DBool CAPHss -> CAPHss
```

```
  bop rest_ : CAPHss -> CAPHss

-- projection operators
  op carr_ : CAPHss -> CArr
  op ptr_ : CAPHss -> Int

  var S : CAPHss
  var B : DBool

-- composition definitions
  eq get S = get(ptr S, carr S) .
  eq ptr put(B, S) = s ptr S .
  eq carr put(B, S) = put(B, s ptr S, carr S) .
  eq ptr rest S = p ptr S .
  eq carr rest S = carr S .
}
```

Secondly, we prove that the inclusion $\varphi_2 : \Sigma_{APHss} \to \Sigma_{CAPHss}$ is an refinement, by showing that *CARR* module is a corresponding composition module of *ARR*. This process is as follows:

```
--> Verifying refinement from ARR to CARR
open .
ops i j : -> Int .
op e : -> DBool .
op a : -> CArr .

--> ceq get(I, put(B, J, A)) = B if I == J .
red get(i, put(e, i, a)) == e .
--> ceq get(I, put(B, J, A)) = get(I, A) if I =/= J .
red get(i, put(e, j, a)) == get(i, a) .
close
```

The result is as follows:

```
--> Verifying refinement from ARR to CARR
-- opening module CARR.. done.
--> ceq get(I, put(B, J, A)) = B if I == J ._*
-- reduce in % : get(i,put(e,i,a)) == e
true : Bool
(0.000 sec for parse, 6 rewrites(0.017 sec), 12 match attempts)
--> ceq get(I, put(B, J, A)) = get(I, A) if I =/= J .
-- reduce in % : get(i,put(e,j,a)) == get(i,a)
true : Bool
(0.017 sec for parse, 5 rewrites(0.000 sec), 16 match attempts)
```

Because each execution of *red* command returns *true*, *CARR* module is a corresponding composition module of *ARR*. Therefore, the inclusion $\varphi_2 : \Sigma_{APHss} \to \Sigma_{CAPHss}$ is an refinement. From Property 28, the inclusion $\varphi_2\ \varphi_1 : \Sigma_{Hss} \to \Sigma_{CAPHss}$ is an refinement, too. $\Box$

As to behavioural equivalence of corresponding composition modules, the following theorem holds.

Theorem 30 *Let PM be a primitive module and CM be a corresponding composition module of PM. Let R be the simple form of behavioural PM-equivalence generated by GSB-algorithm. Then, R is a simple form of behavioural CM-equivalence (regarding $h_{CM} = h_{PM}$ where h_{CM} (h_{PM}) is the sort of CM (PM), respectively), too.*

Proof : Because $AM_{CM} = AM_{PM}$ (regarding $h_{CM} = h_{PM}$ where h_{CM} (h_{PM}) is the sort of CM (PM), respectively) and there are no behavioural operators without attributes and methods in CM, the set of all sequences of behavioural operators of CM (which can be regarded as observational CM-contexts) coincides with those of PM. There is a refinement $i : (H_{PM}, \Sigma_{PM}, AE_{PM}) \to (H_{CM}, \Sigma_{CM}, AE_{CM})$ and this refinement is an inclusion. So, $M' \models\!\equiv_{\Sigma_{CM}} E_{PM}$ for each (Σ_{CM}, AE_{CM})-model M'. Therefore, we can construct an ECRS of observational CM-contexts from E_{PM}. Consequently, the simple form of behavioural CM-equivalence generated by *GSB*-algorithm using this ECRS coincides with R. $\Box$

Example 29 *APHSS (continued)*
From Theorem 30, a simple form $\equiv_{APHSS}$ of behavioural *APHSS*-equivalence is that
$(s\ \equiv_{APHSS}\ s') = \bigwedge_{i \in Nat}(get\ rest^{(i)}[s] == get\ rest^{(i)}[s'])$.
$\Box$

Chapter 8

Related Work

One topic of behavioural semantics — especially, hidden algebras — is a generalization of process algebra [Hoa85, Mil89, BW90]. As to this topic, there is a research by Dr.Goguen and Dr.Malcolm (abbreviate *GM* group) [GM97]. Also, there are researches about hidden algebras themselves [GM97, MG96].

Another topic of behavioural semantics is verifications of refinement from abstract specifications to concrete specifications. As to this topic, there are researches by Dr.Bidoit, Dr.Hennicker et al (abbreviate *BH* group) [Hen90, GP91, BH94, BH96] and researches by *GM* group [GM97, MG96].

BH group researches refinement from abstract specifications to implementations (concrete specifications). For example, context induction [Hen90, GP91], and the method using partial congruences [BH94, BH96].

On the other hand, *GM* group researches refinement from abstract behavioural specifications to concrete behavioural specifications as restriction of models which satisfy specifications [GM97, MG96]. But, these researches are not satisfactory.

In this chapter, we describe the above researches more detail.

8.1 Context Induction

The first verification method of behavioural properties is context induction.

Algorithm 4 *Consider verification of a behavioural property* $s \equiv s'$. **The algorithm of context induction** *is as follows:*

1. *prove that* $at[s] == at[s']$ *for each attribute* at*,*

2. *prove that* $at\ mc\ [s] == at\ mc\ [s']$ *for each attribute* at *and each method context* mc. □

The sort of each *mc* is a hidden sort. So, they thought that any induction hypothesis could not use [GP91]. Note that context induction is not induction over length of contexts.

Example 30 *IHSS* and *IARR*
IARR is a specification of an array and *IHSS* is a specification of an implementation of *HSS* using an array and a pointer. Note that semantics of each module is initial semantics (`mod!`).

```
mod! IARR {
  pr(DATA)

  [ Arr ]
  op get : Int Arr -> DBool
  op put : DBool Int Arr -> Arr

  vars I J : Int
  var B : DBool
  var A : Arr
  ceq get(I, put(B, J, A)) = B
      if I == J .
  ceq get(I, put(B, J, A)) = get(I, A)
      if I =/= J .
}

mod! IHSS {
  pr(DATA)
  pr(IARR)

  [ Hss ]
  op get_ : Hss -> DBool
  op put : DBool Hss -> Hss
  op rest_ : Hss -> Hss

  op _||_ : Int Arr -> Hss

  var I : Int
  var A : Arr
  var B : DBool
  eq get(I || A) = get(I, A) .
  eq put(B, I || A) = s I || put(B, s I, A) .
  eq rest(I || A) = p I || A .
}
```

Consider to prove a property $(rest\ put(t, S)) \equiv (S)$ in IHSS. The process of context induction is as follows [GP91]:

```
--> Prove (rest put(t, S)) Reqv (S) .
-->   at[rest put(t, S)] == at[S]
```

```
open .
red get rest put(t, I || A) == get (I || A) .
close

-->   at mc[rest put(t, S)] == at mc[S]
open .
op mc_ : Hss -> Hss .
red get mc rest put(t, I || A) == get mc (I || A) .
close
```

The result is as follows:

```
--> Prove (rest put(t, S)) Reqv (S) .
-->   at[rest put(t, S)] == at[S]
-- opening module IHSS.. done.
-- reduce in % : get (rest put(t,I || A)) == get (I || A)
true : Bool
(0.000 sec for parse, 8 rewrites(0.017 sec), 19 match attempts)
-->   at mc[rest put(t, S)] == at mc[S]
-- opening module IHSS.. done._*
-- reduce in % : get (mc (rest put(t,I || A))) == get (mc (I || A)
    )
false : Bool
(0.017 sec for parse, 4 rewrites(0.000 sec), 8 match attempts)
```

`red get mc rest put(t, I || A) == get mc (I || A) .` returns *false*. So, to prove the property, we use case analysis as follows:

```
--> (1) mc = z
open .
op mc_ : Hss -> Hss .
red get rest put(t, I || A) == get (I || A) .
close
--> (2) mc = put(B) mc
-->     lemma: get put(B, S) = B
open .
red get put(B, I || A) == B .
close
-->     Prove mc = put(B) mc with the above lemma
open .
op mc_ : Hss -> Hss .
var S : Hss .
eq get put(B, S) = B .
red get put(B, mc rest put(t, I || A)) == get put(B, mc (I || A)) .
close
```

```
--> (3) mc = rest mc
open .
op mc_ : Hss -> Hss .
red get rest mc rest put(t, I || A) == get rest mc (I || A) .
close
```

The result is as follows:

```
--> (1) mc = z
-- opening module IHSS.. done._*
-- reduce in % : get (rest put(t,I || A)) == get (I || A)
true : Bool
(0.017 sec for parse, 8 rewrites(0.017 sec), 19 match attempts)
--> (2) mc = put(B) mc
-->     lemma: get put(B, S) = B
-- opening module IHSS.. done.
-- reduce in % : get put(B,I || A) == B
true : Bool
(0.000 sec for parse, 6 rewrites(0.000 sec), 10 match attempts)
-->     Prove mc = put(B) mc with the above lemma
-- opening module IHSS.. done._*
-- reduce in % : get put(B,mc (rest put(t,I || A))) == get put(B,
    mc (I || A))
true : Bool
(0.033 sec for parse, 6 rewrites(0.000 sec), 12 match attempts)
--> (3) mc = rest mc
-- opening module IHSS.. done._*
-- reduce in % : get (rest (mc (rest put(t,I || A)))) == get (rest
     (mc (I || A)))
false : Bool
(0.033 sec for parse, 4 rewrites(0.000 sec), 10 match attempts)
```

`red get rest mc rest put(t, I || A) == get rest mc (I || A) .` returns *false*. So, we need more case analysis, to prove the property. As discussed in Example 16, a simple form $\equiv$ of behavioural equivalence is that: $(s \equiv s') = \bigwedge_{i \in Nat}(get\ rest^{(i)}[s] == get\ rest^{(i)}[s'])$. Therefore, these case analyses continues forever. $\square$

In view of induction over length of contexts, the problem of context induction (Example 30) is that for proving the n-th step, the $n+1$-th step is necessary. So, induction over length of contexts does not have this problem.

8.2 Finding hidden congruences

The method using partial congruences and coinduction are similar. Consider to verify refinement from a specification Σ to a specification Φ ($\Sigma \subset \Phi$). Partial congruences correspond to Σ-behavioural equivalence or Φ-behavioural equivalence on hidden Σ-algebras which are also hidden Φ-algebras. From this fact, *GSB*-algorithm of test set coinduction is useful for the method using partial congruences.

Users must give hidden congruences to coinduction, or partial congruences to the method using partial congruences. In [BH94, BH96, GM97], firstly, users select $\bigwedge_{A \in AllAttr}$ $(A[s] == A[s'])$ where *AllAttr* denotes the set of all attributes, as a candidate of hidden congruences. If this candidate is not a hidden congruence, then users should find another hidden congruence. A sufficient condition — Δ/Γ-complete — that this candidate coincides with a hidden congruence is given in [GM97]. Heuristic methods to find partial congruences are given in [BH96].

8.3 Refinement

GM group researches refinement from abstract behavioural specifications to concrete behavioural specifications as restriction of models which satisfy specifications [GM97, MG96]. Their method is as follows:

Example 31 *GARR* and *GHSS*
GARR module and *GHSS* module are the following modules:

```
mod* GARR {
  pr(DATA)

  *[ Arr ]*
  bop get : Int Arr -> DBool
  bop put : DBool Int Arr -> Arr

  vars I J : Int
  var B : DBool
  var A : Arr
  ceq get(I, put(B, J, A)) = B
      if I == J .
  ceq get(I, put(B, J, A)) = get(I, A)
      if I =/= J .
}

mod* GHSS {
  pr(DATA)
  pr(GARR)

  *[ Hss ]*
  bop get_ : Hss -> DBool
  bop put : DBool Hss -> Hss
  bop rest_ : Hss -> Hss
```

```
  bop _||_ : Int Arr -> Hss

  var I : Int
  var A : Arr
  var B : DBool
  eq get(I || A) = get(I, A) .
  eq put(B, I || A) = s I || put(B, s I, A) .
  eq rest(I || A) = p I || A .
}

mod* PROOF {
  pr(GHSS)

-- hidden congruence
  op _R_ : Hss Hss -> Bool

  vars I I1 I2 : Int
  vars A A1 A2 : Arr
  eq (I || A) R (I || A) = true .
  eq (I1 || A1) R (I2 || A2) = I1 == I2 and
                               get(I1, A1) == get(I2, A2) and
                               (p I1 || A1) R (p I2 || A2) .
}
```

They proved equations of *HSS* on the states in the form (`I || A`) by using coinduction with this R. This means that they treat Hss sort as the set of the states in the form (`I || A`). But, it is not true. There are states except (`I || A`). Note that equations in *GHSS* are defined on the states in the form (`I || A`). So, for states except (`I || A`), there is no equation. This means that there is no refinement from *HSS* to *GHSS*. □

The reason of this problem (Example 31) is that there are states except (`I || A`). By using projection operators, we eliminate these strange states as in Chapter 7.

They may give a new semantics, that are given by all models whose carriers are constructed from states in the form (`I || A`). Note that states in the form (`I || A`) are inductively defined. So, to prove equations of *HSS*, induction and some lemmas are necessary [GM97]. On the other hand, our method in Chapter 7 does not need induction and any lemmas.

8.4 Projection Operator

As to pseudo-projection operators (we call these operators projection operators in [IMD$^+$, DF98]), there are co-operative researches with Mr.Iida, Dr.Diaconescu, and Dr.Lucanu [IMD$^+$, DF98]. We only wrote our contribution in this thesis. Their interest is to specify dynamic systems using pseudo-projection operators. By changing contents of *ObjId* dynamically, we can specify dynamic systems.

Chapter 9

Conclusion

9.1 Conclusion

We researched verification methods for behavioural specifications. Concretely, we provided test set coinduction, object-oriented specification, and the stepwise refinement methods of object-oriented specifications.

To use coinduction, users must find the simple form of behavioural equivalence. In test set coinduction, this simple form is automatically generated by *GSB*-algorithm. By analysing the structure of the set of all visible contexts, we show the sufficient condition that the simple form generated by *GSB*-algorithm is the simplest form.

Until now, coinduction was regarded more efficient than induction over length of contexts [GM97, BH94, BH96]. By analysing the structure of the set of all visible contexts, we show the case that coinduction (test set coinduction) coincides with induction over length of contexts.

As to research of stepwise refinements of behavioural specifications as restriction of possible implementations, there are researches [GM97, MG96]. But, these are not satisfactory. Firstly, they give the original specification (for example, a stack). Then, they construct it from primitive modules (for example, an array and a pointer) in the refined specification. Finally, they prove that the composed module (for example, a stack constructed from the array and the pointer) satisfy the original specification. In the last process, they treat the composed module as data values. But, in behavioural specification, specifications of systems must be treated as black boxes.

To specify the composed module as a black box, we provided object-oriented specifications composing by projection operators. Then, we provided the method to verify stepwise refinement of object-oriented specifications.

9.2 Acknowledgments

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 Associate Professor Takuo Watanabe, Associate Kazuhiro Ogata, and Associate Răzvan Diaconescu. Next, I thank LDL members, especially Mr. Iida, for helpful discussions about hidden algebras. Finally, I thank my employer PFU Limited, which supports my life at JAIST.

Bibliography

[BD92] Rod Burstall and Răzvan Diaconescu. Hiding and behaviour: An institutional approach. In A.W. Roscoe, editor, *A Classical Mind Essays in Honour of C.A.R. Hoare*, chapter 5, pages 75–92. Prentice Hall, 1992.

[BH94] Michel Bidoit and Rolf Hennicker. Proving behavioural theorems with standard first-order logic. In *4th International Conference, ALP'94*, number 850 in LNCS, pages 41–58. Springer-Verlag, 1994.

[BH96] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165:3–55, 1996.

[Bou97] Adel Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[BW90] J.C.M. Baeten and W.P. Weijland. *Communication and Concurrency.* Prentice Hall, 1990.

[DF96] Răzvan Diaconescu and Kokichi Futatsugi. Logical semantics for CafeOBJ. Technical Report IS-RR-96-0024S, Japan Advanced Institute of Science and Technology (JAIST), 1996.

[DF98] Răzvan Diaconescu and Kokichi Futatsugi. CafeOBJ *Report.* AMAST. World Scientific, 1998. To appear.

[FGJM85] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[FS95] Kokichi Futatsugi and Toshimi Sawada. Design considerations for Cafe specification environment. In *The 10th Anniversary of OBJ2*, 1995.

[Fut97] Kokichi Futatsugi. An overview of cafe specification environment — an algebraic approach for creating, verifying and maintaining formal specifications over the net —. In *First IEEE International Conference on Formal Engineering Methods.* IEEE, 1997.

[GB92] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1), 1992.

[GD94] Joseph Goguen and Răzvan Diaconescu. An Oxford survey of order sorted algebra. *Mathematical Structures in Computer Science*, 4(4), 1994.

[GM92] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2), 1992.

[GM97] Joseph A. Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, UCSD Technical Report, 1997.

[Gog] Joseph A. Goguen. Theorem proving and algebra. To appear.

[GP91] Marie-Claude Gaudel and Igor Privara. Context induction: an exercise. Technical Report 53, PDCS, 1991.

[GWM+93] Joseph Goguen, Timothy Winkler, José Mesegure, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. Technical report, SRI International, Computer Science Laboratory, 1993.

[Hen90] 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.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[IMD+] Shusaku Iida, Michihiro Matsumoto, Răzvan Diaconescu, Kokichi Futatsugi, and Dorel Lucanu. Concurrent object composition in CafeOBJ. To appear.

[Klo92] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford Science Publications, 1992.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 93:73–155, 1992.

[Mes93] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, 1993.

[MG96] Grant Malcolm and Joseph A. Goguen. Proving correctness of refinement and implementation. Technical Report PRG-114, Oxford University Computing Laboratory Technical Monograph, 1996.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[SF95] Toshimi Sawada and Kokichi Futatsugi. Basic features of CHAOS specification kernel language. In *The 10th Anniversary of OBJ2*, 1995.