Inheritance and OMT: a CSP approach

by

Chan Wing Kwong
陳 榮 光

A thesis
presented to The University of Hong Kong
in fulfillment of the
thesis requirement for the degree of
Master of Philosophy
in
Computer Science
September 1995

Abstract of thesis entitled "Inheritance and OMT: a CSP approach"
submitted by Mr Chan Wing Kwong
for the degree of Master of Philosophy
at The University of Hong Kong in September 1995

Much research has been conducted on the formulation of inheritance in object-orientedness. Most proposals retain the semantic relationship between the superclass and subclass, but deliberately ignore the issues of overriding, which plays an important role in object-oriented modelling. Others use class inheritance and module inheritance to distinguish between the cases of inheritance with and without overriding, but, as a result, one fundamental concept is partitioned into two. Some research supports overriding, but not transitivity. This thesis addresses the issue of overriding in object-orientedness, illustrates how a single inheritance concept can be formalized using Communicating Sequential Processes (CSP), proposes a refined concept of conformance which is transitive, and demonstrates the practical application of our theory. This project compares favourably with other related work.

In this thesis, we identify that reflexivity, anti-symmetry, transitivity, overriding, behavioural compatibility, formality, incrementality and multiple inheritance are the desirable features of a theory of inheritance. We formalize the concept of behavioural inheritance as a special kind of conformance relation that links a subclass to its superclass by the Fidge's priority choice operator. Using the priority choice operator, we integrate naturally the concept of overriding into our formalization and provide a support for an incremental development. Transitivity is realized by the introduction of strong conformance which can be used as a unification of extension and reduction. It also provides a framework of reflexivity and anti-symmetry for the inheritance relation. Through strong conformance, behaviour compatibilities between the superclasses and subclasses can be ensured. We also refine our theory to support multiple inheritance and propose a method to resolve ambiguity raised by multiple inheritance. Finally, a criteria for generating an optimal incremental change will be presented.

ACKNOWLEDGMENTS

It is a great pleasure to acknowledge the profound and unusual work of my supervisor Dr T.H.Tse, M.B.E., the exciting insights in his pioneer work on *methods integration*, and his professional enthusiasm in the persuade of knowledge. Without his influence, his support, and his kind and interesting discussions and personal friendship, I can hardly imagine the sunshine and the fresh air flourished in this project.

Sincere thanks should equally and dedicatedly be given to Dr T.Y. Chen, Dr F.C.M. Lau and Dr David W. Cheung for their kind but formal and justifiable appraisal of the research.

I am glad to have a chance to enjoy the warmth and care of the *The Software Engineering Group, Department of Computer Science, The University of Hong Kong* where colleagues mean friends: Professor H.Y. Chen, Dr H. Yan, Mr W.T. Yeung, Mr R.K.P.H. Leung, Mr J.W.F. Cheng, Mr P. Ng, Mr Z. Xu, Mr S.N.P. Suen, Ms I. Kwan, Mr F.T. Chan and Mr H.C.P. Cheng. The happiness and the experience are ever-lasting.

I should thank Ms Maria Lam, Ms Smitty Lam, Ms Brenda Ng, Ms Joanne Chan, Ms Priscilla Chung, Ms Veronica Yim, Ms Irene Wai to maintain an efficient departmental general office, Mr William Chan, Mr Daniel Hung, Mr William Sin, Mr K.M. Fok, Mr W.S. Chan and Mr C.K. Lee to provide a unique, qualified, special and stable computing environment.

I would like to thank Mr H.C. Law, Mr Y.M. Sun, Mr S.B. Lai, Mr S.L. Leung, Mr P.C. Chee, Mr C.T. Kwan, Mr W.K. Tam, Mr W.T. Chan, Ms W.H. Wong, Ms F.L. Yue, Mr W.K. Sung, Mr H.S. Loong, Mr Y.P. Yau, Mr S.M. Yiu and other postgraduate students for their share of my enjoyment including tea and jokes.

Lastly, I would like to express my heartfelt gratitude to my parents, Mr P. Chan and Ms S.K. Shing, for their endless concern, patience, encouragement, love and care throughout the life of the project.

DECLARATION

I hereby declare that this thesis represents my own work which has been done after I registered for the Degree of Master of Philosophy in July 1993 and that it has not been previously included in a thesis, dissertation or report submitted to this or any other institution for a degree, diploma or other qualification.

Chan Wing Kwong
Sept 1995

I dedicate this thesis to
My Parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Forget a while about software engineering and look around other disciplines. Think about an electronic engineer designing a circuit board and a mechanical engineer building an engine and an automobile expert innovating a new car model. The electronic engineer should firstly capture requirements like voltages, current and logics required, then draw various circuit diagrams, analyze whether those alternatives meet the requirements based on analogue and digital electronics, build prototypes and finally fabricate boards. Likewise, the mechanical engineer captures requirements like the torque required to deliver, the pressure to withstand and the power of the engine, then draws blueprints to describe design alternatives, analyzes those blueprints based on theories like mechanics and elasticity, constructs prototypes for testing and finally builds engines. Paradoxically, the automobile expert follows a strikingly similar methodology which makes use of diagrams and theory in the course of design.

Information system development should be no exception. It should be practiced as an engineering process: designers capture the real world, analyze the model and propose feasible alternatives, and finally engineer them. Tools like diagrams and theories should be employed.

The so-called informal methodologies like [Yourdon89] and [Rumbaugh91] play a dominate role in software development. One of their characteristics is they tend to use ambiguous graphical notations to draw diagrams to represent a software system. It may result in anomaly, inconsistency and ambiguity. For instance, Figure 1.1 shows a scenario that a manufacturer produces wooden furniture and paper from fir and the paper is consumed by Press to print newspapers.

It is not clear whether the manufacturer should finish all its jobs first or not. For example, two distinct and yet possible low-level diagrams are shown in Figure 1.2 and Figure 1.3. It illustrates that a false perception may result from looking at the high-level diagrams without examining details.

On the other hand, formal methods are unambiguous, consistent and precise. For instance, Figure 1.4 and Figure 1.5 shows the corresponding CSP specifications (which will be introduced in Appendix A) for the diagrams in Figure 1.2 and Figure 1.3 respectively.

Formal methods are mathematical-based techniques. We can reason and prove the system. For instance, we can reason formally that Press cannot start publishing newspapers before there is fir supply.

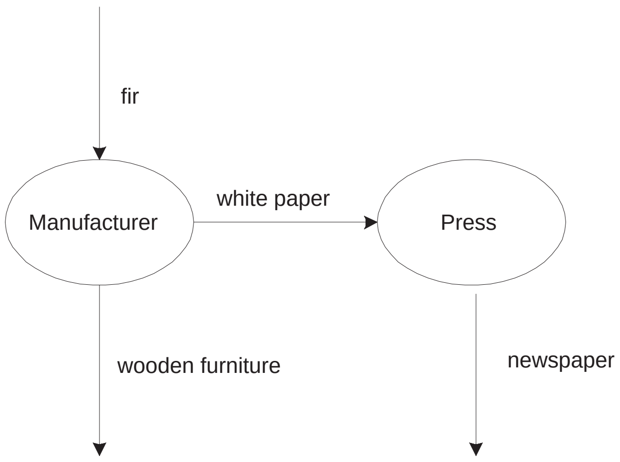

Figure 1.1: Manufacturer and press.

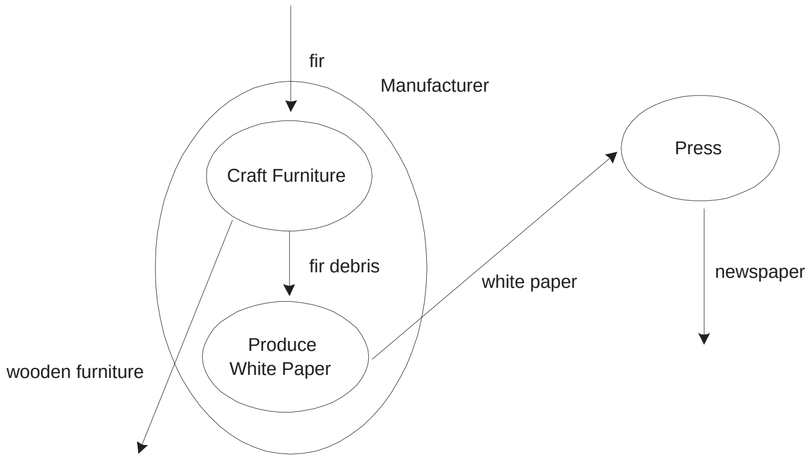

Figure 1.2: Manufacturer before press.

Nevertheless, formal methods are compact and full of jargon, which means that they are more difficult than informal methodologies for lightly grasping the overview of the system. For instance,

$$\text{Second Refinement} = \text{MANUFACTURER} \parallel \text{PRESS}$$

should provide less information than the diagram shown in Figure 1.1.

To align with other engineering disciplines, an unambiguous graphical methodology seems to be a solution to bridge the gap. For instance, designers can use their familiar tools like Data Flow Diagrams, State-Transition Diagrams and Entity-Relationship Diagrams to specify systems and translate their specification into formal methods for rigorous reasoning.

In fact, bridges exist. Tse [Tse91] integrated structured analysis and design [Yourdon89, DeMarco79] with initial algebra and category theory. Pandya and Hoare [Pandya90] used Asynchronous Communicating

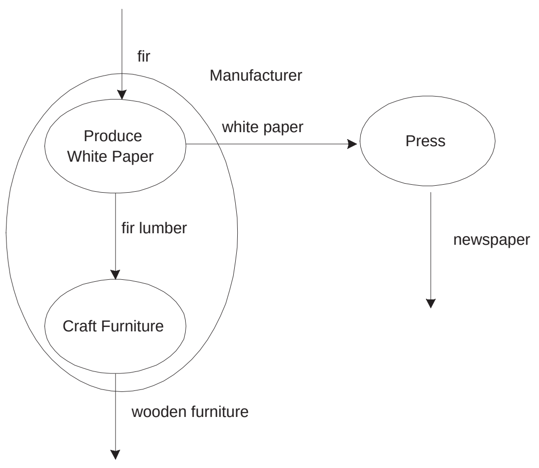

Figure 1.3: Press independent of craft furniture.

First Refinement = MANUFACTURER $\|$ PRESS

MANUFACTURER = fir $\rightarrow$ (wooden furniture $\rightarrow$ **SKIP**)
$\|$
(fir debris $\rightarrow$ paper $\rightarrow$ **SKIP**)

PRESS = paper $\rightarrow$ newspaper $\rightarrow$ **SKIP**

Figure 1.4: Corresponding CSP specification for the first refinement.

Sequential Processes (ACSP) [Hoare89] to support Jackson System Development (JSD) [Jackson82]. Bryant [Bryant90] employed Z [Spivey92] to synthesize and investigate the structured methodologies. As the object-oriented modelling flourishes, there are more and more efforts on integrating the object-oriented paradigm with formal method. For instance, Clark [Clark93] proposed *Rigorous Object Oriented Analysis* with a formal language called LOTOS [ISO8807]. Jacobson [Jacobson92] suggested new graphical notations with logic programming as the backbone.

1.2 Theme

The inheritance formalization plays an important role in these object-oriented methods with formal backbones. Nevertheless, as shown in the review in Chapter 8, they do not simultaneously support overriding, transitivity and behaviour compatibility . This research aims at developing a unified inheritance theory using process algebra so that a more rigorous bridge can be provided for software engineering.

1.3 Organization of this thesis

We firstly introduce inheritance and investigate the desirable features of an ideal theory on inheritance in Chapter 2. It establishes a basis for our own theory and provides a reference point for comparisons. Chapter

```
Second Refinement = MANUFACTURER || PRESS

MANUFACTURER = fir → (fir lumber → wooden furniture → SKIP)
                     ||
                     (paper → SKIP)

PRESS = paper → newspaper → SKIP
```

Figure 1.5: Corresponding CSP specification for the second refinement.

3 justifies the formal language — *Communicating Sequential Processes* (CSP) that we have chosen. In Chapter 4, we analyze problems in inheritance and propose three concepts as our tools for formalizing inheritance. The theory will then be presented in Chapter 5. An account example will be presented in Chapter 6 to demonstrate the practical aspect of our theory. We shall compare our result with those of CSP-related projects in Chapter 8. Finally, we would draw a conclusion and profile future research.

We assume in this thesis that readers are familiar with the basic concepts of CSP. Please refer to Appendix A for a brief summary, and to Hoare [Hoare85] for further details.

Chapter 2

Desirable features of a theory of inheritance

In the last chapter, we propose to practice software development as an engineering process. However, the concept of inheritance is poorly defined. In this chapter, we aim at identifying features of inheritance theory that our formalism should support. Firstly, we will introduce the concept of object-orientedness and, in particular, inheritance. Then, we will identify the desirable features of a theory of inheritance.

2.1 Object-orientedness and inheritance

2.1.1 The flourish of object-orientedness

Object-oriented methodologies are direct extensions from object-oriented programming. However, they were not mature and popular until the late 80s, when software systems were becoming too complex and too large to be managed and understood using top-down structured methodologies like [Yourdon89] and [Jackson82]. The philosophy of structured methodologies is that, to perform a task/function, we decompose it into smaller sub-tasks/sub-functions and repeat this process until the task/function is simple enough to be implemented right away.

This function-oriented viewpoint of system development incurs additional challenge to software maintenance. For instance, to air-freight a shipment from New York to Beijing, the forwarding agent should book cargo space with airlines, calculate charges and prepare shipping documents. A data flow of the simplified air-freight operation is shown in Figure 2.1. To attract business, a new method of charges calculation is made. For example, a prepayment will enjoy discount. To accommodate this change, the shipping documents and the method of calculation should take the prepayment option into account. In other words, this piece of information, prepayment option, should become an input of Calculate Charges and Generate Shipping Document. However, this information should come from Book Space and hence a change in Book Space is required. Imagining a complex system of over 1,000 processes and the influence of each design decision potentially ripples through the whole system, which we can hardly manage.

To tackle the problem of unmanageability of structured methodologies, people start to shift the emphasis from function-orientedness to object-orientedness, which breaks up the system and views components as individual *objects*. Besides, object-oriented analysis and design give the development process a more stable base. For instance, the whole development is based on a single unifying software concept − what objects are rather than how it is used − throughout the development process. As pointed out in [Rumbaugh91] and [Booch94], this type of objects are more stable than their function-oriented counterparts in the long run.

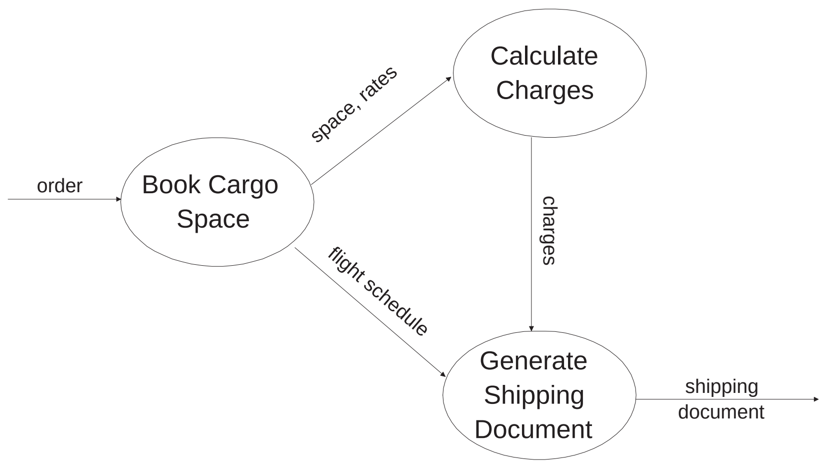

Figure 2.1: A data flow diagram for air-freight forwarding.

A bank account example

To serve as an illustration, we introduce a bank account example and explain the object-oriented features and inheritance properties through the use of this example.

Every bank account should have a a *balance*. We can perform transactions like *withdraw* money and *deposit* cash. To attract more capital, the banker introduces accounts with interest. It is done through *calculating interest* and then depositing the amount into the savings accounts. For instance, *John* holds a savings account which has a balance of $3,000. In this case, an interest of $150 payable to him at the end of a year. On the other hand, a credit account requires the account holder to pay service charge for any overdraft. We picture the scenario in Figure 2.2.

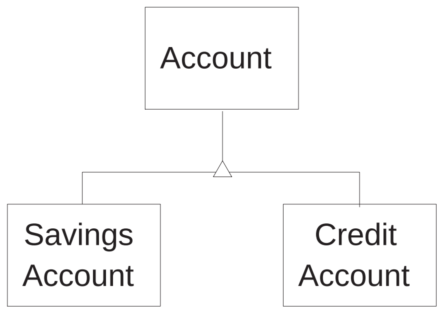

Figure 2.2: Bank accounts.

Object and class

In object-orientedness, a system is composed of a number of autonomous entities communicating with one another by the exchange of messages. These entities are called *objects* and similar objects sharing the same infra-structure are represented by a template called *class*. For instance, an account may be abstracted as an entity with elements like balance, withdraw money and deposit cash. The data element *balance* is called an *attribute* of *Account*. And the function elements, *withdraw* and *deposit*, are called *methods* of *Account*.

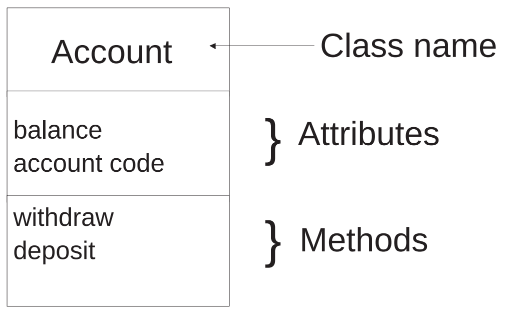

Figure 2.3: Class with attributes and methods.

Encapsulation

To separate the external aspects of an object from the internal details of classes, attributes are protected from arbitrary access. They can only be modified or inquired through the methods of the class. This feature is called *Encapsulation*. According to [Booch94], ”encapsulation is the process of compartment the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction from its implementation.” For instance, if we want to deposit money into a bank account, we should apply the *deposit* method of *Account* rather than a direct increment of the attribute *balance*. (In our abstraction, it means the balance of the account is increased by the amount of deposit.)

Polymorphism

Another object-oriented concept is *polymorphism*. It refers to operations sharing the same calling interface, yet performing distinct functions (in different classes). For instance, *Account* will not charge customers when he *withdraw*s a large amount of money. However, customers are required to pay a service charge for each withdrawal transaction on a *Credit Account*.

2.2 Inheritance

Inheritance links a class with its refined version. The refined class is called the *subclass* and the class being refined is called the *superclass*. A class X inherits from another class Y if X is a subclass of Y having special properties. For instance, *Savings Account* can be made to be a subclass of *Account*, so that it has access to all the operations and attributes defined on *Account* like *balance and withdraw* and the special methods of its own like *calculate interest*.

A subclass may also redefine some methods inherited from the superclass. For instance, as shown in Figure 2.4, *withdraw* is redefined in *Credit Account*, superseding that inherited from *Account*. It means whenever *withdraw* is activated in *Credit Account*, only the new definition (the one defined in *Credit Account*) will be applied. This feature is called *Overriding*.

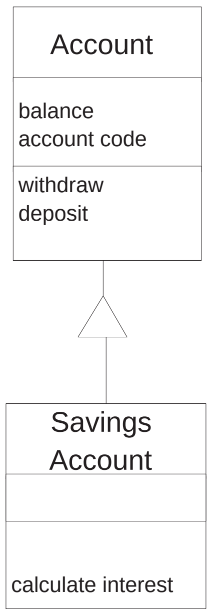

Figure 2.4: Inheritance: Savings Account as a subclass of Account.

However, overriding is too powerful if one may inherit any class and override all inherited properties. For example, one may inherit a *stack* to form a *queue* by redefining the *push* operation to append a new element at the end of a list rather than at the head position. This kind of *ad hoc* overriding does not respect

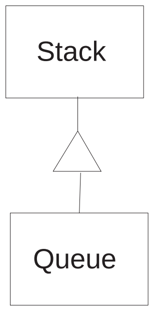

Figure 2.5: An abuse of inheritance.

the semantics relationship between the superclass and subclass. One inherits other classes merely for the convenience of implementation. Hence, it is referred to as the inheritance of implementation. A rule of thumb is never to override unless it is consistent with the original semantics and signature [Rumbaugh91].

2.3 Desirable features

2.3.1 General metrics for a theory of inheritance

Based on the above general understanding of inheritance, we propose the following as the general desirable features of a theory of inheritance.

Behaviourally compatible We would expect the behaviour among subclasses and superclasses to be compatible, particularly during analysis, so that we can apply our knowledge gained from the superclass to its subclasses. Hence, an instance of a class (*Savings Account*) is a polymorphic instance of its superclass (*Account*).

Incremental Features like attributes and operations of a superclass are shared by a subclass. For example, *Savings Account* inherits the attribute *balance* and the operations *withdraw* and *deposit* from *Account*. Subclasses not only inherit features from a superclass, but also add new or redefine existing features. For example, *Savings Account* adds a new operation *calculate interest* which is not available in *Account*.

Automatically propagated Any change or addition of features of the superclass should be automatically propagated to its subclass. For instance, if we modify the *withdraw* operation of the *Account*, the effect of this change should automatically be reflected on *Savings Account*. When adding a new attribute *currency* to *Account*, the new feature should automatically be a feature of *Savings Account*, unless it is explicitly suppressed.

Redefinable The inheritance machanism should support overriding.

Reflexive In theory, a class may inherit itself [1].

Anti-symmetric A class should not simultaneously be a superclass and a subclass of another distinct class. For instance, we do not expect *Savings Account* to be the superclass of *Account*.

Transitive The inheritance relation is transitive. It refers to the class membership properties among objects in the inheritance class hierarchy. An instance of a class should simultaneously be an instance of *all* its (direct or indirect) superclasses. For example, *Vantage Account* is a subclass of *Savings Account* and an instance of *Account*.

2.3.2 Formalism

We would like to provide a formalism for inheritance. We feel that the tool itself should be clear and well-defined; otherwise designers can hardly use it effectively for modelling. For instance, we would like to know which *withdraw* operation of the *Credit Account* is called without ambiguity. We should further want to know whether two classes can form an inheritance relation while preserving the semantics and compatibility.

2.4 A summary of desirable attributes

An inheritance mechanism for analysis should be behaviourally compatible, easily modified, formal, incremental, redefinable, reflexive, anti-symmetric and transitive. In Chapter 8, we will compare our result with some related work.

[1]However, the inheritance mechanism will degenerate to a trivial case as if there were no such inheritance relation defined. In practice, designer would never define (or show) this kind of trivial inheritance explicitly.

Chapter 3

Justification of using CSP

Models essentially capture important characteristics (both structural and dynamic properties) of real world systems and these can then be understood by human through effective manipulation of components like decomposing them into simpler and smaller sub-models. In this research, we propose to use *Communicating Sequential Processes* (CSP) [Hoare85] to model inheritance, a major concept in object-orientedness. CSP, developed by Professor Tony Hoare, is a formal language specialized for concurrency study. A brief introduction of CSP can be found in Appendix A.

We choose CSP as our modelling language for various reasons as listed below.

(1) **Strong concurrency modelling**
Theoretically speaking, autonomous peer objects interact with one another in object-oriented systems. Process algebras like CSP [Hoare85], CCS [Milner86], Asynchronous CSP [Hoare89] and LOTOS [ISO8807], are particularly strong in modelling systems of concurrently communicating objects [Car90].

(2) **Popular formal language**
CSP, a dominant process algebra, has been widely and successfully described distributed communication protocols. This proven track record immensely distinguishes CSP from the rest.

(3) **Excellent reasoning power**
CSP has a formal syntax and semantics by which designers can unambiguously reason on a system. The language is simple and expressive enough to model a wide range of applications.

(4) **Enforcing class reuse discipline**
Class reuse is a key feature in object-oriented development. Enforcing the proper discipline in the description of classes greatly improves the re-usability of these classes.

(5) **Hardware support**
CSP has a successful hardware implementation — transputers. Specifications are ready to be transformed to *real* programs.

(6) **Resemblance between object-orientedness and CSP features**

   (a) Message passing, a mechanism for exchanging messages among objects, and synchronization, a mechanism for synchronizing events among processes, are interchangeable in almost all context.

   (b) The similarity between the notion of processes and the notion of objects drastically reduces the semantics gap between the two categories.

(c) The principle of information hiding [Par72] resembles that of abstraction in CSP. Both aim at defining a hidden and inaccessible infra-structure when necessary.

(d) Class identification is very similar to process equivalence. Classes in object-oriented systems are unique. Any two arbitrary processes of identical semantics have been proved to be equivalent and can be rewritten to an identical syntax.

(7) **State transitions and history**
An object changes states in response to receiving and sending messages, leaving its old states as history. In CSP, a process (in a state) exactly behaves as if it were another process (a new state) after synchronizing an event and records the action as its trace.

There are various research projects in translating CSP specifications to other models like Rewriting Logic [Meseguer92], or defining new CSP-based languages such as LOTOS [ISO8807]. We feel it would be more attractive to carry out our project in a language which has various formal links to other languages so that our findings can readily be applied elsewhere.

Chapter 4

Tools for formalizing inheritance

Having explored the desirable features of a theory of inheritance in Chapter 2, we aim at identifying the tools required to support these features. Firstly, we introduce a conformance relation to serve as a basis for our theory. We will then investigate its problems (with respect to the desirable features) and propose two auxiliary tools — priority choice operator [Fidge93] and weak conformance.

4.1 Cusack conformance

A concept of conformance in terms of CSP notions has been introduced in [Cusack91a]. We reproduce the formal definition below. Further explanation will be given after the definition.

Definition 1 (Cusack Conformance) *A process Q* conforms to *a process P if and only if*

(1) $\alpha\, Q = \alpha\, P$;

(2) $(s,X) \in \mathit{failures}(Q)$ *and* $s \in \mathit{traces}(\mathcal{P}) \Rightarrow (s,X) \in \mathit{failures}(P)$;

(3) $s \in \mathit{divergences}(Q) \cap \mathit{traces}(P) \Rightarrow s \in \mathit{divergences}(P)$.

The alphabet of a process P is denoted as αP. It is also known as the signature of the process. P can only communicate with other processes through event symbols defined in the alphabet. Hence, the first condition of the definition means that conforming processes should have identical signatures which serve as a basis for the comparison of traces, failures and divergences among them. A trace in CSP is a sequence of events. When these events are synchronized among processes, communications happen. We imagine processes in CSP being objects in object-oriented systems, and events being messages. The synchronization will then be the invocations of services (methods) of objects in a system. Thus, a trace of a process is actually a record of a sequence of operations acting upon an object. At the same time, we interpret a failure (s,X) of a process P as an object going into a state (P/s) that it cannot surly proceed further by communicating any message offered by the environment X. The second condition means that, given a common trace, a conforming process should have a smaller (if not the same) set of failures than the original process. The third condition distinguishes the chaotic situation from others. The set of divergences is a set of traces such that the process will diverge or become a chaotic situation where it cannot decide whether to accept or refuse any more event.

```
SimpleStack(⟨⟩) = push?x → SimpleStack(⟨x⟩)
SimpleStack(⟨x⟩ˆ y) = pop!x → SimpleStack(y)
                           □
                           push?z → SimpleStack(⟨z⟩ˆ⟨x⟩ˆ y)
```

Figure 4.1: A simple stack.

For instance, Figure 4.1 shows a CSP specification of a simple stack. If the stack is empty, it can only accept the push operation; otherwise we can apply the push or pop operations upon the stack.

A simple stack SimpleStack(⟨⟩), initially empty, undergoes a sequence of push and pop operations, say 3 pushes and then 3 pops. It will then come to a state that there is no immediate pop to be allowed. In other words, the current state of the stack can be represented as follows:

SimpleStack(⟨⟩)/⟨push?1,push?2,push?3,pop!3,pop!2,pop!1⟩ = SimpleStack(⟨⟩)

As the stack does not support the top operation, a failure associated with the trace ⟨push, push, push, pop, pop, pop⟩ will be

(⟨push, push, push, pop, pop, pop⟩, {pop,top})

(if top is in the alphabet of the stack). It means in this case that the stack cannot handle a further pop or top operation.

4.1.1 Redefinition of services

We intrepret the redefinition of services as the ability of a subclass in object-oriented systems to change the content of services which have been inherited from a superclass. We can further imagine a service consisting of a sequence of operations. Thus, we model it as a trace in CSP. A redefinition of service alters original traces to form new ones (or even new failures or divergences). The role of conformance is to define a semantics relation between the original and new traces so that the subclass can maintain compatible services with respect to the superclass.

Suppose a vending machine always gives a coke if a coin is inserted, and yet breaks down if two consecutive coins are accepted. The CSP definition is as shown below.

```
Coke Machine = coin → coin → STOP
                      □
                      coke → Coke Machine
```

We may want to improve the performance of the Coke Machine by replacing it by a new vending machine — Conforming Coke Machine. When two consecutive coins are inserted, the latter machine will return one coin instead of crashing.

```
Conforming Coke Machine = coin → μ X • ( coin → ( return coin → X)
                                         □
                                         coke → Conforming Coke Machine)
```

The original Coke Machine does not provide a coin rejection facility, and hence will crash after the sequence $\langle coin, coin\rangle$. A set of refusing communications (a refusal) associated with the trace $\langle coin, coin\rangle$ will be the set $\{$coin, return coin, coke$\}$. In other words, after inserting two coins, Coke Machine cannot handle any further request of accepting another coin, returning the second coin or even giving a bottle of coke. However, Comforming Coke Machine will return a coin if two consecutive coins are inserted. An associated refusal of the latter with respect to the trace $\langle coin, coin\rangle$ will be the set $\{$coin,coke$\}$. We can generalize the situation as a refusal of Coke Machine associated with the trace $t\hat{}\langle\text{coin,coin}\rangle$ for some trace t is $\{$coin, return coin, coke$\}$. Coke Machine will crash whenever there are two consecutive insertions of coins irrespective to the history. However, this set will never be a refusal of Conforming Coke Machine.

Furthermore, we can easily check that, according to the definition of conformance, Conforming Coke Machine conforms to Coke Machine.

4.1.2 Section summary

To summarize, a conforming process behaves polymorphically as the original process with respect to some common traces. It allows the redefinition of services in a conforming process. We feel it is strikingly similar to a subclass which behaves as if it were the superclass but overrided some inherited methods. Nevertheless, as we will present, the notion of Cusack conformance is not transitive. Moreover, Cusack conformance cannot the mathematical support incremental development.

4.2 Problems of Cusack conformance

4.2.1 Transitivity

An obvious problem in Cusack conformance is that it cannot guarantee transitivity, which can be demonstrated as shown in the following example.

Ideal Account An ideal account is an account that can perform debit and credit operations. Since it is ideal, no maintenance is required. We model it as IdealAccount. A debit in IdealAccount will first check the balance (checkBalance) and then either draw money (reduceBalance) or reject the operation (rejectDebitRequest). However, we may not know in advance when it rejects the request (and on what criteria it is based). We therefore model it as an internal decision (using the internal choice "⊓"). Similary, IdealAccount increases the balance (increaseBalance) whenever there is a credit request (credit). After performing either debit or credit operation, IdealAccount will then be ready to accept the next request. The CSP specification of IdealAccount is shown in Figure 4.2.

```
IdealAccount = ( debit → checkBalance → ( reduceBalance → IdealAccount
                                          ⊓
                                          rejectDebitRequest → IdealAccount)
                 □
                 credit → increaseBalance → IdealAccount)
```

Figure 4.2: An ideal account

Maintainable Account However, besides credit and debit services, a non-ideal system needs routine services without any notice in advance. The CSP specification of MaintainableAccount is shown in Figure 4.3.

```
MaintainableAccount = ( debit → checkBalance →
                         ( reduceBalance → MaintainableAccount
                           ⊓
                           rejectDebitRequest → MaintainableAccount)
                        □
                        credit → increaseBalance → MaintainableAccount)
                        ⊓
                        service → MaintainableAccount
```

Figure 4.3: A maintainable account

Real Account Unfortunately, in real-life, some accounts cannot resume the daily operation even after on-schedule services. It means the account terminates its life. We cannot control whether termination happens or not. Hence, we model the situation using an internal choice "⊓" in the RealAccount. A CSP specification of RealAccount is shown in Figure 4.4.

```
RealAccount = ( debit → checkBalance → ( reduceBalance → RealAccount
                                          ⊓
                                          rejectDebitRequest → RealAccount)
                □
                credit → increaseBalance → RealAccount)
                □
                service → (STOP ⊓ RealAccount)
```

Figure 4.4: A real account

We can easily check that IdealAccount conforms to MaintainableAccount, and RealAccount conforms to IdealAccount. However, RealAccount does not conform to MaintainableAccount. For instance, after service, RealAccount may not be allowed to perform debit whereas MaintainableAccount can perform debit successfully. It means that the behavioural property of a process cannot be carried to the third process through Cusack conformance. (Hence, Cusack conformance is not transitive.) Fortunately, there are two common special cases of conformance that preserves transitivity. They are known as *extension* and *reduction*.

Special case # 1: Extension

Extension is similar to conformance except that it requires the extended process to have a larger (inclusively) set of traces than the original process. Formally, it is defined as follows:

Definition 2 (Extension) *A process Q* extends *a process P if and only if*
(1) Q conforms to P;
(2) traces(P) ⊆ traces(Q).

For instance, RealAccount extends IdealAccount.

We interpret that if a process *Q* extends a process *P*, it is equivalent to purely adding new features to a superclass *P* to form a subclass *Q*. Furthermore, one can easily prove that extension is reflexive, anti-symmetric and transitive. In short, the extension relation provide a partial order relationship among a set of extension-connected processes. Thus, if we do not remove/override any feature (method) inherited from superclasses to form subclasses, the inheritance relation will be a pure extension hierarchy.

Nevertheless, extension does not allow any trace of the superclass (the original process) to be removed in the subclass (the extended process). It means that it only supports a limited overriding because we cannot remove any unwanted behaviour from the being extended process, that defeating the purpose of overriding. For instance, a generic account may perform the transfer operation as debiting the account and crediting another one in either sequences. This can be specified as shown in Figure 4.5.

```
GenericAccount = transfer → ( credit → debit → GenericAccount
                                ⊓
                              debit → credit → GenericAccount)
                 . . .
```

Figure 4.5: A generic account

If pure extension is used as an inheritance link, we cannot construct a subclass of GenericAccount that only contains

$$\mathit{credit} \rightarrow \mathit{debit} \rightarrow \mathit{GenericAccount}$$

without

$$\mathit{debit} \rightarrow \mathit{credit} \rightarrow \mathit{GenericAccount}.$$

In other words, under all circumstances, $\langle$transfer, debit, credit$\rangle$ must be a valid trace of any extended process of GenericAccount.

Special case # 2: Reduction

Another kind of conformance which provides a partial order relation among processes is reduction. It eliminates unwanted failures from a process to form a reduced process. The formal definition of reduction is shown in Definition 3.

Definition 3 (Reduction) *A process Q* reduces *a process P if and only if*
(1) Q conforms to P;
(2) traces(Q) ⊆ traces(P).

Reduction only allows the reduced process to remove failures and divergences (if any) from the original process. The definition of reduction is similar to that of extension except that the set extension of traces is replaced by set inclusion. For instance, IdealAccount reduces MaintainableAccount. Nevertheless, we cannot define new methods in the subclass (the reduced process), which is an unacceptable outcome from an object-oriented point of view.

From the above discussion, we see that a more general notion than either extension or reduction is required. It should allow a removal of unwanted behaviour, an addition of new services and a redefinition of services. At the same time, the notion should preserve the partial ordering of processes. In Chapter 5, we will propose a relation supporting this general notion.

4.2.2 Incrementality

Cusack conformance gives us a formal relation between two processes. However, it does not provide an incremental approach such that a subclass (a conforming process) can be constructed stepwise from the superclass while preserving behavioural compatibility.

4.2.3 Section summary

Moreover, to be incremental, conformance should also include a syntactic link between the superclass and subclass. In the following section, we will investigate what additional tools will help to support these requirements.

4.3 Priority choice operator

We would like to apply, as a second leg, the priority choice operator defined in [Fidge93], which allows the process to *pick out* a high priority operand to communicate with the environment if both the high and low priority operands (P and Q respectively) can perform the communication. The semantics of this priority choice operator is defined in CSP as follows:

$$\text{traces}(P \overrightarrow{\Box} Q) = \text{traces}(P) \cup (\text{traces}(Q) \setminus \{t \in \text{traces}(Q) : \exists\, u{:}\text{traces}(P) \bullet t_0 = u_0 \wedge t_0 \notin \text{refusals}(P)\}).$$

We illustrate the effect of the priority choice "$\overrightarrow{\Box}$" by a Savings Account example.

Consider a basic account that supports credit and debit only. After performing a credit operation, the balance of the account will be increased by the credited amount. Similarly, after performing the a debit operation, the balance of the account will be decreased by the debited amount. The CSP specification of BasicAccount is shown in Figure 4.6.

```
BasicAccount(Bal) = credit?amount → BasicAccount(Bal + amount)
                    □
                    debit?amount → BasicAccount(Bal – amount)
```

Figure 4.6: A basic account.

We can then build a SavingsAccount on top of the BasicAccount such that a successful debit can be performed only if there is sufficient money. This is shown in Figure 4.7.

```
SavingsAccount(Bal) = ( interest?rate → calculate interest!rate!Bal?NewBal →
                                         SavingsAccount(NewBal)
                        □
                        debit?amount → if Bal ⩾ amount
                                             then SavingsAccount(Bal – amount)
                                             else SavingsAccount(Bal))
                      □⃗
                      ( credit?amount → SavingsAccount(Bal + amount)
                        □
                        debit?amount → SavingsAccount(Bal – amount))
```

Figure 4.7: A savings account

If an event is not accepted by both the high and low priority operands at the same time, the effect of the priority choice operator "$\overrightarrow{\Box}$" is about the same as that of the conventional general choice operator "$\Box$" in CSP. For instance, the event interest?rate will only communicate with the high priority operand, while the event credit will communicate with the low priority operand.

However, the situation is different if an event offered by the environment is acceptable by both the high and low priority operands. The high priority operand (the left operand) will then have the privilege to be selected for communication. For instance, if users want to debit a SavingsAccount, the high priority operand will be chosen. Moreover, since the high priority operand can always successfully perform the debit operation, the debit in the low priority operand will never be selected. Hence, SavingsAccount can be viewed as if it were as shown in Figure 4.8.

```
SavingsAccount(Bal) = interest?rate → calculate interest!rate!Bal?NewBal →
                                      SavingsAccount(NewBal)
                      □
                      debit?amount → if Bal ⩾ amount
                                         then SavingsAccount(Bal – amount)
                                         else SavingsAccount(Bal)
                      □
                      credit?amount → SavingsAccount(Bal + amount)
```

Figure 4.8: A stand-alone Savings Account specification.

We feel that this property — the ability to select an operand in the case of conflict — of the priority choice operator suits our needs in overriding. For instance, we will consider the low priority operand as the superclass and the high priority operand as an incremental change to derivate a new specification of a subclass. Using the Savings Account example, the superclass is

```
( credit?amount → SavingsAccount(Bal + amount)
  □
  debit?amount → SavingsAccount(Bal – amount))
```

and the incremental change is

```
( interest?rate → calculate interest!rate!Bal?NewBal →
                  SavingsAccount(NewBal)
  □
  debit?amount → if Bal ⩾ amount
                     then SavingsAccount(Bal – amount)
                     else SavingsAccount(Bal))
```

Further subclasses can be incrementally constructed as shown in Figure 4.9

4.4 Weak conformance

We purpose, as the third leg, a modified notion of conformance. We call it weak conformance.

Definition 4 (Weak conformance) *A process P weakly conforms to a process Q if and only if*

(1) $\alpha P = \alpha Q$;

SubSubClass = incremental change’

$\overrightarrow{\Box}$

incremental change

$\overrightarrow{\Box}$

superclass

Figure 4.9: An incremental development of a subclass

.

(2) For any non-empty $s \in$ traces(Q), if $(s, X) \in P$, then $(s, X) \in Q$;

(3) For any non-empty $s \in$ divergences(Q) $\cap$ traces(P), $s \in$ divergences(P).

The conformance relation between a superclass and subclass cannot be determined until both the superclass and subclass are specified. However, this does not align with the philosophy of the incremental approach which suggests that the subclass should be the result of applying a modification to an existing class (its superclass). Weak conformance relaxes the constraint of the failure inclusion associated with an empty trace. With this relaxation, we can model the incremental change because a weakly conformed process is not required to re-model the existing behaviour (traces or failures) of the original process. For instance, as shown in the previous section, the incremental change for SavingsAccount weakly conforms to BasicAccount which the behaviour of credit service is not required to repeat in the incremental change.

Lemma 1 (Modifier composition) *If both the processes $\mathcal{A}_1$ and $\mathcal{A}_2$ weakly conform to a process Q, then the process ($\mathcal{A}_1 \Box \mathcal{A}_2$) weakly conforms to Q.*

Proof: Consider a failure (s, X) of $(\mathcal{A}_1 \Box \mathcal{A}_2)$ such that s is also a non-empty trace of Q. We have 3 cases:

1. s is a trace of $\mathcal{A}_1$ but not that of $\mathcal{A}_2$
   (s, X) will be a failure of $\mathcal{A}_1$. Since, $\mathcal{A}_1$ weakly conforms to Q, (s, X) should also be a failure of Q.

2. s is only a trace of $\mathcal{A}_2$ but not that of $\mathcal{A}_1$
   Similarly to the first case, (s, X) should also be a failure of Q.

3. s is an non-empty trace of $\mathcal{A}_1$ and $\mathcal{A}_2$
   From the definition of the general choice operator “$\Box$”, (s, X) should either be a failure of $\mathcal{A}_1$ or $\mathcal{A}_2$ or both. Given that both $\mathcal{A}_1$ and $\mathcal{A}_2$ weakly conform to Q, it follows that (s, X) should also be a failure of Q.

Moreover, any non-empty divergence d of $\mathcal{A}_1 \Box \mathcal{A}_2$ is a divergence of either $\mathcal{A}_1$ or $\mathcal{A}_2$. Since both $\mathcal{A}_1$ and $\mathcal{A}_2$ weakly conform to Q, d should also be a divergence of Q. $\Box$

Lemma 2 (Superclass composition) *If a process A weakly conforms to both the processes Q and R, it weakly conforms to the process ($Q \Box R$).*

Proof: Given a failure (s, X) of A such that s is also a non-empty trace of $(Q \Box R)$. Since A weakly conforms to both Q and R, (s, X) should be either a failure of Q or R or both. From the semantics of the general choice operator “$\Box$”, (s, X) should also be a failure of $(Q \Box R)$. Similarly, any non-empty divergence d of A should be a divergence of Q or R, which means that d should be a divergence of $(Q \Box R)$. $\Box$

4.5 Summary

We have investigated the possibility of using Cusack conformance, extension and reduction to formalize inheritance. We identified problems such as non-transitivity and non-incrementality. Fidge's priority choice operator seems a promising candidate to support overriding. In the next chapter, we will discuss how we can improve on the notion of conformance and, together with the notions of priority choice operator and weak conformance, it helps us to define a theory of inheritance which satisfies the desirable features.

Chapter 5

A Theory of Inheritance

Having explored some tools like Cusack conformance, weak conformance and the priority choice operator, and having identified some desirable features like formality, reflexivity, anti-symmetry, transitivity, incrementality, behavioural compatibility and multiple inheritance, we would like to integrate these tools to form a theory which supports the desirable features. Firstly, we present a stronger notion of conformance which allows the addition of new services and a redefinition of services, supports overriding and preserves the partial ordering of processes. Then, we propose a theory of inheritance and prove some properties to support an incremental development. Following that, we generalize our concept to form the notion of multiple inheritance and the optimization of modifiers. Finally, we profile how the theory supports the desirable features.

5.1 Strong conformance

Cusack conformance allows the addition of new services and the removal of unwanted behaviours, but does not support transitivity; whereas reduction and extension support transitivity but not the addition or removal of services. There should be a kind of conformance which supports all of three features. We call it strong conformance and define it below. The idea is: besides a conformance relation among processes, the conforming processes should not define any trace that has been removed by the original process with respect to yet another process. Our aim is to apply strong conformance to inheritance which supports a partial order relation among classes.

Definition 5 (Strong Conformance) *A process P strongly conforms to a process Q if and only if*

(1) P conforms to Q,

(2) whenever Q strongly conforms to some process R, traces(P) $\cap$ traces(R) $\subseteq$ traces(Q).

Firstly, we prove that strong conformance is a partial order relation. And then, we will show that extension and reduction are its special cases.

5.1.1 Partial order relation

Lemma 3 (Reflexivity) *Strong conformance is reflexive.*

Proof: Trivial. □

Lemma 4 (Anti-symmetry) *Strong conformance is anti-symmetric.*

Proof: Suppose P and Q are processes such that P strongly conforms to Q and Q strongly conforms to P. Consider a trace s of P. According to the second condition of Definition 5, s should also be a trace of Q because Q strongly conforms to P too. However, we have assumed that P conforms to Q. It means a failure (s,X) of P should also be a failure of Q; otherwise P cannot be a conforming process of Q. It follows that all failures of P should be failures of Q. Similarly, we can conclude that all failures of Q should also be failures of P. Furthermore, failure equivalence implies trace equivalence between P and Q. Hence, according to the divergence condition of conformance, any divergence of P should be a divergence of Q, and vice versa. It means that P and Q should have identical sets of traces, failures and divergences, and hence P is equivalent to Q. $\square$

Lemma 5 (Transitivity) *Strong conformance is transitive.*

Proof: Suppose P, Q and R are processes such that P strongly conforms to Q and Q strongly conforms to R. Consider a failure (s,X) of P such that s is a trace of R. If s is also a trace of Q, then (s,X) should be a failure of Q because P strongly conforms to Q. Moreover, given that s is a trace of R, it further infers that (s,X) should also be a failure of R because Q strongly conforms to R. On the other hand, if s is not a trace of Q, according to the definition of strong conformance, s should not be a trace of P; otherwise P cannot strongly conform to Q. Moreover, if s is a divergence of P, then s is also a divergence of Q. Similarly, s should also be a divergence of R. $\square$

Theorem 1 (Partial Order Relation) *Strong conformance is a partial order relation.*

Proof: It follows directly from Lemmas 3, 4 and 5. $\square$

5.1.2 Extension and reduction

(1) Extension: By Definition 2, a process P extends a process Q if traces(P) $\subseteq$ traces(Q). Hence, Condition 2 of Definition 5 will trivially be satisfied. Thus, extension is a special case of strong conformance.

(2) Reduction: Similarly, reduction is another special case of strong conformance.

5.2 Inheritance

We first propose our theory of inheritance and then show that there is one-to-one correspondence relation between the superclass-subclass relation and the superclass-modifier relation. Based on this correspondence, we then propose an incremental technique. The definition of inheritance is shown below.

Definition 6 (Inheritance) *A process P inherits a process Q if and only if*

(1) P strongly conforms to Q and

(2) P behaves like the process $A \overset{\rightarrow}{\square} Q$ for some process A. A is called a modifier for the said inheritance.

As both the notions of strong conformance and priority choice operator form partial order relations, our definition of inheritance can be easily shown to be reflexive, anti-symmetric and transitive. Moreover, if we consider a superclass as the point of origin and a subclass as a point of destination, a modifier will represent a path linking up the origin and the destination.

Moreover, we would like to show that an inheritance relation between two processes will imply a weak conformance relation between the superclass and a modifier.

Lemma 6 *If a process $A \overrightarrow{\Box} R$ behaves like a process P, and P conforms to R, then A weakly conforms to R.*

Proof: Consider a trace s of P. Since P behaves like $A \overrightarrow{\Box} R$, s should be a trace of either A or R or both.

1. Suppose s is a trace of R but not that of A. s is irrelevant to A, and hence does not affect the weak conformance relation between A and R.

2. On the other hand, suppose s is a trace of A but not that of R. Similarly, it does not affect the weak conformance relation between A and R.

3. Lastly, suppose s is a trace of both A and R. We have two cases:

   (i) Suppose s is an empty trace. Weak conformance does not impose any constraint on the empty trace of a weakly conforming process. Hence, it is also an irrelevant case.

   (ii) Suppose s is a non-empty trace. A failure (s, X) of P will be either a failure of A, or R or both. Moreover, according to the semantics of the priority choice operator, all the failures of A associated with non-empty traces will also be failures of P. It means that any failure (s, Y) of A should also be a failure of P. As P conforms to R, (s, Y) should be a failure of R.

$\Box$

Moreover, our inheritance relation is a special kind of strong conformance which, in turn, is a special kind of conformance relation. Thus, we can state that inheritance implies a weak conformance relation between the superclass and a modifier.

Corollary 1 *If a process P inherits a process R and A is a modifier for the inheritance, then A, A weakly conforms to R.*

Proof:
It follows directly from Lemma 6 and the definition of strong conformance (Definition 5). $\Box$
Furthermore, we would like to prove that if there is a weak conformance relation between a modifier and the superclass, then we are guaranteed to have a conformance relation between the superclass and subclass.

Lemma 7 *A process P conforms to a process R if P behaves like $A \overrightarrow{\Box} R$ for some process A such that*

(1) A weakly conforms to R and

(2) R respects the immediate divergence of A.

Proof: Consider a failure (s, X) of P such that s is also a trace of R.

1. Suppose s is an empty trace. From the definition of the priority choice operator "$\overrightarrow{\Box}$", the failures $(\langle\rangle, X)$ of P are the common failures of A and R. It means that $(\langle\rangle, X)$ should also be a failure of R.

2. On the other hand, suppose *s* is not an empty trace. A failure (s,X) of *P* should be a failure of either *A* or *R* or both. Moreover, we know that *A* weakly conforms to *R*. It follows that (s,X) should be a failure of *R*.

Furthermore, suppose *d* is a divergence of *P*. *d* should be a divergence of *A* or *R*. Given that *A* weakly conforms to *R*, so any non-empty divergence *d*' of *A* should also be a divergence of *R*. Moreover, the condition of immediate divergence enforces that if $\langle\rangle$ is a divergence of *P*, it should be a divergence of *R*. Hence, *P* conforms to *R*. □

Lemma 7 can be further strengthened to handle strong conformance.

Lemma 8 *A process P strongly conforms to a process Q if P behaves like A* $\overrightarrow{\Box}$ *Q for some process A such that*

(1) A weakly conforms to Q,

(2) Q respects the immediate divergence of A and

(3) whenever Q strongly conforms to R, traces(A) $\cap$ *traces(R)* $\subseteq$ *traces(Q).*

Proof: By Lemma 7, *P* conforms to *Q*. All traces of *P* should be traces of either *A*, *Q* or both. Moreover, if *A* does not define any trace which is a trace of some *R*, but not a trace of *Q*, then *P* should satisfy the second condition of strong conformance. It follows that *P* should strongly conform to *Q*. □

Based on the above lemmas, we can conclude that there is a one-to-one relationship between the superclass-subclass relation and the superclass-modifier relation.

Theorem 2 *A process P conforms to a process R if and only if P behaves like A* $\overrightarrow{\Box}$ *R for some process A such that*

(1) A weakly conforms to R and

(2) R respects the immediate divergence of A.

Proof: It follows directly form Lemmas 6 and 7. □

Theorem 3 (Incremental Theorem) *A process P inherits a process Q if and only if P behaves like A* $\overrightarrow{\Box}$ *Q for some process A such that*

(1) A weakly conforms to Q,

(2) Q respects the immediate divergence of A and

(3) whenever Q strongly conforms to some process R, traces(A) $\cap$ *traces(R)* $\subseteq$ *traces(Q).*

Proof: It follows directly Definition 6, Corollary 1 and Lemma 8. □

Theorem 3 suggests that we can construct a modifier to create a subclass. Moreover, our notion is transitive, and hence the process can be repeated to generate increasing levels of subclasses. Besides, the lemma on Modifier Composition can be considered as an approach to combine methods into a modifier. The lemma on Superclass Composition, on the other hand, provides us with an approach to integrate two different behaviours patterns to form a single superclass. Finally, if we view the Incremental Theorem as vertical reuse, then Modifier Composition and Superclass Composition provide a means of horizontal reuse.

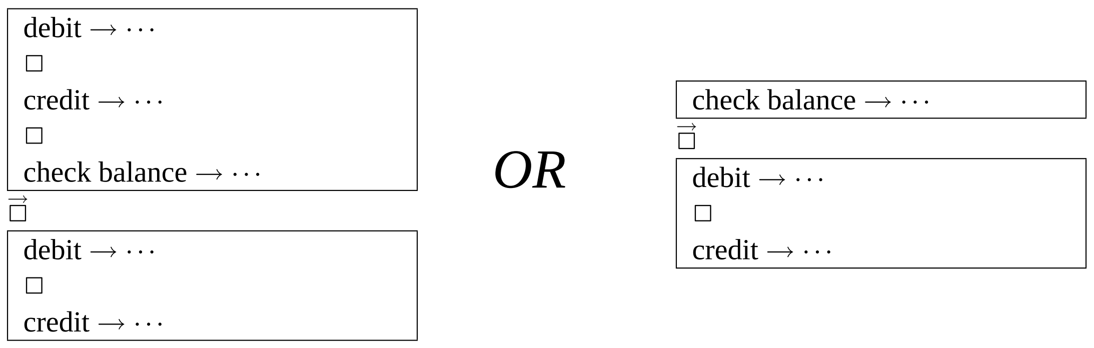

Figure 5.1: Modifier is not unique.

Remark A modifier of a class is not unique. It is up to designers to specify the required modifier. For instance, as illustrated in Figure 5.1, an account may inherit another account and yet override every inherited method and attribute; alternatively, it may simply add a new feature (check balance).

On the other hand, in the object-oriented notion, we would like to manage the complexity by considering individual services in a class and then combine these services together to form a modifier. In fact, we have shown in the last chapter when we are discussing the notion of weak conformance that two modifiers $\mathcal{A}_1$ and $\mathcal{A}_2$ can be combined into $\mathcal{A}_1 \Box \mathcal{A}_2$.

5.3 Multiple inheritance

Multiple inheritance opens up the possibility of specification reuse for more than one superclass. Nevertheless, it also creates new challenges to the formalization of inheritance.

5.3.1 Problem

A problem associated with multiple inheritance is that the inherited behaviours of a subclass from different superclasses may interfere with one another. For instance, we may have a method M in a process Q that conflicts with M in another process R. In other words, the failures of M in Q does not agree with those of M in R. As a result, the failures of a method M in the subclass P cannot conform to both of the methods M in Q and R simultaneously.

5.3.2 Alternatives

To resolve the conflict, there are at least two strategies : redefinition and renaming.

(1) **Redefinition**
The first alternative redefines the methods of conflict in the subclass. To retain the transitivity of the conformance relation, the new definition of these conflicting methods should conform to their overridden counterparts. For instance, the subclass P should define a conforming method, say M_1, to override both the methods M inherited from Q and R respectively. Redefinition is the recommendation of OMT[Rumbaugh91].

(2) **Renaming**
Another alternative is to rename the methods of conflict such that no more conflict will further arise. For instance, we may rename (a change of symbol in CSP terminology) *M* in *R* to *M'*, so that only the other method *M* in *Q* is inherited to *P* using the required signature. Nevertheless, it does not guarantee that the subclass *P* will conform to its superclass *Q* and *R*. For instance, *P* will not conform to *R* since, in the specification of *P*, it does not consider the effect of *M* in *R* but simply substitute it by *M'*.

5.3.3 Our proposal

As we are interested in the formalization of OMT methodology and the renaming approach does not guarantee conforming subclasses, we will choose the first alternative.

An inherited feature will appear in a subclass unless it is overridden. On the other hand, we use the latest definition of a method whenever it has been overridden. It means that this latest definition of features should always be accepted. If we consider services as events, it leads to a conclusion that the redefined events should not be refused, even if there is a conflict among the inherited methods from different superclasses. We formalize the idea as follows.

Theorem 4 (Conflict Resolution) *A process P conforms to both the processes Q and R if P behaves like* $A \overset{\rightarrow}{\Box} (Q \Box R)$ *for some process A such that*

(1) A weakly conforms to Q and R

(2) A always synchronizes with any initial event common to both Q and R. Mathematically, $\langle e \rangle \in traces(Q) \cap traces(R) \Rightarrow (\langle\rangle, \{e\}) \notin failures(A)$

(3) Q and R preserve the immediate divergence of A.

Proof: Consider a failure (s,X) of P.

1. Suppose s is an empty trace. (s,X) will be a common failure of A and $Q \Box R$. Hence, (s,X) will be a failure of $Q \Box R$, which further implies that it will be a failure of both Q and R.

2. Suppose s is a non-empty trace. Let s_o be the first event of s. Moreover, suppose s_0 is also an event refused by A. The second condition of the theorem implies that $< s_0 >$ should not simultaneously be a trace of Q and R; otherwise it should not be refused by A. We have 5 sub-cases:

   a. Suppose s is a trace of both A and R, but not Q. Since $(\langle\rangle,\{s_0\})$ is a failure of A, (s,X) can be a failure of A or R. However, A weakly conforms to R. Hence, (s,X) will be a failure of R.

   b. Suppose s is a trace of both A and Q, but not R. Similarly to case 2a, (s,X) should be a failure of Q.

   c. Suppose s is a trace of neither A nor R. Hence, s should be a trace of Q; otherwise s cannot be a trace of P. Moreover, the failure (s,X) of P should be due to Q.

   d. Suppose s is not a trace of both A and Q. Similarly to case 2c, (s,X) should be a failure of R.

   e. Otherwise, (s,X) should be irrelevant to Q and R and hence does not affect the result.

3. Similarly to case 2, suppose that s is a non-empty trace and the first event s_0 of s is not refused by A. Based on the definition of priority choice operator, the high priority operand A will synchronize with s_0. It means that (s,X) of P should be a failure of A; otherwise s cannot be a trace of P. Since A weakly conforms to both Q and R, it follows that (s,X) should be a failure of Q or R depending whether s is a trace of Q or R, respectively.

Moreover, consider a divergence d of P. It d should be a divergence of A, Q or R. We have 3 cases:

1. Suppose d is a trace of Q but not R. Then, d should be a non-empty trace. If d is also a trace of A, it should be a divergence of A or Q. Since A weakly conforms to Q, it means that d should be a divergence of Q because the divergence of P should come from either A or Q. On the other hand, if d is not a trace of A, it should strictly be a divergence of Q.

2. Suppose d is a trace of R but not Q. Similarly to case 1, d should be a divergence of R.

3. Suppose d is a trace of both Q and R. If d is a non-empty trace, then it should be a trace of A; otherwise the second condition of the theorem will be violated and d cannot be a trace of P. It further means that the divergence d of P should be due to the divergence of A. Moreover, A weakly conforms both Q and R. Therefore, a non-empty divergence d of P should also be a divergence of both Q and R. On the other hand, suppose d is an empty trace. It should be a divergence of A; otherwise P cannot diverge immediately. It means d should be a divergence of both Q and R according to the third condition of the theorem.

□

5.3.4 Optimizing modifiers

A modifier for multiple inheritance can be further simplified. Our prime goal is to keep the conformance relation between a subclass and its superclasses. It means that we may discard some traces of a modifier as long as the relation holds. Moreover, as we only discard traces, the second condition of strong conformance should automatically be satisfied. We formalize the idea as a criterion for optimizing a modifier as follows.

Optimizing criterion If there exists processes A, P, Q and R such that P multiply inherits Q and R, and A is a modifier for the inheritance, we can construct a process A' such that P behaves like $A' \vec{\Box} (Q \Box R)$ (and hence P still multiply inherits Q and R) if

(1) $traces(\mathcal{A}') \subseteq traces(\mathcal{A})$,

(2) $failures(\mathcal{A} \vec{\Box} (\mathcal{Q} \Box \mathcal{R})) = failures(\mathcal{A}' \vec{\Box} (\mathcal{Q} \Box \mathcal{R})) and$

(3) $divergences(\mathcal{A} \vec{\Box} (\mathcal{Q} \Box \mathcal{R})) = divergences(\mathcal{A}' \vec{\Box} (\mathcal{Q} \Box \mathcal{R}))$

Moreover, the modifier is optimized if the following condition is also satisfied:

$\nexists \mathcal{A}'' \bullet$

(a) $traces(\mathcal{A}'') \subset traces(\mathcal{A}')$,

(b) $failures(\mathcal{A}' \vec{\Box} (\mathcal{Q} \Box \mathcal{R})) = failures(\mathcal{A}'' \vec{\Box} (\mathcal{Q} \Box \mathcal{R}))$,

(c) $divergences(\mathcal{A}'' \vec{\Box} (\mathcal{Q} \Box \mathcal{R})) = divergences(\mathcal{A}' \vec{\Box} (\mathcal{Q} \Box \mathcal{R}))$

Finally, we define the multiple inheritance as a special kind of inheritance relation where a subclass should inherit all superlasses and are linked to superclasses through the priority choice operator.

Definition 7 (Multiple Inheritance) *A process P multiply inherits distinct processes* $Q_1, \cdots, Q_n$ *if and only if*

(1) for every Q_i *in the range, P inherits* Q_i*, and*

(2) there is a process A such that the process $A \overset{\rightarrow}{\Box} (\Box\ Q_i)$ *behaves like P.*

5.4 Evaluation

We evaluate our theory according to the desirable features of a theory of inheritance identified in Chapter 2.

(1) **Incremental**
The notions of modifiers and weak conformance provide a quality framework of incremental modification of an inheritance hierarchy. Designers can specify the modifier to existing classes to derive new classes.

(2) **Behaviourally Compatible**
The notion of strong conformance provides a framework for subclasses to be behaviourally compatible to its superclasses.

(3) **Formally defined**
Definitions 6 and 7 formalize single and multiple inheritance under the CSP framework.

(4) **Reflexive, Anti-symmetric and Transitive**
Theorem 1 guarantees that the definition of inheritance is reflexive, anti-symmetric and transitivity.

(5) **Overriding**
The priority choice operator incorporated with the notion of weak conformance enables us to override existing behaviours while maintaining the compatibility of subclasses.

(6) **Automatically propagated**
Given an inheritance hierarchy in our theoretical framework, if we add a new feature, say a method, to a superclass, the feature will automatically appear in its subclasses unless it is overridden.

Chapter 6

An Account Example

6.1 Introduction of an account example

Account Forget about the inheritance theory for a while and pay attention to a concept that we come across in daily life. It is bank account. Generally, we *withdraw* money from an account, *deposit* money in it, and *check* the account balance. Moreover, once the account is open, we can *close* it if required. However, an account may be frozen for some reason and we should wait until the account is reactivated to continue our business upon the account[1].

Savings account Basically, a savings account is an account with interest. It means that, besides monetary transactions, we should have interest from time to time. Nevertheless, banks may only give interest to those savings accounts with a balance over $1000.

Credit account We can also open a credit account which is another kind of account. However, whenever we use our credit card to purchase our favourite products, a service charge will be levied by the credit agent. Moreover, if we use the overdraft facility in a credit account, we should pay the overdraft charge.

Vantage account To have the advantage of a savings account for interest and that of a credit account to allow overdraft, we may introduce a new kind of account − Vantage Account.

6.2 An overview of the relations among accounts

We summarize the relations among accounts in the Object Model of OMT [Rumbaugh91]. Basically, both *Savings Account* and *Credit Account* are subclasses of *Account*, whereas *Vantage Account* is a subclass of both *Savings Account* and *Credit Account*.

In *Account*, we define methods like *deposit*, *withdraw*, *freeze*, *reactivate* and *close* and an attribute *balance*, whereas, a new method *calculate interest* is added to *Savings Account*. This method is applicable only if the balance of the account is at least $1000. On the other hand, in *Credit Account*, we override the

[1]Some people feel that *transfer*ring money from one account to another should be an operation of an account. Paradoxically, it (*transfer*) involves at least two accounts, and if one of them has been destroyed, the transfer operation among these accounts will become invalid. It shows that *transfer* should be an association (in OMT terminology) between two accounts and hence not a method or an attribute of an account class.

withdraw operation inherited from *Account* and add a feature *charge overdraft*. This feature is applicable only if the account is being overdrawn.

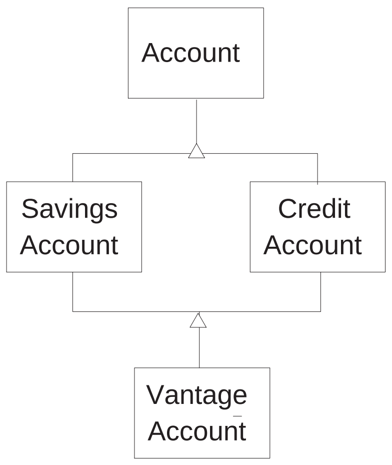

Figure 6.1: The account example : Object Model

6.3 Name substitution

We link OMT and CSP and introduce a syntactic construct which we call name substitution. We substitute a process by another process. For instance, when we write:

$$Q = a \longrightarrow b \longrightarrow Q$$

$$P = c \longrightarrow \mathbf{STOP}$$
$$\Box$$
$$Q[Q/P]$$

it is a short cut to iteratively mean:

$$P = c \longrightarrow \mathbf{STOP}$$
$$\Box$$
$$a \longrightarrow b \longrightarrow P[Q/P]$$

However, name substitution is not a special kind of change of symbols. The change of symbols is a one-to-one function such that the symbol for an event is changed to another symbol, while the semantics of the processing are still maintained; whereas name substitution may replace a process by an entirely distinct unrelated process: there may not be any semantics relationship between the original process and the new one. Nevertheless, the link between OMT and CSP is not formally defined, we translate a OMT specification into a CSP specification by our craftsman skills.

6.4 Account

We will concentrate our discussion on the control aspect of the example.

An account starts its life once a deposit is made, and it enters the *Normal* state. When a message is received, it will trigger a state transition. For instance, in response to receiving the *withdraw* message, it will perform the function to debit the account and then return to the *Normal* state. However, if *freeze* is received, it will enter the *Frozen* state. It will wait until the account is *reactivated*. In the later case, it comes back to the *Normal* state. We can also apply the check balance method to check our account balance. The Dynamic Model of *Account* is shown in Figure 6.2.

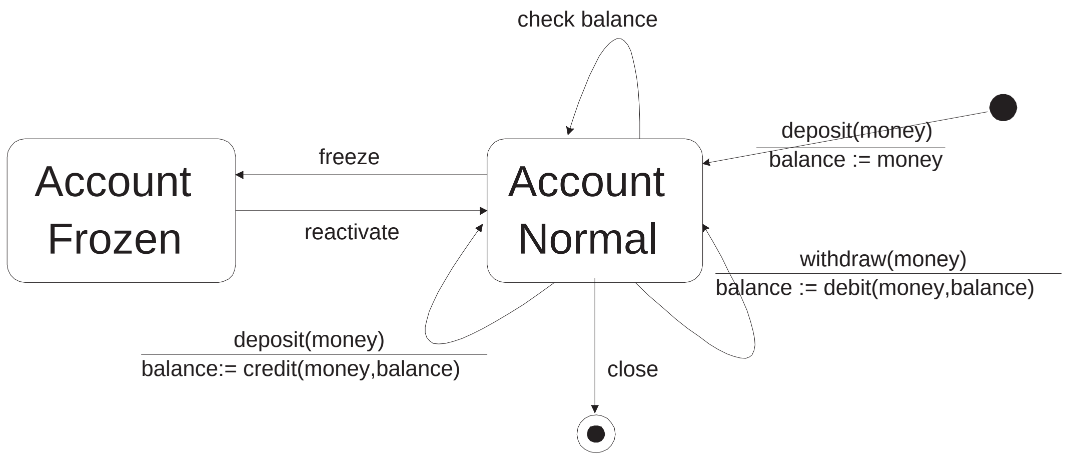

Figure 6.2: Account: Dynamic Model

We translate the Dynamic Model of *Account* into a CSP specification, which is shown in Figure 6.3. Starting from the black dot in Figure 6.2, Account receives a deposit(money) message and enters the AccountNormal state. We model it as

$$\text{Account} = \text{deposit?money} \longrightarrow \text{AccountNormal(money)}.$$

In the normal state, AccountNormal can receive check balance?, deposit(money), withdraw(money), freeze. We model it as a choice combination. The triggered actions like balance:=debit(money,balance) triggered by withdraw(money) are modelled as events following the triggering event. For instance, we model the withdraw operation as

$$\begin{aligned}&\text{withdraw?money} \longrightarrow \text{debit!money!balance?newbalance}\\&\qquad\qquad\longrightarrow \text{AccountNormal(newbalance)}.\end{aligned}$$

Similarly, the message freeze will drive the account into the Account Frozen state and the message reactivate will active the account again and bring it back to the Normal state. We model it as

$$\text{freeze} \longrightarrow \text{AccountFrozen(balance)}$$

$$\text{AccountFrozen(balance)} = \text{reactivate} \longrightarrow \text{AccountNormal(balance)}$$

The account will terminate if we close the account. In the Dynamic Model, it is denoted by an arc connected to a circle with a black dot inside. Since it represents a successful termination of an object, we use the process **SKIP** to represent the situation in CSP.

$$\text{close} \longrightarrow \mathbf{SKIP}$$

```
Account = deposit?money → AccountNormal(money)

AccountNormal(balance) = deposit?money → credit!money!balance?newbalance
                                       → AccountNormal(newbalance)
                         □
                         withdraw?money → debit!money!balance?newbalance
                                        → AccountNormal(newbalance)
                         □
                         freeze → AccountFrozen(balance)
                         □
                         check balance? → check balance!balance
                                        → AccountNormal(balance)
                         □
                         close → SKIP

AccountFrozen(balance) = reactivate → AccountNormal(balance)
```

Figure 6.3: Account: CSP specification

6.5 Savings account

Savings Account refines the AccountNormal state into two separate substates: EligibleForInterest and NotEligibleForInterest. If a savings account is in the EligibleForInterest state, we can apply calculate interest. The Dynamic Model of *Savings Account* is shown in Figure 6.4.

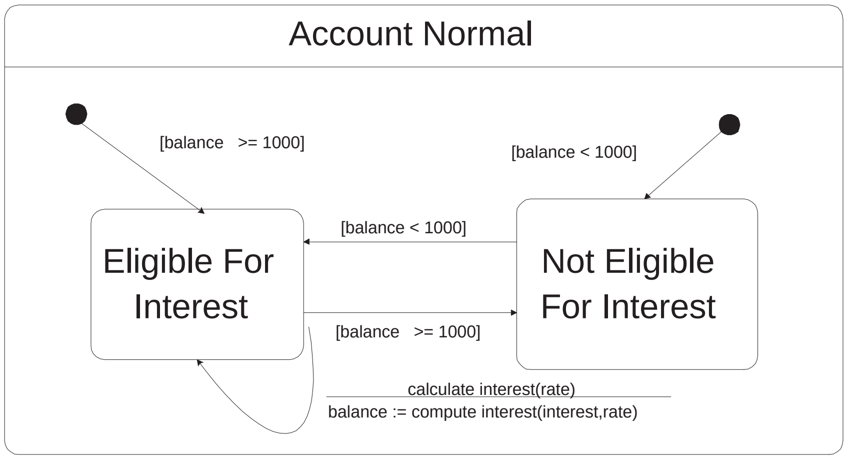

Figure 6.4: Savings Account: Dynamic Model

Similarly, we present the CSP specification of *Savings Account* in Figure 6.5. Since the normal state of a savings account is the normal state of an account. We refer to it as SavingsAccountNormal. From the Dynamic Model, there are two entry points: one goes into NotEligibleForInterest and the other goes into EligibleForInterest. We will consider these two substates as incremental changes acting upon the AccountNormal. Hence, we can apply our theory to firstly combine the two incremental change (Modifier Composition) and then link them with AccountNormal using the priority choice operator “$\vec{\Box}$”.

$$\begin{array}{ll}\text{SavingsAccountNormal(balance)} = & \\ & ([0 \leq \text{balance} < 1000] \rightarrow \text{NotEligibleForInterest(balance)} \\ & \Box \\ & [\text{balance} \geq 1000] \rightarrow \text{EligibleForInterest(balance))} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)[SavingsAccountNormal/AccountNormal]}\end{array}$$

For the NotEligibleForInterest state, there is a newly defined transition which causes the account to enter the EligibleForInterest state. Since it is also an inherited substate of AccountNormal, we also apply our theory to form:

$$\begin{array}{ll}\text{NotEligibleForInterest(balance)} = & [\text{balance} \geq 1000] \rightarrow \text{EligibleForInterest(balance)} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)[SavingsAccountNormal/AccountNormal]}\end{array}$$

Similarly, we translate the EligibleForInterest state into a CSP process.

$$\begin{array}{ll}\text{EligibleForInterest(balance)} = & ([0 \leq \text{balance} < 1000] \rightarrow \text{NotEligibleForInterest(balance)} \\ & \Box \\ & \text{calculate interest?rate} \rightarrow \text{compute interest!rate!balance?newbalance} \\ & \qquad\qquad\qquad\qquad\quad \rightarrow \text{EligibleForInterest(newbalance))} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)[SavingsAccountNormal/AccountNormal]}\end{array}$$

$$\text{Savings Account} = \text{Account[SavingsAccountNormal/AccountNormal]}$$

$$\begin{array}{ll}\text{SavingsAccountNormal(balance)} = & ([0 \leq \text{balance} < 1000] \rightarrow \text{NotEligibleForInterest(balance)} \\ & \Box \\ & [\text{balance} \geq 1000] \rightarrow \text{EligibleForInterest(balance))} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)} \\ & \qquad\quad \text{[SavingsAccountNormal/AccountNormal]}\end{array}$$

$$\begin{array}{ll}\text{NotEligibleForInterest(balance)} = & [\text{balance} \geq 1000] \rightarrow \text{EligibleForInterest(balance)} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)} \\ & \qquad\quad \text{[SavingsAccountNormal/AccountNormal]}\end{array}$$

$$\begin{array}{ll}\text{EligibleForInterest(balance)} = & ([0 \leq \text{balance} < 1000] \rightarrow \text{NotEligibleForInterest(balance)} \\ & \Box \\ & \text{calculate interest?rate} \\ & \qquad\quad \rightarrow \text{compute interest!rate!balance?newbalance} \\ & \qquad\quad \rightarrow \text{EligibleForInterest(newbalance))} \\ & \overrightarrow{\Box} \\ & \text{AccountNormal(balance)[SavingsAccountNormal/AccountNormal]}\end{array}$$

Figure 6.5: Savings Account: CSP specification

We would like to give a few remarks here:

(1) **Incremental and reuse**
We build *Savings Account* on top of *Account*. Hence, we can reuse the existing specification.

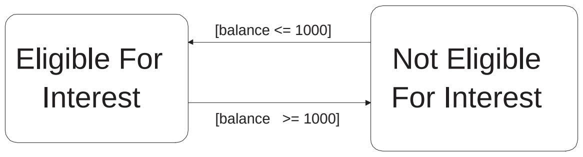

Figure 6.6: Problematic state transitions.

.

(2) **Weakly conformed**
In specifying EligibleForInterest, its modifier does not contain all the methods inherited from AccountNormal. Nevertheless, EligibleForInterest still conforms to AccountNormal.

(3) **Behaviourally compatible**
Savings Account conforms to *Account*.

(4) **Anti-symmetric**
Moreover, *Account* does not conform to *Savings Account*. For instance, *Account* will refuse to engage in calculate interest even if the balance is well over \$1000. On the other hand, *Savings Account* will synchronize with this event.

(5) **Reflexive**
In addition, instead of refining AccountNormal, we keep it intact. It means that if no NotEligibleForInterest or EligibleForInterest is defined, SavingsAccountNormal is equivalent to AccountNormal, thus:

$$\begin{aligned} \text{SavingsAccountNormal'} &= \text{STOP} \\ &\quad \vec{\Box} \\ &\quad \text{AccountNormal[SavingsAccountNormal'/AccountNormal]} \\ &= \text{AccountNormal[SavingsAccountNormal/AccountNormal]} \\ &= \text{AccountNormal} \end{aligned}$$

$$\boxed{\begin{aligned} \text{SavingsAccountNormal(balance)} &= ([0 \leq \text{balance} < 1000] \longrightarrow \text{NotEligibleForInterest(balance)} \\ &\quad \Box \\ &\quad [\text{balance} \geq 1000] \longrightarrow \text{EligibleForInterest(balance))} \\ &\quad \vec{\Box} \\ &\quad \ldots \\ \text{NotEligibleForInterest(balance)} &= [\text{balance} \geq 1000] \longrightarrow \text{EligibleForInterest(balance)} \\ &\quad \vec{\Box} \\ &\quad \ldots \\ \text{EligibleForInterest(balance)} &= ([0 \leq \text{balance} \leq 1000] \longrightarrow \text{NotEligibleForInterest(balance)} \\ &\quad \Box \\ &\quad \ldots \end{aligned}}$$

Figure 6.7: A problematic specification.

(6) **Error Detection**
If the transition condition [balance $<$ 1000] is changed to [balance $\leq$ 1000] (Figure 6.7), the OMT

specification will become that shown in Figure 6.6. We can hardly detect the problem of infinite inter-state transitions. However, if we translate to this OMT specification into a CSP specification, SavingsAccountNormal will be as shown in Figure 6.7. If the balance is $1000, SavingsAccountNormal will behave like **CHOAS**, since the transitions between NotEligibleForInterest and EligibleForInterest are always satisfied..

6.6 Credit account

Similarly, we refine *Account* into *Credit Account*. We override withdraw to include service charges and add a feature charge overdraft to compute the service charges. The Dynamic Model of *Credit Account* is shown in Figure 6.8.

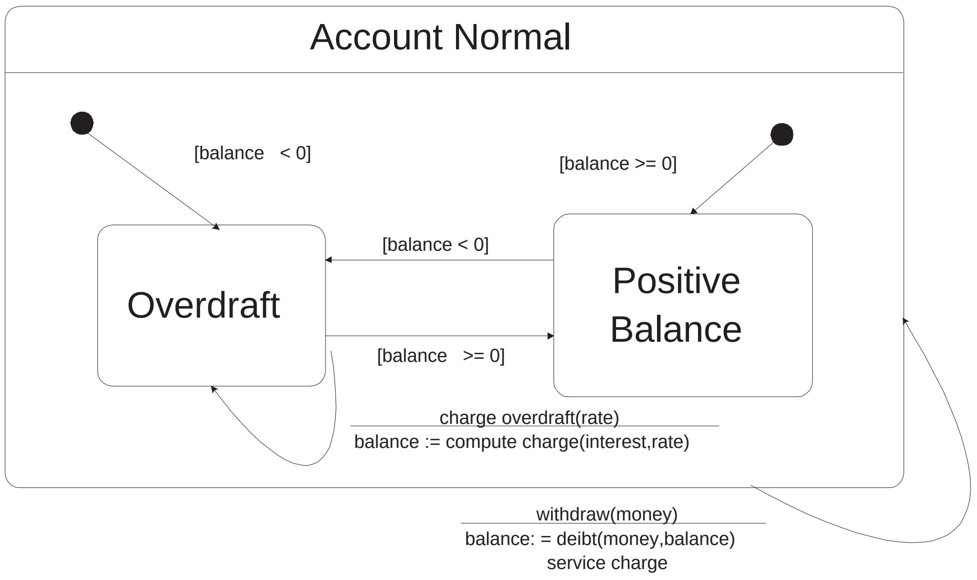

Figure 6.8: Credit Account: Dynamic Model

We also apply the technique described above to translate the Dynamic Model of Credit Account into a CSP specification, which is shown in Figure 6.9.

Remark

Overriding
In CreditAccountNormal, the method withdraw is redefined. The previous definition (the one with no service charge) is concealed by means of the priority choice operator "$\overrightarrow{\Box}$".

6.7 Vantage account

Vantage Account multiply inherits *Credit Account* and *Savings Account*. It means that methods like deposit and withdraw are inherited twice. According to Theorem 4, we should redefine these methods in *Vantage*

Credit Account = Account[CreditAccountNormal/AccountNormal]

CreditAccountNormal(balance) = ([balance $\geq$ 0] $\rightarrow$ PositiveBalance(balance)
$\Box$
[balance $<$ 0] $\rightarrow$ Overdraft(balance))
$\overrightarrow{\Box}$
AccountNormal(balance)[CreditAccountNormal/AccountNormal]

PositiveBalance(balance) = [balance $<$ 0] $\rightarrow$ Overdraft(balance)
$\overrightarrow{\Box}$
AccountNormal(balance)[CreditAccountNormal/AccountNormal]

Overdraft(balance) = ([balance $\geq$ 0] $\rightarrow$ PositiveBalance(balance)
$\Box$
charge overdraft?rate $\rightarrow$ compute charge!rate!balance?newbalance
$\rightarrow$ Overdraft(newbalance)
$\Box$
withdraw?money $\rightarrow$ debit!money!balance?newbalance
$\rightarrow$ (Overdraft(newbalance) $\|$
service charge $\rightarrow$ **SKIP**))
$\overrightarrow{\Box}$
AccountNormal(balance)[CreditAccountNormal/AccountNormal]

Figure 6.9: Credit Account: CSP specification

Account. Moreover, the two superclasses should be composited by the general choice operator “□” according to the definition of multiple inheritance (Definition 7). The Dynamic Model and CSP specification of *Vantage Account* are shown in Figures 6.10 and 6.11 respectively.

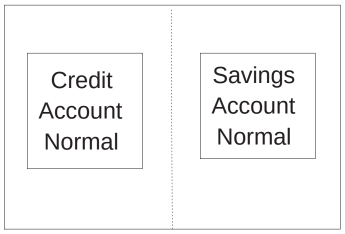

Figure 6.10: Vantage Account: Dynamic Model

(1) **Behaviourally Compatible**
VantageAccount behaves compatibly with *CreditAccount* and *SavingsAccount* since *VantageAccount* conforms to both *CreditAccount* and *SavingsAccount*.

(2) **Transitive**
Moroever, *VantageAccount* behaves compatibly with *Account* since *VantageAccount* conforms to *Account*.

(3) **Multiply inherited**
VantageAccount multiply inherits *CreditAcccount* and *SavingsAccount* despite the name substitution.

```
VantageAccount(balance) = (deposit?money → credit!money!balance?newbalance
                                         → VantageAccount(newbalance)
                           □
                           withdraw?money → debit!money!balance?newbalance
                                          → (VantageAccount(newbalance) ||
                                             ([balance < 0] service charge → SKIP
                                             □
                                             [balance ≥ 0]free service → SKIP)
                           □
                           freeze → AccountFozen(balance)[VantageAccount/Account]
                           □
                           check balance? → check balance!balance
                                          → VantageAccount(balance)
                           □
                           close → SKIP)
                           □⃗
                           (CreditAccount □ SavingsAccount)
                           [VantageAccount/CreditAccount,VantageAccount/SavingsAccount]
```

Figure 6.11: Vantage Account: Initial CSP specification

```
VantageAccount(balance) = withdraw?money → debit!money!balance?newbalance
                                         → (VantageAccount(newbalance) ||
                                            ([balance < 0] service charge → SKIP
                                            □
                                            [balance ≥ 0]free service → SKIP) )
                          □⃗
                          (CreditAccount □ SavingsAccount)
                          [VantageAccount/CreditAccount,VantageAccount/SavingsAccount]
```

Figure 6.12: Ventage Account: Optimized CSP specification

(4) From the OMT Dynamic Model of Vantage Account, we do not know that we have to redefine *withdraw*. The theory, however, tells us that we have to.

(5) **Optimized modifier**
We can apply the optimizing modifier criterion on *Vantage Account* to yield another CSP specification as shown in Figure 6.12

6.8 Summary

We have illustrated an application of our theory using the bank account example. It demonstrates its capability in

- reuse, where a substate is built on top of another state. For instance, we construct SavingsAccountNormal on top of AccountNormal.
- error discovery. For instance, a collection of simple and trivial state transitions, as shown in the problematic Savings Account example, will cause a CHAOS situation. This kind of error is harder to be discovered using diagrams.

- multiple inheritance. We have shown that a simple diagram like the Dynamic Model of Vantage Account, in fact, contains, ambiguities. We have to resolve these ambiguities to get a valid subclass. Our theory suggests to redefine inconsistent services and optimize the incremental change.

The example gives us an insight on a formal link between OMT and CSP. We propose to further develop this link in our future research.

Chapter 7

Discussions and suggestions

Formalizing inheritance may be challenging, exciting and fruitful, but there is one thing it never is. Formalizing inheritance is never complete. After investigating other related research and the painful struggle that we passed, we would like to share our opinion with you.

This is what the following sections are about. We have formally given the syntactic and semantical concepts of inheritance. Semantics relation is defined by strong conformance. Using the technique of incremental modification and the priority choice operator, it (inheritance) gains its syntax too. Nevertheless, challenges are still there: generalization may conflict with exceptions; incremental modification reverses the process of generalization; the priority choice operator facilitates overriding but is constrained by the conformance relation. How about other aspects of inheritance? Inheritance of implementation (with or without semantics relation)? Every solution may have its problems.

7.1 Generalization, exceptions and transitivity

Generalization links classes to their refined version. It helps designers to reason about the inter-class relations and facilitates reuse. Incidentally, formal methods put their strength in reasoning, and refinement solidifies abstract classes to concrete versions. We find that it is a natural to formalize generalization.

Nevertheless, designers model and capture similarities in objects and organize them in an inheritance hierarchy for simplicity. Usually, in a generalization hierarchy, a subclass is a refinement of the superclass. To have an easy life, some disappointed non-refined classes are added to the hierarchy. Designers use the technique of exceptions to handle these disappointments. For instance, a *Queue* can be considered as a *Stack* except that the push operation of the *Stack* class is overridden by the "append" operation of the *Queue* class. Because of the presence of exceptions, subclasses may be inheriting without a transitive relation. Any subclass of the *Queue* class cannot be considered as a kind of stack semantically. Knowledge from superclasses apply (to those exceptional subclasses) but is not guaranteed. We assume we have last-in-first-out (LIFO) in the *Stack* class, but we cannot guarantee this in the *Queue* class which, is actually first-in-first-out (FIFO). If we have a subclass of both the *Stack* class and *Queue* class, it will confuse designers and users. Thus we do not support exceptions in our formalism. On the other hand, are we modelling reality?

7.2 Generalization and refinement

Designers sometimes start with a class and incrementally refine it into different subclasses for different purposes, and sometimes generalize classes to form a superclass. This project addresses the former situation (using the technique of incremental modification).

7.2.1 Generalizing classes: the superclass

We draw an analogy between optics and generalization. Telescope craftsmen make lenses and mirrors. Physicists developed the principles of optics to explain these phenomena. Likewise, we are looking for a similar generalization. Most software designers today are still craftsmen: they construct the mysterious superclass by experience, like the telescope craftsman who manipulates lenses and mirrors without realizing the principle of optics. In fact, if we can formulate a theory, some unwarranted problems may be avoided.

From the theoretical point of view, our inheritance relation is reflexive, anti-symmetric and transitive. Given two classes, we can find a process to represent the more general class (generalization) since theory suggests that the more general form should always exist in this situation.

For instance, the superclass of

$$\begin{array}{ll} P = & \mathrm{a} \rightarrow \mathrm{b} \rightarrow P \\ & \Box \\ & \mathrm{c} \rightarrow \mathrm{d} \rightarrow \mathbf{CHAOS} \end{array} \qquad \text{and} \qquad \begin{array}{ll} Q = & \mathrm{a} \rightarrow \mathrm{b} \rightarrow Q \\ & \Box \\ & \mathrm{e} \rightarrow \mathrm{f} \rightarrow Q \\ & \Box \\ & \mathrm{g} \rightarrow \mathbf{CHAOS} \end{array}$$

is $T = \mathrm{a} \rightarrow \mathrm{b} \rightarrow T$.
Conversely, P can be constructed from T as

$$\begin{array}{ll} \mathrm{P} = & \mathrm{c} \rightarrow \mathrm{d} \rightarrow \mathbf{CHAOS} \\ & \overrightarrow{\Box} \\ & T[\mathrm{T/P}] \end{array}$$

and, similarly, for process Q.

There are, however, many questions concerning the construction of superclasses and modifiers:

(1) Does T represent the least upper bound of the partial order set of superclasses of P and Q?

(2) Given a set of classes C, is there any systematic and incremental technique to produce a generalized superclass and modifiers to represent them?

(3) Can the construction be automatic?

7.3 Overriding

We unavoidably redefine inherited features (attributes and methods) to suit the need of subclasses. For instance, a *History Stack* may inherit a *Stack* but extending the *push* and *pop* operations to record the history. However, it may confuse and hampers the designers' reasoning if used to the extreme. For instance, *Queue* may inherit *Stack* but override the *push* operation to append items at the end rather than at the beginning. The latter kind of overriding is, as discussed earlier, *ad hoc* and should be discouraged; otherwise a third party may assume the effect of applying *push* on *Queue* very similarly to that of *Stack*, and hence may build a subclass based on this wrong assumption.

$$\begin{aligned}\text{Queue} = {} & \text{push} \longrightarrow \text{append item at the end} \longrightarrow \text{Queue} \\ & \overset{\rightarrow}{\Box} \\ & \text{Stack[Stack/Queue]}\end{aligned}$$

$$\begin{aligned}\text{Stack} = {} & \text{push} \longrightarrow \text{concatenate item at the beginning} \longrightarrow \text{Stack} \\ & \Box \\ & \text{pop} \longrightarrow \text{return the first element} \longrightarrow \text{Stack}\end{aligned}$$

Figure 7.1: Improper inheritance.

On the other hand, although we use strong conformance to achieve proper overriding, *ad hoc* overriding should not be totally discouraged. For example, in the implementation stage, inheritance of implementation, a kind of *ad hoc* overriding that shares the implementation of classes regardless of the meaning, will become more important. Instead of strictly insisting a modifier to be weakly conforming to a superclass, we can relax the definition merely allowing a non-conforming modifier to be used. Thus, *ad hoc* overriding can be allowed at the expense of a theoretical concern for transitivity.

7.4 Different kinds of inheritance support

Our result, however, solves only half of the problem. The other half includes: generalization with no semantics support and inheritance of implementation with semantics support. Table 7.1 summarizes those 4 combinations and cells with a tick $\surd$ are those we have addressed.

	Generalization	Inheritance of implementation
With semantics	$\surd$	
Without semantics		$\surd$

Table 7.1: Possible combination of inheritance support.

7.4.1 Generalization without semantics support

Generalization without semantics support can be considered as a general form of inheritance, and suggests that the subclass may not be a refinement of that superclass. For instance, a simple Copy will copy a stream bit by bit. New Copy want to know the end of the stream by reading three consecutive 0s. To handle this situation, the original stream will be inspected, and if two consecutive 0s are copied, New Copy will generate a 1 before copying the next bit; otherwise it will function as if it were a normal Copy. We do not support this kind of generalization because an important asset of our formalism, namely transitivity, cannot be guaranteed under these circumstances.

7.4.2 Inheritance of implementation with semantics support

Another combination is the *inheritance of implementation with semantics relation*. We feel that this option resembles closely the concept of *delegation*, which means that an unrelated class (outside the inheritance hierarchy) is made to have an ability to behave like a class in the inheritance class hierarchy. Designers keep the distinction for various reasons: security, integrity and politics, to name a few.

7.5 Conclusion

A theory of inheritance should not only tackle mathematical niceties such as transitivity, not only selected practical issues such as overriding. It would also be nice to have a more general theory to unify the four combinations of inheritance support. Clearly, the concept of inheritance is not fully defined and fully generic. However, given the fact that

- exceptions will conflict with transitivity;
- mathematical definitions may not fully capture the real-life practice and;
- a general theorem prover does not exist,

a totally formal definition of inheritance may not be feasible if not impossible. We are still in the dark age when the Periodic Table is the only means to organize chemical elements and reflects their similarities and differences.

Chapter 8

A comparison with CSP-related Inheritance formalizations

8.1 Introduction

Over the years, much research has been conducted in formalizing the inheritance relation. As a comparison with these research, we have taken some CSP-related approaches and evaluated them using the desirable features of a theory of inheritance identified in Chapter 2. We will then summarize the evaluation and compare them with our project.

8.2 Review

8.2.1 Cusack approach

Cusack [Cusack91a] illustrates the idea of inheritance as the conformance relation and realizes it in the CSP framework. A brief review of Cusack conformance has been given in Chapter 4.

$$\boxed{\begin{array}{l} \mathsf{CoffeeMachine} = \mathsf{coin} \rightarrow \mathsf{coffee} \rightarrow \mathsf{CoffeeMachine} \\ \mathsf{TeaAndCoffeeMachine} = \mathsf{coin} \rightarrow \mathsf{x}{:}\{\mathsf{coffee}, \mathit{tea}\} \rightarrow \mathsf{TeaAndCoffeeMachine} \end{array}}$$

Figure 8.1: An example of conforming process.

For example, the TeaAnd*CoffeeMachine*, as shown in Figure 8.1, conforms to the *CoffeeMachine* because the former simply extends the latter to produce a cup of *tea* as an alternative.

Cusack addresses the issue of behavioural compatibility as a conformance relation. Nevertheless, rather than formalizing the object–oriented inheritance, it tackles the class-based inheritance where properties like methods and queries cannot be propagated to a conforming process. It means that a conforming process should duplicate those properties in its process specification, and hence the approach does not provide an incremental development mechanism.

Although conformance relation is reflexive and anti-symmetric, it is unfortunately not transitive as illustrated in Chapter 4. We can redefine the process behaviour in the conforming process. For instance, TeaAnd*CoffeeMachine* can be viewed as an extension of the properties of *CoffeeMachine*. Nevertheless, we cannot reuse the original specification, in *CoffeeMachine*, to redefine those properties that need to be modified. Because of a lack of a process construction operator (like the use of the priority choice operator "$\overrightarrow{\Box}$" in our formalism) to link up the conforming process and the original process, overriding cannot be supported explicitly.

8.2.2 Rudkin approach

Rudkin [Rudkin92] proposes an inheritance hierarchy with incremental development in the LOTOS framework. To summarize Rudkin's work:

- It is based on the notion of *extension*[1], which ensures that a class hierarchy forms a subtyping hierarchy, since extension is reflexive, anti-symmetric and transitive. Nevertheless, as we note in Chapter 4, extension does not provide the necessary mechanism for overriding and redefinition since the formalism requires that the extending process should only add new behaviour. Furthermore, the very first event of these new behaviours cannot be in common with any first event of the superclass (the original process). For instance, the very first event of the *CoffeeMachine* is *coin*, and hence, the TeaAnd*CoffeeMachine* should not have any behaviour which starts with *coin*. Thus, the definition of TeaAnd*CoffeeMachine* stated in Figure 8.1 is invalid according to this approach.

- It facilitates an incremental modification approach using the choice operator [] to link up a superclass and a subclass under a few constraints.

- In addition of the standard processes stop and exit, it adds a process **self** as a primitive process to denote a source of redirection. However, the semantics of self is not formally defined in the paper.

- It adds a new modifier "*" to denote the redirection operator to identify the destination initiated by the self process.

For example, FlushableBuffer is a subclass of the Buffer as shown in Figure 8.2 (where symbol "⊕" denotes the inheritance operator). In this way, methods likes in and out need not be repeated in the specification of FlushableBuffer. Moreover, LOTOS enables the definition to be formal.

```
process Buffer [in, out] (q:queue): noexit:=
        in?x:integer; Buffer[in,out](x appends q)
    []
        [q ne empty] -> out!hd(q); Buffer[in,out](tl(q))
endproc

process FlushableBuffer [in, out, flush] (q:queue): exit(nat) :=
        Buffer[in,out](q)
    ⊕
        (flush; FlushableBuffer [in, out, flush](empty)
endproc
```

Figure 8.2: Inheritance example of Rudkin's View

[1]See Section 4.2.1 of this thesis.

The approach restricts the initial communication of the modifier (subclass template) to be distinct from the initial communication of its superclasses. Hence, there cannot have any overriding or redefinition. Multiple inheritance is not addressed.

8.2.3 Clark approach

In Rigorous Object-Oriented Analysis [Clark93], two proposals for inheritance, namely pure extension and inheritance with redefinition of services, are presented.

Proposal #1: Pure extension

A subclass extends its superclass if it only adds new features without removing any original behaviour. The subclass should be behaviourally compatible with its superclass. Based on the transitivity of simple extensions, transitive relation among classes in the inheritance hierarchy can be guaranteed. The proposal employs incremental modification technique in constructing subclasses. It also addresses the multiple inheritance. Services in superclasses can be readily used by subclasses. However, as the name implies, no redefinition or overriding is allowed. More general concepts like conformance and exceptions are not addressed.

```
process Account[g](bal:Money) : exit(Money) :=
        g!deposit?m:Money;exit(Credit_Account(bal,m))
    []
        g!getbalance;exit(bal)
    []
        g!withdraw;exit(Debit_Account(bal,m))
endproc
process RedefineChequeAccount[g](bal:Money, cards:Card) : exit(Money,Card) :=
        g!deposit?m:Money;exit(any Money, Card_Op(cards))
    []
        g!getbalance;exit(any Money, cards)
    []
        g!withdraw;exit(any Money, cards)
endproc
process ChequeAccount[g](bal:Money, cards:Card) : exit(Money,Card):noexit :=
        (((Account[g](bal) >> accept newbal: Money
                    in exit(newbal, any Card))
              |[g]| RedefineChequeAccount[g](bal,card))
    []
        g!print_statement;exit(bal,card))
    >> accept update_account: Account, update_card:Card
        in ChequeAccount[g](update_account,update_card)
endproc
```

Figure 8.3: Inheritance in ROOA

Proposal #2: Inheritance with redefinition of services

The second proposal is *inheritance with redefinition of services*. A simplified bank account example is shown in Figure 8.3. Account allows users to deposit, get balance and withdraw money. ChequeAccount redefines the service deposit by allowing user to use a card too. In this approach, a redefinition process is required for specifying the interface of the services to be inherited. The process is then made to have a parallel composition with the superclass, as shown in the figure, to form a part of the ChequeAccount specification.

This approach demonstrates an incremental modification technique. However, transitivity and behavioural compatibility are not supported. It does not present a formal definition of inheritance. Moreover, the redefinition of services is advised by means of a low-level technique. Because of the requirement to repeat the inherited methods (interface) in the redefinition processes, the services of the superclass cannot be used easily by the subclasses. Extension, conformance and exceptions are not explicitly supported.

8.2.4 Summary

We summarize the review presented in this section in the following table. The last column is the comparison of this project. In the table, $ROOA_1$ and $ROOA_2$ are the proposes one and two of Clark's approach respectively.

	Cusack	Rudkin	$ROOA_1$	$ROOA_2$	**This Project**
Incremental		√	√	√	√
Behavioural Compatible	√	√	√		√
Formal	√	√			√
Transitive			√		√
Re-definable	√			√	√
Overriding				√	√
Automatically Propagated		√	√		√
Multiple Inheritance			√	√	√
Extension Guaranteed	√	√	√		√
Conformance Guaranteed	√				√
Exception				√	

Chapter 9

Conclusion

This thesis proposes a theoretical framework for inheritance. We have formalized the concept of behavioural inheritance as a special kind of conformance relation that links a subclass by Fidge's priority choice operator to its superclass. With the priority choice operator, we also integrated naturally the concept of overriding into our formalization. Our definition is reflexive, anti-symmetric and transitive. Thus, classes form a partial ordering in an inheritance hierarchy. Guidelines for incremental modification of a superclass to form subclass is also proposed to facilitate reuse.

We have studied the effect of an inheritance with more than one superclass, their interference and impacts on the subclass. Based on our study, we have proposed theories to tackle the behaviourally conflicting superclasses and generalize our results to multiple inheritance.

The research compares favourably with other related work. Our approach reinforces the transitivity of behavioural compatibility while others either ignore the issue, require personal discipline of users in the specification process, or support only extension which is a special case of our formalism. Moreover, we have successfully demonstrated a simple application of our theory in an account example using the Object Modeling Technique (OMT) [Rumbaugh91].

In short, this research focuses on the formalization of a key concept in object-orientedness — inheritance — and also successfully tackles the issue of overriding. At the same time, it refines conformance into a transitive relation. Furthermore, it provides great potential for further development of a formal framework behind OMT methodology using the elegant theory of CSP.

Appendix A

A CSP Overview

A.1 Introduction

Communicating Sequential Processes, CSP, was developed by Tony Hoare in late 70s to early 80s. In this research, we use the definition of CSP together with the notation and behaviour laws given in [Hoare85].

CSP is a kind of process algebra that specialize in concurrency modeling. The basic concept is a *process*. The interactions between a system and its environment are modelled mathematically as events.

A.2 Processes

A system is described in CSP as a set of communicating processes. Processes are black boxes and communicate through their externally observable behaviours with other processes. Hence, a system is specified by defining the temporal relationships among these interactions. It uses a synchronous communication. Asynchronous communication can be established through the use of bufferred processes. Events are atomic actions and an event takes place only if all the components of a system are ready to synchronize that event. For example, in the event:

channel.data

"channel" is the channel of communication, "data" is the actual information for synchronization in that channel. The data can be a variable; synchronization causes these variables to instantiate. Table A.2 summarizes the three types of synchronizations.

Process *P*	**Process *Q***	**Interaction Type**	**Effect**
channel!data	channel!data	Value matching	Events synchronize
channel!data	channel?x	Value passing	Synchronization occurs and "x" will be instantiated to "data"
channel?y	channel?x	Value generation	After synchronization, "y" = "x"

Table A.1: Three synchronization types.

Semantically, every process *P* has a signature called the alphabet, a failure set and a divergence set. The signature is donated by αP.

STOP and **CHAOS** are two fundamental processes in CSP. The former represents the end of lifespan of processes and the latter, as the named implies, denotes the most non-deterministic situation where the process may or may not refuse or accept any event. CSP introduces an event $\surd$ to represent a successful termination of a process and use the process **SKIP** to denote

$$\textbf{SKIP} = \surd \rightarrow \textbf{STOP}$$

A.3 Operators

In the following sections, we will give the basic ideas of some operators. We adopt the following convention:

- a is an event.
- P and Q are processes.
- C is a set of alphabet.

A.3.1 Prefix

($a \rightarrow P$) is pronounced as "a then P". It describes a process that first engages in an event a and then behaves exactly like P. For instance, when we use our bank card to withdraw cash, we first insert the card, and then key in the password and the amount required. Finally, we get some cash. The whole situation is abstractly captured by the following process.

$$\text{insert card} \rightarrow \text{key-in password} \rightarrow \text{key}-\text{in amount} \rightarrow \text{get cash} \rightarrow \textbf{SKIP}$$

A.3.2 Recursion

Although the prefix notation can be used to describe the entire behaviour of a process that eventually stops, it will be impossible to write out the full behaviour of any non–terminating process. For instance, a simple CLOCK will *tick* and then *tick* again forever. We use

$$\text{CLOCK} = \text{tick} \rightarrow \text{CLOCK}$$

to provide a recursive definition for CLOCK. In general, a recursion is an equation $P = f(P)$.

A.3.3 Or

($P \sqcap Q$) is pronounced as "P or Q". It describes a process that behaves either exactly like P or exactly like Q, where the selection is made arbitrarily not within the control or the knowledge of the external environment. For instance, to change a \$10 note to coins, a change-giving machine always gives the right change in one of two combinations as shown in next page.

$$\begin{aligned}\text{Ch10} = (\text{in10} \rightarrow (\ &(\text{out1} \rightarrow \text{out2} \rightarrow \text{out2} \rightarrow \text{out5} \rightarrow \text{Ch10})\\ &\sqcap\\ &(\text{out5} \rightarrow \text{out2} \rightarrow \text{out2} \rightarrow \text{out1} \rightarrow \text{Ch10}))\end{aligned}$$

A.3.4 Choice

($P \square Q$) is pronounced as "P choice Q". It describes a process that allows the environment to select P or Q by exercising the very first action. For example, we are allowed to select to get a coke or an apple juice after we insert a coin in a vending machine.

$$\text{insert coin} \rightarrow ((\text{coke} \rightarrow \textbf{STOP}) \square (\text{apple juice} \rightarrow \textbf{STOP}))$$

However, if some of the very first actions are shared by P and Q, then the choice between P and Q among those common actions will become non–deterministic.

A.3.5 Concurrency

($P \parallel Q$) describes a process that behaves as if the system is composed of processes P and Q interacting in lock-step synchronization.

For instance, a greedy customer of a vending machine is happy if he obtains a toffee or a chocolate free of charge. However, if he has to pay, he would insist on getting a chocolate.

$$\begin{aligned} \text{Greedy Customer} = (\ &(\text{toffee} \rightarrow \text{Greedy Customer} \\ &\ \sqcap \\ &\ \text{chocolate} \rightarrow \text{Greedy Customer}) \\ &\square \\ &\text{coin} \rightarrow \text{chocolate} \rightarrow \text{Greedy Customer}) \end{aligned}$$

A well–designed vending machine should request the customer to insert a coin before it gives a toffee or a chocolate.

$$\begin{aligned} \text{Vending Machine} = \text{coin} \rightarrow (\ &(\text{chocolate} \rightarrow \text{Vending Machine}) \\ &\square \\ &(\text{toffee} \rightarrow \text{Vending Machine})) \end{aligned}$$

When the greedy customer comes to this vending machine, his greed is frustrated since the machine does not allow anything to be extracted before payment. Hence, the only choice the customer has is to insert a coin and then get a chocolate.

$$\begin{aligned} &(\text{Greedy Customer} \parallel \text{Vending Machine}) = \\ &\qquad \text{coin} \rightarrow \text{chocolate} \rightarrow (\text{Greedy Customer} \parallel \text{Vending Machine}) \end{aligned}$$

A.3.6 Interleaving

($P \mid\mid\mid Q$) is pronounced as "P interleave Q"" Unlike the $\parallel$ combinator, $\mid\mid\mid$ introduces a possibility of a system of non-interacting components with the same alphabet. For instance, we can place 4 independent vending machines together on a street.

$$\text{Vending Machine} \mid\mid\mid \text{Vending Machine} \mid\mid\mid \text{Vending Machine} \mid\mid\mid \text{Vending Machine}$$

A.3.7 Concealment

$(P \backslash \mathrm{C})$ describes a process that behaves like P except that each occurrence of any event in C is concealed. Hence, those concealed events becomes internal actions and represent internal communication among components of P.

For instance, a noisy vending machine may *clink* for inserting a coin and *clunk* for dropping a toffee.

$$\begin{aligned}\text{Noisy Vending Machine} = {} & \text{coin} \rightarrow \text{clink} \rightarrow \text{toffee} \rightarrow \\ & \text{clunk} \rightarrow \text{Noisy Vending Machine}\end{aligned}$$

The noisy vending machine can be placed in a soundproof box. The resulting process is equivalent to a simple vending machine.

$$\begin{aligned}\text{Simple Vending Machine} &= \text{Noisy Vending Machine} \backslash \{\text{clink,clunk}\} \\ &= \text{coin} \rightarrow \text{toffee} \rightarrow \text{Simple Vending Machine}\end{aligned}$$

A.4 Failure–divergence model

In failure–divergence model, a process is represented by a tuple (A,F,D) where “A” is the alphabet set, “F” is the set of failures and “D” is the set of divergence.

A.4.1 Trace

A trace of a process is a sequential record of the behaviour of that process up to some moment in time. A complete set of all possible traces of a process P is denoated by traces(P).

For example, traces(Vending Machine) denotes the complete set of traces of the process

$$\text{Vending Machine} = \text{coin} \rightarrow \text{coke} \rightarrow \textbf{SKIP}$$

and is $\{\ \langle\rangle, \langle coin\rangle, \langle coin, coke\rangle, \langle coin, coke, \surd\rangle\}$.

A.4.2 Failures

Suppose a process P can engage in a sequence of events denoted by the trace s, and then fails to proceed if the environment is prepared to engage in any event in a set X. The tuple (s,X) will be known as a failure of P. We use failures(P) to denote the set of possible failures of a P.

For example, the set of failures of Vending Machine is $\{(\langle\rangle,\{\text{coke}\}), (\langle coin\rangle,\{\text{coin}\}), (\langle coin, coke\rangle, \{\text{coin,coke}\}), (\langle coin, coke, \surd\rangle,\{\text{coin,coke}\}\}$.

A.4.3 Divergence

A divergence of a process P is defined as any trace t of P after which P behaves chaotically. We use divergence(P) to denote the set of possible diverging traces. Since a divergence is a trace, divergence(P) should be a subset of traces(P).

A.4.4 Other models

The failure–dievergence model of CSP uses the traces, failures and divergence to attach a meaning to a process. We can extend the model to other models. For instance, by including the real–time consideration, we would have the real-time specification language TCSP [Davies93].

A.5 Summary

CSP has a rich set of operators and encourages users to define their own operators. The following table summarizes the basic operators and the priority choice operator.

Name	***Syntax***	***Meaning***
Prefix	(a $\rightarrow$ *P*)	A process that firstly engages in an event *a* and then behaves exactly like *P*.
Recursion	*P* = f(*P*)	Describes recursive behaviour patterns.
Choice	$P \,\square\, Q$	A process that allows the environment to choose *P* or *Q* by exercising the very first event.
Or	$P \sqcap Q$	A process where the selection of *P* and *Q* is internal and non–deterministic.
Priority Choice	$P \,\overset{\rightarrow}{\square}\, Q$	Similar to the choice operator, but *P* is given a higher preference.
Concurrency	$P \parallel Q$	*P* parallels with *Q*. interacting in a lock–step synchronization.
Interleaving	$P \,|||\, Q$	*P* interleaves with *Q* non–interactively.
Concealment	$P \backslash C$	A process which behaves like *P*, except that any alphabet in C would be concealled.

Table A.2: A summary of basic operators in CSP.

Bibliography

[Armstrong94] J.M. Armstrong and R.J. Mitchee, “Uses and abuses of inheritance”, *Software engineering journal*, January 1994, pp 19 – 26.

[Bear88] S. Bear, “Structuring for the VDM specification languages”, *VDM’88*, Springer–Verlag Berlin, 1988, pp 2 – 25.

[Boehm81] B.W. Boehm, *Software engineering economics*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Booch94] G. Booch, *Object oriented analysis and design with applications*, 2nd Edition, Benjamin-Cummings, Redwood City, California, 1994.

[Bryant90] A. Bryan, “Structured methodologies and formal notations: developing a framework for synthesis and investigation”, *Z user workshop: proceedings of 4th annual Z user meeting*, Springer–Verlag Berlin, 1990.

[Car90] V. Carchiolo and G. Pappalardo, “On the design of communication systems by ECCS and CSP based approaches”, *Proceedings of IEEE region 10 conference on computer and communication systems (TENCON’90)*, 1990, pp 380 – 385.

[Cusack91a] E. Cusack, “Refinement, conformance and inheritance”, *Formal aspects of computing*, Vol 3, 1991, pp 129 – 141.

[Cusack91b] E. Cusack, “Inheritance in object-oriented Z”, *ECOOP ’91, European conference on object-oriented programming, lecture notes in computer science* Vol 512, Spring-Verlag Berlin, 1991, pp 167-179.

[DeMarco79] Tom DeMarco, *Structured analysis and system specification*, Prentice-Hall Englewood Cliffs, New Jersey, 1979.

[Davies92] J. Davies and S. Schneider, *A brief history of timed CSP*, Programming Research Group, Oxford University Computing Laboratory, Oxford, 1992.

[Davies93] J. Davies, *Specification and proof in real time CSP*, Distinguished Dissertations in Computer Science, Vol 6, Cambridge University Press, Cambridge, 1993.

[FDR93] *Failures divergence refinement*, Formal Systems (Europe) Ltd, Oxford, 1993.

[Fidge93] C.J. Fidge, “A Formal definition of priority in CSP”, *ACM transaction on programming languages and systems*, Sept 93, Vol 15, No 4.

[Fraser91] M.D. Fraser, K. Kumar and V.K. Vaishnavi, “Informal and formal requirements specification languages: bridging the gap”, *IEEE transactions on software engineering*, 17(5), 1991, pp 454 – 466.

[Fuchs92] N.E. Fuchs, “Specifications are (preferably) executable”, *Software engineering journal*, 7(5) September 1992, pp 323 – 334.

[Gaskell94] C. Gaskell and R. Phillips, “Executable specifications and CASE”, *Software engineering journal*, July 1994.

[Goldberg94] A.Goldberg and D. Robson, *Smalltalk-80: the interactive programming environment*, Addison-Wesley, Reading, Massachusett,1984.

[Hall90] A. Hall, “Seven myths of formal methods”, *IEEE software*, September 1990, pp 11-20.

[Hayes89] I.J. Hayes and C.B. Jones, “Specifications are not (necessarily) executable”, *Software engineering journal*, 4(6) November 1989, pp 330 – 338.

[Hoare85] C.A.R Hoare, *Communicating sequential processes*, Prentice-Hall, Hamel Hempstead Hertfordshire, 1985

[Hoare89] C.A.R. Hoare *et al.*, *A theory of asynchronous processes*, Technical Report PRG-TR-6-89, Oxford University Computing Laboratory, Oxford, 1989.

[HOOD92] HOOD User Group, *HOOD reference manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[ISO8807] *LOTOS: a formal description technique based on the temporal logic*, ISO 8807.

[Jackson82] M. Jackson, *Systems development*, Prentice-Hall, Hamel Hempstead Hertfordshire, 1982.

[Jacobson92] I. Jacobson, *Object–oriented software engineering : a use case driven approach*, Addison-Wesley, 1992.

[Jones90] C.B. Jones, *Systematic Software Development using VDM*, Prentice-Hall International Series in Computer Science, Prentice-Hall, Hemel Hempstead, Hertfordshire, UK, 1990.

[Martin92] J. Martin and J.J. Odell, *Object–oriented analysis and design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Meseguer92] J. Meseguer, “Conditional rewriting logic as a unified model of concurrency”, *Theoretical computer science* 96, 1992, pp 73 – 155.

[Meyer90] B. Meyer, *Eiffel: the Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Milner86] R. Milner, *A calculus of communicating systems*, Technical Report ECS-LFCS-86-7, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, 1986.

[Clark93] A. Moreira and R. Clark, *ROOA: Rigourous object-oriented analysis method*, Technical Report TR109, Department of Computing Science and Mathematics, University of Strirling, October 1993.

[Nerson91] J.-M. Nerson, “Extending Eiffel toward O–O analysis and design”, *Technology of object-oriented languages and systems: proceedings of 5th international conference TOOLS 5*, 1991.

[Par72] D.L. Parnas, “On criteria to be used in decomposing systems into modules”, *Communication of ACM*, Vol 14, No 1, April 1972, pp 221 – 227.

[Rudkin92] S. Rudkin, “Inheritance in LOTOS”, *Formal description techniques IV*, KR. Parker and G.A. Rose (Eds.), North–Holland Amsterdam , pp 409 - 424.

[Rumbaugh91] , J. Rumbaugh *et al.*, *Object–oriented modeling and design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Pandya90] P.K. Pandya and C.A.R. Hoare, *JSD in asynchronous CSP* , Programming Research Group, Oxford University Computing Laboratory, Oxford, 1990.

[Peterson81] J.L. Peterson, *Petri net theory and the modeling of systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Porter92] H.H. Porter III, “Separating the subtype hierarchy from the inheritance of implementation”, *Journal of object-oriented programming* February 1992, pp 20 – 29.

[Pressman92] R.S. Pressman, *Software engineering: a practitioner’s approach*, McGraw-Hill, New York, 1992, 3rd Edition.

[Schn93] S. Schneider, “An operational semantics for timed CSP”, *Information and computation*, 1993.

[Seidel92] K. Seidel, *Probabilistic communicating processes*, D.Phil. Thesis, University of Oxford, Oxford, 1992.

[Sommerville92] I. Sommerville, *Software engineering*, Addison-Wesley, 4th Edition, 1992.

[Spivey92] J.M. Spivey, *The Z notation: a reference manual*, Prentice-Hall, Hemel Hempstead, Hertfordshire, UK, 1992.

[Stepney92] S. Stepney, R. Barden and D. Cooper, “A survey of object-orientation in Z”, *Software engineering joural*, 7(2), March 1992, pp 150 – 160.

[Stroustrup91] B. Stroupstrup, *The C++ programming language*, 2nd Edition, Addison Wesley, Reading, Massachusetts, reprinted in 1992.

[Tse91] T.H. Tse, *A unifying framework for structured analysis and design models: an approach using initial algebra semantics and category theory*, Cambridge University Press, Cambridge, 1991.

[Tse92a] T.H. Tse and C.P. Cheng, “Towards a 3–dimensional net-based object-oriented development environment (NOODLE)”, 5th international congress on computational and applied mathematics (ICCAM ’92), 1992.

[Tse92b] T.H. Tse and J.A. Goguen, “Functional object-oriented design”, *Dagstuhl seminar report*, International Conference and Research Center for Computer Science, Wadern, Germany, 1992.

[Tse93] T.H. Tse, “Formal or informal, practical or impractical: towards integrating formal methods with informal practices in software engineering education”, *software engineering education: proceedings of IFIP WG 3.4 working conference*, North-Holland, Amsterdam, 1993, pp 189 – 197.

[Tsvi92] B.D. Tsvi, *Practical consequences of formal definitions of inheritance*, Journal of object-oriented programming, July/August 1992, pp 43 – 49.

[Wegner88] P. Wegner *et al.*, “Inheritance as an incremental modification mechanism, or what like is and isn’t like”, *Proceedings of 2nd european conference on object-oriented programming*, 1988.

[Wing90] J.M. Wing, “A Specifier’s introduction to formal methods”, *IEEE computer*, Vol 23, No 9, September 1990, pp 8 – 24.

[Yourdon89] E. Yourdon, *Modern structured analysis*, Yourdon Press, 1989.

[Yourdon91a] P. Coad and E. Yourdon, *Object–oriented analysis*, Prentice-Hall, Englewood Cliffs, New Jerse, 1991.