# Towards a New Extension Relation for Compositional Test Case Generation for CSP Concurrent Processes

**Chan, Wing Kwong**

A thesis submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy
at The University of Hong Kong.

September 2003

Abstract of thesis entitled

**Towards a New Extension Relation for Compositional Test Case Generation for CSP Concurrent Processes**

submitted by

**Chan, Wing Kwong**

for the degree of Doctor of Philosophy
at The University of Hong Kong
in September 2003

Software concurrent systems such as electronic financial services are very popular today. Rigorous testing of such systems is indispensable. Nevertheless, the state-explosion problem, in which the size of a system grows exponentially with the number of concurrent components, is a severe obstacle to the generation of test cases.

In this study, we propose a new relationship assuring the conformities of test cases for concurrent systems. We stipulate it in Communicating Sequential Processes (CSP), which is an excellent tool to study concurrent systems. Specification of concurrent systems is expressed as processes that are composed sequentially and concurrently. Our proposal supports both abstraction and process composition.

This approach decomposes a given process into sequential compositions of component processes. Each component can be anti-extended into abstract forms. Sequential and concurrent combinations of abstract forms can substitute their corresponding extending components in a process to form aggregated abstract forms. Since the given processes, components and abstract forms are all processes, this approach can be applied recursively and compositionally. We prove that these aggregated abstract processes should be anti-extensions of the corresponding processes under a few necessary and sufficient conditions. Hence, test cases generated from these processes will be conformance test cases for the implementations.

We examine three major testing relations (conformance, extension and reduction) from the point of view of their ability to support abstraction. Extension and reduction are special kinds of conformance, which are partially ordered. However, only extension from a specification can provide abstract forms that guarantee to be conformed by an implementation of the specification. Extension is further investigated to support the compositional approach. We identify five weaknesses of extension, which we call problems of preceding, succeeding, prefix, decomposition and paralleling. Each will result in non-conforming abstraction.

To resolve the identified weaknesses of extension in a compositional and abstraction-oriented approach, we formulate $\sigma$-extension and identify the necessary and sufficient conditions. We also discuss a restricted sub-class of process specification so that these conditions can easily be satisfied. In the restricted sub-class, successful terminations of processes are deterministic.

Under these conditions, we achieve a few desirable properties. First, both preceding and succeeding components of a given sequential composition can be anti-extended. The

given composition should extend the sequential composition of its corresponding anti-extended processes. Secondly, a given parallel composition whose concurrent components are sequential compositions should extend a parallel composition of anti-extensions of preceding components of corresponding sequential compositions, followed by a parallel composition of anti-extensions of corresponding succeeding components. Furthermore, a given sequential composition should $\sigma$-extend the preceding component of a resultant sequential composition. We also illustrate the generation of test cases and discuss possible enhancements.

In conclusion, this study achieves three important aims:

(i) we identify extension as a superior tool for abstraction over conformance and reduction from the testing perspective, and identify its weaknesses in compositional approach;

(ii) we propose an abstraction-oriented and compositional approach to test case generation; and

(iii) we recommend a restricted sub-class of CSP specification that can utilize our results easily.

# Towards a New Extension Relation for Compositional Test Case Generation for CSP Concurrent Processes

**Chan, Wing Kwong**

A thesis submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy
at The University of Hong Kong.

September 2003

*To my parents*

# Declarations

I hereby declare that this thesis, submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy and entitled "Towards a New Extension Relation for Compositional Test Case Generation for CSP Concurrent Processes", represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

________________________

Chan, Wing Kwong

September 2003

# Acknowledgements

It is my honour to express my gratitude to people caring me in my research study by this acknowledgments.

No doubt, the first person that I would like to express my sincere thanks is my research supervisor Dr T.H. Tse for his profound guidance, encouragement and endless support on my pursuit of this research study throughout these years. Special thanks also go to Prof T.Y. Chen, my external examiner, for his valuable comments. I would like to thank Dr Francis C.M. Lau, my internal examiner, for his evaluation of the research and his great help to support my research study. It is truly my luck to have the opportunity to learn from them, especially when I lost in the dark.

It is also a real pleasure to thank Prof H.Y. Chen, Dr F.T. Chan, and Dr Karl R.P.H. Leung for their discussions, Mr W.C. Ying and Dr Michael Luk for their support of my study during my time at Computer Centre, The University of Hong Kong and HKU SPACE, Mr Paul Y.H. Hsi, Mr Y.N. Lee, Mr W.T. Yeung and my brother Mr W.K. Chan for their valuable helps at my most critical time, and my sister Ms Biddy K.W. Chan for her help in preparation of this thesis. There are plenty on the list for me to acknowledge such as the colleagues at the Department, members of The Software Engineering Group and the friends around campus. I should thank ASM Assembly Automation Ltd. to provide domain knowledges of bonding machines.

I would like to express my deepest gratitude to my parents Mr Pat Chan and Ms Siu Kee Shing. To them, I dedicate this thesis.

Finally, I would like to thank The University of Hong Kong for partial financing by grants of the Research Grants Council of Hong Kong (Project Nos. HKU 7033/99E and 7029/01E), a grant of the Innovation and Technology Fund (Project No. UIM/77), and a research and conference grant of The University of Hong Kong.

# Contents

**Declarations** i

**Acknowledgements** iii

**Contents** v

**List of Figures** vii

**List of Tables** ix

**1 Introduction** 1
- 1.1 Background . . . 1
- 1.2 Software Testing . . . 1
- 1.3 Difficulties in Testing Concurrent Programs . . . 2
- 1.4 Our Proposal . . . 2
- 1.5 Organization of Thesis . . . 5

**2 Literature Review on Conformance Testing** 7
- 2.1 Functional Approach . . . 7
- 2.2 Parallel Composition Approach . . . 8
- 2.3 Checking Sequence Approach . . . 9
- 2.4 Conventional Approach . . . 10
- 2.5 Other Techniques . . . 10
- 2.6 Comparison with Our Work . . . 11

**3 Communicating Sequential Processes** 12
- 3.1 Syntax and Semantics . . . 12
- 3.2 Graphical Illustration . . . 18
- 3.3 Sequential Composition in CSP . . . 20
  - 3.3.1 Special Algebraic Laws for Sequential Composition . . . 22
- 3.4 Parallel Composition in CSP . . . 22
  - 3.4.1 Special Algebraic Laws for Parallel Composition . . . 23
- 3.5 Chapter Summary . . . 23

**4 Conformance Testing** **24**
4.1 The Notion of Conformance Testing . . . . . 24
4.1.1 Conformance, Reduction and Extension . . . . . 24
4.2 Justification of Our Proposal . . . . . 27

**5 The Problems of Extension Hierarchy** **29**
5.1 Problem 1: Preceding . . . . . 29
5.2 Problem 2 : Suceeding . . . . . 31
5.3 Problem 3: Prefix . . . . . 32
5.4 Problem 4: Decomposition . . . . . 33
5.5 Problem 5: Paralleling . . . . . 37
5.6 Chapter Summary . . . . . 41

**6 A New Relation for Compositional Test Case Generation** **42**
6.1 Introduction . . . . . 42
6.2 Sequential Extension . . . . . 42
6.3 $\sigma$-Extension . . . . . 50
6.4 Sequential Constraints . . . . . 57
6.5 Parallel Constraints . . . . . 76
6.6 Chapter Summary . . . . . 93

**7 Discussions** **94**
7.1 The Chosen Semantic Model . . . . . 94
7.2 Applicability to Other Process Algebra . . . . . 95
7.3 Anti-Extension and Testing Criteria . . . . . 95
7.4 Our Framework and Test Case Generation Approaches . . . . . 96
7.5 Modules with Manageable Number of States . . . . . 97
7.6 Method Integration . . . . . 97
7.7 Comparisons to Selected Work . . . . . 98
7.7.1 Abstraction as Extension and Refinement . . . . . 98
7.7.2 Parallel Composition . . . . . 99
7.8 Our Inspiration, Real-Life Examples and Testbed . . . . . 100

**8 Conclusions** **102**

**A Appendix** **105**
A.1 A Few Results on Conformance, Extension and Reduction . . . . . 105

**Bibiography** **109**

# List of Figures


1.1 Conventional approach . . . 3
1.2 Our proposal . . . 4

3.1 CSP and graphical convention . . . 19
3.2 Sequential composition of CSP processes . . . 21

4.1 Examples of conformance, reduction and extension relations . . . 25

5.1 Problem Preceding . . . 30
5.2 Problem Succeeding . . . 31
5.3 Problem Prefix . . . 32
5.4 Problem Decomposition (a) . . . 33
5.5 Problem Decomposition (b) . . . 34
5.6 Problem Decomposition (c) . . . 36
5.7 Problem Decomposition (d) . . . 37
5.8 A pick arm mounted with a movable vision camera . . . 38
5.9 Process *SimplePickArm* and its sequential components . . . 39
5.10 Process *Vision* and its sequential components . . . 39
5.11 Parallel composition of processes . . . 40
5.12 Problem Paralleling . . . 40

6.1 Examples of sequential extension . . . 45
6.2 Sequential Extension . . . 45
6.3 Example 1 of σ-Extension. . . . 51
6.4 Example 2 of σ-Extnesion . . . 52
6.5 σ-Extension implies overall extension . . . 58
6.6 Tail Anti-Extension . . . 59
6.7 Head Anti-Extension . . . 61
6.8 An example of head anti-extension preserving overall extension . . . 62
6.9 Compositional Anti-Extension Hierarchy by σ-Extension . . . 64
6.10 Sub-testcase extraction from various levels of abstraction . . . 67
6.11 Extension and Reduction hierarchies at a glance . . . 75
6.12 Parallel constraints problem . . . 76
6.13 A bonding machine . . . 84
6.14 The bonding machine in CSP processes . . . 86
6.15 Components of other processes of the bond machine . . . 90

# List of Tables

3.1 Trace sets of processes *Inspection*, *Ranking* and *Inspection*;*Ranking* . . . . . . 21

6.1 A summary of our solution to extension problems . . . . . . . . . . . . . . . . . 42
6.2 A reference table of condition list in Definition 6.2.1 . . . . . . . . . . . . . . . 44

A.1 A summary of a few useful results in conformance, extension and reduction . . 105

# Chapter 1

# Introduction

## 1.1 Background

Concurrent applications, such as mobile games, network protocols, embedded control software, multi-threaded programs, operating systems and databases, are ever-present in modern systems. Assuring the quality of these applications is indispensable.

Generally speaking, software testing and formal proving are two complementary and non-competing means to assure the quality of any implementation. Formal proving relies on precise models of the implementation to deduce important properties such as safety, liveness and refinement. Nevertheless, the reachability problem and the boundedness problem in the communicating finite state machines model, one of the basic models of concurrent systems, are undecidable [12, 46]. Finkel and Mckenzie even conclude that identical communicating processes are "extremely difficult to verify" [46]. If both specification and implementation fall into this class of processes, such as the case of the concurrent applications mentioned above, formal proving of important properties will be very difficult. Consequently, software testing remains one of the most practical means to assure the quality of an implementation of a system of communicating processes.

## 1.2 Software Testing

Software testing is a process to assure software quality by discovering symptoms caused by faults [7]. It can be used to show the presence of bugs, but not to show their absence [41]. In general, we can classify the process into black-box or white-box testing. We can also classified it as specification-based or program-based. Black-box testing is also known as functional testing. The goal is to verify whether the implementation behaviour is consistent with the specification. In this approach, implementation details are ignored. Black-box testing techniques such as random testing and partition testing usually makes use of the information on the input domain. Low cost is a key advantage of this approach. On the other hand, white-box testing assumes knowledge on implementation details. The program-based approach considers different kinds of models applicable to the implementation structure to design test suites. Specification-based white-box testing is also known as grey-box testing. In this approach, both specification and implementation are taken into account.

In general, software testing consists of four steps, namely defining testing objectives,

test design, test execution and test result verification. The definition of testing objectives describes specifically the purposes and scope of testing to accomplish. Test design includes the formulation of testing strategy and the construction of test suites. Test suite construction is the procedure to select test cases to satisfy the chosen testing criteria. Test execution concerns the application of test cases to implementation. It can also be static or dynamic. Static techniques include static analysis and simulation. Dynamic techniques often refer to the execution of test cases by programs. Test case for execution can in general be symbolic or concrete. Symbolic test cases are often used with simulation techniques. Concrete test cases are those test cases that can be directly executed by the implementation. Test execution methods can also be automatic, semi-automatic or manual. Test result verification is a procedure to verify the outcome of a set of tests. It can be done with or without a testing oracle. A testing oracle is an object that specifies the expected outcome of a set of tests as applied to a program under test. All four steps can be formal or informal.

In view of the limited budget and time, the intractable amount of legitimate and illegitimate test cases, as well as the potential absence of a testing oracle, software testing is challenging.

## 1.3 Difficulties in Testing Concurrent Programs

Concurrent programs incur the difficulties of their sequential counterparts as well as new challenges arising from their internal concurrencies. These programs can be considered as a collection of smaller processes running concurrently and communicating with one another from time to time. A process, such as the presence of an infinite WHILE-loop in C++, may be non-terminating. Moreover, if the processes are depicted as a process graph, it is well known to be PSPACE-complete[1] to conclude whether a node in the graph can be reachable from the initial state. This problem is known as the *reachability problem*. In addition to these sequential difficulties, owing to concurrency amongst processes, the number of possible global states increases exponentially with respect to the number of processes. This is known as the *state-explosion problem*. Hence, the size of the flattened structure of the composite process graph is formidably large, which makes direct structured test case generation impractical. Hierarchical or compositional approaches are often used to tackle this problem. [68] reports that the minimal number of transitions in a path to connect two nodes in the flattened graph can grow exponentially with the number of processes in the concurrent system. Non-determinism also complicates the testing. A feasible sequence in an experiment does not warrant that it can be repeated in another experiment. In software testing, it is frequently required to make the complete testing assumption, namely that any feasible output of a non-determinism can be obtained in a finite number of repeated experiments; or the forced execution assumption, namely that non-determinism can be resolved deterministically by some means, such as controlling the process execution priority at the operating system level [57, 103].

## 1.4 Our Proposal

We propose a compositional method to facilitate the generation of test cases for systems of communicating processes. We suggest decomposing processes into a sequential composition of component processes. Each component can be anti-extended into abstract forms. Sequential

[1] In other words, we need to try all possible alternatives.

and concurrent combinations of abstract forms can substitute their corresponding extending components in a process to form aggregated abstract forms. Since the given processes, components and abstract forms are all processes, this approach can be applied recursively and compositionally. We shall show that these aggregated abstract processes should be anti-extensions of the corresponding processes under a few necessary and sufficient conditions. Hence, test cases generated from these processes will be conformance test cases for the implementations.

To our best knowledge, the novelty of our approach includes, but are not limited to, the following:

I. It is the first proposal to identify anti-extensions from specifications as test cases for conformance testing.

II. It improves on the weaknesses in the conventional notion of extension in conformance testing in terms of compositional support.

III. It integrates the improved notion of extension with the compositional property in CSP for sequential and parallel compositions.

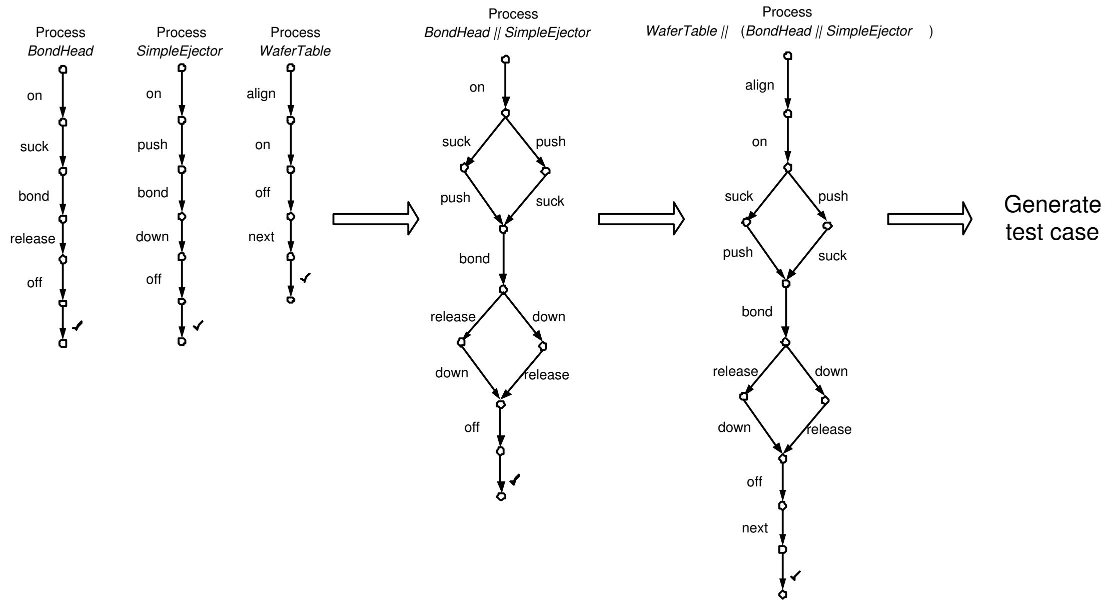


Figure 1.1: Conventional approach

Figure 1.1 depicts a conventional technique to generate test cases for concurrent systems. There are three processes, namely the *BondHead*, the *SimpleEjector* and the *WaferTable*. The *BondHead* and the *SimpleEjector* coordinate to hold a die firmly so that the *BondHead* can make a bonding. The conventional technique would first compose, say, the *BondHead* and the *SimpleEjector* to form an intermediate process depicted as the process in the middle. Then, this intermediate process will compose with the *WaferTable* to become the process on the right. Test cases will be generated according to this composed process. For instance,

$$\langle align, on, suck, push, bond, release, down, off, next \rangle$$

is a test sequence (that is, a path) of this process.

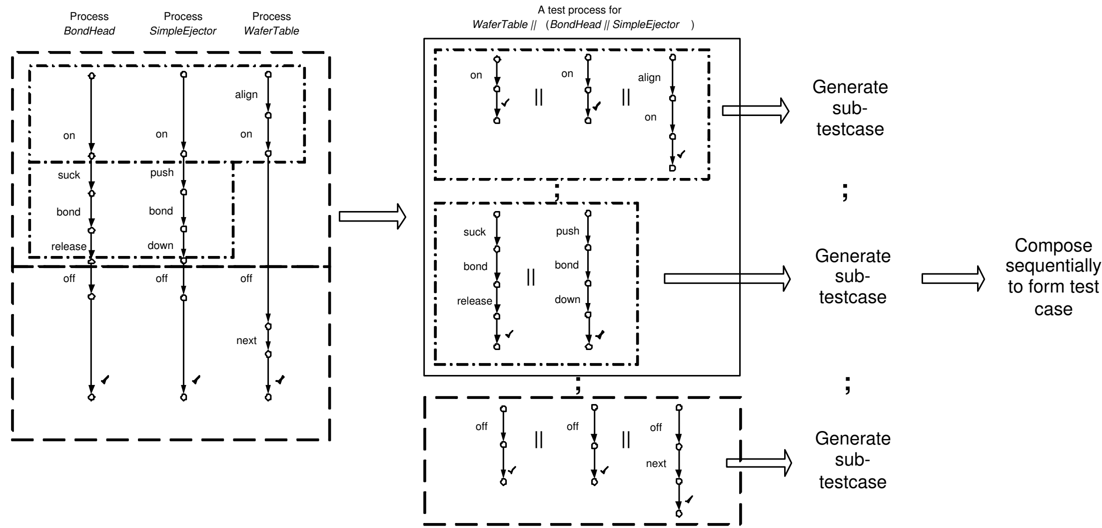


Figure 1.2: Our proposal

Figure 1.2 depicts our proposal. Each concurrent component is decomposed into sub-tasks. Corresponding sub-processes from concurrent components are composed in parallel. The $BondHead$ and the $SimpleEjector$ are decomposed into three sub-tasks and the $WaferTable$ is decomposed into two sub-tasks. The first sub-tasks of the these three components are composed in parallel to form a sub-task of the whole system. Similarly, we form the second and third sub-tasks as shown in the figure. Test cases are generated from each subtask of the process on the right.

| | |
|---|---|
| $\langle align, on \rangle$ | from the first sub-task |
| $\langle suck, push, bon, release, down \rangle$ | from the second sub-task |
| $\langle off, next \rangle$ | from the third sub-task |

Finally, they are composed together to construct a test case for the system. For instance,

$$\langle align, on, suck, push, bond, release, down, off, next \rangle$$

is a test case constructed from the compositional method.

The conventional approach always produces an explicit graph for the whole system before any fragment of a test case is generated. Obviously, when the number of concurrent components increases, the explicit graph in general will be formidably huge. The construction of this explicit graph will therefore be a burden in test case generation.

Our approach attempts to (i) analyse the relationships of processes in parallel and sequential compositions and (ii) decompose processes into sub-processes before any parallel composition takes place. In Figure 1.2, for instance, the process $WaferTable$ is divided into three sub-processes,

I. $on \rightarrow SKIP$ ,

II. $suck \rightarrow bond \rightarrow release \rightarrow SKIP$ , and

III. $off \rightarrow SKIP$ .

Each of these sub-processes can be used to form a parallel composition with sub-processes of processes *BondHead* and *SimpleEjector*. Intuitively, each sub-process is less complex than the corresponding enclosing process. For example, the process $suck \rightarrow bond \rightarrow release \rightarrow SKIP$ contains 5 nodes and 4 transitions only; whereas process *WaferTable* contains 6 nodes and 6 transitions. In this connection, intuitively, each explicit graph for the parallel composition of a sub-process cluster contains less nodes and less transitions than those for the whole system. For instance, the explicit graph for the sub-process cluster formed by the parallel composition of the first sub-process from each of the three concurrent processes in the system (that is, $(on \rightarrow SKIP) \parallel (on \rightarrow SKIP) \parallel (align \rightarrow on \rightarrow SKIP))$ consists of 4 nodes and 4 transitions. In the conventional approach (Figure 2.4), the explicit graph for the whole system consists of 13 nodes and 14 transitions. Intuitively, this smaller graph allows testers to generate test case from each sub-process cluster easier. If our proposal can maintain the relationship amongst sub-processes so that these fragments of test cases can be put together to form test cases for the whole system, software testers will not need to construct an explicit graph for the whole system to produce test cases.

Moreover, via abstraction, some sub-process clusters may be shown to behave equivalently, and hence the generation of test cases from one such unified and abstract cluster can be applied to all the more concrete forms of clusters. For instance, testers can decompose the process $(align \rightarrow on \rightarrow SKIP)$ into $(align \rightarrow SKIP);(on \rightarrow SKIP)$. The succeeding process in this sequential composition (that is, $on \rightarrow SKIP$) shares the same process definition as the first sub-process for both processes *WaferTable* and *BondHead*. In this case, the sequence of operations $\langle on, \surd \rangle$ generated from the common abstract form can be used as a trace-based test case for all these the sub-processes.

In summary, we consider it useful to explore the compositional support and the abstraction support for test case generation in CSP.

## 1.5 Organization of Thesis

The organization of this thesis is as follows:

**Chapter 1** is this chapter for general introduction. It points out that formal proving and software testing are two major means to assure software quality. It further presents the general problem in concurrent programs. On one hand, concurrent programs are extremely difficult to verify. On the other, testing of these programs need to take into account the problems related to testing sequential programs and new problems for concurrency and non-determinism. We propose our idea for specification-based testing to close this chapter.

**Chapter 2** is the review on related work in testing concurrent programs by conformance testing approaches. We classify them into four approaches. Each of them has their merits and limitations. The functional approach is the most general one. It only considers the initial and final observations. However, as internal structures of concurrent system are not modelled, it does not effectively address the testing issues related to inter-module integration. The parallel composition approach takes process as the unit. The way of testing is to compose a tester process against an implementation. Tester processes are

often derived from the specification directly and may be infinite. There are, however, methods to derive a finite tester process. The checking sequence approach is based on automata. The techniques can usually distinguish a program state from others. The state-explosion problem is not explicitly addressed. The conventional approach uses testing criteria to construct test cases. The emphasis is usually to address the state-explosion problem. We close this chapter by covering other techniques.

**Chapter 3** describes a formal language called Communicating Sequential Processes (CSP). Our proposal is developed on this platform. We introduce the syntax, the failures-divergences semantics, sequential composition and parallel composition. They form building blocks of this thesis.

**Chapter 4** introduces the notion of conformance testing. The process equivalence based on this notion is the same as testing equivalence. The latter is well known to be failures-divergences equivalence. The intransitive nature of this conformance notion, however, diminishes its use in an abstraction way to deal with the state-explosion problem. Subclasses of this notion are identified. One is reduction, which is also the refinement in CSP. Another is extension. We close this chapter by justifying our proposal in Chapter 1 about the use of extension hierarchy rather than reduction hierarchy as the building blocks to generate test cases.

**Chapter 5** further investigates the extension hierarchy from the compositional point of view. It presents five kinds of problems. Four of them will result in unreliable composition or decomposition, in either a sequential or concurrent way. The remaining one is a desirable property in decomposition.

**Chapter 6** contains the main contribution of this thesis. It proposes a $\sigma$-extension to preserve the desirable properties of process decomposition. It formulates the necessary and sufficient conditions to maintain the overall extensional property for processes to be compositional sequentially and concurrently. As a result, compositional test cases can be generated at different levels of abstraction in the anti-extension hierarchy.

**Chapter 7** discusses potential future work and compares our project with other related work.

**Chapter 8** is the conclusion. It summarizes the contributions of our research.

# Chapter 2

# Literature Review on Conformance Testing

In this chapter, we shall review the techniques used to facilitate the generation of conformance test cases for concurrent systems. Generally speaking, conformance testing is the assessment of an implementation to determine if it behaves as expected [64]. In particular, for off-the-shelf packages or black-box components, it establishes customer confidence in the products and services that they buy [84]. From the point of view of protocol conformance testing, ISO standards [59] states that a system "exhibits conformance if it complies with the conformance requirements of the applicable ... standard". Microsoft, however, states on their *MSDN Network* [72] that "the conformance tests for OLE DB are meant to supplement a provider writer's own testing, not to replace it. They form just a part of the complete testing suite."

In short, there is no universal definition of conformance testing. For the clarity of presentation, we organize different kinds of conformance testing into four basic complementing approaches in our review.

## 2.1 Functional Approach

The functional approach treats testing as a functional input and output relation. In other words, they relate preceding and/or succeeding observations about test cases. They do not impose any constraint in achieving them. In general, a preceding observation is not necessarily the initial activity of a program and a succeeding observation is also not necessarily the last activity of a program.

Abramsky [1] uses observational equivalence (of processes) as a testing criterion. Intuitively, their method checks whether an implementation is observed to behave compatibly to a specification after the succeeding observation of the test case is observed. Their method does not model the preceding observation constraints on the test cases, however. On the other hand, Tai *et al.* [22, 23, 24, 25, 61] develop more general relations, called sequencing constraints, between the preceding and the succeeding observations. Three kinds of constraints are identified, informally known as "always", "never" and "possibly". Intuitively, the occurrence of the preceding observation (that is, a synchronization event in their terminology) "must", "must not" and "may" immediately cause the occurrence of the succeeding observation, respectively. Constraints can be generated from the specifications.

Violations of constraints by test cases by the implementation would imply an implementation inconsistent to a specification. Intuitively, their method can also be considered as a kind of refusal testing [81].

Abstraction is also used in test case constructions. Stepney [97], Derrick and Boiten [44] and Aichernig [4] apply the concept of refinements in model-based languages. Informally, two components composed sequentially are readily replaced by their corresponding abstractions as the basis to generate test cases, so long as the post-condition of the preceding component with the pre-condition of the succeeding component can remain valid. In Chapter 5, like [44], we illustrate via a counter-example that the conformance relation in parallel composition approach (see Section 2.2) may not in general be preserved when the relationship between a test case and the specification is the refinement relation. It should also be noted that this approach does not model the internal structure of a component; whereas the other three approaches address structural properties. Consequently, if the interest is in the structural integration testing among components, the applicability of abstraction techniques in functional approach may be limited.

## 2.2 Parallel Composition Approach

Another approach is to treat testing as the parallel composition of a tester with a program to be tested. Informally, a tester is a model of a test suite. There are two schools of thought, one on equivalence and the other on an implementation relation between a specification and an implementation.

De Nicola and Hennessy [43] propose the notion of testing equivalence to distinguish processes by tests. Two processes are *testing equivalent* if they cannot be distinguished by the class of tests being considered. In their paper, they propose to use observational equivalence. There are a lot of testing equivalence relations proposed. A few examples are trace equivalence [98] and failures equivalence [17] as in CSP, observational equivalence and bisimulation relation as in CCS and barbed bisimulation [76, 82] in $\pi$-calculus.

From the point of view of software testing, one process can be considered as a specification, and another process can be considered as a program. Obviously, testing by equivalence will imply that the specification should be as detailed as the implementation. In most circumstances, it is very limited. A similar idea is used elsewhere, such as in the testing of object-oriented programs by Chen *et al.* [28, 29]. Their method generates two normally equivalent (and hence observationally equivalent) objects, and then applies identical sequences to check their observational equivalence. Since both objects and the equivalence checking are at the implementation level, it relaxes the aforementioned limitation. The violation of observational equivalence will mean errors.

Brinksma [13, 15] proposes a well-known implementation relation, denoted by $conf$, as the conformance relation between a specification and an implementation. He also finds out two special cases which are preorder relations, namely extension and reduction. Pitt and Freestone [83] provides a way to generate canonical tester for LOTOS processes. A significant difference between a canonical tester and a process constructed from our approach is that each canonical tester has to include every possible potential behaviour of the system; whereas our approach via anti-extension can construct processes that contains behaviour to be tested.

The inclusion of behaviour to be tested in the test process is not new. Abramsky [1]

shows that test scenarios can classify behaviour equivalences and pre-orders under different assumptions. Van Glabbeek [102] provides an extensive survey on this topic. Tretmans [99] enhances the notion of the tester process to consider test cases rather than a test suite. Informally, a tester is a process that restricts communication of a program to be tested. Since these two processes (a tester and a program) are intended to be executed concurrently, by such restriction, the composite process will reach a deadlock state if the program refuses to proceed with the synchronization offered by the tester, or vice versa. In other words, there is non-conforming behaviour of the program to be tested.

If the objective of a test is to check whether the program allows the tester to reach a state marked as successful, then it is commonly called *may testing* [53]. Similarly, if the objective of a test is to check whether the program allows the tester to reach all states marked as successful, then it is commonly called *must testing* [53]. As pointed out in [16], *must testing* is an alternative way to state the failure model of CSP [56]. May testing are commonly used in testing of non-interference properties [80] of security protocols [91, 47]. There are other conformance relations such as [67]. They consider a different interpretation on failures and divergences of processes. [16] also provides additional references in testing transition systems.

A major problem about this approach is to have to consider a whole process or a whole specification, which is difficult in most situations. Brinksma *et al.* [14] simplify a specification by a parallel composition of test process. They call it a loosening of the specification. Suppose that a specification has concurrent processes. In order to loosen the specification, individual components are required to compose with test process components concurrently. Hence, their simplification requires coordination between test process components.

## 2.3 Checking Sequence Approach

The third approach is to generate checking test cases from the specification to distinguish program states, and then to execute test cases by the program. There are four basic alternatives, namely transition tour, distinguishing sequences (DS), W-method and unique input-output sequence (UIO).

The transition tour method [77] finds test cases to cover all transitions at least once. From the point of view of software testing, it requires one test case to fulfil the all-state-transition-coverage criteria. The use of testing criteria will be discussed in the conventional approach in the next section.

The other three, on the other hand, consider program states. A program to be tested is first suspended at a location (that is, a particular state of a program). The DS approach [54, 65] executes a test case (that is, a sequence) capable of differentiating every state of the program. Conformance is checked by determining whether the series of output from a *DS* sequence is as expected as its specification. The W-method [36] executes a set of test cases so that the combined effect of the outputs from all test cases in a test suite can distinguish any program state. The UIO approach [92, 20, 85, 107, 40, 3, 94, 93], on the other hand, assumes known program suspension points, and executes a test case (that is, a sequence) which is capable of distinguishing this particular program state from all the others.

However, all these methods are applied to a program in which internal concurrency is seldom explicitly addressed. Moreover, owing to the state-explosion problem in composing concurrent components into a composite system or a graph, these methods are less attractive in

testing concurrent programs.

## 2.4 Conventional Approach

The most common paradigm is to generate test cases to fulfil testing criteria from a specification and to apply them to the program. Tripahty and Sarikaya [100] translate a LOTOS process into a chart which is a kind of extended finite state machine. They then use data-flow criteria to generate test case from these charts. A similar approach is used in [63, 62].

Hartmann, Imoberdort and Meisinger [52] models individual components as startcharts and stepwise composes them to form a global and flattened statechart heuristically. They then build a hierarchical graph by mapping the composite statecharts to the user-supplied category-partition scheme. Finally, reachability search is performed to generate test suites fulfilling the desired testing criteria. There are other techniques to produce a hierarchical structure of compositional processes such as reduction of individual component based on the contextual information of other components [34] or based on the preservation of some desirable property [19, 18, 60, 86]. Luo *et al.* [70] reduces the communicating finite state machine model into a single machine, and generate test cases from the latter.

On the other hand, besides explicit composition of concurrent components into a global structure, there are other methods that consider local constraints to generate test cases. Lee *et, al.* [68] generates test sequences starting from individual components incrementally. The method to construct a test sequence is, first, to construct an independent subsequence for each component incrementally. It selects one subsequence and extends that subsequence by a succeeding transition, which is randomly selected amongst transitions that can minimize the outstanding states and/or transitions incrementally. The test sequence constructed is then checked against each component to ensure it is a feasible path.

Li and Wong [69] use a similar construction method as [68]. They use the all-local-branch-coverage testing criteria. However, they require a global hierarchical statechart as a part of a specification. Starting from the statechart, their incremental heuristics is to select the succeeding transition that maximizes the aggregated number of outstanding states in each component dominated by this transition.

Chang *et al.* [27] use usage profiles as the heuristics to assign weights to transition edges of an extended finite state machine, and then uses all-state-transition-coverage criteria to generate a set of basic paths. Those basic paths are then composed sequentially in heuristic ways to obtain more complex forms of test cases. Simplification techniques are also used, such as process abstraction by means of contextual information [34], or by embedding sub-paths of lower tiers in the edges of higher tiers in a process hierarchy [66].

## 2.5 Other Techniques

There are other techniques that can be used to test concurrent programs. Fault-based techniques inject faults to the program to produce program mutants [42, 109]. Data-flow techniques check data-flow or synchronization anomalies [21, 89, 31, 49, 48, 32]. There are a lot of other detective techniques, such as data-race detection [35, 90, 57] and deadlock detection [37]. Some technique are based on the comparative analyses of the probabilities of detecting failures, such as partition testing and random testing [33, 79].

It should be noted that some of the techniques, such as data-flow testing, are concerned about the structure of some formal objects. Hence, they share a lot of common testing idea with the conformance testing approaches discussed above. Some of them are also relevant to integration testing [26] of other kinds of software.

## 2.6 Comparison with Our Work

Unlike the functional approach, our method takes structural details into account. We define $\sigma$-extension and the notion of converging process to split a task (known as a process) into sub-tasks (known as a sequential composition of processes). We use anti-extension rather than refinement as the building block of abstraction. Since structural details are considered, anti-refinement (that is, anti-reduction) may result in a process containing unnecessary traces that will hamper the generation of test cases. Example 6.4.8 in Chapter 6 illustrates this point.

Similar to the parallel composition approach, in Chapter 6, we also show that testing equivalence is preserved if two processes are mutually and sequentially extended. Our relation is also a preorder relation. Nevertheless, unlike the specification loosening proposal of Brinksma *et al.*, we use anti-extension hierarchy to manipulate specification process so as to manage the structural complexity.

Our method can work complementarily with the checking sequence approach and the conventional approach. It imposes no restriction on the construction of test cases to distinguish a program state from others. Moreover, our relation can combine with these approaches to produce test cases from anti-extended processes.

# Chapter 3

# Communicating Sequential Processes

As we will discuss in Section 4.1, the failures-divergences model of Communicating Sequential Processes (CSP) is strongly related to notions of conformance testing. Building our proposal on CSP will let us concentrate on the testing area, rather than deal with the irregularity in the underlying model[1]. In this chapter, we will introduce the CSP language accompanied with process graph illustrations. In particular, we will introduce the process composition via sequential and concurrent means.

## 3.1 Syntax and Semantics

According to [9], a process algebra is a formal description technique, especially for those systems with communicating and concurrently executing components. Amongst them, the most-known process algebras are ACP [8], CCS [74], CSP [56, 17] and their variants. Process algebras are also closely related to LOTOS [58] and $\pi$-calculus [76]. Characteristics of process algebra are (1) compositional modelling, building larger processes from smaller ones through a small number of constructs, (2) operational semantics and rich sets of algebraic laws and (3) behavioural reasoning via equivalence and preorder.

CSP describes concurrent systems in terms of communications amongst processes. The basic units in these communication patterns are activities, called events, which the system can perform. The set of all possible events for process $P$ is denoted as $\alpha(P)$ or $\alpha P$ for short. This set categorizes the static behaviour (i.e. interface) of the process concerned. A process will never participate into a communication with others if the event is not a part of its alphabet set.

**Example 3.1.1 (Alphabet set example)** The alphabet set of the process *CoffeeMachine*2 below contains two events, namely *coin* and *coffee*.

$$CoffeeMachine2 = coin \rightarrow coffee \rightarrow STOP$$

$$\alpha CoffeeMachine2 = \{coin, coffee\}$$

■

[1]However, it should be noted that there are a large variety of studies in translation and unification amongst different process algebras and Petri Net[11].

Events can be atomic or compound with *values*. An atomic event can be considered as a compound event in which the communication channel is the atomic event itself without value component. For instance, *coin* is an atomic event, and $coin.3$ is a compound event modelling the occurrence of message 3 dollars that sends along the channel *coin*. An event is said to occur if every process that carries this event as one of its alphabets participates in the communication.

The dynamic behaviour of a process is described by CSP expressions. The process *STOP* will not perform any event. The process *SKIP* does nothing except signalling successful termination. The process *RUN* repeatedly engages in any event in its alphabet set. The process *CHAOS* diverges immediately. Informally, it means error states. And in CSP, the process will become unable to control. It may proceed to any event, and at the same time, refuse to proceed to any event non-deterministically. All CSP expressions are then constructed based on a small number of operators and on these primitive processes.

The prefix operator is the most fundamental one which links up an event to a process to form another process. The prefixed process $a \rightarrow P$ (pronounced as '*a* then *P*') will perform event *a* initially, and then will behave like the process *P*.

**Example 3.1.2 ("Prefix" operator example)** The process *CoffeeMachine*2 will accept a *coin*, then provide a coffee and finally behave like the process *STOP*.

$$CoffeeMachine2 = coin \rightarrow coffee \rightarrow STOP$$

■

Processes can also be composed sequentially. The process $P;Q$ behaves like the process *P* initially and, if the process *P* terminates successfully, it passes control over to the process *Q*. The process *Coin*;*TeaChoiceCoffee* (will introduce later on page 20) shows an example of sequential composition.

The process $P \square Q$ (pronounced as '*P* choice *Q*') behaves like either the process *P* or the process *Q*. It offers all possible initial events of both processes *P* and *Q* to be chosen by the environment. However, whether it then behaving like *P* or like *Q* is totally within its own control, as long as it includes the chosen event as its first event.

**Example 3.1.3 (Choice composition example)** For instance, the process *TeaChoiceCoffee* offers users to select either a cup of *tea* or a cup of *coffee*. If the user selects a cup of *tea*, then the process *TeaChoiceCoffee* should behave like the process $(tea \rightarrow STOP)$. On the other hand, if the user selects a cup of *coffee*, then the process *TeaChoiceCoffee* should behave like the process $(coffee \rightarrow STOP)$.

$$TeaChoiceCoffee = (tea \rightarrow STOP) \square (coffee \rightarrow STOP)$$

■

There is another kind of choice in CSP. The process $P \sqcap Q$ (pronounced as '*P* internal choice *Q*') can behave either like the process *P* or the process *Q*, and the environment has no control over the selection.

**Example 3.1.4 (Internal choice composition example)** The process *VM3*, after accepting a *coin*, will make a cup of *tea* or a cup of *coffee* non-deterministically. A customer may want

*coffee*, but the process *VM3* may decide to serve a cup of *tea*, and vice versa. The only way that a customer gets a right kind of drink is to prepare to receive either kind of drinks.

$$VM3 = coin \rightarrow ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP))$$

■

**Example 3.1.5 (Special parallel compositions example)** A special case is that any process except the process *CHAOS* will form a deadlock with a deadlock if they have the same alphabets. The process *CHAOS*, usually means program error, will result in a chaotic process.

$$\begin{array}{l} VM3 \parallel STOP = STOP \\ VM3 \parallel CHAOS = CHAOS \end{array}$$

■

The process $P \backslash A$ behaves like the process $P$ with all events in the set $A$ not observable.

**Example 3.1.6 (Concealment example)** The process *FreeVM* will make a cup of *tea* or a cup of *coffee* free of charge.

$$\begin{array}{ll} FreeVM & = VM3 \backslash coin \\ & = ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP) \end{array}$$

■

The process $P \parallel Q$ (pronounced as '$P$ parallel $Q$') is a composite process in which the process $P$ and the process $Q$ should synchronize whatever activities common to them.

**Example 3.1.7 (Parallel composition example)** As illustrated in Example 3.1.4, a customer should prepare either kinds of drink in order to work perfectly with the vending machine *VM3*. When the process *Customer* and the process *VM3* are composed concurrently, the process *Customer* will synchronize the event *coin* with the process *VM3*, after the customer *read instructions*. And then, the process *VM3* will behave like the process *FreeVM* and the process *Customer* will behave like the process *TeaChoiceCoffee*.

$$\begin{array}{l} Customer = read\ instructions \rightarrow coin \rightarrow ((tea \rightarrow STOP) \Box (coffee \rightarrow STOP)) \\ Customer \parallel VM3 = read\ instructions \rightarrow coin \rightarrow (TeaChoiceCoffee \parallel FreeVM) \end{array}$$

■

The process $P ||| Q$ is a composite process in which the process $P$ and the process $Q$ do not interact at all.

**Example 3.1.8 (Interleaving example)** For instance, putting the free vending machine *FreeVM* aside the vending machine *VM3*, we shall have two drinks by inserting one coin; or may even have a drink free of charges.

$$\begin{array}{rcl} FreeVM ||| VM3 & = & ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP) \\ & & ||| (coin \rightarrow ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP))) \\ & = & tea \rightarrow (coin \rightarrow ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP))) \sqcap \cdots \end{array}$$

■

The process $P/tr$ behaves like the process $P$ after it has performed the sequence of activities as defined in the trace $tr$.

**Example 3.1.9 ("After" operator example)** For instance, the process *VM3* after the trace $\langle coin \rangle$ will behave like the process *FreeVM*.

$$VM3/\langle coin \rangle = ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP) = FreeVM$$

■

Processes may also be recursively defined by equations, $X = \mu X.F(X)$.

**Example 3.1.10 (Recursion example)** The process $Hello$ says "hello" repeatedly.

$$Hello = hello \rightarrow Hello$$

■

A sequence of events that a process $P$ may perform is called a trace. Putting all feasible traces of the process $P$ in a set will form the trace set of the process $P$, $traces(P)$.

**Example 3.1.11 (Trace example)** For instance, $\langle coin, coffee \rangle$ and $\langle coin, tea \rangle$ are two traces of the process *VM3* above.

$$traces(VM3) = \{\langle\rangle, \langle coin \rangle, \langle coin, coffee \rangle, \langle coin, tea \rangle\}$$

■

The divergences of a process $P$, *divergences*($P$), is a subset of the trace set of the process and are those traces whose occurrence the process $P$ might behave like the process *CHAOS*.

**Example 3.1.12 (Divergences example)** The process *Clock* (= *click* → *CHAOS*) will diverge after the first *click* occurs.

$$\langle click \rangle \in divergences(Clock)$$

■

The failures of a process $P$, $failures(P)$, is the set of tuple $(tr, X)$ that the process $P$ can exhibit. $X$ is a set of events ($\subseteq \alpha P$), called refusal set, that the process $P/tr$ may initially refuse to participate in.

**Example 3.1.13 (Failures example)** After a *coin* is inserted, the vending machine *VM3* may

unwillingly serve a cup of *tea*.

$$Failures(VM3) = \left\{ \begin{array}{l} (\langle\rangle, \emptyset), \\ (\langle\rangle, \{tea\}), \\ (\langle\rangle, \{coffee\}), \\ (\langle\rangle, \{coffee, tea\}), \\ (\langle coin\rangle, \emptyset), \\ (\langle coin\rangle, \{coffee\}), \\ (\langle \mathbf{coin}\rangle, \{\mathbf{tea}\}), \\ (\langle coin\rangle, \{coffee, coin\}), \\ (\langle coin\rangle, \{tea, coin\}), \\ (\langle coin, coffee\rangle, \{coffee, coin, tea\}), \\ (\langle coin, coffee\rangle, \{coin, tea\}), \\ (\langle coin, coffee\rangle, \{coin, coffee\}), \\ (\langle coin, coffee\rangle, \{coffee, tea\}), \\ (\langle coin, coffee\rangle, \{coin\}), \\ (\langle coin, coffee\rangle, \{tea\}), \\ (\langle coin, coffee\rangle, \{coffee\}), \\ (\langle coin\rangle, \emptyset), \end{array} \right\}$$

$$refusals(VM3/\langle coin\rangle) = \{\emptyset, \{\mathbf{tea}\}, \{coffee\}, \{coffee, coin\}\{tea, coin\}\}$$

However, if the customer prepares to accept either kind of drinks, then the vending machine *VM3* will meet this customer's expectation. Consequently, tuple

$$(\langle read\ instructions, coin\rangle, \{coffee, tea\})$$

is not a failure of the process *VM3*. ■

The failures-divergences semantics of a process is a triple $FD(P) = (\alpha P,\ failures(P),\ divergences(P))$. In this semantics model, two processes are equivalent if and only if they have identical failures and identical divergences. A process $P$ is said to *refine* a process $Q$, denoted by $Q \sqsubseteq_{refine} P$ or $Q \sqsubseteq P$ whenever unambiguous, if and only if process $P$ is less likely to diverge and less likely to fail. Precisely, processes $P$ and $Q$ should share the same alphabet set, and both the failures and the divergences of process $Q$ are corresponding subsets of those of process $P$. A process is said to be deterministic if it never refuses to engage in communications that it may perform.

There are 7 conditions identified by Hoare [56] for which a failures-divergences ($FD = (A, F, D)$) define a process uniquely. In other words, in this model, two processes are said to be (failures-divergences) equivalent if and only if they have the same failures set and the same divergences set. The set of conditions is as follows:

A process is a triple $(A, F, D)$
where $A$ is any set of symbols
$F$ is a relation between $A^*$ and $2^A$
$D$ is a subset of $A^*$
providing that they satisfy the following conditions:

I. $(\langle\rangle, \emptyset) \in F$

II. $(s\hat{\ }t, X) \in F \implies (s, \emptyset) \in F$

III. $(s, Y) \in F \wedge X \subseteq Y \implies (s, X) \in F$

IV. $(s, X) \in F \wedge a \in A \implies (s, X \cup \{a\}) \in F \vee (s\hat{\ }\langle a\rangle, \emptyset) \in F$

V. $D \subseteq domain(F)$

VI. $s \in D \wedge t \in A^* \implies s\hat{\ }t \in D$

VII. $s \in D \wedge X \subseteq A \implies (s, X) \in F$

Intuitively, all traces should be prefix-closed, all divergences should be suffix-closed and be parts of traces, and all failures should be sub-refusal-set-closed. Full descriptions of CSP can be found in [56, 87]. We also follow two more definitions on trace set to ease the subsequent discussion.

**Definition 3.1.1 (Complete traces)** *[102] A complete trace of a process P is a trace which will lead P to a deadlock state immediately after the trace. The set of complete traces for a process P is denoted by* $CT(P)$.

**Example 3.1.14 (Complete trace example)** For instance, the complete trace set for the vending machines *CoffeeMachine*1 and *CoffeeMachine*2 are as follows.

$$CT\,(CoffeeMachine1) = \{\langle coin, coffee, \surd\rangle\}$$

$$CT\,(CoffeeMachine2) = \{\langle coin, coffee\rangle\}$$

■

Obviously, a process may never stop (such as the process $Hello$ in Example 3.1.10) or diverge immediately (such as the process $Clock$ in Example 3.1.12 after the first $click$). In these cases, the complete trace sets for those processes would be empty. Moreover, as sequential composition (Section 3.3) has special interest in the successful termination of processes, we restrict the above definition to formulate the successful complete trace set.

**Definition 3.1.2 (Successful complete trace)** *A successful complete trace of a process is a complete trace of that process with the last event on the trace is the successful termination symbol* ($\surd$)*. The set of successful complete traces for a process P is denoted by* $CT^{\surd}(P)$.

**Example 3.1.15 (Successful complete trace example)** For the process *CoffeeMachine*1, the set $CT\,(CoffeeMachine1)$ contains one complete trace. Moreover, as the trace is terminated with the $\surd$ symbol, and so, the set of successful complete traces is the same as the set of complete traces.

$$CT^{\surd}\,(CoffeeMachine1) = \{\langle coin, coffee, \surd\rangle\}$$

■

## 3.2 Graphical Illustration

Sometimes, a process is represented as a *process graph*, a *synchronization tree* or a *state transition diagram*. We follow the common conventions used in CSP, CCS and ACP, and use the definition of *process graph* in [102].

**Definition 3.2.1 (Process Graph)** *[102] A process graph over an alphabet A is a rooted, directed graph whose edges have labels from A. Formally, a process graph P is a triple* $(N,R,E)$*, where*

I. *N is a set, of which the elements are called the nodes or states of P,*

II. $R \in N$ *is a special node: the root or initial state of P and*

III. $E \subseteq N \times A \times N$ *is a set of triples* $(s,a,t)$ *where* $s,t \in N$ *and* $a \in A$*: the edges or transitions of P.*

A node without an outgoing edge represents termination (or deadlock). If $e = (s,a,t) \in E$, one says that $e$ goes from the state $s$ to the state $t$. A path $tr$ in a process graph is an alternating sequence of nodes and edges, starting and ending with a node, such that each edge goes from the node before it to the node after it. If the last node of a path is a deadlock state or a termination, then the path is called a complete path. Moreover, if $\surd$ is the event label on the last edge, then the path is called a successful complete path. If only events but not states of a process graph are observable, then a path $tr$ is a trace in CSP sense. We also use the convention of unnamed edges to stand for internal transitions. Hence, whenever we show an unnamed edge, the label on this edge should be some internal event so that it coincides with CSP that the alphabet set of any process contains no internal event.

It should be noted that we use process graphs as an auxiliary means to illustrate our examples on CSP processes in this thesis. Formally, we always refer to CSP definitions. Below are a few illustrations of process graph.

**Example 3.2.1 (Illustrations of process graph)** Figure 3.1(a) shows three processes. The process *CoffeeMachine*1 will accept a *coin*, and make a cup of *coffee*, and then *terminate successfully* (event $\surd$). The process *CoffeeMachine*2 behaves quite similar to the process *CoffeeMachine*1, however, it cannot terminate successfully. The process *VendingMachine* describes a vending machine which will accept a *coin* and then it *may* behave like either a process which will make a cup of *tea* with *cream*, or a process which will allow users to select either a jar of *coke* with *ice* or a cup of *coffee* with *cream*. The process graph for the process *VendingMachine* also illustrates that if a node has more than one outgoing edges labelled with the same label (*coin*), the selection will become non-deterministic between these two edges.

$$CoffeeMachine1 = coin \rightarrow coffee \rightarrow SKIP$$

$$CoffeeMachine2 = coin \rightarrow coffee \rightarrow STOP$$

$$\begin{aligned} VendingMachine = {} & (coin \rightarrow tea \rightarrow add\ cream \rightarrow STOP) \\ & \sqcap (coin \rightarrow ((coke \rightarrow add\ ice \rightarrow STOP) \Box (coffee \rightarrow add\ cream \rightarrow STOP))) \end{aligned}$$

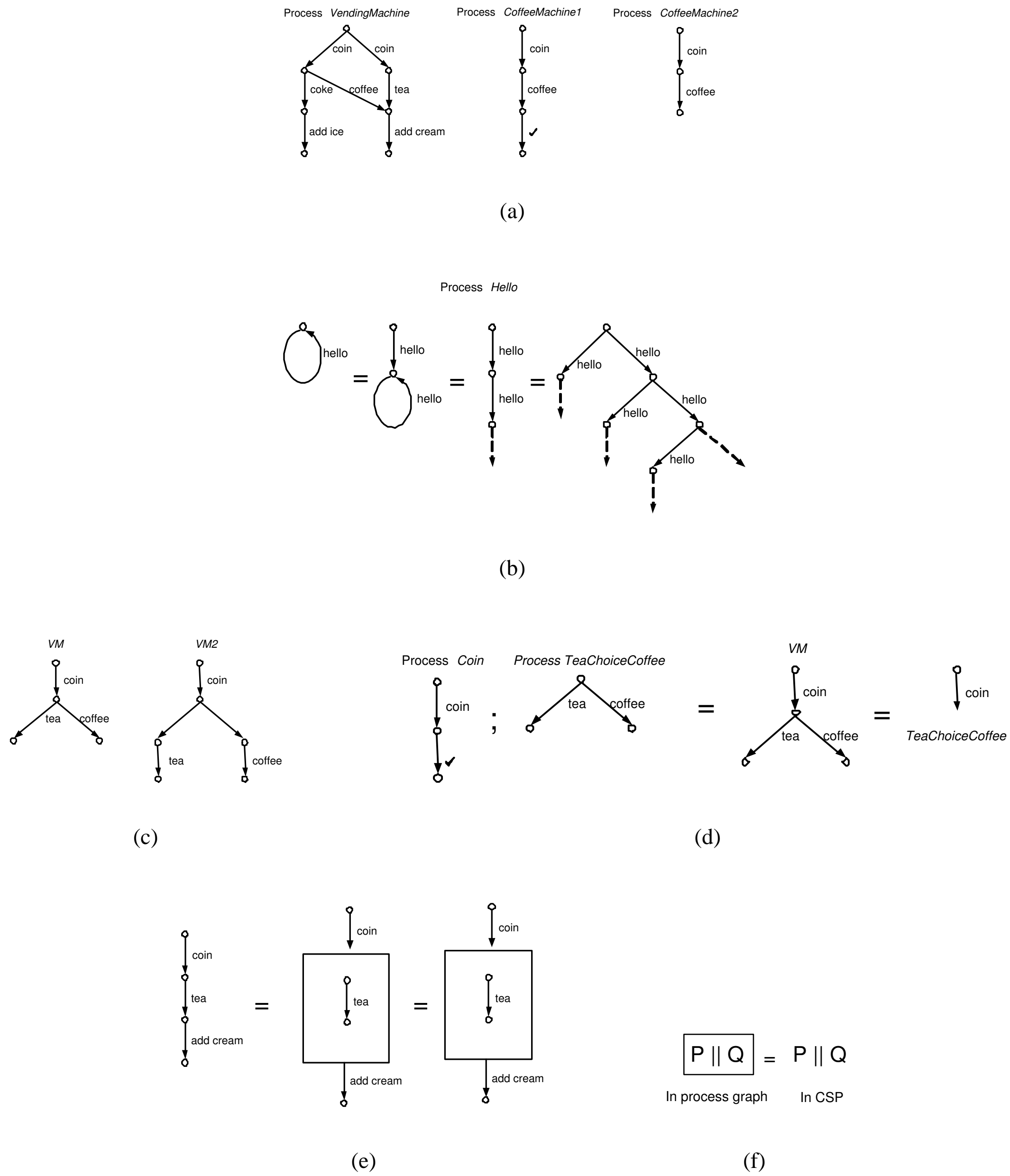


Figure 3.1: CSP and graphical convention

Figure 3.1(b) shows recursion. The process $Hello$ will say $hello$ repeatedly.

$$Hello = hello \rightarrow Hello \qquad = hello \rightarrow \mu X.(hello \rightarrow X)$$

Figure 3.1(c) shows the graphical illustration of choice and internal choice respectively. The process *VM* is a vending machine which allows users to select either $tea$ or *coffee*; the process $VM2$ forbids user from selection on the contrary.

$$VM = coin \rightarrow ((tea \rightarrow STOP) \Box (coffee \rightarrow STOP))$$

$$VM2 = coin \rightarrow ((tea \rightarrow STOP) \sqcap (coffee \rightarrow STOP))$$

Figure 3.1(d) shows the effect of sequential composition of processes in CSP. The process *Coin* will accept a *coin* and then it will terminate successfully. However, in the sequential composition *Coin*;*TeaChoiceCoffee*, the process *Coin*, however, will then transfer the control to the process *TeaChoiceCoffee* which allows users to select amongst *tea* and *coffee*. Hence, the process resulted from this sequential composition will be the same as process *VM*. And in CSP, it is also same as the process $coin \rightarrow TeaChoiceCoffee$.

$$Coin = coin \rightarrow SKIP$$

$$TeaChoiceCoffee = (tea \rightarrow STOP) \Box (coffee \rightarrow STOP)$$

$$\begin{aligned} Coin;TeaChoiceCoffee &= coin \rightarrow (tea \rightarrow STOP) \Box (coffee \rightarrow STOP) \\ &= coin \rightarrow TeaChoiceCoffee \end{aligned}$$

Figure 3.1(e) shows the convention that we, sometimes, box up a process for clarity using a solid or dotted surrounding box. Figure 3.1(f) shows that the parallel composition operator ($\|$) is used in process graph as if it were in CSP. ■

More examples can be found in [56, 87] on CSP, in [75] on CCS and in [6] on ACP.

## 3.3 Sequential Composition in CSP

### The Semantics of Sequential Composition

The sequential composition (a process $P;Q$) has something interesting. Semantically, it hides the successful termination symbol $\surd$ in a trace of its preceding process (i.e. the process $P$) and appends the traces of its succeeding process (i.e. the process $Q$). It also treats all traces of the preceding process as the traces of the composite process. Consequently, the divergences of the sequential composition ($P;Q$) should be the divergences of the process $P$, plus those from the process $Q$ prefixed with any $\surd$-pruned successful complete trace of the process $P$. The failures of the sequential composition can be worked out in a similar fashion. Formally, the trace set, the failures set and the divergences set of the sequential composition [56] are defined as follows:

$$\begin{aligned} traces(P;Q) = \; & \{s | s \in traces(P) \wedge \neg \langle \surd \rangle in\ s\} \\ & \cup \{s\hat{\ }t | s\hat{\ }\langle \surd \rangle \in traces(P) \wedge t \in traces(Q)\} \end{aligned}$$

$$\begin{aligned} divergences(P;Q) = \; & \{s | s \in traces(P) \wedge \neg \langle \surd \rangle in\ s\} \\ & \cup \{s\hat{\ }t | s\hat{\ }\langle \surd \rangle \in traces(P) \\ & \quad \wedge \neg \langle \surd \rangle in\ s \\ & \quad \wedge t \in divergences(Q)\} \end{aligned}$$

$$\begin{aligned} failures(P;Q) = \; & \{(s,X) | (s, X \cup \{\surd\}) \in failures(P)\} \\ & \cup \{(s\hat{\ }t, X) | s\hat{\ }\langle \surd \rangle \in traces(P) \wedge (t,X) \in failures(Q)\} \\ & \cup \{(s,X) \subseteq \alpha(P;Q) | s \in divergences(P;Q)\} \end{aligned}$$

| $traces\,(Inspection)$ | $traces\,(Ranking)$ | $traces\,(Inspection;Ranking)$ |
|---|---|---|
| $\langle\rangle$ | | $\langle\rangle$ |
| $\langle capture\rangle$ | | $\langle capture\rangle$ |
| $\langle capture, inspect\rangle$ | | $\langle capture, inspect\rangle$ |
| | $\langle\rangle$ | $\langle capture, inspect\rangle$ |
| | $\langle align\rangle$ | $\langle capture, inspect, align\rangle$ |
| $\langle capture, inspect, \surd\rangle$ | $\langle align, rank\rangle$ | $\langle capture, inspect, align, rank\rangle$ |
| | $\langle align, rank, \surd\rangle$ | $\langle capture, inspect, align, rank, \surd\rangle$ |
| | $\langle align, discard\rangle$ | $\langle capture, inspect, align, discard\rangle$ |
| | $\langle align, discard, \surd\rangle$ | $\langle capture, inspect, align, discard, \surd\rangle$ |

Table 3.1: Trace sets of processes *Inspection*, *Ranking* and *Inspection*;*Ranking*

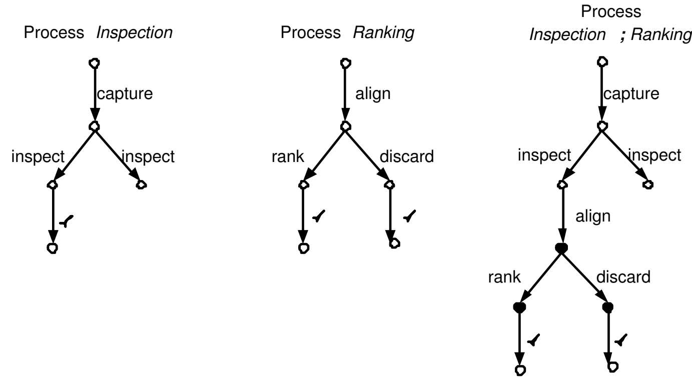


Figure 3.2: Sequential composition of CSP processes.

**Example 3.3.1 (Sequential composition example)**

$$Inspection = capture \rightarrow ((inspect \rightarrow STOP) \,\square\, (inspect \rightarrow SKIP))$$

$$Ranking = align \rightarrow ((rank \rightarrow SKIP) \,\square\, (discard \rightarrow SKIP))$$

$$Inspection;Ranking = capture \rightarrow ((inspect \rightarrow STOP)$$
$$\square\, (inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \,\square\, (discard \rightarrow SKIP))))$$

Figure 3.2 depicts the effect of sequential composition. The process *Inspection captures* a photo of a die, and then *inspects* the bonding result. However, it is faulty. It may or may not terminate successfully. The process *Ranking* will first align the ranking table so that a die may drop into a desired collection bin, and then it either actually *discards* a die (perhaps the die is defeat) or starts the ranking process (*rank*). It then terminates successfully. The sequential composition may mean that it becomes the process (*Inspection*;*Ranking*). It captures a photo of a die, recognizes any defeat, moves to the required collection bin based on the inspection result and then, discards the defeat or ranks the acceptable. The composite process will terminate successfully. Table 3.1 shows the trace sets of processes *Inspection*, *Ranking* and *Inspection*;*Ranking* respectively. One can easily observe that all traces in the third column come from a composition of traces from the first and the second columns. ■

### 3.3.1 Special Algebraic Laws for Sequential Composition

In CSP, there are a few special laws for sequential composition, namely, for processes $A$ and $B$,

I. $STOP;A = STOP$,

II. $CHAOS;A = CHAOS$ and

III. $B;A = B$ if $B$ never stops.

Intuitively, they mean that sequential composition is ineffective to them.

#### Choice Composition with *SKIP*

The successful termination process *SKIP* has special treatment in CSP. In the original CSP due to Hoare (page 178 of [56]), it states "whenever a process can terminate, it can do so without offering any alternative event to its environment.". It further restricts the process *SKIP* to never appear unguarded in an operand of $\Box$. [2]

## 3.4 Parallel Composition in CSP

#### The Semantics of Parallel Composition

The semantics of the parallel composition follows a standard treatment. Informally, a trace of a parallel composition (a kind of processes) is the possible interleaving of traces of its concurrent component subjected to the synchronization constraints. In other words, it may proceed with a component's action as long as other components allow. Otherwise, it will form a deadlock. The failures of the parallel composition are just the unions of the refusals from its components

[2] Roscoe [87] imposes additional restrictions on the original failures-divergences model and allow the process *SKIP* to be an operand of $\Box$. He introduces an algebraic law, referred as "$\Box - SKIP$ resolve". Given a process $P$,

$$P \Box SKIP = (P \sqcap STOP) \Box SKIP$$

This law states that the process $P \Box SKIP$ will behave like the process *SKIP* but can offer the initial choices of the process $P$ non-deterministically. Essentially, it opts to terminate successful whatever the environment (because the environment is assumed to have no control over the successful termination of a process). It may refuse any other alphabet initially.

$$(\langle\rangle, \alpha(P \Box SKIP) \backslash \{\surd\}) \in failures(P \Box SKIP) \tag{3.1}$$

However, we do not cover his CSP variant in this research.

that may form its trace part. Formally [56],

$$
\begin{aligned}
traces(P \parallel Q) = \{t| \\
&(t \upharpoonright \alpha P) \in traces(P) \\
&\wedge (t \upharpoonright \alpha Q) \in traces(Q) \\
&\wedge t \in (\alpha P \cup \alpha Q)^*\}
\end{aligned}
$$

$$
\begin{aligned}
failures(P \parallel Q) = \{s, (X \cup Y) | s \in (\alpha P \cup \alpha Q)^* \\
&\wedge (s \upharpoonright \alpha P, X) \in failures(P) \\
&\wedge (s \upharpoonright \alpha Q, Y) \in failures(Q)\} \\
&\cup \{s, X | s \in divergences(P \parallel Q)\}
\end{aligned}
$$

$$
\begin{aligned}
divergences(P \parallel Q) = \{u | \exists s, t. u\ interleaves(s, t) \\
&\wedge ((s \in divergences(P) \wedge t \in traces(Q)) \\
&\quad \vee (s \in traces(P) \wedge t \in divergences(Q)))\}
\end{aligned}
$$

where $A^*$ means the transitive closure of sequences over the alphabet set $A$.

### 3.4.1 Special Algebraic Laws for Parallel Composition

In this thesis, we use the after operator, and hence, we quote the related algebraic laws. The after operator is distributive through parallel composition. Informally, if a sequence of actions (i.e. a trace and, say $s$,) is feasible for processes $P$ and $Q$ which are composed concurrently, then the behaviour of parallel composition after executing that sequence of actions should be the same as the process $P$ after executing its relevant part of actions and the process $Q$ after executing its relevant part of actions and composed concurrently.

$$(P \parallel Q)/s = (P/(s \upharpoonright \alpha P)) \parallel (Q/(s \upharpoonright \alpha Q))$$

## 3.5 Chapter Summary

We introduce the language of Communicating Sequential Processes (CSP). We also describe sequential composition and parallel composition. There is a special treatment of successful termination in CSP. However, special treatment of successful termination is not unique to CSP. For instance, Aceto and Hennessy [2] provide a detailed treatment of termination in process algebra in general. In the following chapters, we will develop our work on CSP.

# Chapter 4

# Conformance Testing

## 4.1 The Notion of Conformance Testing

We are both interested in the area of theory of conformance testing (Section 2.2) and the applicability to the conventional approach (Section 2.4). We will define formally the notion of conformance testing to be used in this research and evaluate our motivation to focus on extension. We adopt the definition of conformance testing problem from Bernhard [10] and re-cast it into CSP. The *conformance testing problem* can be stated as follows.

**Definition 4.1.1 (Conformance Testing Problem)** *Let S and P be CSP processes. Suppose that a specification S of an implementation program P. The question we wish to ask is whether or not program P is a correct implementation of specification S. The conformance testing approach answers this question by prescribing a way to generate a set of tests from specification S, which is then used to test implementation P.*

### 4.1.1 Conformance, Reduction and Extension

We follow the implementation relation proposed by Brinksma (see Section 2.2). Brinksma shows that two processes are said to be *testing equivalent* if they mutually conform to each other under their formalism. Intuitively, *testing equivalence* means that applying any testing to a pair of processes separately will always observe the same result (failed or passed).

Unfortunately, conformance relation is known to be intransitive[1]. In other words, suppose that we are given three program versions implementing the same functionality. We would like to know whether they are consistent amongst themselves. So, we test whether the first program version conforms to the second one, whether the second program version conforms to the third one and whether the third program version conforms to first one. However, we cannot conclude that the first program version *will* then conform to the third one. In fact, we need to conduct three more testing of conformance in another way round. In other words, we have to test all possible combinations.

Brinskma identifies two special cases that are preorder relations. A preorder relation is attractive; and one of the reasons is that it allows indirect relations to be established amongst processes that minimize the overhead of repeated validations in different combinations of

[1] Interested readers please refer to Proposition A.1.3.

processes. From the software testing point of view, it suggests a way to perform *incremental testing* [99]. The first preorder relation proposed is *extension* relation. Another preorder relation proposed is *reduction* relation.

Reduction is the same as refinement in failures-divergences model of CSP [83]. Moreover, as pointed out by van Gblabbeek [102], testing equivalence is the same as failures-divergences equivalence in CSP. These strong co-relations between the standard semantic model of CSP and the notions of conformance testing facilitate CSP to be an excellent vehicle to develop further theoretical results on testing. We introduce these three relations below and then argue about extension for conventional approach (see Section 2.4).

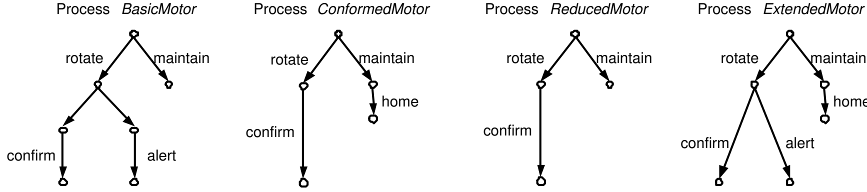


Figure 4.1: Examples of conformance, reduction and extension relations

**Definition 4.1.2 (Conformance)** *[13, 83, 39] Let P and Q be processes. Assume P and Q have the same alphabet set. P* conforms *to Q if and only if P deadlocks less often than Q whenever placing in an environment whose traces are limited to those common to both P and Q. Formally, P conforms to Q iff*

*I.* $\alpha Q = \alpha P$,

*II.* *for all* $s \in traces(P) \cap traces(Q)$, $(s,X) \in failures(P) \implies (s,X) \in failures(Q)$ *and*

*III.* $divergences(P) \cap traces(Q) \subseteq divergences(Q)$.

Informally, conformance checks whether the failures and divergences of a process respect those of another process.

**Example 4.1.1 (Conformance example)**

$$BasicMotor = (maintain \rightarrow STOP)$$
$$\Box(rotate \rightarrow ((confirm \rightarrow STOP) \sqcap (alert \rightarrow STOP)))$$

$$ConformedMotor = (maintain \rightarrow home \rightarrow STOP)\Box(rotate \rightarrow confirm \rightarrow STOP)$$

In Figure 4.1, besides a maintenance service, the process *BasicMotor* may unwillingly *confirm* users after it has finished its rotation. Rather, it may *alert* them for some reason. The process *ConformedMotor* enhances the process *BasicMotor*. It always *confirms* users after a rotation and returns to its home position if a maintenance service is required. Observe that it conforms to the process *BasicMotor*. It improves the failures of the process *BasicMotor* after the rotation by securing an confirmation to users. It also introduces new traces. It allows a new action *home* after the maintenance service call. The *alert* action is irrelevant to their conformity, as the sequence $\langle rotate, alert \rangle$ is not a common trace of these two processes. ■

**Definition 4.1.3 (Reduction)** *[13, 83, 39] Let P and Q be processes. Assume P and Q have the same alphabet set. The behaviour that P is permitted to exhibit can also be exhibited by Q. Moreover, P can deadlock only when Q can deadlock. Sometimes, we use "Q anti-reduces P" and "P reduces Q" interchangeably. Formally, P reduces Q iff*

*I.* $\alpha P = \alpha Q$,

*II.* $failures(P) \subseteq failures(Q)$ *and*

*III.* $divergences(P) \subseteq divergences(Q)$.

Reduction (relation) is a special kind of conformance relation by restricting all traces of a *reduced* process ($Q$) to be those of its counterpart ($P$).

**Example 4.1.2 (Reduction example)**

$$ReducedMotor = (maintain \rightarrow STOP)\Box(rotate \rightarrow confirm \rightarrow STOP)$$

The process *ReducedMotor* (in Figure 4.1) is similar to the process *ConformedMotor* in Example 4.1.1. It improves the failures of the process *BasicMotor* after rotation. However, it is limited by the reduction relation, and so it provides no homing action after the maintenance service call. ■

**Definition 4.1.4 (Extension)** *[13, 83, 39] Let P and Q be processes. Assume P and Q have the same alphabet set. P extends Q, denoted by* $Q \sqsubseteq_{ext} P$*, if and only if the P can exhibit all permitted behaviour of Q and if Q can refuse some set of events whenever P can. Sometimes, we use "Q anti-extends P" and "P extends Q" interchangeably*[2]*. Formally,* $Q \sqsubseteq_{ext} P$*, iff*

*I.* $traces(Q) \subseteq traces(P)$,

*II.* $s \in traces(Q) \wedge (s,X) \in failures(P) \implies (s,X) \in failures(Q)$ *and*

*III.* $divergences(P) \cap traces(Q) \implies divergences(Q)$.

Intuitively, an implementation $P$ *adds* information that is consistent to its specification $Q$. Extension is also a special kind of conformance relation by restricting, on the contrary, the traces of extended process ($P$) to include all traces of its counterpart ($Q$).

**Example 4.1.3 (Extension example)**

$$\begin{aligned} ExtendedMotor = &(maintain \rightarrow home \rightarrow STOP) \\ &\Box(rotate \rightarrow ((confirm \rightarrow STOP)\Box(alert \rightarrow STOP))) \end{aligned}$$

Performing everything that the process *ConformedMotor* (in Example 4.1.1) can perform, the process *ExtendedMotor* (in Figure 4.1) can alert users if they wish to. Observe that it includes all sequences of actions of the process *BasicMotor* and, hence, extends the latter. ■

**Definition 4.1.5 (Testing equivalence)** *[13] Let P and Q be processes. Assume P and Q have the same alphabet set. P is testing equivalent to Q if and only if* $P = Q$.

[2] Informally, $Q$ is a kind of abstracted form of $P$.

**Proposition 4.1.1 (Testing Equivalence = Mutual Conformance, Reduction or Extension)** *[13] Let P and Q be processes. Assume P and Q have the same alphabet set. P and Q are testing equivalent if and only if they mutually*

I. *conform to,*

II. *extend or*

III. *reduce*

*each other.*

The proof directly follows the definitions of these conformance and the process equivalence in the failures-divergences model. In the failures-divergences model, two processes are said to be equivalence if and only if they have the same alphabet set, the same failures and the same divergences.

## 4.2 Justification of Our Proposal

As discussed in previous sections, one of the most difficult obstacles in testing concurrent system is the state-explosion problem. In particular, the most often used techniques are abstraction and compositional approaches. Suppose that we are interested in testing the conformity of an implementation against a specification. We choose extension in CSP as the basis for the following reasons.

I. Extension is a kind of conformance relation and is a preorder. A framework supporting a multi-level abstraction (i.e. an anti-extension hierarchy) can be built on extension from specification. The transitive property of extension ensures that each abstraction tier will be extended by the specification. It further means that they should be conformed by an implementation of the specification[3]. In other words, conformance test cases can be generated at different levels of abstraction.

   Conformance, in general, is intransitive[4] however. So, this promising approach cannot be applied to the situation when one builds abstractions from specification based on conformance.

II. Reduction hierarchy is not as efficient as extension hierarchy. Reduction suffers from at least three problems. First, although in a reduction hierarchy, reduction is preserved, yet it does not guarantee that the anti-reduction from specification will be conformed by an implementation of the specification[5]. Secondly, an anti-reduced process *allows* the introduction of extra operation sequences. Those sequences may be completely irrelevant to both the specification and the implementation. Test cases generated from this kind of abstraction will include illegitimate test cases. It also enlarges the size of a test suite unnecessarily. Moreover, the effectiveness of the test suites may be affected if test suite minimization techniques [106, 105, 30] are used because some legitimate

[3] Interested reader may refer to Proposition A.1.1

[4] Interested reader may refer to Proposition A.1.3

[5] Interested reader may refer to Proposition A.1.2

test case will be removed during test suite minimization. Thirdly, an abstraction through reduction should include all traces and all failures of the specification. Specification, in practice, is already very complex. Introducing extra complexity in the abstraction is difficult to make the testing task to be more effective.

III. Specification in CSP offers the potential of test automation. Testing is costly. Generation of test cases and executions based on informal specification are inefficient and unreliable. Formal specification provides an unambiguous source of reference to generate test cases. We will further illustrate in Chapter 5 that undue extension in sequential and parallel compositions of processes will result in non-conformity which defies the purpose of abstraction (i.e. anti-extension) as a way to attack state-explosion and at the same time to preserve conformity between an abstraction tier and an implementation.

IV. The model of CSP fits perfectly with the notion of conformance testing. In CSP model, we can concentrate on testing. Mutual conformance is the same as testing equivalence. Reduction, one of the two well-known transitive sub-classes of conformance, is the same as refinement. The excellent alignment between the model of CSP and the model of testing is unparallel. CSP has tools (such as ProBE and FDR) to support.

In the next chapter, we will investigate the weaknesses of extension towards the objective of a compositional and abstraction-oriented approach to facilitate test case generation. In Chapter 6, we will propose our solutions and other related results.

# Chapter 5

# The Problems of Extension Hierarchy

In the last chapter, we find it justified to use anti-extension hierarchy in CSP as the architectural framework to generate conformance test cases. It is not used without problems, however. In particular, one of the major problems is still the explosion of states. Anti-extension provides a kind of abstraction. We investigate the compositional way on top of anti-extension hierarchy in this chapter.

We identify 5 problems associated with extension in sequential and parallel compositions. The first two show that extension in general has problems associated with failures, even if the traces are respected by their *paradoxically* extended processes in sequential composition. The next one shows a counter-intuitive example. There are three processes which do not extend one another. A sequential composition of the first two processes, however, will form an extension relation to the third. Strictly speaking, it is not a problem of extension; nevertheless, we are interested in this kind of decomposition. The fourth one illustrates that undue decomposition or "pruning" of processes do not guarantee extension. Sometimes, it may introduce *extra* traces, on the contrary. The last, but not the least, illustrates a problem of extension in parallel composition. Sequential composition of corresponding components of concurrent processes of a system violates the extension property to be respected. The parallel composition of concurrent processes may not extend this kind of composition, which is also counter-intuitive.

We will discuss them in more detail in this chapter. We will first present examples and follow with an analysis of problems. They provide insights for us to build our theoretical results to be presented in the next chapter.

## 5.1 Problem 1: Preceding

**Example 5.1.1 (Preceding process extension problem)** Consider the processes in Figure 5.1. It illustrates that extending the preceding process in a sequential composition does not guarantee overall extension. The processes $Inspection$, *Ranking* and $Inspection$;*Ranking* are introduced on page 21. Suppose that we extend process $Inpsection$ with some alignment capability to form a process such as the process $Inspection_Alignment$. It will make an alignment for a die after the first inspection, and then re-inspect the die again. Finally, it terminates successfully. On the other hand, if the alignment is classified as failed, it will discard the die automatically and terminate. It is easy to see that the process $Inspection_Alignment$

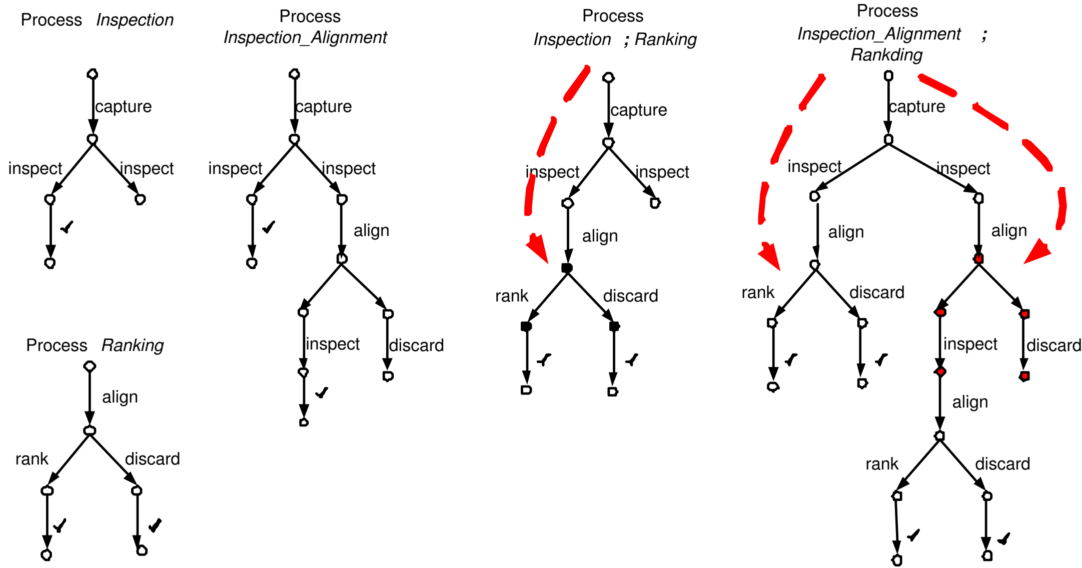


Figure 5.1: Problem Preceding. Extending the preceding process in a given sequential composition does not guarantee extension to that sequential composition.

extends the process *Inspection*.

$$Inspection_Alignment = capture \rightarrow ((inspect \rightarrow SKIP)$$
$$\Box(inspect \rightarrow align \rightarrow ((discard \rightarrow STOP) \sqcap (inspect \rightarrow SKIP))))$$

Suppose that the process *Inspection_Alignment* is composed sequentially with the process *Ranking* as shown in the figure. The process *Inpsection_Alignment*;*Ranking* may discard a die after the first alignment non-deterministically. The reason could be due to either the process *Inspection_Alignment* requires an additional alignment, or the process *Ranking* allows users to discard sub-quality dies. On the other hand, the process *Inpsection*;*Ranking* will always allow users to discard a die after the first alignment. In this connection, the process *Inspection_Alignment*;*Ranking* does not extend the process *Inspection*;*Ranking*, through the process *Inspection_Alignment* extends the process *Inspection*. The situation is illustrated in the figure by thick arrows indicating feasible paths for the trace $\langle capture, inspect, align \rangle$. It shows that the process *Inspection_Alignment*;*Ranking* may fail the action *discard*. On the other hand, the process *Inspection*;*Ranking* will not. ■

In the last example, it shows that when a preceding process is extended arbitrarily, the situation may turn out that the intuitively extended sequential composition may contain traces that are not parts of the intuitively being extended sequential composition.

**Problem 1 (Preceding Problem)** *Let P, Q and R be processes. Suppose that the process R extends the process P. The extended sequential composition R;Q, in general, does not extend the original sequential composition P;Q.*

## 5.2 Problem 2 : Suceeding

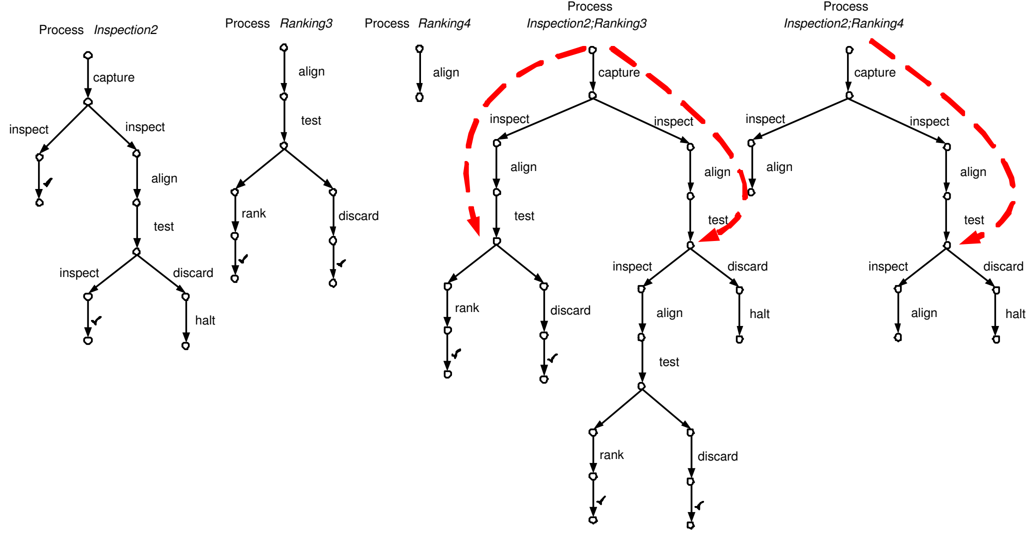


Figure 5.2: Problem Succeeding. Extending the succeeding process in a given sequential composition does not guarantee overall extension to that sequential composition.

**Example 5.2.1 (Succeeding process extension problem)** Consider the processes in Figure 5.2. It illustrates that extending the succeeding process in a sequential composition does not guarantee overall extension.

$$
\begin{aligned}
&Inspection2 = capture \rightarrow (inspect \rightarrow SKIP) \\
&\quad \Box(inspect \rightarrow align \rightarrow test \rightarrow \\
&\qquad ((inspect \rightarrow SKIP)\Box(discard \rightarrow halt \rightarrow STOP))
\end{aligned}
$$

$$Ranking3 = align \rightarrow test \rightarrow ((rank \rightarrow SKIP) \sqcap (discard \rightarrow SKIP))$$

$$Ranking4 = align \rightarrow STOP$$

The process *Inspection*2 behaves like the process *Inspection_Alignment* (see Example 5.1.1) except that it will *halt* after discarding a die. It is also enhanced to perform an extra *test* after an alignment. Similarly, the process *Ranking*3 behaves like the process *Ranking* (see Example 3.3.1) except that it introduces an extra test after the alignment. The process *Ranking*4, on the other hand, is a faulty ranking process which will terminate, whenever it makes an alignment. Hence, no test or ranking can be performed. Observe that the process *Ranking*3 extends the process *Ranking*4. The process *Ranking*3 includes a trace $\langle\rangle$ and $\langle align\rangle$. It improves the failures associated with these traces common to the process *Ranking*4. For instance, the process *Ranking*3 can proceed to perform a testing after the alignment; whereas the process *Ranking* will reach a deadlock. Nevertheless, the sequential composition *Inspection*2;*Ranking*3 may fail to proceed definitely to discard a die. On the other hand, the

process $Inspection2;Ranking4$ will always allow users to choose whether to discard a die or not. ■

In the last example, like the Problem Preceding, it shows that when a succeeding process is extended arbitrarily, the situation may turn out that the intuitively extended sequential composition may contain traces that are not parts of the intuitively being extended sequential composition. They are introduced via extending the succeeding process, rather than extending the preceding process. A similar argument as for Problem Proceeding is applied.

**Problem 2 (Succeeding Problem)** *Let P,Q and R be processes. Suppose that the process R extends the process Q. The extended sequential composition P;R, in general, does not extend the original sequential composition P;Q.*

## 5.3 Problem 3: Prefix

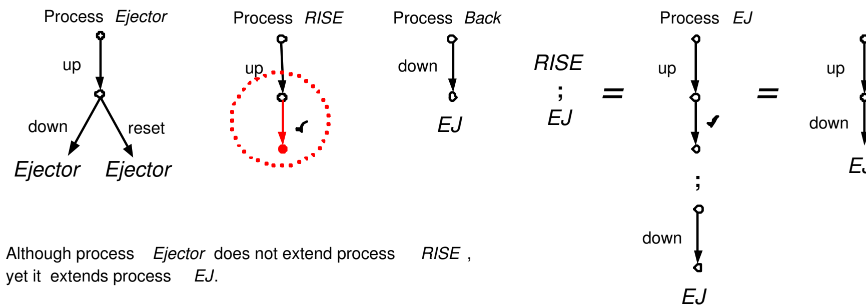


Figure 5.3: Problem 3. Process $RISE$ acts like a part of process $Ejector$, however, process $Ejector$ does not extends process $RISE$, which is counter-intuitive.

**Example 5.3.1 (Prefix process problem)** Consider the process *Ejector* which moves up and down continuously unless it resets to restart. It is easy to observe that it extends the process *EJ* which is similar to *Ejector* but without the reset capability. Figure 5.3 shows their process graphs.

$$Ejector = moveUp \rightarrow (moveDown \rightarrow Ejector) \Box (reset \rightarrow Ejector)$$

$$RISE = moveUp \rightarrow SKUP$$

$$\begin{aligned} EJ &= (moveUp \rightarrow SKIP);(moveDown \rightarrow EJ) \\ &= RISE;(moveDown \rightarrow EJ) \end{aligned}$$

It is also easy to observe that the processes *Ejector* and *RISE* do not extend each other. For instance, the sequence $\langle moveUp, \surd \rangle$ is a trace of the process *RISE*, but not a trace of the process *Ejector*. On the other hand, the sequence $\langle moveUp, moveDown \rangle$ is a trace of the process *Ejector*, but not a trace of the process *RISE*. However, all non-successful complete traces of

the process *RISE* are traces of the process *Ejector*. Moreover, suppose that the process *RISE* is composed with the process $(moveDown \rightarrow EJ)$ in a sequential manner to form the process $EJ$. The process *Ejector* will extend to this composed process (i.e. *EJ*). In a sense, the process *RISE* is a "prefix" of the process *Ejector*, which, in a sense, "extends" the process *RISE* in the sequential composition. ■

The sequential composition operator hides the intermediate successful termination event $\surd$ of the proceeding process. Informally, the kinds of failure exception (those related to successful complete trace) are resolved by CSP naturally.

**Problem 3 (Prefix Problem)** *Let P and Q be processes. The process P respects the behaviour of the process Q whenever the process Q may not terminate successfully. The process P may not extend the process Q.*

## 5.4 Problem 4: Decomposition

In this section, we give a few counter-examples to illustrate that the problems of extension with compositionality. Informally, an anti-extended process means that it has fewer traces and fails more often. First example demonstrates that although, intuitively, CSP agrees in its syntax and semantics, yet removal of process branches syntactically may violate extension. Compositional properties of processes are not maintained.

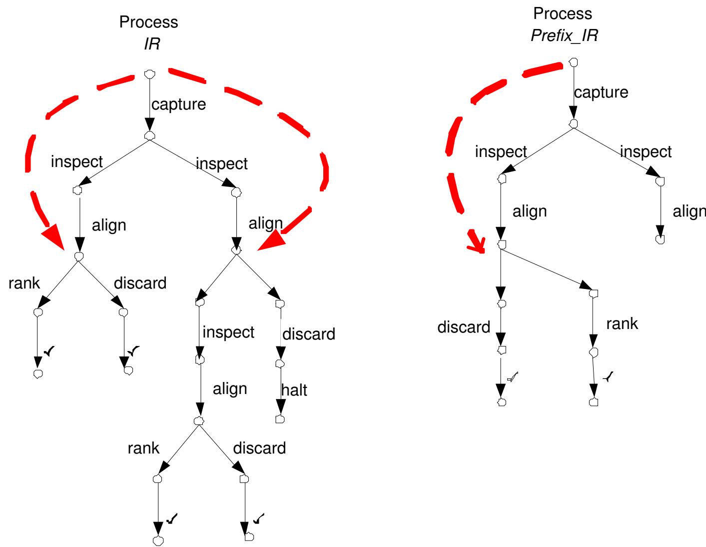


Figure 5.4: Problem Decomposition (a). A branch-pruned version of a process does not guarantee the extension to the original process.

**Example 5.4.1 (Decomposition problem 1)**

$$\begin{aligned} IR = capture \rightarrow (&inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP)\Box(discard \rightarrow SKIP))) \\ &\sqcap(inspect \rightarrow align \rightarrow ( \\ &\quad((inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP)\Box(discard \rightarrow SKIP))) \\ &\quad\sqcap(discard \rightarrow halt \rightarrow STOP)))) \end{aligned}$$

$$\begin{aligned} Prefix_IR = capture \rightarrow (&inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP)\Box(discard \rightarrow SKIP))) \\ &\sqcap(inspect \rightarrow align \rightarrow STOP) \end{aligned}$$

Figure 5.4 illustrates a problem in decomposition, namely, the failures of a "pruned" process may cause failure exception of an original process. The process $IR$ is similar to the process $Inspection2;Ranking3$ in Problem Succeeding. It has no test action. It first *captures* photo, and then inspects an image captured. It always makes an alignment, no matter a die is misaligned or not. After the alignment, it may, sometimes, rank the quality of the die, or discard the sub-quality one if the users want to. On the other hand, it may, sometimes, choose to repeat the inspection process, make alignment and so on. Unfortunately, the program is written wrongly with the second inspection action. It may, sometimes, start the re-inspection and sometimes, discard a die and then halt. The process $Prefix_IR$ is a pruned version of $IR$, a software engineer discovers the problem in the second inspection, and so one removes the second inspection and the subsequent behaviour. In a sense, a purpose of the process $Prefix_IR$ is to test the remaining behaviour of the process $IR$. Counter-intuitively, the process $IR$ may refuse to perform the ranking process non-deterministically; whereas the process $Prefix_IR$ will not. If there are errors associated to this failure, the pruned version would be ineffective to them. The figure also highlights the locations of failure exception. ■

Informally, a branch-pruned version of a process may remove some bad refusals from the original process. So, it may possibly perform better than the original process. In an extreme case, a process $Q$ actually refines the process $P \sqcap Q$ where $P$ is a process. Sometimes, decomposition through branch-pruning may even introduce unexpected traces in a sequential composition as illustrated in the following example.

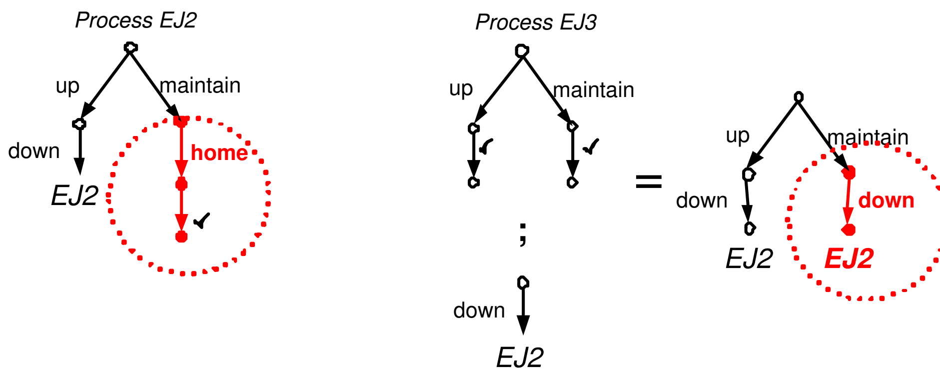


Figure 5.5: Problem Decomposition (b). Intuitive extension may have problems in sequential composition.

**Example 5.4.2 (Decomposition problem 2)** The process $EJ2$ (in Figure 5.5) is a modified version of $EJ$ (see Example 5.3.1), an ejector, which allows maintenance to be performed. It will also *move* to a safe place (i.e. its home) after the maintenance service call. After that, it terminates successfully. Similar to the process $Prefix_IR$ (see Example 5.4.1), the process $RISE2$ in certain sense is a pruned version of the process $EJ2$ which allows the program to *move up* or to be *maintained*. Unlike the process $Prefix_IR$, there is no failure exception, except that it is counter-intuitive in the sense of Problem Prefix. Despite the successful complete traces, the process $RISE2$ is intuitively extended by the process $EJ2$. The process ($moveDown \rightarrow EJ2$) is a part of behaviour of the process $EJ2$ right after the $moveUp$ operation in the process $EJ2$.

$$\begin{aligned} EJ2 &= (moveUp \rightarrow SKIP)\,;(moveDown \rightarrow EJ2) \\ &\quad \Box(maintain \rightarrow home \rightarrow SKIP) \\ RISE2 &= (moveUp \rightarrow SKIP)\Box(maintain \rightarrow SKIP) \\ EJ3 &= RISE2;(moveDown \rightarrow EJ2) \end{aligned}$$

Naturally, since the process $RISE2$ describes the $moveUp$ operation and is intuitively extended by process $EJ2$, so it "supposedly" combines with the process $(moveDown \rightarrow EJ2)$ to form the process $EJ3$, a process which is intuitively described as a part of the process $EJ2$'s behaviour. Accidentally, the sequence $\langle maintain, moveDown\rangle$ is a trace of the process $EJ3$ but is not a trace of the process $EJ2$; and sequence $\langle maintain, home\rangle$ is a trace of the process $EJ2$ but not a trace of the process $EJ3$. Unlike the counter-intuitive case in Problem Prefix, $EJ$ does not extend $EJ3$. In fact, it is easy to be observed the other way round. ■

Intuitively, the two examples above demonstrate that there is a problem if we cannot resolve the failures before sequential decomposition. The next example will demonstrate that there is also a problem even if we resolve the failures before sequential decomposition. The final example shows that even the composition can be shown to be equivalent to the normalized version, yet failures of a sub-process may still cause failure exception.

**Example 5.4.3 (Decomposition problem 3)**

$$\begin{aligned} &Improved_Inspection2_Ranking2 = \\ &\quad (inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \sqcap (maintain \rightarrow SKIP))) \\ &\quad \sqcap(inspect \rightarrow align \rightarrow \\ &\qquad ((inspect \rightarrow align \rightarrow Ranking2)\Box(maintain \rightarrow align \rightarrow Ranking2))) \end{aligned}$$

$$\begin{aligned} &Inspection2;Ranking2 = (inspect \rightarrow align \rightarrow Ranking2) \\ &\quad \sqcap(inspect \rightarrow align \rightarrow \\ &\qquad ((inspect \rightarrow align \rightarrow Ranking2)\Box(maintain \rightarrow align \rightarrow Ranking2))) \end{aligned}$$

$$\begin{aligned} &Inspection2 = (inspect \rightarrow SKIP) \\ &\quad \sqcap(inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP)\Box(maintain \rightarrow SKIP))) \end{aligned}$$

$$Ranking2 = align \rightarrow ((rank \rightarrow SKIP)\Box(maintain \rightarrow SKIP))$$

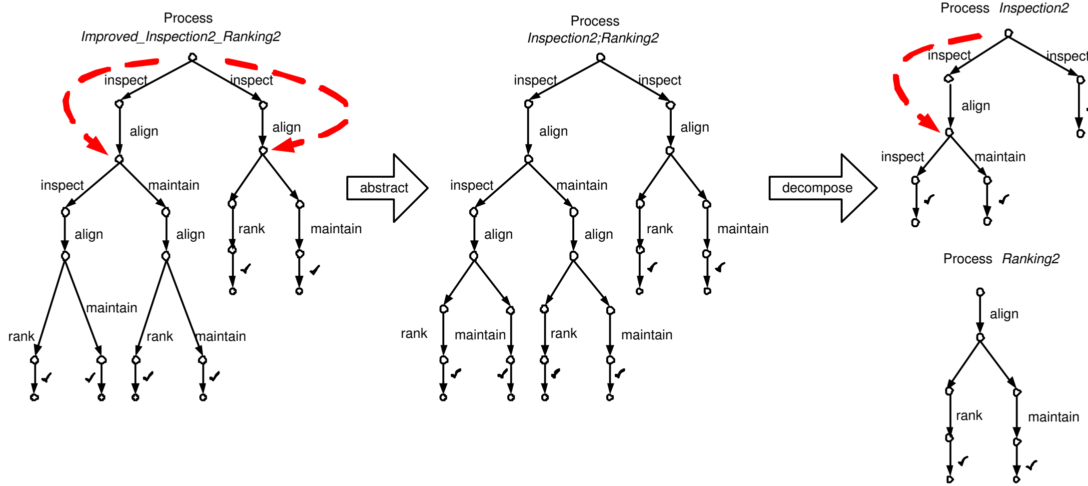


Figure 5.6: Problem Decomposition (c). Anti-extension followed by process decomposition does not preserve extension in general.

Consider the process *Impoved_Inspection2_Ranking2*, which is illustrated in Figure 5.6. Suppose that we are interested in decomposing this process into two sequential components, namely the process *Inspection2* and the process *Ranking2*. The process *Ranking2* may behave like the process *Impoved_Inspection2_Ranking2* after some (but not all) successfully completed trace. One may attempt to *generalize* the behaviour of the process *Impoved_Inspection2_Ranking2* so that the process *Ranking2* becomes a viable candidate in decomposition. Changing the choice composition to an internal choice composition will not introduce any trace.

Hence, one possible way of abstraction on the process *Impoved_Inspection2_Ranking2* is to construct the process *Inspection2;Ranking2* as shown in the figure. Observe that the process *Impoved_Inspection2_Ranking2* extends the process *Inspection2;Ranking2*. This process *Impoved_Inspection2_Ranking2* can be considered as the sequential composition of processes *Inspection2* and *Ranking2*. Observe that, as highlighted in the figure, the process *Inspection2* may proceed to perform a maintenance service definitely, however, the process *Impoved_Inspection2_Ranking2* may not[1]. ■

**Example 5.4.4 (Decomposition problem 4 : False Alarm in Refusal Testing)** Example 5.4.3 also shows another problem in decomposition. The process *Inspection2;Ranking2* is exactly the sequential composition from the processes *Inspection2* and *Ranking2*. However, the process *Inspection2* will not fail to proceed if the environment offers both *inpsect* and *maintain*; whereas the sequential composition will refuse to proceed. Technically, the set $\{inspect, maintain\}$ is not a refusal for the process *Inspection2*; whereas it is a refusal for the process *Inspection2;Ranking2* after the same trace $\langle inspect, align\rangle$. Figure 5.7 illustrates this point.

[1]In fact, processes *Inspection2* and *Inspection2;Ranking2* suffers from a similar problem as illustrated in Example 5.4.1. The process *Inspection2* will not refuse to the second inspection after the first alignment. However, the process *Inspection2;Ranking2* may.

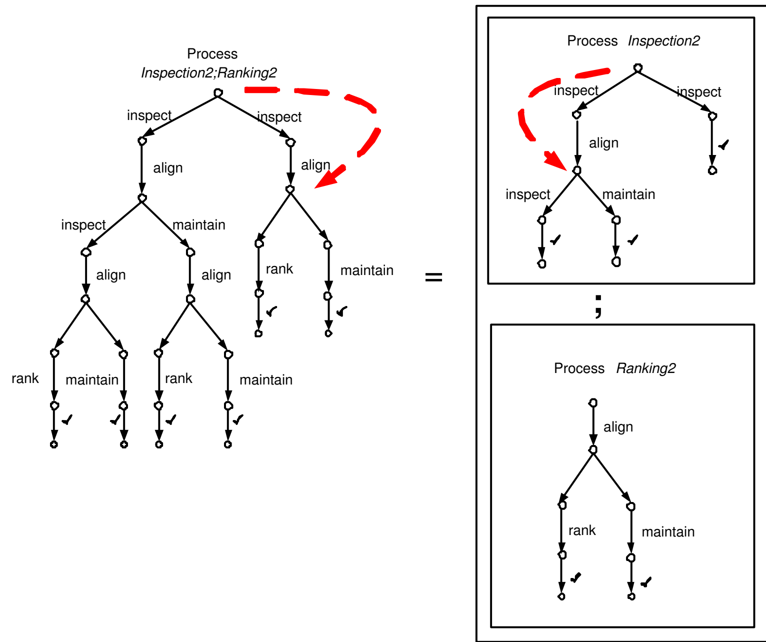


Figure 5.7: Problem Decomposition (d). Even two processes are equivalent; a modular view may not be used alone to derive the refusals.

An implication is that it can raise a *false alarm* if we perform refusal testing based on the information provided by process *Inspecton*2 on an implementation of the process *Inspection*2;*Ranking*2. An implementation will fail to proceed with the environmental offer $\{inspect, maintain\}$, which in fact, is not required by the specification *Inspection*2;*Ranking*2. So, it is, in general, not reliable to generate test cases from individual modules, and then composed sequentially. ■

As illustrated in the last example, the generation of test cases in a compositional way may not be reliable even if sequential composition is the same as specification semantically. So obviously, the conformance property will not be reliable if it is obtained from a compositional extension hierarchy either. A common approach is to verify test cases against specification to ensure that they are also reliable from specification point of view.

**Problem 4 (Decomposition Problem)** *Process decomposition and sequential composition may not preserve extension. Even overall extension is preserved, test cases generated in a compositional way may still be unreliable.*

## 5.5 Problem 5: Paralleling

Extension also has problems in parallel composition. We illustrate the problem by an example.

**Example 5.5.1 (Paralleling Problem)** Figure 5.8 shows a configuration of a pick arm mounted with a movable vision camera. The simple pick arm, the process *SimplePickArm*

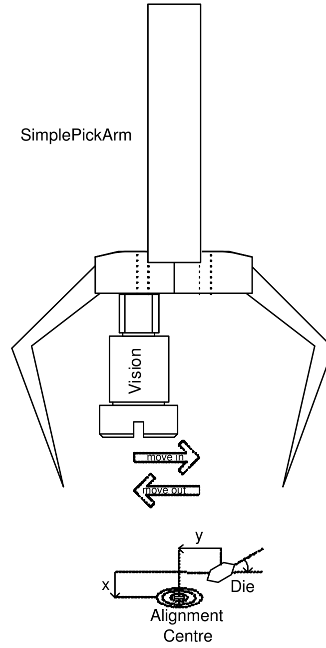


Figure 5.8: A pick arm mounted with a movable vision camera

(Figure 5.9), consists of two sequential activities, namely an inspection phase and a picking phase. The inspection phase, the process *Inspection*3 will trigger a photo capturing of the vision system attached to it and will inspect the image captured. After the inspection, this phase may be completed. Sometimes, it detects that there is a mal-alignment between the die concerned and the centre of alignment. It will try to re-align the die for a number of times. At each time, if the alignment completes, it will decide to perform one of the two activities. Either it *warns* the users that an alignment is done and re-starts the capturing process completely to ensure that the die is perfectly located to be picked up by the pick arm, or it may discard the die for some reason. The picking phase, the process, *SimplePick*, allows further (fine) alignment repeatedly. It also allows users to either pick a die up or discard it.

$$SimplePickArm = Inspection3; SimplePick$$

$$Inspection3 = capture \rightarrow ((inspect \rightarrow SKIP) \Box (inspect \rightarrow align \rightarrow (\mu X.(align \rightarrow X) \Box ((discard \rightarrow SKIP) \sqcap (warn \rightarrow Inspection3)))))$$

$$SimplePick = (align \rightarrow SimplePick) \Box (pick \rightarrow SKIP) \Box (discard \rightarrow SKIP)$$

$$Vision = Capturing; SimpleAlignment$$

$$Capturing = move\ in \rightarrow capture \rightarrow moveout \rightarrow SKIP$$

$$SimpleAlignment = align \rightarrow SKIP$$

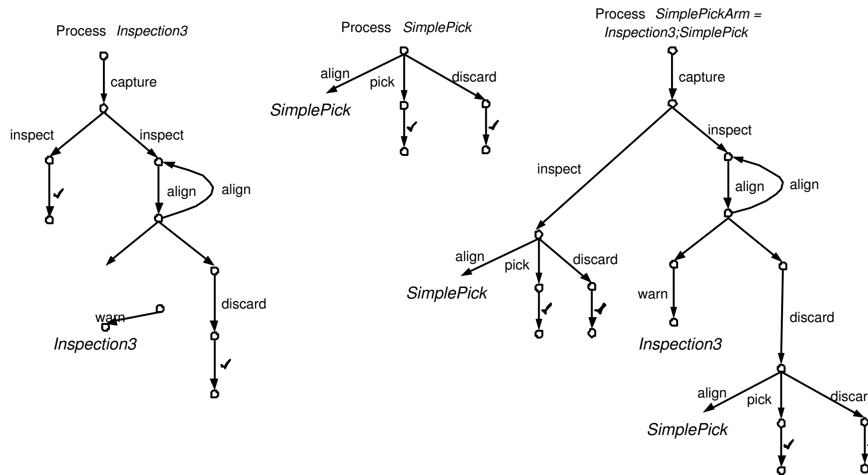


Figure 5.9: Process *SimplePickArm* and its sequential components

The process *Vision* (Figure 5.10) is the vision process attached to the pick arm. It also consists of two sequential phases of activities, namely a photo capturing and a simple die alignment. The photo capturing activity is a process *Capturing* which will *move the camera in*, capture the die image and finally *move the camera out*. The second phase simply waits until alignment occurs so that it can then continue to perform the next round of photo capturing.

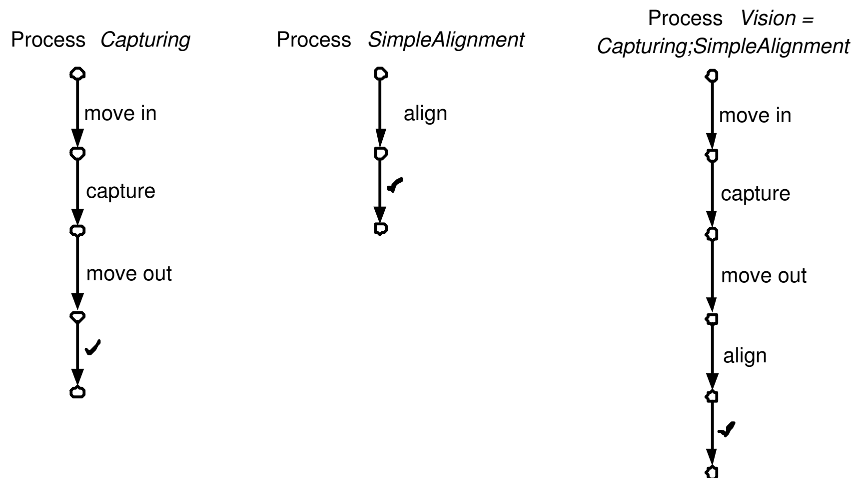


Figure 5.10: Process *Vision* and its sequential components

Supposed that two sequential compositions (*SimplePickArm* and *Vision*) are intended to operate concurrently. Each of them is composed of two sequential processes. Intuitively, one possible scenario of the machine behaviour could be expressed as the parallel composition of respective proceeding modules followed by the parallel composition of respective following modules. In other words, it forms the process composition which behaves like the process *Bundle* below. Figure 5.11 shows these two parallel compositions.

$$Bundle = (Capturing \parallel Inspection3);(SimpleAlignment \parallel SimplePick)$$

Intuitively, the process (*SimplePickArm*||*Vision*) should extend this parallel composition. However, as depicted in Figure 5.12, in fact, the process *Vision*||*SimplePickArm* may refuse to discard a die; whereas the actual parallel composition (*SimplePickArm*||*Vision*) will never refuse to.

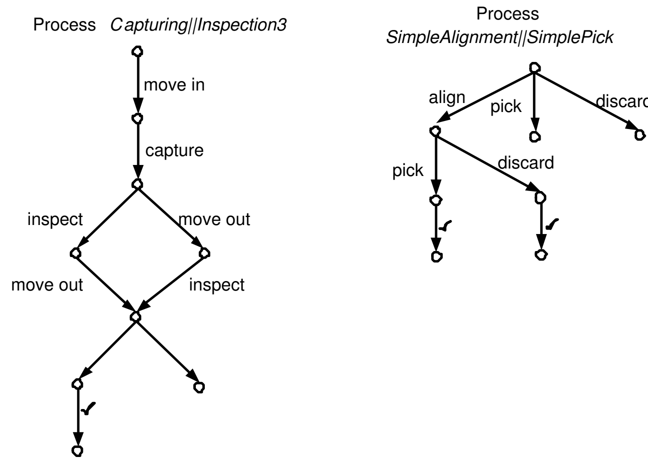


Figure 5.11: Parallel composition of processes

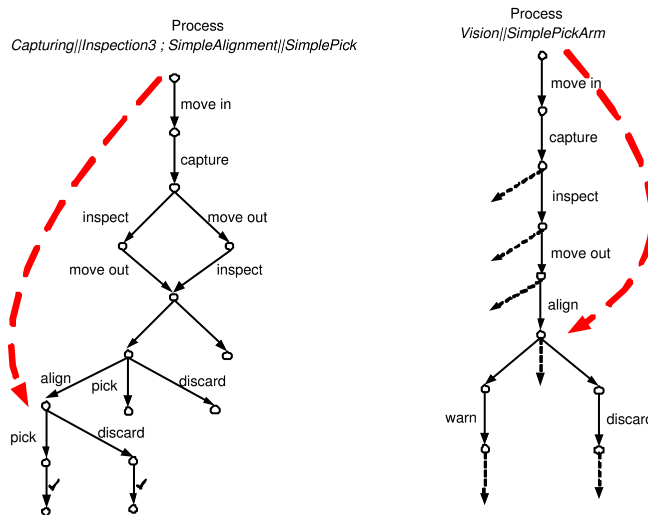


Figure 5.12: Problem Paralleling. A concurrent system, in general, does not extend a system of paralleling sequential sub-processes of this concurrent system, aggregated by sequential composition.

Consequently, the process *Vision* || *SimplePickArm* does not extend the process *Bundle*. Intuitively, the problem arises because a trace, such as $\langle capture, inspect, align \rangle$, of the process *SimplePickArm* can be obtained from the process *Inspection*3 alone, as well as from a trace, such as $\langle capture, inspect \rangle$, of the process *Inspection*3 appended with a trace, such as $\langle align \rangle$, of process *SimplePick*. Unfortunately, in the composition *Bundle*, owing to the sequential

composition operator, the process ($Capturing \parallel Inspection3$) should be completed before any non-empty traces of the process ($SimpleAlignment \parallel SimplePick$) are observed. The process *Capturing* always rejects the communication of event *align* and so, the alignment feature of the process *Inspection*3 is disallowed to be observed.

$$\begin{aligned} &Capturing \parallel Inspection3 = move\ in \rightarrow capture \rightarrow \\ &\quad ((move\ out \rightarrow SKIP) \parallel (inspect \rightarrow SKIP)); (SKIP \sqcap STOP) \end{aligned}$$

Concidentally, in the process $SimpleAlignment \parallel SimplePick$, alignment is possible. It happens that it allows the selection of choices by the users. In other words, the parallel composition serving as the preceding process in the sequential composition actually removes the poorer failures, leaving those traces with better failures. ■

**Problem 5 (Paralleling Problem)** *A concurrent system, in general, does not extend the sequential composition of processes, which are parallel compositions of corresponding components of that system.*

## 5.6 Chapter Summary

We have shown the problems of extension in sequential and parallel composition. Both direct sequential composition (Preceding process extension problem and Succeeding process extension problem) and indirect sequential composition (Process decomposition extension problem) have problems. To respect the notion that testing should be a finite task, it is also desirable to allow finite behaviour about (infinite) process to be derived (Prefixing problem extension problem), in particular when the systems composed of concurrent processes (Parallel processes extension problem) when state-explosion problem is usually intractable in the global process. In the next chapter, we will formulate our solution towards these problems.

# Chapter 6

# A New Relation for Compositional Test Case Generation

## 6.1 Introduction

In this chapter, we propose a new extension relation and first summarize our solutions to these problems identified in the last chapter in Table 6.1. We generalize the notion of extension and investigate the properties of sequential composition and parallel composition of CSP processes. We discover the necessary and sufficient conditions to ensure a process to be decomposable in serial and maintain overall extension in a compositional manner. We further show that this nice property is also preserved if the preceding process or the succeeding process in a sequential composition is replaced by their respective anti-extended versions. In this connection, we are able to build abstraction hierarchy in a compositional way that preserves anti-extension. Test cases can be generated from different levels of abstraction and composed together to form extended test cases for the specification. Hence, they are conformance test cases of an implementation of the specification.

| Extension problems | Our solutions |
|---|---|
| Problem Preceding | Head Anti-Extension Theorem 6.4.2 |
| Problem Succeeding | Tail Anti-Extension Theorem Theorem 6.4.1 |
| Problem Prefix | Definition 6.3.1 |
| Problem Decomposition | Decomposition of Processes Theorem 6.4.4 |
| Problem Paralleling | Decomposition of Concurrent Processes Theorem 6.5.3 |

Table 6.1: A summary of our solution to extension problems

## 6.2 Sequential Extension

In this section, we *propose* a generalized extension, called sequential extension, which tackles Problem Prefix. We shall restrict it to a new relation, $\sigma$-extension, to be introduced later in this chapter. In particular, we shall show that

- sequential extension is a generalization of extension (Proposition 6.2.2),

- test equivalence is preserved if and only if sequential extension equivalence is preserved (Proposition 6.2.3), and
- sequential extension is a partial order (Proposition 6.2.1).

Informally, the notion of extension for a process $P$ extending a process $Q$ is that the operation sequences of the process $Q$ should be the operation sequences of the process $P$ and that, at any time, the process $P$ may come to a deadlock only when the process $Q$ can come to a deadlock. The notion of sequential extension follows this train of thought. The difference is that it allows the process $Q$ to terminate successfully "earlier" than that of the process $P$. On the other hand, it is not too generous. When the process $Q$ may terminate successfully, the process $P$ should terminate successfully whenever the process $P$ cannot find a possible way to perform another action. Moreover, the process $P$ even forces the process $Q$ to be able to terminate successfully whenever the former finds itself may terminate successfully and the process $Q$ may proceed with a user action.

**Definition 6.2.1 (Sequential extension)** *Let P and Q be processes. The process P* extends *the process Q sequentially, denoted by* $Q \sqsubseteq_{sext} P$, *iff*

*I.* $\alpha P = \alpha Q$,

*II.* $\forall s \in traces(Q) . s \in traces(P)$,

*III.* $\forall s\hat{}\langle\surd\rangle \in traces(Q), \nexists s\hat{}\langle\theta\rangle \in traces(P), s\hat{}\langle\surd\rangle \in traces(P)$,

*IV.* $\forall s\hat{}\langle\surd\rangle \in traces(Q), s\hat{}\langle\surd\rangle \notin traces(P), \exists \theta . s\hat{}\langle\theta\rangle \in traces(P)$,

*V.* $\forall s\hat{}\langle\surd\rangle \notin traces(Q), (s,X) \in failures(P), s \in traces(Q)$.
$$\begin{cases} (a)\ Q/s \text{ deadlocks immediately} & \text{if } s\hat{}\langle\surd\rangle \in traces(P) \wedge \nexists s\hat{}\langle\theta\rangle \in traces(P) \\ (b)\ (s,X) \in failures(Q) & \text{otherwise} \end{cases}$$

*VI.* $\forall s\hat{}\langle\surd\rangle \in traces(Q), (s,X) \in failures(P), s \in traces(Q)$,
$$\begin{cases} (a)\ (s, X \setminus \{\surd\}) \in failures(Q) & \text{if } s\hat{}\langle\surd\rangle \notin traces(P) \wedge s\hat{}\langle\theta\rangle \notin traces(Q) \\ (b)\ (s,X) \in failures(Q) & \text{otherwise} \end{cases}$$

*and*

*VII.* $\forall t \in divergences(P), t \in traces(Q) . t \in divergences(Q)$.

*where s is not a successful complete nor a diverging trace and event* $\theta$ *is a non-*$\surd$ *symbol in the alphabet. Event* $\theta$ *is called a user alphabet, and throughout the remaining paragraphs of this thesis, we shall use* $\theta$ *consistently as a user alphabet.*

**Note** To simplify the cross-reference in subsequently proofs in this chapter, whenever a proof refers to any numbered item in the definition of sequential extension, it will be either introduced with the clause "The proof of Condition" before the refereed items, or use "Condition" without quantifier before the refereed items provided that the scope is clearly referring to a numbered item in the definition of sequential extension.

The full mapping is as follow:

| Clauses In Subsequent Proofs | Referred Item in Definition 6.2.1 |
|---|---|
| The proof of Condition I | Condition I |
| The proof of Condition II | Condition II |
| The proof of Condition III | Condition III |
| The proof of Condition IV | Condition IV |
| The proof of Condition V(a) | Condition V(a) |
| The proof of Condition V(b) | Condition V(b) |
| The proof of Condition VI(a) | Condition VI(a) |
| The proof of Condition VI(b) | Condition VI(b) |
| The proof of Condition VII | Condition VII |

Table 6.2: A cross-reference for a referred item of the definition of sequential extension in subsequent proofs

Informally, the failures conditions above refer to the general case (that is, Conditions V(b) and VI(b)), a forced termination case (that is, Condition V(a)) and an intended early termination case (that is, Condition VI(a)).

**Discussions** In the definition, the sub-case V(a) states that the process being sequentially extended (that is, the process $Q$ in the definition) will reach a deadlock after a trace. In fact, it is a natural consequence because, first, Condition II requires that any non-successful complete trace in the process $Q$ should be a trace in the process $P$ and, secondly, Condition V(a) explicitly requires that the process $P$ cannot have any non-successful trace extension after the trace $s$. Moreover, in Condition V, it requires that $Q$ cannot reach successful termination. Hence, there is no choice but to form a deadlock immediately. Informally, it can merge with Condition V(b) to simplify the definition because the deadlock condition can naturally be obtained. We keep it in the definition to clarify the proving steps in the following Lemma, Propositions and Theorems. ⧫

**Example 6.2.1 (Sequential extension example)**

(1) $STOP$
(2) $SKIP \sqcap STOP = (\surd \rightarrow STOP) \sqcap STOP$
(3) $SKIP = \surd \rightarrow STOP$
(4) $SKIP \Box (align \rightarrow SKIP) = SKIP \Box (STOP \sqcap (align \rightarrow SKIP))$
(5) $align \rightarrow rank \rightarrow SKIP$

Consider the above five simple processes (Figure 6.1). The process $STOP$ does not extend the process $SKIP$ sequentially. This is because Condition IV is violated, which requires the process $STOP$ to have at least a non-empty trace. Obviously, $STOP$ has no such trace. In fact, owing to the violation of Condition IV, the processes $STOP$ and $SKIP$ do not extend processes $SKIP \sqcap STOP$ and $SKIP \Box (align \rightarrow SKIP)$ sequentially. Similarly, all the first four processes (that is, (1) to (4)) cannot extend the process $align \rightarrow rank \rightarrow SKIP$ sequentially. This is because when Condition II is fulfilled, the sequence $\langle align, rank \rangle$ should be parts of their traces, which is not the case. In fact, extensions hold amongst the first four processes ((1) to (4)). However, there is no extension relation between the fifth one and processes (2) to (4). For instance, the sequence $\langle \surd \rangle$ is a trace of any of these processes, but it is not a trace of the fifth. On the other hand, the five processes are extended sequentially from (1) to (5).

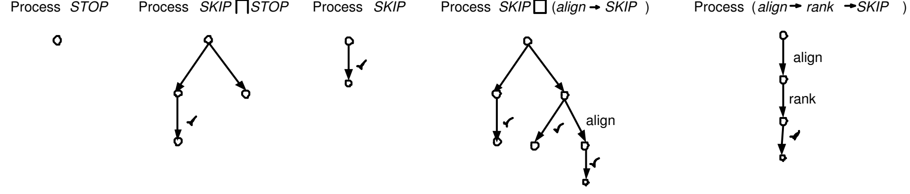


Figure 6.1: Examples of sequential extension. The processes extend the processes on their respective left hand side sequentially.

$$\begin{array}{ll}
(i) & STOP \sqsubseteq_{sext} SKIP \sqcap STOP \\
(ii) & SKIP \sqcap STOP \sqsubseteq_{sext} SKIP \\
(iii) & SKIP \sqsubseteq_{sext} SKIP \Box (align \rightarrow SKIP) \\
(iv) & SKIP \Box (align \rightarrow SKIP) \sqsubseteq_{sext} align \rightarrow rank \rightarrow SKIP
\end{array}$$

■

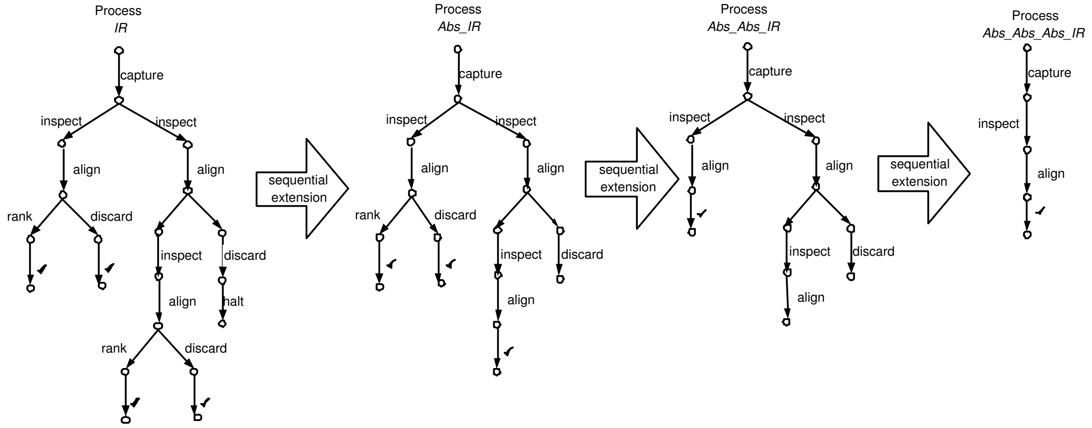


Figure 6.2: Sequential extension is a generalized extension relation, and is a preorder relation.

**Example 6.2.2 (Sequential extension example on transitive property)** The process *Abs_IR* captures a photo, inspects an image, makes necessary alignment, and then may discard a die, perform ranking tasks, or re-inspect the alignment result and make further alignment. Finally, it may terminate successfully. However, it may not if it discards a die.

$$\begin{aligned}
Abs_IR = capture \rightarrow & (inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \Box (discard \rightarrow SKIP))) \\
& \sqcap (inspect \rightarrow align \rightarrow ( \\
& \quad ((inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \Box (discard \rightarrow SKIP))) \\
& \quad \sqcap (discard \rightarrow STOP))))
\end{aligned}$$

The process *IR*, (see page 34), on the other hand,

$$\begin{aligned}
IR = capture \rightarrow & (inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \Box (discard \rightarrow SKIP))) \\
& \sqcap (inspect \rightarrow align \rightarrow ( \\
& \quad ((inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP) \Box (discard \rightarrow SKIP))) \\
& \quad \sqcap (discard \rightarrow halt \rightarrow STOP))))
\end{aligned}$$

after discarding a die (that is, after the trace $\langle capture, inspect, align, discard\rangle$), will reach a state which is

| | |
|---|---|
| either | $SKIP$ |
| or | $halt \rightarrow STOP$ |

non-deterministic. In other words, it will behave like the process $SKIP \sqcap halt \rightarrow STOP$. In a similar manner, the process $Abs_IR$ will behave like $SKIP \sqcap STOP$ after the same trace. Since the constraints on checking the special event $\surd$ is relaxed by the definition, we may concentrate on the second branch in the internal choice compositions. Obviously, $STOP$ will refuse any event immediately; whereas $halt \rightarrow STOP$ will refuse all except $halt$. Hence, the failures for the process $Abs_IR$ associated with this particular trace are respected by the process $IR$. Similarly, the failures after the second alignment of the process $Abs_IR$ is also respected by the $IR$. $Abs_IR$ will behave like the process $SKIP$ after the second alignment action; whereas $IR$ will behave like the process $(rank \rightarrow SKIP)\Box(discard \rightarrow SKIP)$. The same situation applies to other sequences of operations and hence the process $IR$ extends the process $Abs_IR$ sequentially[1].

Consider the process $Abs_Abs_IR$. It is similar to the process $Abs_IR$, but it has no ranking task and may terminate successfully after the first alignment. Moreover, it may terminate (unsuccessfully) after the second alignment. It can be observed that the process $Abs_IR$ extends the process $Abs_Abs_IR$ sequentially.

$$\begin{aligned} Abs_Abs_IR = capture \rightarrow & (inspect \rightarrow align \rightarrow SKIP) \\ & \sqcap(inspect \rightarrow align \rightarrow ( \\ & \quad ((inspect \rightarrow align \rightarrow ((rank \rightarrow SKIP)\Box(discard \rightarrow SKIP))) \\ & \quad \sqcap(discard \rightarrow STOP)))) \end{aligned}$$

Similarly, it is easy to observe that the process $Abs_Abs_Abs_IR$, which can only capture a photo, inspect an image and make an alignment, is extended by the process $Abs_Abs_IR$ sequentially.

$$Abs_Abs_Abs_IR = capture \rightarrow inspect \rightarrow align \rightarrow SKIP.$$

$$Abs_Abs_Abs_IR \sqsubseteq_{sext} Abs_Abs_IR \sqsubseteq_{sext} Abs_IR \sqsubseteq_{sext} IR$$

■

In fact, sequential extension relations are transitively held among the processes in the above example. They are guaranteed by the following proposition that shows sequential extension to be a preorder (relation).

**Proposition 6.2.1** *Sequential extension is a partial order relation.*

*Proof:* We need to prove the reflexivity, anti-symmetry and transitivity.

**Reflexive** Trivial.

[1] It also shows that sequential extension is not a kind of conformance relation (Definition 4.1.2). Consider another example as follows. The process $B = b \rightarrow STOP$ refines the process $AB = (a \rightarrow STOP) \sqcap (b \rightarrow STOP)$. Refinement in CSP is the same reduction, a kind of conformance relation. Hence, the process $B$ conforms to $AB$. However, sequential extension does not hold between these two processes.

**Anti-symmetric** We prove by contradiction. Let $P$ and $Q$ be processes. Suppose that the processes $P$ and $Q$ extend each other sequentially, but they are not failures-divergences-equivalent. We first prove that they should be trace-equivalent. In other words, they should have the same trace set. Consider a trace $s$ of the process $Q$. There are two scenarios to be considered, namely, that the trace $s$ is a successful complete trace, and that it is not. In other words, $s$ ends with or without the alphabet $\surd$.

I. Suppose $s$ is not a complete trace of the process $Q$. Obviously, by Condition II of $Q \sqsubseteq_{sext} P$, $s$ should be a trace of the process $P$.

II. Now, suppose $s$ is a complete trace of the process $Q$. Let us rewrite $s$ as $t\hat{}\langle\surd\rangle$. Obviously, $t$ is not a successful complete trace because the symbol $\surd$ can only appear at the end of a trace (such as $s$). By exhaustion, $s$ can either be a trace or not a trace of the process $P$. We shall prove by contradiction that $s$ should be a trace of the process $P$. Assume the contrary. By Condition IV of $Q \sqsubseteq_{sext} P$, there is a user alphabet $\theta$ which will form a trace $t\hat{}\langle\theta\rangle$ of the process $P$. We observe that this trace is not a complete successful trace. Hence, by Condition II of $P \sqsubseteq_{sext} Q$, $t\hat{}\langle\theta\rangle$ should be a trace of the process $Q$. By Condition III of $Q \sqsubseteq_{sext} P$, $s$ should be a trace of the process $P$. This contradicts the assumption that $s$ is not a trace of the process $P$.

In a similar manner, we can show that all traces of the process $P$ should also be traces of the process $Q$. The processes $P$ and $Q$ are trace-equivalent. Revisiting the conditions of sequential extension, the conditions required to the failures are Conditions (V) and (VI). In particular, since the processes $P$ and $Q$ are trace-equivalent, Condition VI(a) will become irrelevant. Hence, all failures of the process $Q$ should be failures of the process $P$ and vice versa. The two processes are failures-equivalent. The equivalence on the two sets of divergences for these two processes follows directly from the conditions about divergences in the definition and the trace-equivalent property between these two processes. Consequently, the processes $P$ and $Q$ are equivalent in the failures-divergences model. In other words, $P = Q$.

**Transitive** Let $P$, $Q$ and $R$ be processes. Suppose that

$$\text{the process } P \text{ extends the process } Q \text{ sequentially and} \tag{6.1}$$

$$\text{the process } Q \text{ extends the process } R \text{ sequentially.} \tag{6.2}$$

The following notations are used in the proving steps below:

$$u = s\hat{}\langle\surd\rangle \text{ and}$$
$$v = s\hat{}\langle\theta\rangle$$

where the symbols $\theta$ and $s$ are defined in the definition of sequential extension (Definition 6.2.1). In other words, if the notation $v$ is used, it means that there exists a non-successful complete trace. A trace $v$ of a process is not necessary a trace, also denoted by $v$, of another process. We use a looser term to simplify the over-introduction of symbols to distinguish various $v$'s in the proof and will state more clearly whenever there is any ambiguity. Throughout the following proving steps, we assume either a failure $(s,X)$ or a failure $(s,X\backslash\{\surd\})$ of the process $P$ is inferred to be a failure of the process $R$.

Traces We first prove the trace condition (that is, Conditions II to IV).

**The proof of Condition II** Consider a non-successful trace $s$ of the process $R$. By Condition II of Equation (6.2), $s$ should be a trace of the process $Q$. Similarly, by Condition II of Equation (6.1), $s$ should also be a trace of the process $P$.

**The proof of Condition III** Suppose $u$ is a trace of the process $R$ and $v$ is not a trace of the process $P$. Suppose that there is no such trace like $v$ for the process $Q$. By Condition III of Equation (6.1), $u$ should be a trace of the process $Q$. Combined with the given condition on $P$, by Condition IV of Equation (6.1), $u$ should be a trace of the process $P$. On the other hand, suppose that there is a trace like $v$, denoted by another $v$ for the process $Q$. By Condition II of Equation (6.1), $v$ should be a trace of the process $Q$. This is a contradiction. Hence, $u$ should be a trace of the process $P$.

**The proof of Condition IV** Suppose $u$ is a trace of the process $R$ but not a trace of the process $P$. Let us consider two scenarios, namely, whether or not $u$ is also a trace of the process $Q$. First, suppose $u$ is not a trace of the process $Q$. By Condition IV of Equation (6.2), $v$ should be a trace of the process $Q$. Consequently, by Condition II of Equation (6.1), it should be a trace of the process $P$. Secondly, suppose that $u$ is a trace of the process $Q$. It follows directly from Condition IV of Equation (6.1) that $v$ should be a trace of the process $P$.

Failures We are now going to prove the failures conditions (Conditions V and VI).

**The proof of Condition V(a)** Suppose $u$ is a trace of the process $P$ but not a trace of the process $R$. We are going to show that any failure with the $\surd$-symbol removed (that is, $(s, X\backslash\{\surd\})$) of the process is a failure of the process $R$. Suppose $s$ is a trace of the process $R$, and $(s,X)$ is a failure of the process $P$. By Condition II of Equation (6.2), $s$ should be a trace of the process $Q$. By exhaustion, there are two possible scenarios, namely, whether or not $u$ is a trace of the process $Q$.

A. We first consider the scenario that $u$ is a trace of $Q$. By Condition VI(b) of Equation (6.1), the failure $(s,X)$ should be a failure of the process $P$. Since $u$ is not a trace of the process $R$, by Condition V(a) of Equation (6.2), the failure $(s, X\backslash\{\surd\})$ should be a failure of the process $R$.

B. Secondly, consider the scenario that $u$ is not a trace of $Q$. By Condition V(a) of Equation (6.1), the failure should be a failure of the process $Q$. Like the previous case, the subset $(s, X\backslash\{\surd\})$ should be a failure of the process $R$.

**The proof of Condition V(b)** Suppose $u$ is not a trace of both the processes $R$ and $P$. We also consider whether or not $u$ is a trace of the process $Q$.

A. First, suppose $u$ is not a trace of the process $Q$. By Condition V(b) of both relations (6.1 and 6.2), a failure $(s,X)$ of the process $P$ should be a failure of the process $R$.

B. Secondly, suppose $u$ is a trace of the process $Q$. We need to consider whether or not the process $R$ will reach a deadlock.

   i. Suppose that it reaches a deadlock after the trace $s$. Obviously, if this is

really the case, the process $R$ will refuse any event and, hence, a failure $(s,X)$ of the process $P$ should be a failure of the process $R$.

ii. On the other hand, suppose that the process $R$ will not reach a deadlock after $s$. We have two more scenarios to consider. *Case* $(i)$ : Suppose $v$ is a trace of the process $R$. In this case, $v$ should be a trace of the process $Q$; otherwise it will violate Condition II of Equation (6.2). *Case* $(ii)$ : Suppose $v$ is not a trace of the process $R$. Another $v$ may or may not be a trace of the process $Q$. However, if the latter $v$ is not a trace of the process $Q$, it will either correspond to the deadlock case above, or case $(i)$ above (which violates Condition II). In other words, the two scenarios can be considered as one case, which is to consider $v$ to be a trace of the process $Q$. By Conditions VI(b) of Equation (6.1) and V(b) of Equation (6.2), $(s,X)$ should be a failure of the process $R$.

Thus, the failures conditions of the "transitive property" of the sequential extension hold.

Divergences The divergences conditions follow directly from the definition.

□

Informally, sequential extension differs from extension in the treatment of successful complete traces. It follows from the definition of extension that if we know that all traces of a process being extended are included in an extending process, then Conditions (II) and (III) above should be automatically fulfilled. Conditions IV and VI(a) are irrelevant because they will violate the trace inclusion property imposed by the extension relation. All other failures conditions are not as strict as that of extension. It requires either the failures inclusion as extension or a subset of the latter. The divergences condition is the same as extension. Consequently, extension is a special kind of sequential extension. Looking at it from another perspective, sequential extension is a generalized extension relation.

**Proposition 6.2.2 (Extension $\Longrightarrow$ Sequential Extension)** *Extension is a special kind of sequential extension, but not vice versa.*

*Proof:* It directly follows the definitions of extension and sequential extension, and the illustrative examples of sequential extension above. □

**Proposition 6.2.3 (Mutual Sequential Extension = Testing Equivalence)** *Let P and Q be processes. Suppose that the processes P and Q mutually extend themselves sequentially. The process P should be testing equivalent to the process Q.*

*Proof:* For the complete proof, please refer to the proof of anti-symmetric property in Proposition 6.2.1. It shows that if they mutually extend themselves, then they should be equivalent in the failures-divergences model. The situation is exactly the same in testing equivalence. □

**Example 6.2.3 (Extension as a special case of sequential extension)** Refer to the processes in Example 6.2.1. There is no extension relation between the fifth one and processes (2) to (4). For instance, the sequence $\langle\surd\rangle$ is a trace of any of these 3 processes, but not a trace of the fifth. ■

These examples also illustrate that sequential extension is not a special case of conformance relation (Definition 4.1.2). Obviously, reduction relation is not a special case of sequential case and, hence, conformance relation is not a special case of sequential extension either.

## 6.3 σ-Extension

Although sequential extension can solve Problem Prefix, it does not solve the problems arisen from extension associated with the sequential operator. We observe that the main reason for the failures non-compliance in the sequential composition (Problems Preceding and Succeeding) is the fact that an extending process may introduce undesirable non-determinism in sequential composition. Problem Prefix and Problem Decomposition further illustrate that we have to "control" the kind of process after the processes being extended. Otherwise, it may even introduce extra traces.

These observations are important. This is because extra traces in the abstraction (as the case in anti-reduction) will render test case generation unreliable. We shall further justify this claim with an example later. In fact, we actually want this special case "converging" to a process that fulfills the sequential extension property. Consequently, we introduce the concept of σ-extension.[2]

**Definition 6.3.1 (σ-Extension)** *A process $P$ σ-extends a process $Q$, denoted by $Q \sqsubseteq_\sigma P$, iff*

I. *the process $P$ extends the process $Q$ sequentially, and*

II. *any pair of successful complete traces of the process $Q$ will cause the process $P$ to behave equivalently ($\forall s\hat{}\langle\surd\rangle, t\hat{}\langle\surd\rangle \in CT^{\surd}(Q).\ P/s = P/t$).*

**Example 6.3.1 (σ-Extension example 1)** (Figure 6.3 shows the processes in this example.) The Process *Inspection_Alignment*;*Ranking*, discussed in the section for Problem Preceding, can be anti-extended sequentially to form the process *IAR*.

$$\begin{aligned} IAR &= capture \rightarrow inspect \rightarrow align \\ &\quad \rightarrow ((rank \rightarrow STOP) \sqcap (discard \rightarrow STOP) \sqcap (inspect \rightarrow align \rightarrow SKIP)) \end{aligned}$$

It is abstracted from the process *Inspection_Alignment*;*Ranking* by replacing the process $discard \rightarrow SKIP$ by the process $dicard \rightarrow STOP$, and removing some branch of inspections after the first alignment. This process can be further abstracted into the process *IAR*2.

$$IAR2 = capture \rightarrow inspect \rightarrow align \rightarrow SKIP$$

[2] We have chosen to annotate our relation with the symbol "σ" because "σνμβολη" in Greek means convergence.

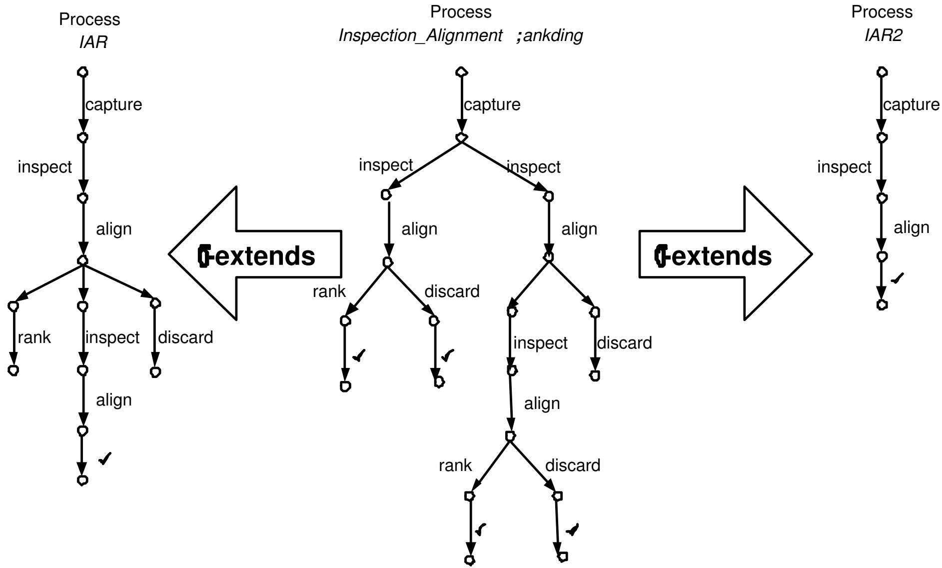


Figure 6.3: Example 1 of σ-Extension.

We observe that the set $CT^{\surd}(IAR)$ contains one trace.

$$CT^{\surd}(IAR) = \{\langle capture, inspect, align, inspect, align, \surd \rangle\}$$

Since it is a singleton, Condition (II) of σ-extension should be satisfied automatically. The situation for the process *IAR*2 is similar. ■

**Example 6.3.2 (σ-Extension example 2)** (Figure 6.4 illustrates another example.) The process *PickArm* and their anti-σ-extensions illustrate the situation with multiple successful complete traces causing the process *PickArm* to behave like non-equivalent processes. The process *PickArm*, "stealing" time, always captures a photo even if the camera is blocked by some object. If the image is blocked, it requires the pick head to *move in* to unblock the vision system. Then, the process repeats itself. On the other hand, if the captured image is not blocked (or in other words, it successfully steals time!), the inspection job will start. It may, sometimes, require a further die alignment so that the pick head can pick the die up firmly, or sometimes requires the pick head to *move in*. In the latter case, it will decide whether the picking process can continue. Chances are, either there is a pick head movement error (in which case it will *alert* an operator and start the *UserService* process), or it decides to skip the processing of the present die, proceed to the *next die* and repeat the picking process. Alternatively, after the aforementioned die alignment, the pick head will *move down*. It may sometimes encounter errors, in which case the pick head needs to *move up* again, *move in* to shelter itself, and then *alert* an operator to start the *UserService* process. Otherwise, it will start the vacuum pump to *suck* up the die. After detecting a firm hold, it will *move up*, and the pick arm will transfer the die by *moving to the ranking bin*, and then *dropping* the die if the pick arm reaches the required destination. Finally, the arm will *move back* to its home position, and the cycle repeats itself. The process *UserService* simply fixes the error (that is, do *service*) and then *restarts*. Finally,

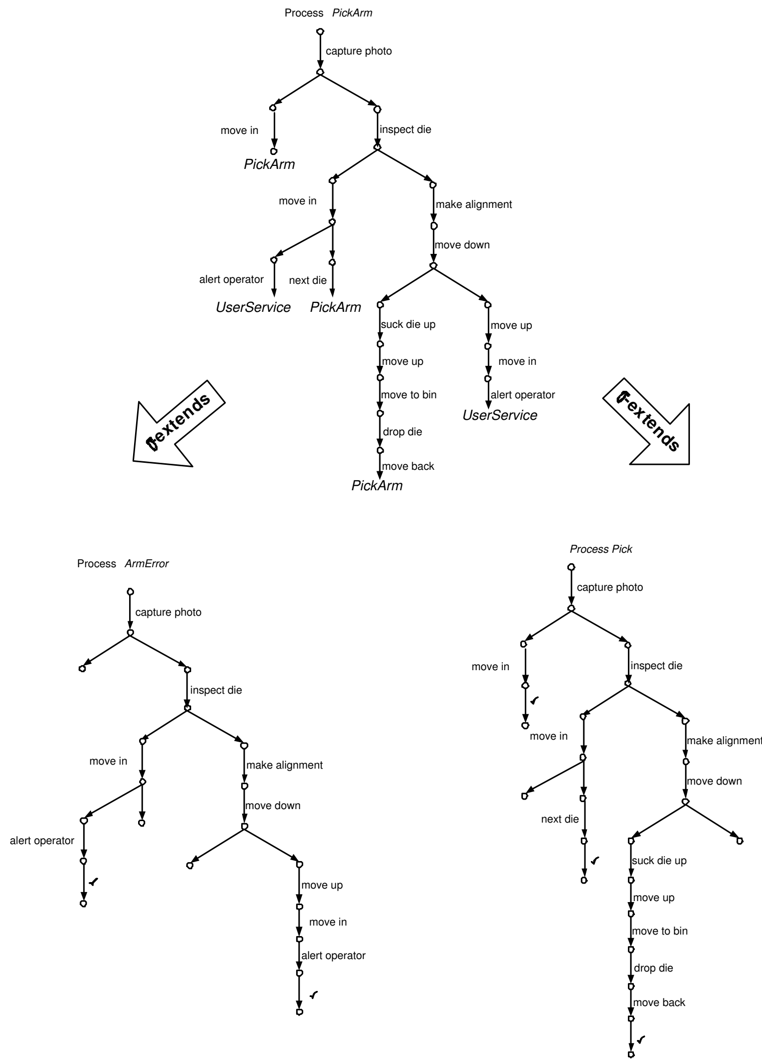


Figure 6.4: Example 2 of σ-Extnesion

it terminates successfully.

$$UserService = service \rightarrow restart \rightarrow SKIP$$

$$\begin{aligned}
PickArm = capture\ photo \rightarrow (((move\ in \rightarrow PickArm)\\
\sqcap(inspect\ die \rightarrow (move\ in \rightarrow ((alert\ operator \rightarrow UserService)\\
\sqcap(next\ die \rightarrow PickArm))\\
\sqcap(make\ alignment \rightarrow move\ down\\
\rightarrow ((move\ up \rightarrow move\ in \rightarrow alert\ operator \rightarrow UserService)\\
\sqcap(suck\ die\ up \rightarrow move\ up \rightarrow move\ to\ bin\\
\rightarrow drop\ die \rightarrow move\ back \rightarrow PickArm)))))
\end{aligned}$$

The process *Pick* is an anti-σ-extension of the process *PickArm*. It will eventually behave like the process *PickArm* again if it composes with the process *PickArm* sequentially. Informally, it syntactically turns the recursion identifiers of the process *PickArm* into the process *SKIP*, and prunes out those actions that may result in the process *UserService*.

$$
\begin{aligned}
Pick = {} & capture\ photo \rightarrow (((move\ in \rightarrow SKIP) \\
& \sqcap(inspect\ die \rightarrow (move\ in \rightarrow (STOP \sqcap (next\ die \rightarrow SKIP) \\
& \qquad \sqcap(make\ alignment \rightarrow move\ down \\
& \qquad \rightarrow (STOP \sqcap (suck\ die\ up \rightarrow move\ up \rightarrow move\ to\ bin \\
& \qquad\qquad \rightarrow drop\ die \rightarrow move\ back \rightarrow SKIP)))))
\end{aligned}
$$

It is easy to observe that the set of successful complete traces for the process *Pick* is as follows:

$$
\begin{aligned}
& CT^{\surd}(Pick) = \\
& \left\{ \begin{array}{l}
\langle capture\ photo, move\ in, \surd\rangle, \\
\langle capture\ photo, inspect\ die, move\ in, next\ die, \surd\rangle, \\
\langle capture\ photo, inspect\ die, make\ alignment, move\ down, suck\ die\ up, \\
\qquad move\ up, move\ to\ bin, move\ back, \surd\rangle
\end{array} \right\}
\end{aligned}
$$

Moreover, it is also easy to check that the process *PickArm* will "converge" to processes that have equivalent behaviours.

$$
\begin{aligned}
& PickArm/\langle capture\ photo, move\ in\rangle = PickArm \\
& PickArm/\langle capture\ photo, inspect\ die, move\ in, next\ die\rangle = PickArm \\
& PickArm/\langle capture\ photo, inspect\ die, make\ alignment, move\ down, suck\ die\ up, \\
& \qquad move\ up, move\ to\ bin, move\ back\rangle = PickArm
\end{aligned}
$$

Similarly, the process *PickArm* σ-extends the process *ArmError* defined below. When the process *PickArm* follows any trace, say $t\hat{}\langle\surd\rangle$, of the successful completed traces of the process *ArmError*, the process *PickArm* should always evolve into the process *UserService* right before the successful termination symbol $\surd$ is reached (that is, $PickArm/t\hat{}\langle\surd\rangle = UserService$).

$$
\begin{aligned}
ArmError = {} & capture\ photo \rightarrow ((STOP \sqcap (inspect\ die \rightarrow (move\ in \\
& \qquad\qquad \rightarrow ((alert\ operator \rightarrow UserService) \sqcap STOP) \\
& \sqcap(make\ alignment \rightarrow move\ down \\
& \qquad \rightarrow ((move\ up \rightarrow move\ in \rightarrow alert\ operator \rightarrow UserService) \sqcap STOP)))
\end{aligned}
$$

Informally, the process *PickArm* σ-extends the processes *Pick* and *ArmError*. ■

Since this converging process behaves like a σ-extended process after any successful trace of a process being σ-extended and, at the same time, it should behave compatibly with the σ-extended process, it follows that we may define this kind of "converging" processes using the descriptions of the latter two processes.

By exhaustion, a trace, say $tr$, in a trace set of a process $P$ can only fall into the following situations:

I. $tr$ is a divergent trace

II. $tr$ is a trace terminated with a $\surd$ event (that is, a successful complete trace)

III. $tr$ is a trace terminated without a $\surd$ event (that is, a complete but not successful trace)

IV. $tr$ is an infinite trace

V. $tr$ is a prefix of some trace in the above cases.

**Discussions** A complementary view is that not all processes have successful complete traces. The following are some examples.

I. The process $Ejector$ σ-extends the process $EJ$ (on page 32). Since, the process $EJ$ never stops, it has no complete trace.

II. The process $STOP$ stops immediately. By definition, the process $Ejector$ also σ-extends the process $STOP$. Nevertheless, the process $STOP$ has no successful complete trace.

III. The process *CHAOS* can only be σ-extended by itself[3]. It has no complete trace.

In CSP, if they (as preceding processes) are composed sequentially with some processes (as succeeding processes), the sequential composition will be the processes themselves (page 20). In other words, the sequential composition can be considered as having no effect on them. On the other hand, suppose that a process $P$ may terminate successfully. There is a CSP law,

$$P = P;SKIP$$

, which guarantees process equivalence. ♦

Only case (II) will constitute to the successful complete trace set. Consequently, we define the process $Q/\triangle_P$ to be the converging process as follows:

**Definition 6.3.2 (Representative trace for σ-extension)** *Suppose that a process P σ-extends a process Q. A representative trace (for σ-extension) of the process Q with respect to the process P, denoted by $t_{rep(Q)_P}$, is a successful complete trace of the process Q.*

**Example 6.3.3 (Representative trace for σ-extension example)**

| | |
|---|---|
| $\langle capture\ photo, move\ in, \surd \rangle$ | for the process *Pick* |
| $\langle capture\ photo, inspect\ die, move\ in, alert\ operator, \surd \rangle$ | for the process $ArmError$ |

Let us use the processes in Example 6.3.2 as an illustration. A representative trace for σ-extension for the processes *Pick* with respect to the process *PickArm* is to capture a photo, move the camera into the right position and then terminate successfully. Similarly, one for the process $ArmError$ with respect to the process *PickArm* is to capture a photo, inspect a die, move the camera into the right position, alert operators and then terminate successfully. ■

**Definition 6.3.3 (Converging process)** *Let P and Q be processes. A converging process of the process P with respect to the process Q, denoted by $P/\sqcap_Q$, is the ⊓-composition for all processes defined as follows: The process should be one such that the process P may behave after some traces. The traces, concatenated with $\langle \surd \rangle$, should be successful complete traces of*

[3] In CSP, the process *CHAOS* is the bottom process.

*the process Q. If there is no such trace to be defined, this converging process will behave like the process SKIP.*

$$P/\sqcap_Q = \begin{cases} \underset{t\hat{\ }\langle\surd\rangle \in CT^{\surd}(Q)}{\sqcap} P/t & \textit{if } CT^{\surd}(Q) \textit{ is non-empty} \\ \\ SKIP & otherwise \end{cases}$$

**Discussions** The use of the process *SKIP* in the "otherwise" condition above respects the notion of process composition in CSP. Suppose that a process $Q$ has no successful complete trace. By CSP laws, $Q$ should be equivalent to $Q;STOP$. Moreover, a standard CSP law states that $STOP$ should be equivalent to $STOP;SKIP$. Consequently, $Q$ should behave like $(Q;STOP);SKIP$. ♦

Definition 6.3.3 can be further simplfied when $P$ σ-extends $Q$. By Condition II of Definition 6.3.1, the processes $P/t_1$ and $P/t_2$ should be equivalent, whenever $t_1\hat{\ }\langle\surd\rangle$ and $t_2\hat{\ }\langle\surd\rangle$ are successful complete traces of the process $Q$. Moreover, by the CSP law "$R = R \sqcap R$", the process $P/\sqcap_Q$ can be expressed as $P$ after a representative trace for σ-extension. The above definition for converging processes can be simplified to a simpler one as follows: Informally, whatever the internal path chosen to be executed in the process $Q$, it will always ensure the same result if it can terminate successfully. This resultant behaviour is exactly the same as the process $P$ after executing any one of its representative traces for σ-extension with respect to the process $Q$.

**Definition 6.3.4 (Converging process for σ-extension)** *Let P and Q be processes. Suppose* $P/\triangle_Q$ *is a converging process of the process P with respect to the process Q. The converging process for the σ-extension* $Q \sqsubseteq_\sigma P$ *is a converging process in which all branches of* $\sqcap$ *(in the definition of converging process above) should be equivalent.*

$$P/\triangle_Q = \begin{cases} P/u & \textit{if } u\hat{\ }\langle\surd\rangle = t_{rep(Q)_P} \textit{ and } t_{rep(Q)_P} \in CT^{\surd}(Q) \\ \\ SKIP & otherwise \end{cases}$$

**Example 6.3.4 (Converging process for σ-extension example)** Using the example in Figure 6.4, the converging processes for the process *PickArm* with respect to processes *Pick* and *ArmError* are the processes *PickArm* and *UserService* respectively.

$$\begin{aligned} PickArm/\triangle_{Pick} &= PickArm/t_{rep(Pick)_{PickArm}} \\ &= PickArm/\langle capture\ photo, move\ in\rangle \\ &= PickArm \end{aligned}$$

$$\begin{aligned} PickArm/\triangle_{ArmError} &= PickArm/t_{rep(Pick)_{PickArm}} \\ &= PickArm/\langle capture\ photo, inspect\ die, move\ in, alert\ operator\rangle \\ &= UserService \end{aligned}$$

■

**Lemma 6.3.1** *Let P and Q be processes. Suppose that the process P σ-extends the process Q. Then, the converging process for this σ-extension cannot behave like the process STOP.*

*Proof:* According to Definition 6.3.4, a trace (say $u$) of the converging process (that is, the $\sqcap$-composition) should be one of the following two cases.

First, it is a trace from the process $SKIP$. It can happen when there is no successful complete trace for the process $Q$. It is obvious that processes $SKIP$ and $STOP$ are not equivalent.

Secondly, it (trace $u$) is the suffix part of certain trace, (say $t$), of the process $P$. Such a trace $t$ should be subject to a constraint that its corresponding prefix part (say $s$) should become a successful complete trace of the process $Q$ providing that the prefix part $s$ is appended with the successful termination symbol $\surd$. (In other words, $s\hat{}\langle\surd\rangle \in traces(Q)$ and $s\hat{}u = t$).

Hence, according to Conditions III and IV of Sequential Extension[4], if $Q$ terminates successfully after a trace $s$, the process $P/s$ should either terminate successfully or come to no immediate deadlock. In either case, the behaviour of the process $P$ after the trace $s$ cannot be equivalent to the process $STOP$.

By combining the two cases, the result follows. □

**Discussions** The successful termination of intermediate modules (that is, processes) in sequential composition is not observable, and in the extreme case, undecidable. Consider the following standard CSP laws and definitions:

$$SKIP;STOP = STOP = STOP;STOP = STOP;SKIP$$
$$SKIP = \surd \rightarrow STOP$$

It is easy to observe that the process $SKIP$ σ-extends the process $STOP$, but not vice versa. In the above composition, however, if these two processes are composed with the process $STOP$ in a sequential manner as shown, they will behave equivalently as the process $STOP$. This is because, unlike other events, the successful termination event $\surd$ of the preceding process is hidden by the sequential composition operator (;). Since it is hidden in the sequential composition, there is no way to tell whether the preceding process actually terminates successfully or not.

I. If we reject the σ-extension between the process $SKIP$ in composition $SKIP;STOP$ and another process $STOP$, then by the CSP laws above, we should reject the σ-extension between the process $STOP$ in composition $SKIP;STOP$ and the sole process $STOP$.

II. If we accept the σ-relation between the preceding process $STOP$ in composition $STOP;STOP$ and another process $STOP$, then by CSP laws, we must accept the σ-extension between the process $SKIP$ in the composition $SKIP;STOP$ and the sole process $STOP$.

However, we would like to preserve the order between the processes $SKIP$ and $STOP$ in σ-extension. We observe that the anomaly can be resolved if the succeeding process is restricted to a behaviour not equivalent to the process $STOP$. Informally, this is because, in that case, the event in the succeeding process can be observed and, hence, it is feasible to tell whether or not the proceeding process reaches a successful termination[5].

In this way, does it mean that we are restricted only to deadlock-free processes? The answer is negative. This is because, once the σ-extension is established, we may anti-extend the

[4] Σ-extension is a special kind of Sequential Extension.

[5] By definition, the sequential composition is a binary operator.

process *SKIP* (which is a succeeding process) into the process *STOP* (by Theorem 6.4.1 to be introduced later). Hence, the process in question will be transformed into a deadlock-secured process. Applying the definition, the converging process $STOP/\sqcap_{SKIP}$ should be defined as *STOP* (as expected!). By eliminating this irregularity, the preceding process (that is, *SKIP* or *STOP*) will respect the trace constraints of the reference process (that is, *SKIP*) to determine σ-extension. We shall use this observation in Lemma 6.4.2 (Sufficiency Lemma) to decompose processes from a sequential composition. In σ-extension, the process *STOP* does not σ-extend the process *SKIP* and so $STOP/\vartriangle_{SKIP}$ should not be applied. Alternatively, by insisting on the existence of the process $P/\vartriangle_Q$ for the σ-extension relation between some process $P$ and $Q$, Lemma 6.3.1 ensures that it will not behave like the process *STOP*. In other words, we only allow (1) a "poorer" process to terminate (successfully) whenever the better process terminates successfully and (2) a "poorer" process to be forced to terminate whenever the better process should be terminated. ♦

## 6.4 Sequential Constraints

We are now in the position to show the important results. Informally, σ-extension ($Q \sqsubseteq_\sigma P$) ensures all non-successful complete traces of the process $Q$ are traces of the process $P$. Moreover, if it is composed sequentially with a process (such as the process $Q$ is followed by the process $R$), all successful complete traces will intuitively "evaporate". The process $Q$ will transfer control to the process $R$. Incidentally, if the process $R$ is a converging process for the σ-extension, it will guarantee that all jointed traces should be traces of the process $P$. A similar intuition can be applied to failures and divergences. We introduce this necessary property.

**Lemma 6.4.1 (Necessity)** *Let P and Q be processes. Suppose that a process P σ-extends a process Q. Then, P extends process* $Q;P/\vartriangle_Q$.

*Proof:* Consider a failure $(s,X)$ of the process $P$.

I. First, suppose that $s$ is also a trace of the process $Q$. Since the process $P$ σ-extends the process $Q$, by definition, $(s,X)$ should be a failure of the process $Q$, unless the process $P$ cannot terminate successfully after $s$ while the process $Q$ can. However, the exceptional case means that $s\hat{}\langle\surd\rangle$ should be a trace of the process $Q$, and in sequential composition with the process $P/\vartriangle_Q$, it implies that the failures associated with trace $s$ in $Q;P/\vartriangle_Q$ should include those from the process process $P/s$ (see the failure definition of sequential composition on page 20). Hence, $(s,X)$ should be a failure of the process $Q;P/\vartriangle_Q$.

II. Secondly, suppose that $t$ ($< s$) is a trace of the process $Q$. Suppose that sequence $s$ is a trace of the process $Q;P/\vartriangle_Q$. We can find a sequence $u$ which appends to sequence $t$ to form sequence $s$. If $t\hat{}u$ is a trace of the process $Q$, then it will fall within the first case discussed above. Hence, without loss of generality, $u$ is a composite trace of the process $P/\vartriangle_Q$, where a part of it is from the process $Q$ and the subsequent part is from the process $P/\vartriangle_Q$. By definition, $u$ is a trace of the process $P/t$; otherwise the process $P$ cannot σ-extend the process $Q$. In other words, it is a failure of the process $P$ itself. Thus, $(s,X)$ should be a failure of the process $Q;P/\vartriangle_Q$.

As for the traces, suppose $s$ is not a successful complete trace of the process $Q$. By Condition II and the trace definition of sequential composition, it should be a trace of the process $Q;P/\vartriangle_Q$.

Otherwise, the process $Q$ will pass control over to the process $Q;P/\triangle_Q$, which should behave equivalently as the process $P$ after some representative traces for σ-extension. Hence, in either case, $s$ should be a trace of the process $Q;P/\triangle_Q$. The divergences condition follows directly from the definitions. □

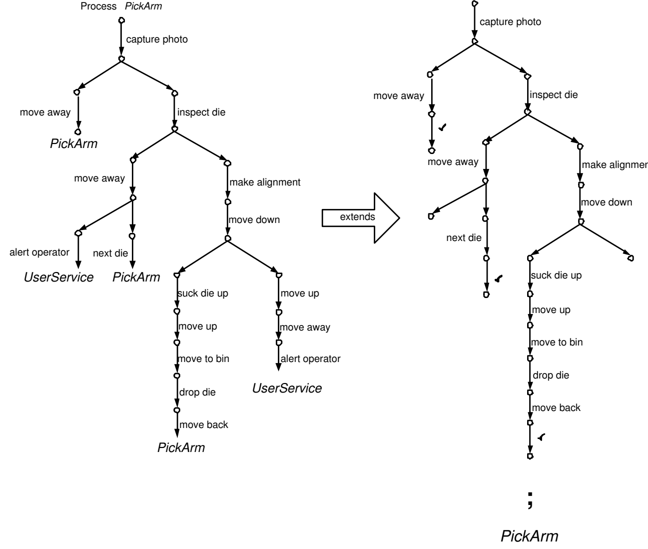


Figure 6.5: σ-Extension implies overall extension

**Example 6.4.1 (σ-Extension + Converging process $\Longrightarrow$ Extension example)**
Figure 6.5 illustrates the effect of σ-extension and converging process for the Example 6.3.2. As shown in Example 6.3.4, the converging process for the process *PickArm* can be the process *PickArm* or the process *UserService*. Suppose that we are interested in the σ-extension between the process *PickArm* and the process *Pick*. As we have illustrated in Example 6.3.4, the converging process $PickArm/\triangle_{Pick}$ for the process *PickArm* with respect to the process *Pick* is the process *PickArm* itself. Consider the sequential composition

$$Pick;PickArm$$

Let us focus on the process *Pick* in the first place. All the traces of the process *Pick* except those that terminate with a successful termination event ($\surd$) are traces of the process *PickArm*. Similarly, all the failures associated with the traces of the process *PickArm* are failures of the process *Pick*. Moreover, after any successful complete trace, such as $tr\hat{\ }\langle\surd\rangle$, the process *Pick* will then pass control to the process *PickArm* in the sequential composition. Following the same trace $tr$, the process *PickArm* actually repeats. It is easy to perceive that the process *PickArm* extends the process $Pick;PickArm$. ■

**Theorem 6.4.1 (Tail anti-extension)** *Let P, Q, R and S be processes. Suppose that*

*I. the process P σ-extends the process Q,*

*II. the process P extends the process Q;R,*

*III. the process R anti-extends the process $P/\triangle_Q$ and*

*IV. the process S anti-extends the process R.*

*Then, the process P extends the process Q;S.*

*Proof:* The process $R$ respects the failures of the process $S$ and, hence, if the process $P$ respects the failures of the process $R$, it should also respect those from $S$.

Moreover, following the same proof as in Lemma 6.4.1, and since extension preserves failures, the removal of traces from the process $R$ to form an anti-extended process $S$ will not affect the failure preservation. σ-extension is not affected by the extension relation between $R$ and $S$ and hence the result follows. The traces and divergences conditions are simple and follow the definitions directly. □

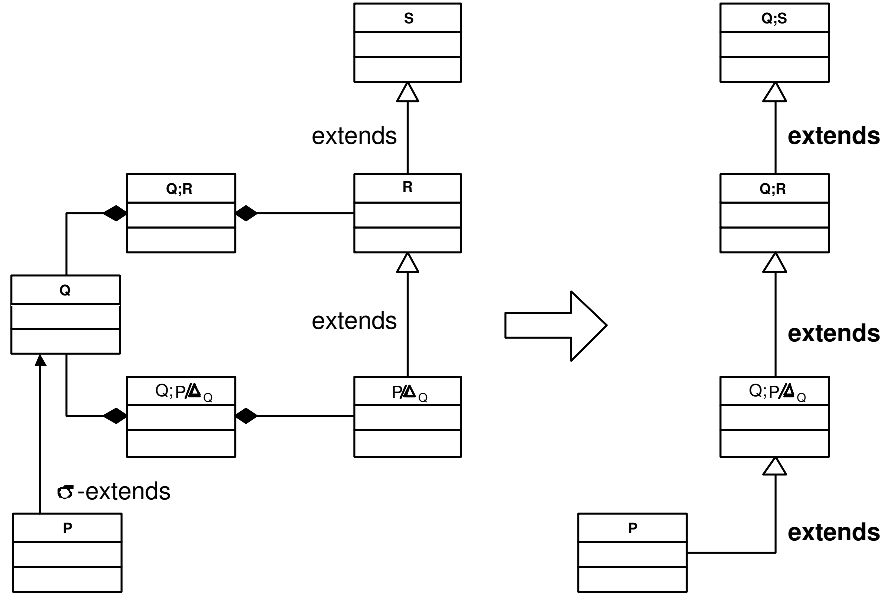


Figure 6.6: Preservation of overall extension by tail anti-extension.

Figure 6.6 illustrates the result of the tail anti-extension. Informally, it shows that if a specification (such as the process $P$) maintains a σ-extension with its abstraction (such as the process $Q$), it will maintain the overall extension to any anti-extended abstraction (such as a process $R$) of its tail (that is, the converging process $P/\triangle_Q$).

**Example 6.4.2 (Tail anti-extension example)** Further to Example 6.4.1, which shows that the process *PickArm* should extend the process *Pick*;*PickArm*,

*PickArm* extends *Pick*;*PickArm*

we observe that

$capture\ photo \rightarrow STOP$ anti-extends $Inspection_r$ and

$Inspection_r$ anti-extends$PickArm$

where

$Inspection_r = capture\ photo \rightarrow (inspect\ die \rightarrow STOP)\Box(move\ in \rightarrow SKIP)$

. By the Tail Anti-Extension Theorem, we may *anti-extend* the succeeding process (that is, the process *PickArm*) in the sequential composition above by the process $capture\ photo \rightarrow STOP$ to form the following sequential composition, which anti-extends the process *PickArm*.

$$PickArm \text{ extends } Pick; capture\ photo \rightarrow STOP$$

■

Another example is the extension hierarchy as shown in Figure 6.11 for Example 6.4.5.

We can generalize Lemma 6.4.1. Suppose that a process $Q$ is better[6] than a process $R$. Moreover, suppose we know that the process $Q$ will form a sequential composition with a process $S$. We may then replace $Q$ by its less perfect version $R$ in the sequential composition $Q;S$ to form the process $R;S$. Intuitively, if a process $P$ is even better than the process $Q;S$, it should be better than the process $R;S$. In other words, we may replace the proceding process of a sequential composition by another process.

**Theorem 6.4.2 (Head anti-extension)** *Let P, Q, R and S be processes. Suppose that*

*I. the process $P$ σ-extends the process $Q$,*

*II. the process $S$ anti-extends the process $P/\triangle_Q$ and*

*III. the process $R$ anti-extends a process $Q$.*

*Then, the process $P$ σ-extends the process $R$, and extends the process $R;S$.*

*Proof:* First, we show that the process $P$ σ-extends the process $R$. By definition, we need to show that there is a sequential extension relation between the processes $P$ and $R$, and $R$ will cause the process $P$ to behave like some equivalent process, no matter how it may terminate successfully.

Since sequential extension is a partial order (Proposition 6.2.1), and extension is a special kind of sequential extension (Proposition 6.2.2), the process $P$ should extend the process $R$ sequentially. Moreover, all the traces of the process $R$ should be traces of the process $Q$, and so the complete trace $CT^{\surd}(R)$ should be a subset of the complete trace set $CT^{\surd}(Q)$. Given that the process $P$ σ-extends the process $Q$, any pair of traces from the successful complete trace $CT^{\surd}(Q)$ will cause the process $P$ to behave equivalently. Hence, the process $P$ should σ-extends the process $R$.

Now, we are going to show that the process $P$ extends the process $R;S$ for a process $S$ anti-extending the process $P/\triangle_Q$. Given that the process $S$ anti-extends the process $P/\triangle_Q$,

[6] In the sense of failure inclusion.

by Theorem 6.4.1, the process *P* should extend the process *Q*;*S*. By showing that the process *Q*;*S* extends the process *R*;*S*, we can conclude that *P* extends *R*;*S*, since the extension relation is transitive. The trace inclusion is simple. Suppose that *s* is a trace of the process *R*;*S*. We have two sub-cases, namely that *s* is a trace from the process *R* alone, or a successful complete trace of the process *R* followed by a trace of the process *S*. The first sub-case is trivial as this trace should then be a trace of the process *Q* and hence a trace of the process *Q*;*S*. For the second sub-case, since the process *Q* extends the process *R*, any successful complete trace of the process *R* should be a successful complete trace of the process *Q*. Hence, it should be a trace of the process *Q*;*S*. Consequently, we can establish the trace condition. By the same token, a failure associated with the trace *s* for the process *Q*;*S* should be a failure of the process *Q* alone or a failure in which the refusals come from the process *S*. Following the same argument as in Theorem 6.4.1, the process *R* should be even poorer than the process *Q* in the failure sense[7] and by keeping process *S* as the succeeding process to form the target sequential process, the process *P* should respect the process *R*;*S*. The divergence condition is trivial. Thus, the result follows. □

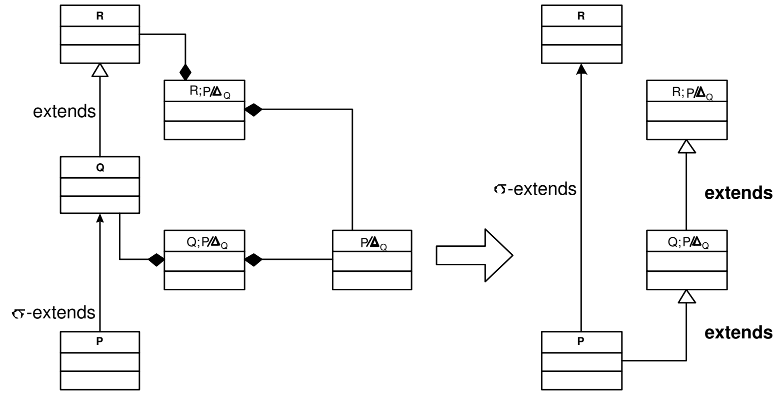


Figure 6.7: Preservation of overall extension by head anti-extension.

Figure 6.7 shows the theorem diagrammatically. The processes *S* and *R* have been set to be equivalent to simplify the illustration.

**Example 6.4.3 (Head anti-extension example)** We have illustrated in Example 5.4.2 that the process *RISE*2 is not a proper decomposition from the process *EJ*2. This is because it may introduce unexpected traces in the sequential composition. We may limit the ability of the process *RISE*2 by disallowing it from terminating successfully and immediately after the maintenance call. The process *RISE*3 is an example of such restriction. We form the process *EJ*5, which is similar to the process *EJ*3 in Example 5.4.2. We observe that the process *EJ*2 σ-extends the process *EJ*3, and the converging process is the original succeeding

[7] σ-extension is a kind of sequential extension which is transitive, so that *P* should respect *R*.

process $(moveDown \rightarrow EJ2)$ in Example 5.4.2.

$$\begin{aligned} EJ2 &= ((moveUp \rightarrow SKIP);(moveDown \rightarrow EJ2)) \\ &\quad \Box(maintain \rightarrow home \rightarrow SKIP) \\ RISE3 &= (moveUp \rightarrow SKIP)\Box(maintain \rightarrow STOP) \\ EJ5 &= RISE3;(moveDown \rightarrow EJ2) \end{aligned}$$

It is ease to observe that the process $EJ2$ extends the process $EJ5$.

$$EJ2 \text{ extends } EJ5$$

Now, we are able to abstract the process $EJ2$ into two processes in sequential composition. Obviously, if we would like to focus on testing the sequences of movement operations, we may temporarily not consider the maintenance service call. In this connection, the process $RISE3$ can be further abstracted into the process $RISE4$.

$$RISE4 = moveUp \rightarrow SKIP \text{ anti-extends } RISE3$$

By the Head Anti-Extension Theorem,

$$EJ2 \text{ extends } EJ6 = RISE4;(moveDown \rightarrow EJ2)$$

Figure 6.8 shows these processes and their relationships. ■

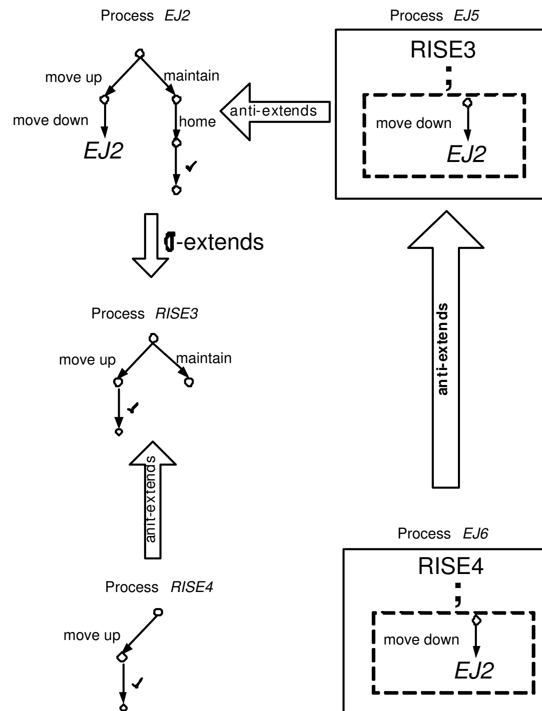


Figure 6.8: An example of head anti-extension preserving overall extension

**Theorem 6.4.3 (Serial anti-extension)** *Let P, Q, R, S and T be processes. Suppose that*

I. *the process P and R σ-extends the processes Q and S, respectively, and*

II. *the processes R and T anti-extends the processes $P/\triangle_Q$ and $R/\triangle_S$, respectively.*

*Then, the process P extends the process $Q;S;T$.*

*Proof:* By Theorem 6.4.1, the process $R$ extends the process $S;T$. Owing to the transitivity of extension, the process $P/\triangle_Q$ should then extend the process $S;T$. By re-applying Theorem 6.4.1 again, the result follows. □

**Example 6.4.4 (Serial anti-extension example)** Let us refer to the processes *PickArm*, *Pick* and *ArmError* in Example 6.4.1. It is known that the process *PickArm* σ-extends both the processes *Pick* and *ArmError*. Moreover, the converging process for the process *PickArm* with respect to the process *Pick* is the process *PickArm* itself, and that with respect to the process *ArmError* is the process *UserService*. Hence, we have the following two extension relationships:

$$PickArm \text{ extends } Pick;PickArm \text{ and} \tag{6.3}$$

$$PickArm \text{ extends } ArmError;UserService \tag{6.4}$$

By the Serial Anti-Extension Theorem, it follows that we may serialize the process *PickArm* repeatedly by abstracting it as the process *Pick* (using Equation (6.3)), and then composed with the process *ArmError*;*UserService* (using Equation (6.4)).

$$PickArm \text{ extends } \overbrace{Pick;\cdot;Pick}^{n};ArmError;UserService,$$

We observe that the process *UserService* extends the process *STOP*, so that

$$ArmError;UserService \text{ extends } ArmError;STOP.$$

By the Tail Anti-Extension Theorem,

$$PickArm \text{ extends } \overbrace{Pick;\cdot;Pick}^{n};ArmError;STOP.$$

■

**Example 6.4.5 (An example combining Head, Tail and Serial Anti-Extensions)**
Figure 6.9 shows the situation for the process *PickArm*. As discussed in Example 6.3.2 (Figure 6.4), there are two processes, namely *Pick* and *ArmError*, which are both σ-extended by the process *PickArm*. By Lemma 6.4.1, both the processes *Pick*;*PickArm* and *ArmError*;*UserService* should anti-extend the process *PickArm*.

**(Level 1)**

$$\begin{array}{l} PickArm \text{ extends } Pick;PickArm \quad \text{and} \\ PickArm \text{ extends } ArmError;UserService \end{array} \tag{6.5}$$

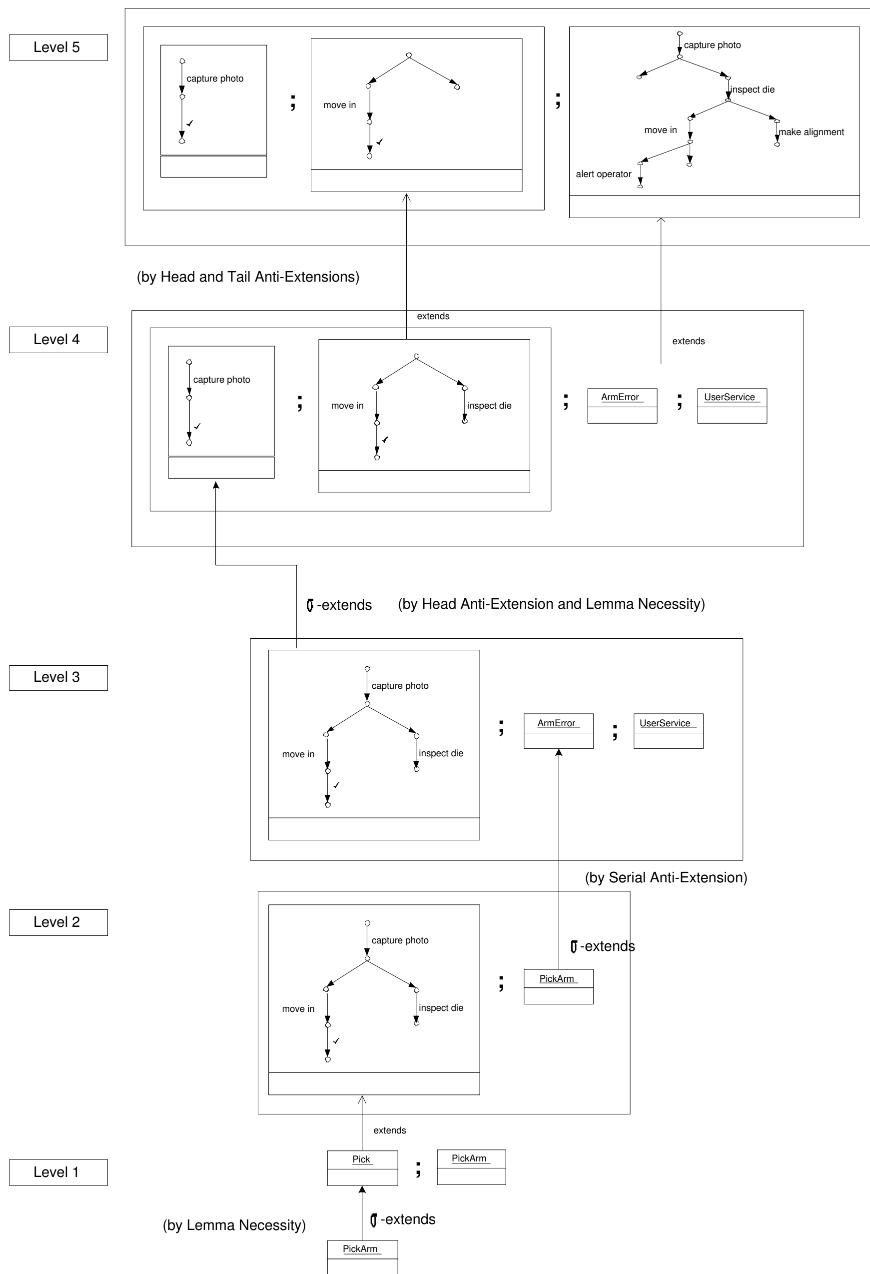


Figure 6.9: Compositional Anti-Extension Hierarchy by σ-Extension

Suppose that we anti-extend the process *Pick* into a simpler version, giving the process $SimplePick_1$.

$$SimplePick_1 = capture\ photo \rightarrow ((inspect\ die \rightarrow STOP) \sqcap (move\ in \rightarrow SKIP))$$

By the Head Anti-Extension Theorem (Theorem 6.4.2), the process *PickArm* should extend the sequential composition formed from the processes $SimplePick_1$ and *PickArm*.

**(Level 2)**

$$PickArm \text{ extends } SimplePick_1;PickArm \tag{6.6}$$

Moreover, the process *PickArm* should σ-extend the process $SimplePick_1$. Knowing that the process $ArmError;UserService$ anti-extends the process *PickArm*, by the Serial Anti-Extension Theorem, the process *PickArm* should extend the sequential composition formed from the processes *SimplePick* and $ArmError;UserService$.

**(Level 3)**

$$PickArm \text{ extends } SimplePick_1;ArmError;UserService \tag{6.7}$$

On the other hand, the process *SimplePick* may be treated as a standalone process, which can be decomposed sequentially. For instance, it may be decomposed into a process *SimpleCapture*, which captures photos, and a movement process *PickDecision*.

$$SimpleCapture = capture\ photo \rightarrow SKIP$$

$$PickDecision = (move\ in \rightarrow SKIP) \sqcap (inspect\ die \rightarrow STOP)$$

We observe that the process $SimplePick_1$ σ-extends the process *SimpleCapture* and its converging process is as follows.

$$SimplePick_1 / \triangle_{SimpleCapture} = ((inspect\ die \rightarrow STOP) \sqcap (move\ in \rightarrow SKIP))$$

Clearly, the process *PickDecision* is the process $SimplePick_1 / \triangle_{SimpleCapture}$. By Lemma 6.4.1, the process $SimplePick_1$ should extend the process $SimpleCapture;PickDecision$.

$$SimplePick_1 \text{ extends } SimpleCapture;PickDecision$$

By the Head Anti-Extension Theorem, the process *Pick* should extend a sequential composition which replaces the process *SimplePick* by the sequential composition $SimpleCapture;PickDecision$.

**(Level 4)**

$$Pick \text{ extends } (SimpleCapture;PickDecision);(ArmError;UserService) \tag{6.8}$$

In the same fashion, the process $ArmError;UserService$ can be anti-extended into the process *ErrorHandler*, which removes the unwanted "*make alignment*" event.

$$\begin{aligned} ErrorHandler = {} & capture\ photo \rightarrow STOP \\ & \sqcap (inspect\ die \rightarrow STOP \sqcap \\ & \quad move\ in \rightarrow alert\ operator \rightarrow service \rightarrow restart \rightarrow SKIP \end{aligned}$$

Likewise, the process *PickDecision* may discard the unwanted inspection to form the process *SimpleMove*.

$$PickDecision \text{ extends } SimpleMove$$
$$SimpleMove = ((inspect\ die \rightarrow STOP) \sqcap STOP)$$

Applying Tail Anti-Extension to the sequential composition *SimpleCapture*;*PickDecision*, we obtain

$$SimpleCapture;PickDecision \text{ extends } SimpleCapture;SimpleMove,$$

and by applying Head Anti-Extension again, we further obtain the following extension relationship:

$$(SimpleCapture;PickDecision);(ArmError;UserService)$$
$$\text{extends } (SimpleCapture;SimpleMove);(ArmError;UserService)$$

Finally, by applying Tail Anti-Extension to the above sequential composition, we can produce yet another version of the process *PickArm*.

**(Level 5)**

$$PickArm \text{ extends } (SimpleCapture;SimpleMove);ErrorHandler \tag{6.9}$$

■

**Example 6.4.6 (Test case construction from an abstraction tier)** We demonstrate how to extract test sequences from an extension hierarchy. Figure 6.10 shows the situation that test sequences can be extracted from the extension hierarchy proposed in Example 6.4.5. Suppose that we intend to generate test cases from Abstraction Level 3. That is, test cases are generated from the sequential composition below.

$$SimplePick_1;ArmError;UserService$$

Since sequential composition is associative (that is, $(P;Q);R = P;(Q;R)$), without loss of generality, let us rewrite it as

$$(SimplePick_1;ArmError);UserService.$$

We can compute test cases as follows:

I. Construct sets of traces for the processes $SimplePick_1$, *ArmError* and *UserService* based on some testing criteria on the processes. For instance, using the all-transition-coverage testing criteria, we can produce the following sets of traces:

A set of traces for the process *SimplePick*:

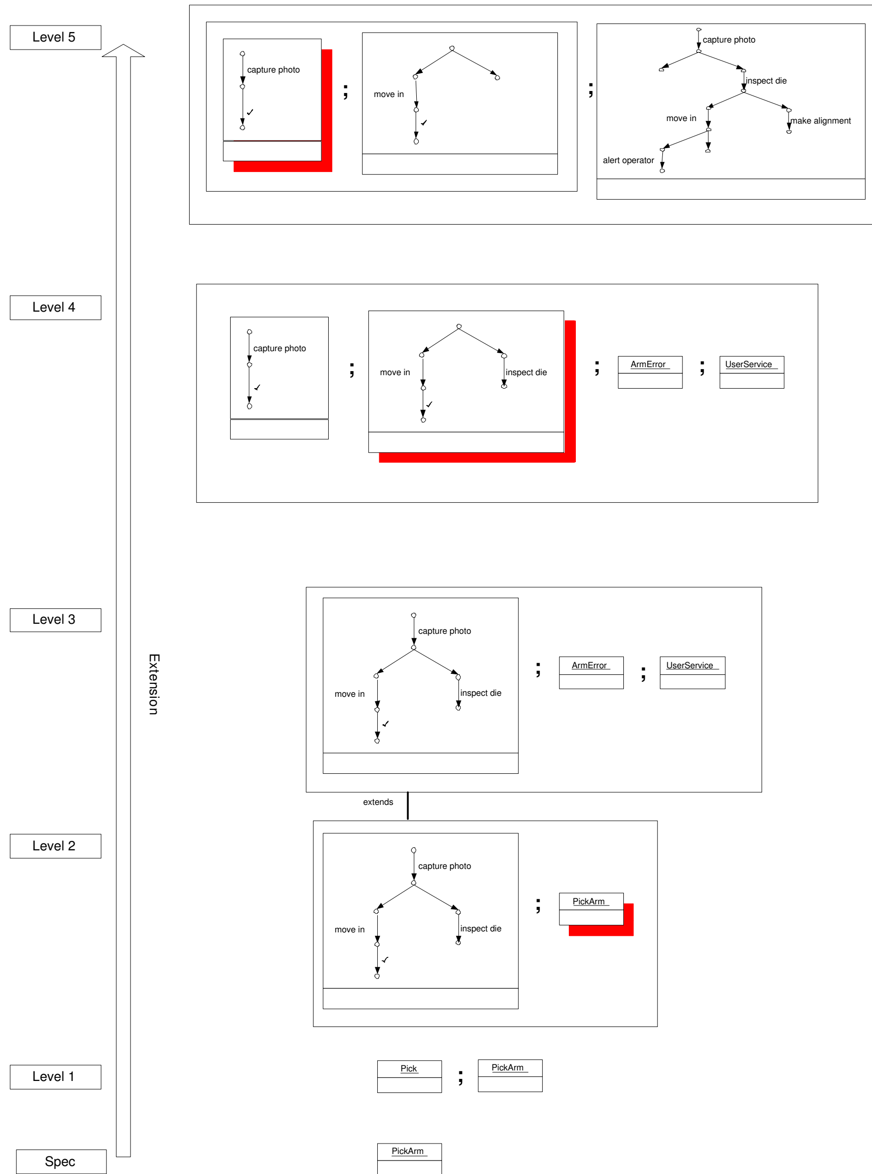


Figure 6.10: Sub-testcase extraction from various levels of abstraction

$$\left\{\begin{array}{l}\langle\rangle\\ \langle capture\ photo\rangle\\ \langle capture\ photo, move\ in\rangle\\ \langle capture\ photo, move\ in, \surd\rangle\\ \langle capture\ photo, inspect\ die\rangle\\ \langle capture\ photo, inspect\ die, \surd\rangle\end{array}\right\}$$

A set of traces for the process $ArmError$:

$$\left\{\begin{array}{l}\langle\rangle\\ \langle capture\ photo\rangle\\ \langle capture\ photo, inspect\ die\rangle\\ \langle capture\ photo, inspect\ die, move\ in\rangle\\ \langle capture\ photo, inspect\ die, move\ in, alert\ operator\rangle\\ \langle capture\ photo, inspect\ die, move\ in, alert\ operator, \surd\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment, move\ down\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment, move\ down, move\ up\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment, move\ down, move\ up,\\ \quad move\ in\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment, move\ down, move\ up,\\ \quad move\ in, alert\ operator\rangle\\ \langle capture\ photo, inspect\ die, make\ alignment, move\ down,\\ \quad move\ up, move\ in, alert\ operator, \surd\rangle\end{array}\right\}$$

A set for the process $UserService$:

$$\left\{\begin{array}{l}\langle\rangle\\ \langle service\rangle\\ \langle service, restart\rangle\\ \langle service, restart, \surd\rangle\end{array}\right\}$$

II. Construct the set of traces based on the CSP sequential composition operator for the process $SimplePick_1;ArmError$.[8]

$$\begin{array}{l}\langle capture\ photo, move\ in\rangle\\ \langle capture\ photo, inspect\ die\rangle\\ \langle capture\ photo, inspect\ die, capture\ photo, inspect\ die, move\ in, alert\ operator\rangle\\ \langle capture\ photo, inspect\ die, capture\ photo, inspect\ die, make\ alignment, move\ down,\\ \quad move\ up,\\ \quad move\ in, alert\ operator, \surd\rangle\end{array}$$

III. Construct a set of traces in which every trace is either a trace obtained Step (I), or constructed from appending a trace from the set obtained in Step (II) to a $\surd$-purged successful complete trace from the set obtained in Step (I). Further testing criteria or minimization techniques can be used. For instance, we may eliminate those traces which are prefixes of some other traces in the final set of traces.

[8] Minimization techniques can be applied to minimize the set of composed traces.

$\langle capture\ photo, move\ in \rangle$
$\langle capture\ photo, inspect\ die \rangle$
$\langle capture\ photo, inspect\ die, capture\ photo, inspect\ die, move\ in, alert\ operator \rangle$
$\langle capture\ photo, inspect\ die, capture\ photo, inspect\ die, make\ alignment, move\ down,$
$move\ up, move\ in, alert\ operator, service, restart, \checkmark \rangle$

IV. Hence, we construct a set of test cases for the process

$$SimplePick_1; ArmError; UserService.$$

By extension relation, they should be a set of test cases (that is, traces) for the process *PickArm*. ■

**Example 6.4.7 (Test cases construction from different levels of abstraction)** It is also feasible to generate test cases from different levels of abstraction. This is because the transitive nature of the extension relation allows the traces of a process at an abstract level to be the traces at lower levels of abstraction (that is, the extending processes). For instance, the figure (obtained from Example 6.4.5) shows that there are 5 levels of abstraction over the process *PickArm* denoting by levels 5, 4, 3, 2 and 1 from top to bottom. Suppose that the process *PickArm* is a specification and we are interested in constructing test cases from the processes in shadow.

I. Similarly to the last example, construct the sets of traces from the required processes (such as the processes in shadow in the figure) according to some testing criteria. For instance, the following are the traces of the shadowed processes at various abstraction levels:

| | |
|---|---|
| *SimpleCapture* at abstraction level 5: | $t_1 = \langle capture, \checkmark \rangle$ |
| *PickDecision* at abstraction level 4: | $t_{2_1} = \langle move\ in, \checkmark \rangle, t_{2_2} = \langle inspect\ die \rangle$ |
| *PickArm* at abstraction level 2: | $t_3 = \langle capture\ photo, inspect\ die, capture\ photo, inspect\ die, move\ in, alert\ operator \rangle$ |

II. Start from the most abstract layer (Abstraction Level 5). Since there is no test case set to compose with the trace $t_1$, it propagates downward to Abstraction Level 4 and we locates its corresponding process. For this particular case, it is also the process *SimpleCapture*. We also find a shadowed process, the process *PickDecision*, which can form a sequential composition with the process *SimpleCapture* in the given abstraction level in the hierarchy. Hence, composing these traces, we obtain the following two traces:

$$t_1 ⨟ t_{2_1} = \langle capture, move\ in \checkmark \rangle$$
$$t_1 ⨟ t_{2_2} = \langle capture, inspect\ die \rangle$$

These two traces cannot be further sequentially composed at Abstraction Level 4, and hence they will be propagated downward to Abstraction Level 3.

III. At Abstraction Level 3, the corresponding process is $SimplePick_1$. This process should extend the sequential composition *SimpleCapture*;*PickDecision*. (This extension is guaranteed in the given hierarchy. Readers may refer to Example 6.4.5 for the details of the construction.) Hence, the two traces $t_1 ⨟ t_{2_1}$ and $t_1 ⨟ t_{2_2}$ should be traces of the process $SimplePick_1$. As there is no additional "shadowed" process to compose, the two traces are propagated downward to Abstraction Level 2.

IV. At Abstraction Level 2, the corresponding processes are both the process *SimplePick*$_1$. Obviously, the traces calculated at Abstraction Level 3 are traces for this process. However, these two traces encounter the trace $t_3$ from the process *PickArm*. The processes *SimplePick*$_1$ and *PickArm* form a sequential composition at this abstraction layer. Applying trace composition, we construct traces $s_1$ and $s_2$ for the sequential composition *SimplePick*$_1$;*PickArm*.

$$s_1 = \langle capture, move\ in, capture\ photo, inspect\ die, capture\ photo, inspect\ die, move\ in, alert\ operator\rangle$$
$$s_2 = \langle capture, inspect\ die\rangle$$

V. The two latest constructed traces are propagated downward to Abstraction Level 1. The corresponding process is the sequential composition *Pick*;*PickArm*. This process extends the sequential composition *SimplePick*$_1$;*PickArm*. Hence, these two traces should be traces for the sequential composition *Pick*;*PickArm*.

VI. Finally, the traces $s_1$ and $s_2$ will be propagated downward to the Specification Level. The relationship between the sequential composition *Pick*;*PickArm* and *PickArm* is that the latter extends the former. Hence, $s_1$ and $s_2$ should be traces of the process *PickArm*.

■

We are going to present the sufficient conditions such that extension can result in σ-extension and the previous established results can be applied. Informally, apart from the requirements of extension, σ-extension needs two more conditions. The first one requires the preceding process to allow users to observe the worst cases of any common activity that may be performed by the preceding process alone and jointly by preceding and succeeding processes. The second condition is to deal with the problem of non-observability of intermediate successful termination.

**Lemma 6.4.2 (Sufficiency)** *Let P and Q be processes. Suppose that the process P and the process Q have the same alphabets. If*

*I. the process P extends the process* $Q;P/\sqcap_Q$*,*

$(*)$ *the process Q respects the failures of the process* $P/\sqcap_Q$ *associated with those traces in common between Q and* $Q;P/\sqcap_Q$*,*

$$\forall s,t,t\hat{\ }\langle\surd\rangle \in traces(Q), u \in traces(P/\sqcap_Q).$$
$$t < s \wedge s = t\hat{\ }u \implies refusals(P/\sqcap_Q/u) \subseteq refusals(Q/s)$$

$(**)$ *the process* $P/\sqcap_Q$ *does not behave like the process STOP,*

*then the process P σ-extends the process Q.*

*Proof:*

We prove the conditions of sequential extension one by one. Throughout the proof, we assume that $(s,X)$ is a failure of the process $P$. Like in the proof of the partial order of sequential extension, we let

$$u = s\hat{\ }\langle\surd\rangle \quad and$$
$$v = s\hat{\ }\langle\theta\rangle$$

where the symbol θ means a user alphabet as in the definition of sequential extension.

**The proof of Condition I** Given.

**The proof of Condition II** It is trivial.

**The proof of Condition III** Suppose $u$ is a trace of the process $Q$ and there is no $v$ which can be a trace of the process $P$. For this particular case, the process $Q$ will transfer control to the process $P/\sqcap_Q$. However, as there is no trace of the process $P$ that is the same as the sequence $v$, the process $P/\sqcap_Q$ should behave like either the process $STOP$ or the process $SKIP$. Given Condition $(**)$ above, $P/\sqcap_Q$ cannot behave like the process $STOP$. Hence, it should behave like the process $SKIP$. Thus, $u$ should be a trace of the process $P$.

**The Proof of Condition IV** Suppose $u$ is a trace of the process $Q$ and there is no trace like $u$ for the process $P$. The process $Q$ will transfer control to the converging process. Like the above case, the converging process cannot behave like the process $STOP$. Moreover, since $u$ is not a trace of the process $P$, it follows that there exists a trace like $v$ in the trace set of the process $P$.

**The proof of Condition V(a)** This is trivial. As explained in the sequential extension section, the process $Q$ should come to a deadlock after the trace $s$.

**The proof of Condition V(b)** Suppose $u$ is not a trace of the process $Q$ and $s$ is a trace of the process $Q$. (There are two disjunctive conditions for this sub-case to be fulfilled, namely $u$ is not a trace of the process $P$, or $v$ is a trace of the process $P$.) Suppose that there is a trace $t$ which is a proper prefix of the trace $s$ and, at the same time, $t\hat{}\langle\surd\rangle$ is a trace of the process $Q$. Consider the first sub-case that $u$ is a trace of the converging process. Since the process $P$ extends the given sequential composition, all refusals of the process $P$ after the trace $s$ should be refusals of the sequential composition of the process $Q$ and the converging process. The refusals set of this sequential composition after the trace $u$ should be, by definition of sequential composition, the same as the union of the refusals set of the process $Q$ after the trace $s$ if every refusal is enlarged with the event $\surd$, and the refusals set of the converging process. By Condition $(*)$, the refusals of the process $Q$ after the trace $s$ should include the refusals of the converging process after the trace $u$. Hence, the refusals of the sequential composition after the trace $s$ should be the refusals of the process $Q$ after the same trace enlarged with $\surd$. On the other hand, $u$ is supposedly not a trace of the process $Q$. In this connection, the enlargement of the refusal set carries no effect. Consequently, $(s,X)$ should be a failure of the process $Q$.

Alternatively, consider the case the where $u$ does not exist in the converging process. In other words, the trace $s$ can only come from the process $Q$. Given that the process $P$ extends the sequential composition, $(s,X)$ should be a failure of the process $Q$.

Now, consider the case that there is no such $t$ (a proper prefix of the trace $s$) that can be found in the process $Q$. In other words, the trace $s$ can only come from the process $Q$. Given that the process $P$ extends the sequential composition, $(s,X)$ should be a failure of the process $Q$.

**The proof of Condition VI(a)** Suppose $u$ is a trace of the process $Q$ but not a trace of the process $P$. Moreover, $v$ is not a trace of the process $Q$. For this particular case, the process

$Q$ also transfers control to the converging process after the trace $s$. By Condition $(*)$ and following the same argument as for the proof of Condition V(b) above, $(s,X)$ should be a failure of the process $Q$ after the trace is enlarged with the successful termination symbol $\surd$. However, since $u$ is a trace of the process $Q$, we can only conclude that the subset $(s,X\setminus\{\surd\})$ is a failure of the process $Q$.

**The proof of Condition VI(b)** For this particular case, the process $Q$ may transfer control to the converging process. By Condition $(*)$ and following the same argument as above, $(s,X)$ should be a failure of the process $Q$ after the trace is enlarged with the successful termination symbol $\surd$. However, either $u$ is a trace of the process $P$ or $v$ is a trace of the process $Q$. Consider the former case. Given that the process $P$ extends the sequential composition and Condition $(*)$, the refusals of the process $P$ should be respected by the process $Q$; otherwise Condition $(*)$ will not be fulfilled. The remaining scenario is that $u$ is not a trace of the process $P$ and $v$ should be a trace of the process $Q$. Since the process $P$ extends the sequential composition, sequence $v$ should be a trace of the process $P$. In other words, the process $Q$ is able to terminate "earlier" than the process $P$ whenever $P$ cannot terminate immediately after $s$. Hence, by Condition $(*)$, $(s,X)$ should also be a failure of the process $Q$.

**The proof of Condition VII** It follows from the definitions.

□

**Theorem 6.4.4 (Decomposition of Processes)** *Let P and Q be processes. Suppose P and Q have the same alphabet.*

*I. The process P extends the process* $Q;P/\sqcap_Q$[9]*,*

*II. the process Q respects the failures of the process* $P/\sqcap_Q$ *associated with the traces common to the latter and the process* $Q;P/\sqcap_Q$*,*

$$\forall s,t,t\hat{}\langle\surd\rangle \in traces(Q), u \in traces(P/\sqcap_Q).$$
$$t < s \wedge s = t\hat{}u \implies refusals(P/\sqcap_Q/u) \subseteq refusals(Q/s)$$

*III. and the process* $P/\sqcap_Q$ *does not behave like the process* $STOP$

*iff*

*(*) The process P σ-extends the process Q.*

*Proof:* It follows directly from Lemma 6.4.1, Lemma 6.4.2, Definition 6.3.3 and Lemma 6.3.1.
□

**Discussions** Condition (III) in the above Theorem 6.4.4 is due to the fact that it is undecidable to conclude whether the preceding process in a sequential process terminates successfully or not. Consider the following two processes $Q_1$ and $Q_2$.

$$Q_1 = a \rightarrow SKIP \qquad \text{and} \qquad Q_2 = a \rightarrow STOP$$

[9] In other words, it requires that the converging process cannot behave like the process $STOP$

They differ only in the termination. If they form a sequential composition with the process $STOP$, then they will be the same.

$$Q_1;STOP = a \rightarrow STOP = Q_2;STOP$$

However, if we can also derive anti-extensions based on the specification, then this condition may not be a serious problem. This is because, given a specification process, whether it uses the process $STOP$ or the process $SKIP$ immediately before the sequential composition operator can be obtained syntactically. Hence, we can follow a basic rule never to anti-extend the process $STOP$ by the process $SKIP$[10], in which case, we shall not encounter the undecidable problem as illustrated by the processes $Q_1$ and $Q_2$. ♦

**Discussions** Condition (II) presents a limitation in our work. It requires us to check all refusals of the related traces.

I. In our opinion, the situation may not be as limited as it appears to be. First, let us make a reasonable interpretation of the traces $s$, $t$ and $u$. The process $Q$ may reach a termination after the execution of the trace $t$. At the same time, it may perform the sequence $u$ instead of choosing to terminate. A likely scenario is that the process $Q$, as part of a specified system, would like to perform some activity which is not supported by the succeeding process ($P/\sqcap_Q$); otherwise from the design's point of view, it should transfer control over to the process $P/\sqcap_Q$. Let us refer to these activities as a process $U$.

$$Q/s = SKIP \sqcap U$$

Intuitively, the initial events of the succeeding process and that of the process $U$ are possibly different. Hence, the chances of encountering this kind of system specification should not be common. Nevertheless, more empirical studies are required to confirm this intuition.

II. We recommend to restrict CSP specifications to those in which all successful terminations of processes should be deterministic[11]. This is reasonable because we would not want to have programs which cannot secure termination. In this way, Condition (*II*) is degenerated into a trivially fulfilled condition because, by definition, the process $Q$ will refuse all alphabets except $\surd$ at the point of successful termination.

The only case that cannot be covered by this refusal set is where the converging process $P/\sqcap_Q$ will refuse all alphabets. In other words, under the restricted class of specifications as recommended, the process $P/\sqcap_Q$ behaves like the chaotic process $CHAOS$, since Condition (*III*) requires it to behave not like the process $STOP$. The condition will force the process $Q$ to behave like the process $CHAOS$ after the sequence $t$. In this case, the specification ($P$) should behave like the process $CHAOS$; otherwise it cannot extend the sequential composition. Moreover, by $\sigma$-extension, a trace that ends with a successful termination event should be a divergent trace; otherwise it cannot be $\sigma$-extended in a chaotic process. In this situation, Condition (*II*) should also be satisfied automatically.

In fact, since the process $CHAOS$ means totally uncontrollable behaviour, practical specifications will rarely specify chaotic process behaviour and require it to be implemented. It is therefore reasonable to excluding the chaotic case.

[10] In other words, $SKIP$ can be anti-extended by $SKIP$ or $STOP$ and $STOP$ can only anti-extended by $STOP$ (but not $SKIP$).

[11] That is, $SKIP \sqcap P$ is not a part of the specification for some process $P$.

**Theorem 6.4.5 (Decomposition of Deterministic Processes)** *Let P and Q be processes which are deterministic.*

*(*) The process P σ-extends the process Q*

*iff*

*I. the process P extends the process Q;$P/\sqcap_Q$ and*

*II. the process $P/\sqcap_Q$ is well defined (that is, it does not behave like the process STOP).*

*Proof:* This is a special case of Theorem 6.5.4. □

From the condition list, the decomposition approach is more encouraging in deterministic specifications.

**The problem in Reduction Hierarchy** How about using reduction rather than extension? It is known [83] that *reduction* and *refinement* are equivalent. Hence, there is also a good motivation to use reduction. Informally, if the process *P* reduces the process *Q*, then *Q* should include all the traces and failures of *P*. Moreover, the process *Q* may include extra traces or failures. The first problem, however, is that in concurrent systems, the processes are already complex. The inclusion of extra complexity will only further complicate the testing task. The second problem is that the extra operation sequences are not a part of the specification. The conformity to these observations is irrelevant to the conformity to the specification.

**Example 6.4.8** Consider the following processes. Note that the process $EJ_r$ reduces the process $EJ2$ and the process $EJ_{rr}$ also reduces the process $EJ_r$.

$$EJ2 = (moveUp \rightarrow moveDown \rightarrow EJ2)\Box(maintain \rightarrow home \rightarrow SKIP)$$

$$\begin{aligned} EJ2_r = &((moveUp \rightarrow SKIP)\Box(maintain \rightarrow SKIP)) \\ &;((moveDown \rightarrow EJ2)\Box(home \rightarrow SKIP)) \\ =^{12} &moveUp \rightarrow moveDown \rightarrow EJ2 \\ &\Box maintain \rightarrow home \rightarrow SKIP \\ &\Box moveUp \rightarrow home \rightarrow SKIP \quad \textbf{extra behaviour!} \\ &\Box maintain \rightarrow moveDown \rightarrow EJ2 \quad \textbf{extra behaviour!} \end{aligned}$$

$$\begin{aligned} EJ2_{rr} = &((moveUp \rightarrow SKIP)\Box(maintain \rightarrow SKIP)) \\ &;((moveDown \rightarrow EJ2)\Box(home \rightarrow SKIP)) \\ =^{13} &moveUp \rightarrow ((moveDown \rightarrow EJ2)\Box(\textbf{newOperation} \rightarrow \textbf{EJ2})) \\ &\Box maintain \rightarrow home \rightarrow SKIP \\ &\Box moveUp \rightarrow home \rightarrow SKIP \quad \textbf{extra behaviour!} \\ &\Box maintain \rightarrow moveDown \rightarrow EJ2 \quad \textbf{extra behaviour!} \end{aligned}$$

The process $EJ2_r$ introduces “extra” traces. Worse still, the process $EJ_3$ further introduces banned action $newOperation$[14].

Since reduction is a special kind of conformance, besides the problem of intransitivity, conformance also suffers from these problems as in anti-reduction hierarchy. ■

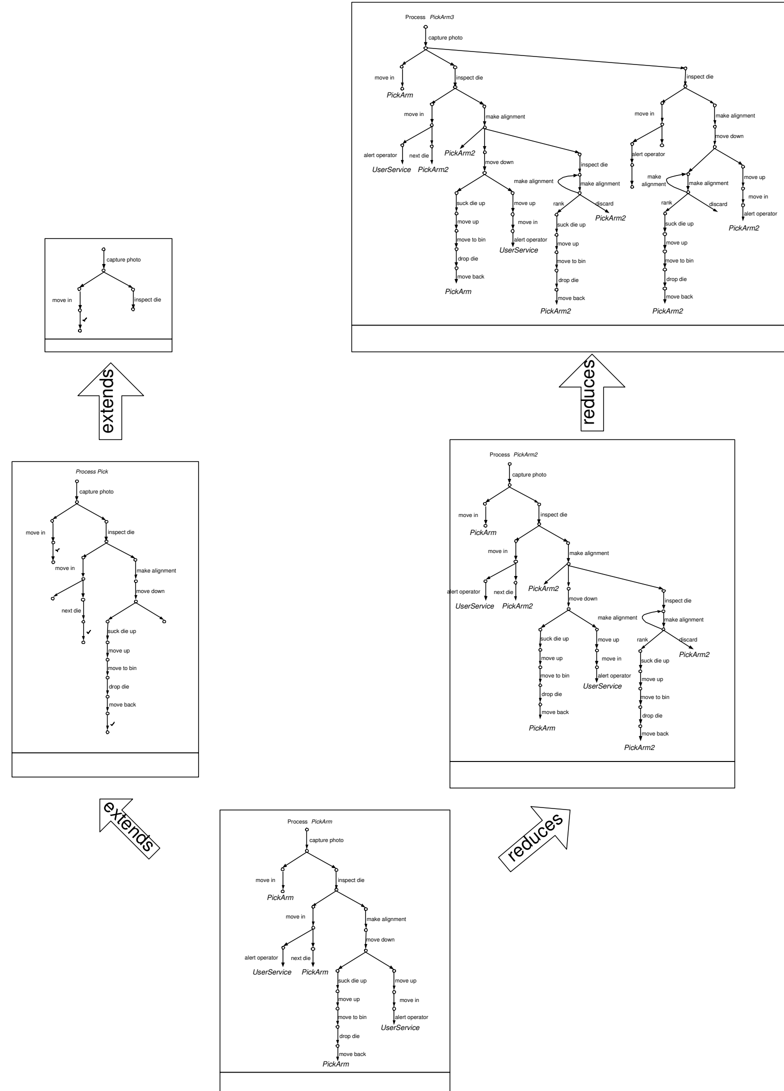


Figure 6.11: Extension and Reduction hierarchies at a glance

Figure 6.11 further gives an impression on the process *PickArm* if we use anti-reduction arbitrarily. The abstraction produced will be more complex and may specify some unexpected operation sequences or strange actions.[15]

[14] All processes $EJ2$, $EJ2_r$ and $EJ2_{rr}$ share the same alphabets. The operation $newOperation$ should not be available to the process $EJ$ or $EJ_r$. They should always refuse it.

[15] Interested readers may observe that reduction incurs the problem that it should “contain” all traces and all failures of the process being reduced (that is, the process *PickArm*). Hence, it may be bulky and may provide

## 6.5 Parallel Constraints

In this section, we shall present our results in tackling Problem Paralleling identified in the last chapter. We identify additional conditions for parallel composition. Generally speaking, it is due to the interference of other concurrent components, which may turn the original successful termination infeasible to occur; whereas the preceding sub-task (that is, a σ-extended component) may allow such termination to occur because it "simply" defers the decision to its converging process. We illustrate this point by the follow example.

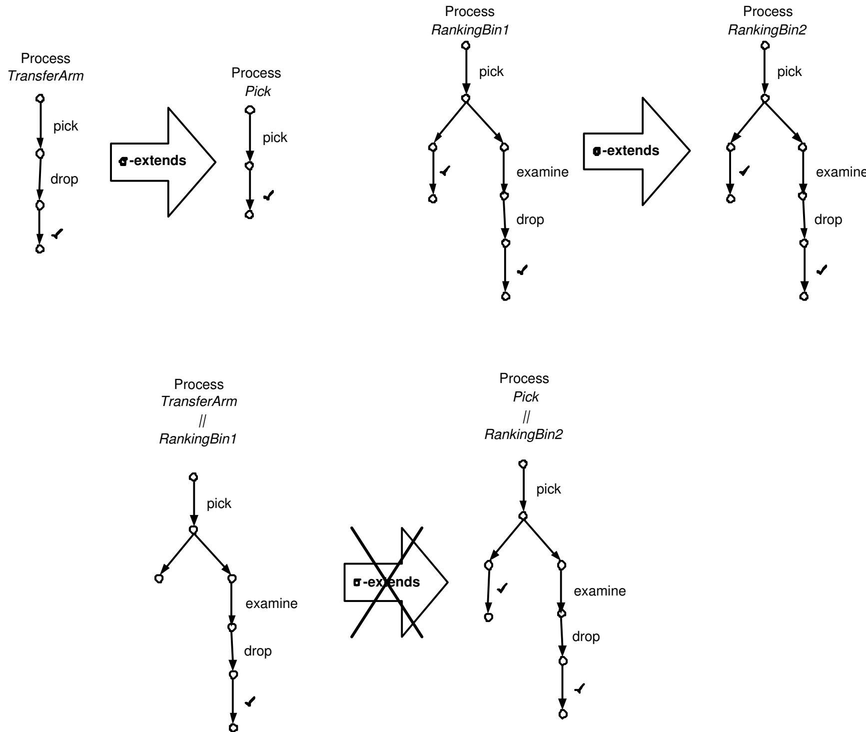


Figure 6.12: Parallel constraints problem

**Example 6.5.1 (Counter-parallel constraints example)** Consider four processes *TransferArm*, *PickHead*, *RankingBin*1 and *RankingBin*2, as illustrated in Figure 6.12. The process *TransferArm* will *pick* up a die and then *drop* it again. The process *PickHead* is more focused — it only performs the *pick* action. The processes *RankingBin*1 and *RankingBin*2 are identical. They can synchronize the *pick* and *drop* actions with the *TransferArm* process. In between these two actions, it *examines* the die. However, they are faulty and coordinately *pick* up the die and then may terminate immediately (and non-deterministically). The *examination*

undesirable result. This figure shows that it may even introduce banned "operations" such as "rank". Hence, it may generate irrelevant test cases if the corresponding test sequences and refusals contains the event "rank".

action is a private action of ranking bins.

$$
\begin{aligned}
&TransferArm = pick \rightarrow drop \rightarrow SKIP \\
&PickHead = pick \rightarrow SKIP \\
\\
&RankingBin1 = pick \rightarrow (SKIP \sqcap examine \rightarrow drop \rightarrow SKIP) \\
&RankingBin2 = RankingBin2
\end{aligned}
$$

We observe that the process *TransferArm* σ-extends the process *PickHead*. Let us consider the parallel compositions *TransferArm* ‖ *RankingBin*1 and *PickHead* ‖ *RankingBin*2.

$$TransferArm \parallel RankingBin1 = pick \rightarrow (STOP \sqcap examine \rightarrow drop \rightarrow SKIP)$$

$$PickHead \parallel RankingBin2 = pick \rightarrow (SKIP \sqcap examine \rightarrow drop \rightarrow SKIP)$$

It is ease to observe that the process *TransferArm* ‖ *RankingBin*1 does not σ-extend the process *PickHead* ‖ *RankingBin*2. This is because both $\langle pick, examine\rangle$ and $\langle pick, \surd\rangle$ are traces of the process *PickHead* ‖ *RankingBin*2. According to the definition of sequential extension (Definition 6.2.1), it falls into Condition VI(b), which requires the parallel composition *PickHead* ‖ *RankingBin*2 to include all refusals associated with the trace $\langle pick\rangle$ for the process *TransferArm* ‖ *RankingBin*1. It means that $(\langle pick\rangle, \{\surd\})$ should be a failure of the process *PickHead* ‖ *RankingBin*2, which is obviously not correct. ■

**Discussions** Condition VI(b) of Sequential Extension (Definition 6.2.1) is to force the process being σ-extended (*PickHead* in the above example) to respect the successful termination of the σ-extended process (the process *TransferArm*). Unfortunately, in a parallel composition environment, the successful termination is forbidden. To restore the spirit of sequential extension, the additional condition is to require that the successful termination events of constitutive processes can be "floated" up to be observed in the parallel composition. ♦

**Theorem 6.5.1 (Parallel σ-Extension)** *Suppose that*

I. *a process P σ-extends a process R,*

II. *a process Q σ-extends a process T,*

III. *the process $P \| Q \,/\sqcap_{R\|T}$ is well defined (that is, it does not behave like the process STOP), and*

IV. *the parallel composition of σ-extending processes should allow successful terminations, which are feasible for the parallel composition of their corresponding anti-σ-extended processes, to occur.*

$$
\begin{aligned}
&\forall s \in traces(P \parallel Q) \cap traces(R \parallel T), u = s \upharpoonright \alpha P, v = s \upharpoonright \alpha Q, \exists \theta \in \alpha P \cup \alpha Q. \\
&(u\hat{}\langle\theta\rangle \in R \wedge u\hat{}\langle\surd\rangle \in R \wedge u\hat{}\langle\theta\rangle \in P \wedge u\hat{}\langle\surd\rangle \in P \implies s\hat{}\langle\surd\rangle \in P\|Q) \wedge \\
&(v\hat{}\langle\theta\rangle \in T \wedge v\hat{}\langle\surd\rangle \in T \wedge v\hat{}\langle\theta\rangle \in Q \wedge v\hat{}\langle\surd\rangle \in Q \implies s\hat{}\langle\surd\rangle \in P\|Q)
\end{aligned}
$$

*where s is not a divergent trace and θ is a user alphabet.*

*Then, the process $P \parallel Q$ will σ-extend the process $R \parallel T$.*

*Proof:* In the sequential extension definition, there are 3 conditions for traces and 4 conditions for failures. Hence, in general, for the parallel composition $R \parallel T$, we need to show 9 combinations for traces and 16 combinations for failures. To establish the sequential extension, in general, we need to consider 27 conditions for traces and 64 conditions for failures. We shall use arguments to eliminate the infeasible cases. The divergences condition is trivial. We first show the trace conditions for the parallel composition $P \parallel Q$ to σ-extend the process $R \parallel T$.

Traces The trace condition is in fact fairly simple. Let us consider a trace $s$ of the process $R \parallel T$. Let $u$ and $v$ denote the projection of the trace $s$ on to processes $R$ and $T$ respectively (that is, $u = s \upharpoonright \alpha P$ and $v = s \upharpoonright \alpha Q$).

**The proof for Condition II** Suppose $s$ is not a successful complete trace. It means that both $u$ and $v$ should not be a successful complete trace because they should synchronize the symbol $\surd$ otherwise (that is, it eliminates four combinations in which one of $u$ and $v$ is a successful complete trace and the other one is not). By Condition II of $R \sqsubseteq_\sigma P$, $u$ should be a trace of the process $P$. Similarly, $v$ should be a trace of the process $Q$. The inverse projection $s$ should then be a trace of the process $P \parallel Q$.

**The proof for Condition III** Now, suppose $s$ is a complete trace of the process $R \parallel T$. Both $u$ and $v$ should be complete traces of the processes $R$ and $T$, respectively. In other words, we need to concentrate on Conditions III and IV. Let us rewrite $s$ as $t\hat{}\langle\surd\rangle$.

Consider the case for Condition III as if the target σ-extension holds. That is, there is no non-successful termination event such as $\theta$[16] for the process $P \parallel Q$ that will form a trace $t\hat{}\langle\theta\rangle$ of the process $P \parallel Q$. Since events should only be synchronized or interleaving, this trace situation will mean that both $u$ and $v$ should not be followed by any non-successful termination event. Given that the process $P$ σ-extends the process $R$, by Condition III, $u\hat{}\langle\surd\rangle$ should be a trace of the process $P$. Similarly, $v\hat{}\langle\surd\rangle$ should be a trace of the process $Q$. Hence, the processes $P$ and $Q$ can synchronize on the successful termination event. Thus, $s\hat{}\langle\surd\rangle$ is a trace of the process $P \parallel Q$.

**The proof for Condition IV** Consider the case for Condition IV as if the target σ-extension holds. That is, the process $P \parallel Q$ will never terminate successfully and immediately after the trace $t$. Since the process $P \| Q \,/\, \triangle_{R\|T}$ is well defined, $P \parallel Q$ after the trace $t$ should not behave like the process $STOP$. In other words, it must be able to proceed (albeit non-deterministically or divergently). Hence, the trace set conditions should be fulfilled.

Failures Now, we would like to prove that the failures conditions for sequential extension should be satisfied. In the subsequent steps in this proof, Conditions V, V(a), V(b), VI, VI(a) and VI(b) refer to the corresponding numbered item in Definition 6.2.1.

Let $(s, X \cup Y)$ be a failure of the parallel composition $P \parallel Q$. Without loss of generality, let $(u, X)$ be a failure of the process $P$ and $(v, Y)$ be a failure of the process $Q$. $u$ and $v$ are the corresponding trace projections of $s$ on to the process $P$ and $Q$, respectively.

[16] Interested readers may refer to Definition 6.2.1 for details.

A. It is known that, in CSP, all events should either be synchronized or interleaved. In particular, the successful termination event should be synchronized. Hence, suppose that a trace, say $s$, of the parallel composition $R \parallel T$ satisfies Condition V. It means that its projection $u$ and $v$ (as above) on the processes $R$ and $T$ cannot both satisfy Condition VI with respect to their corresponding anti-σ extended processes $P$ and $Q$. Otherwise $s\hat{}\langle\surd\rangle$ will be a trace of the process $R \parallel T$, which will contradict the assumption that it is not.

B. On the other hand, if both of them satisfy Condition V(a), it will reduce to the deadlock case. Obviously, the parallel composition $R \| T$ will be a deadlock. Condition V(a) for the parallel composition $R \parallel T$ should hold.

C. Besides, the refusals of any parallel composition (such as the process $R \parallel T$) after a trace should be the union of the refusals of the parallel components (such as $R$'s and $T$'s) after the corresponding projections of the trace $s$. Revisiting the condition list of sequential condition, Conditions V(a), V(b) and VI(b) do not impose further restriction on the refusals and, hence, their parallel composition should be failures of the processes $T$ and $Q$. In other words, $(s, X \cup Y)$ should be a failure of the parallel composition $R \parallel T$.

D. Similarly, if both $u$ and $v$ satisfy Condition VI(a), then $(s, (X \cup Y) \setminus \{\surd\})$ should be a failure of the parallel composition $R \parallel T$.

E. The most difficult case is such that one of them satisfies Condition VI(a), but not both. (In this case, all other combinations are symmetric, and parallel composition will include both of them as its parts.) We observe that in this theorem, the treatments of the processes $P$ and $Q$ and processes $R$ and $T$, as well as other conditions such as the well-definedness of the converging process, are symmetric. Without loss of generality, suppose $u$ satisfies Condition VI(a) (as $s$ in the condition) but not $v$.

   i. Suppose $v$ can be followed by the successful termination symbol $\surd$ (that is, $v$ is in Condition VI(b)). In this case, $s\hat{}\langle\surd\rangle$ is a trace of the parallel composition $R \parallel T$. Condition V between compositions $P \parallel Q$ and $R \parallel T$ will be irrelevant. On the other hand, suppose $v$ satisfies Condition V (that is, $v$ cannot be followed by a successful termination event in the process $T$, no matter whether it actually satisfies Condition V(a) or V(b)). The parallel composition $R \parallel T$ must be able to refuse the successful termination event. It contradicts Condition VI that $s$ is a feasible complete trace of the parallel composition $R \parallel T$.
   Hence, whether $v$ satisfies Conditions V(a), V(b) or VI(b) should be irrelevant to Condition V between compositions $P \parallel Q$ and $R \parallel T$.

   ii. In this connection, the remaining case is that $v$ satisfies Condition VI(b) and $u$ satisfies Condition VI(a).

   iii. Consider Condition VI(a) for the composition $R \parallel T$ to be extended by the composition $P \parallel Q$ sequentially. In this case, $s\hat{}\langle\surd\rangle$ should not be a trace of the parallel composition $P \parallel Q$. Moreover, the parallel composition $R \parallel T$ may only terminate successfully. There are two scenarios to consider. The first is

that the parallel composition $P \parallel Q$ forbids a non-successful termination event, say $\theta$, after the traces $s$ to be observed. The other one is the contrary.

A. Let us consider the first scenario. Suppose that the parallel composition $P \parallel Q$ allows the event $\theta$ to occur after the trace $s$. The process $P$ can guarantee the failure $(u, X \setminus \{\surd\})$ to be a failure of the process $R$. Based on simple mathematical on set manipulation, the failure $(s, X \setminus \{\surd\} \cup Y)$ $(=(s, (X \cup Y) \setminus \{\surd\}))$ should be a failure of the parallel composition $R \parallel T$.

B. Let us consider the second scenario. Suppose that the parallel composition $P \parallel Q$ does not allow the event $\theta$ to occur. Follow exactly the same logic as the first scenario above, $(s, (X \cup Y) \setminus \{\surd\})$ should also be a failure of the parallel composition $R \parallel T$.

iv. Consider Condition VI(a) for the composition $R \parallel T$ to be extended by the composition $P \parallel Q$ sequentially. It will be feasible provided that the two sub-conditions (the negation of the sub-case in Condition VI(a)) can be disjunctively held.

A. Consider the first term in the disjunctive sub-condition. That is, $s\hat{}\langle\surd\rangle$ is a trace of the parallel composition $P \parallel Q$. It also means that both the processes $P$ and $Q$ after the traces $u$ and $v$, respectively, should be feasible to terminate successfully. However, it contradicts the assumption that $u$ is in Condition VI(a) (for the relationship between processes $P$ and $R$) under which the process $P$ cannot terminate successfully after the trace $u$ (that is, violating the first conjunctive sub-condition of Condition VI(a)). Hence, it is an infeasible case.

B. Now, consider the second term in the disjunctive sub-condition. That is, there is at least an event, say $\theta$, with which the parallel composition $R \parallel T$ may progress after the trace $s$. The trace $s\hat{}\langle\theta\rangle$ must be a trace of the process $P \parallel Q$. Since $u$ is assumed to satisfy Condition VI(a), it means that the process $R$ cannot progress with event $\theta$ after the trace $u$. Hence, if the case is self-consistent, the event $\theta$ should come from the process $T$ after the trace $v$. Given that the process $Q$ includes all non-successful complete traces of the process $T$, $v\hat{}\langle\theta\rangle$ should be a trace of the process $Q$. We further consider whether or not the first term in the disjunctive sub-condition can be true.
($Case$ 1) Suppose $s\hat{}\langle\surd\rangle$ is a trace of the process $P \parallel Q$. This is not feasible, because it will contradict the assumption in this case that the process $P$ does not contain $u\hat{}\langle\surd\rangle$ as one of its traces.
($Case$ 2) Suppose $s\hat{}\langle\surd\rangle$ is not a trace of the process $P \parallel Q$. It will be further required to consider whether or the process $Q$ may allow $v\hat{}\langle\surd\rangle$ as a trace.
($Case$ 2$a$) However, by the given Condition IV, it is not feasible for the process $Q$ to have a trace $v\hat{}\langle\surd\rangle$ but finally removed in the composition $P \parallel Q$.
($Case$ 2$b$) Suppose the process $Q$ does not allow $v\hat{}\langle\surd\rangle$ to be one of its traces. In this case, the process $Q$ must refuse the $\surd$ event after the trace $v$. Hence, the failure relationskip for this particular trace between the

processes $T$ and $Q$ are given by Condition VI(b). Thus, the process $T$ must be able to refuse the event $\surd$. On the other hand, the process $T$ has $v\hat{}\langle\theta\rangle$ as its trace (because $s\hat{}\langle\theta\rangle$ is considered as a trace of the parallel composition $R \parallel T$). It means that the process $T$ may choose to terminate successfully or to progress with some event. Consequently, the process $T$ must be able to refuse all such events $\surd$ and $\theta$. Hence, the parallel composition $R \parallel T$ should be able to refuse every alphabet in its alphabet set after the trace $s$. Naturally, $(s, X \cup Y)$ should be a failure of this composition.

Moreover, since the after operator (that is, $/s$) is distributive across the parallel composition[17], $(P\|Q)/s$ should be the same as $(P/(s \upharpoonright \alpha P)) \parallel (Q/(s \upharpoonright \alpha Q))$. In other words, it is the same as $P/u \parallel Q/v$. Hence, the converging process for σ-extension is defined. The result follows. □

**Theorem 6.5.2 (Parallel anti-extension)** *Suppose that*

*I. a process $P$ σ-extends a process $R$,*

*II. a process $Q$ σ-extends a process $T$,*

*III. processes $S$ and $U$ anti-extend the processes $P/\triangle_R$ and $Q/\triangle_T$, respectively,*

*IV. the process $P\|Q\ /\sqcap_{R\|T}$ is well defined (that is, it does not behave like the process $STOP$), and*

*V. the parallel composition of processes being σ-extended should allow successful terminations, which are feasible for the parallel composition of the corresponding anti-σ-extended processes, to occur.*

$$\forall s \in traces(P \parallel Q) \cap traces(R \parallel T), u = s \upharpoonright \alpha P, v = s \upharpoonright \alpha Q, \exists \theta \in \alpha P \cup \alpha Q.$$
$$(u\hat{}\langle\theta\rangle \in R \wedge u\hat{}\langle\surd\rangle \in R \wedge u\hat{}\langle\theta\rangle \in P \wedge u\hat{}\langle\surd\rangle \in P \implies s\hat{}\langle\surd\rangle \in P\|Q) \wedge$$
$$(v\hat{}\langle\theta\rangle \in T \wedge v\hat{}\langle\surd\rangle \in T \wedge v\hat{}\langle\theta\rangle \in Q \wedge v\hat{}\langle\surd\rangle \in Q \implies s\hat{}\langle\surd\rangle \in P\|Q)$$

*where $s$ is not a divergent trace and $\theta$ is a user alphabet.*

*The process $P \parallel Q$ extends the process $(R \parallel T)\,;(S \parallel U)$.*

*Proof:* The proof is similar to that of Theorem 6.5.1. The process $P \parallel Q$ should σ-extend the process $R \parallel T$. Hence, the process $P\|Q\ /\sqcap_{R\|T}$ will be the process $P\|Q\ /\ \triangle_{R\|T}$ by definition. We can use the results on extension to show that the parallel composition $P/\triangle_R\| Q/\triangle_T$ extends the process $S \parallel U$. Consequently, by the Tail Anti-Extension Theorem (Theorem 6.4.1), the result follows. The detailed steps are as follows. By Proposition A.1.5, the process $P/\triangle_R$ should extend the process $P/\triangle_R\| Q/\triangle_T$. By Proposition A.1.6, it should extend the parallel compostion $S \parallel Q/\triangle_T$. Applying the same proposition again, it should also extend $S \parallel U$. Similarly, the process $Q/\triangle_T$ should extend $S \parallel U$. By Corollary A.1.1, the composition $P/\triangle_R\| Q/\triangle_T$ should extend $S \parallel U$. The result follows. □

By equating the processes $Q$ and $P$ and equating the processes $R$ and $T$, this theorem can be seen to be compatible with Lemma 6.4.1.

[17] See Section 3.4 on CSP.

Combining with the Sufficiency Lemma (Lemma 6.4.2), we have the following theorems.

**Theorem 6.5.3 (Decomposition of Concurrent Processes)** *Let P, Q, R, S, T and U be processes.*

*I. A process $P$ $\sigma$-extends a process $R$,*

*II. a process $Q$ $\sigma$-extends a process $T$,*

*III. the parallel composition of $\sigma$-extending processes should allow successful terminations, which are feasible for the parallel composition of their corresponding anti-$\sigma$-extended processes, to occur*

$$\forall s \in traces(P \parallel Q) \cap traces(R \parallel T), u = s \upharpoonright \alpha P, v = s \upharpoonright \alpha Q, \exists \theta \in \alpha P \cup \alpha Q.$$
$$(u\hat{\ }\langle\theta\rangle \in R \wedge u\hat{\ }\langle\surd\rangle \in R \wedge u\hat{\ }\langle\theta\rangle \in P \wedge u\hat{\ }\langle\surd\rangle \in P \implies s\hat{\ }\langle\surd\rangle \in P \| Q) \wedge$$
$$(v\hat{\ }\langle\theta\rangle \in T \wedge v\hat{\ }\langle\surd\rangle \in T \wedge v\hat{\ }\langle\theta\rangle \in Q \wedge v\hat{\ }\langle\surd\rangle \in Q \implies s\hat{\ }\langle\surd\rangle \in P \| Q)$$

*where $s$ is not a divergent trace and $\theta$ is a user alphabet*

*IV. processes $S$ and $U$ anti-extend the processes $P/\triangle_R$ and $Q/\triangle_T$, respectively, and*

*V. the process $P \| Q / \sqcap_{R\|T}$ is well defined (that is, it does not behave like the process $STOP$).*

*iff*

*A. the process $P \parallel Q$ extends the process $(R \parallel T);(S \parallel U)$,*

*B. the process $P \| Q / \sqcap_{R\|T}$ is well defined (that is, it does not behave like the process $STOP$),*

*C. the parallel composition $R \parallel T$ respects the failures of the process $(P \parallel Q)/\sqcap_{R\|T}$ associating with those traces in common between $R \parallel T$ and $(R \parallel T);(P\|Q)/\sqcap_{R\|T}$*

$$s, t, t\hat{\ }\langle\surd\rangle \in traces(R \parallel T), u \in traces\big((P \parallel Q)/\sqcap_{R\|T}\big).$$
$$t < s \wedge s = t\hat{\ }u \implies refusals\big(P\|Q/\sqcap_{R\|T}\big) \subseteq refusals(R\|T/s)$$

*D. processes $S$ and $U$ anti-extend the processes $P/\sqcap_R$ and $Q/\sqcap_T$, respectively.*

*Proof:* It follows directly from Lemma 6.4.2 and Theorem 6.5.2. □

**Discussions**

I. One can observe that, in the last theorem, Conditions (IV) and (D) are similar and Conditions (V) and (B) are similar.

   Conditions (IV) and (D) are included in the theorem to make it general. We may use the processes $P/\triangle_R$ and $P/\sqcap_R$ instead of the process $S$, and use the process $Q/\triangle_T$ instead of the process $Q/\sqcap_T$. Although anti-extensions of these processes are already proved in Theorem 6.5.2, we would like to show them in explicit terms.

   Conditions (V) and (B) are very similar. They appear in the "if" and "only if" parts, respectively. This is because the remaining conditions in the "if" part and the "only if" part, respectively, are not sufficient to show the result.

II. Similarly to the discussion for increasing the applicability of Theorem 6.5.3 (page 72), one possible way to increase the applicability of this theorem is to restrict on the class of specifications. Using the exact proposal for Theorem 6.5.3, we may limit the use of this technique for those specifications in which successful terminations are deterministic. In this case, Conditions (III) and (C) in the theorem can be reduced to trivial cases.

   Consider Condition (III) above. Since successful termination is deterministic, it will not be possible for the processes $P$, $Q$, $R$ and $T$ to have a trace which can immediately be followed by the successful termination event $\surd$ and a user alphabet, say $\theta$, at the same time[18]. Hence, the pre-requisite of the condition will never appear in this kind of specification. The arguments for Condition (C) follows the same logic as that on page 72.

In practice, there are a lot of deterministic specifications. The conditions can be simplified.

**Theorem 6.5.4 (Decomposition of Concurrent Deterministic Processes)** *Let P, Q, R, S, T and U be deterministic processes.*

*I. The process $P$ $\sigma$-extends the process $R$,*

*II. the process $Q$ $\sigma$-extends the process $T$,*

*III. processes $S$ and $U$ anti-extend the processes $P/\triangle_R$ and $Q/\triangle_T$, respectively, and*

*IV. the process $P\|Q\ /\sqcap_{R\|T}$ is well defined (that is, it does not behave like the process STOP)*

*iff*

*A. the process $P \parallel Q$ extends the process $(R \parallel T);(S \parallel U)$,*

*B. the process $P\|Q\ /\sqcap_{R\|T}$ is well defined (that is, it does not behave like the process STOP) and*

*C. processes $S$ and $U$ anti-extend the processes $P/\sqcap_R$ and $Q/\sqcap_T$, respectively.*

*Proof:* Consider Condition (6.5.3) of the "if" part of Theorem 6.5.3. Suppose that all processes are deterministic. Let $s$, $t$ and $u$ be traces. Suppose $s$, $t$ and $t\hat{}\langle\surd\rangle$ are traces of the process $R \parallel T$, and suppose $u$ is a trace of the process $(P \parallel Q)/\triangle_{R\|T}$. Since all these processes should be deterministic, the trace $t$ should be a proper prefix of the trace $s$ (that is, $s = t\hat{}\langle\surd\rangle$). In other words, the trace $u$ is $\langle\surd\rangle$. Obviously, after a successful termination, the process $R \parallel T/s$ should refuse everything and behaves like the process $STOP$. Similarly, the process $(P \parallel Q)/\triangle_{R\|T}$ $/\langle\surd\rangle$ should also behave like the process $STOP$. The refusal condition is trivially fulfilled.

Similarly, consider Condition (III) of the "only if" part of Theorem 6.5.3. Since all the processes are deterministic, it is not possible to have a trace, say $u$, of a process, say $R$, which can be followed by both $\surd$ and an event other than the successful termination event. Hence, this condition is also trivially fulfilled. □

Obviously, by setting

[18] For the CSP restriction on the use of *SKIP*, please refer to the sequential composition section of the CSP chapter in this thesis.

I. the processes $P$ and $Q$ to be equivalent,

II. the processes $R$ and $T$ to be equivalent, and

III. the processes $S$, $U$, $P/\sqcap_R$ and $Q/\sqcap_T$ to be equivalent,

the parallel composition will be degenerated into the case of a single process decomposition. Hence, we have Theorem 6.4.5 on page 74.

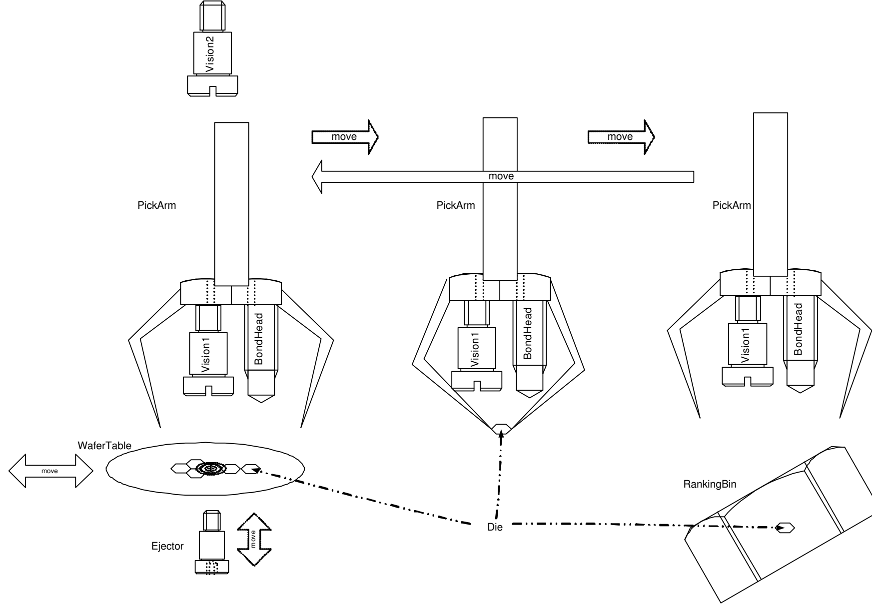


Figure 6.13: A bonding machine

**Example 6.5.2 (Extract from Example 6.5.3)** Consider the deterministic processes in Example 6.5.3, which is the full version of the present example. It describes the operations of a bonding machine. In brief, the pick arm is responsible for picking up and transferring a die. The wafer table is responsible feeding a die to be picked up by the pick arm. The vision system is responsible for letting the wafer table know whether or not there is a die in the picking position. Like other previous examples, we may "serialize" these processes and produce anti-extensions.

I. The process $ThePickArm$ may produce the anti-extension $PA_{unblock}$;$ThePickArm$,

II. the process *TheWaferTable* may produce the anti-extension $WT_{vision}$;$WT_{unblock}$;*TheWaferTable* and

III. the process *TheVision2* may produce the anti-extension $V2_{capture}$;*TheVision2*.

Moreover, it is ease to observe that the processes $ThePickArm$, *TheWaferTable* and $Vision2$ $\sigma$-extend the processes $PA_{unblock}$, $WT_{vision}$;$WT_{unblock}$ and $V2_{capture}$, respectively. The corresponding converging processes can be shown to be the same as the specified

processes. In this case, the condition for the theorem will be fulfilled automatically[19]. Hence, by this theorem, we can decompose the parallel composition

$$ThePickArm \parallel TheWaferTable \parallel TheVision2$$

into the process

$$\begin{array}{l}(PA_{unblock} \parallel WT_{vision};WT_{unblock} \parallel V2_{capture});\\ \quad (ThePickArm \parallel TheWaferTable \parallel TheVision2)\end{array}$$

It can be further serialized into the process

$$\begin{array}{l}(WT_{vision} \parallel V2_{capture});\\ \quad (PA_{unblock} \parallel WT_{unblock});\\ \quad\quad (ThePickArm \parallel TheWaferTable \parallel TheVision2)\end{array}$$

Thus, we can generate test cases from this process (such as using the techniques described in Example 6.4.6). ■

**Example 6.5.3 (Concurrent process decomposition example)** This is the full version of Example 6.5.2. Consider a bonding machine as shown in Figure 6.13. The movement is as follows. Initially, the Ejector should push a die on the Wafer Table up a little bit. At the same time, the Pick Arm should try to pick the die from the top. If it can be done, the die will be firmly held. Hence, the Bond Head attached to the Pick Arm will move the bonding point and start bonding.

The Bond Head will first turn on its bonding engine, feed wire and then release a little bit if any additional bonding for the same wire is required. If there is no additional bonding requirement, it will tear up the wire. Of course, to allow additional bonding, the Pick Arm should coordinately move to the required wiring position. Similarly, it also requires the Ejector to release a little; otherwise the die or the wafer table will easily be broken.

After the bonding is complete, the Bond Head will turn off its engine and the Pick Arm will inform the Ejector to release the holding. Then, the Bond Head will set back to unblock the vision system attached to the Pick Arm (Vision 1) to capture the bonding result. The Pick Arm will inform the Ranking Bin to inspect this result. The latter will analyze the image and rank the quality of the die accordingly. It will also move the bin for the appropriate ranking quality to prepare it for receiving the die.

Meanwhile, the Pick Arm should accomplish two things. First, it will move next to the Ranking Bin and, if the Ranking Bin is ready, it will drop the die. Secondly, it will inform the Ejector to push down a little to release the Wafer Table so that the latter can advance to the next slot. The cycle repeats itself.

In fact, the Wafer Table is also accomplished with another vision system (Vision 2) fixed at a higher place. In order to align the die to be ready for being held by the Ejector and the Pick Arm, it first requests Vision 2 to capture an image of a target die and then it will inspect the result. There are a few scenarios. First, it will read the photo, and make necessary alignment so that the die can be picked easily. Secondly, the Pick Arm may block the image of the die. In this case, it will request the Pick Arm to unblock Vision 2. Thirdly, if the die slot is empty

[19] Hence, we can compose and decompose these processes without the need to refer to the internal structures of these processes. In other words, we can work at the process level.

or the die is out of reach, it will try to skip that slot. Finally, if there is any unfortunate error, it will alert the operator and request for service.

The processes for these entities are as follows. Figure 6.14 shows the bonding machine in the processes. All these processes are assumed to be composed concurrently together.

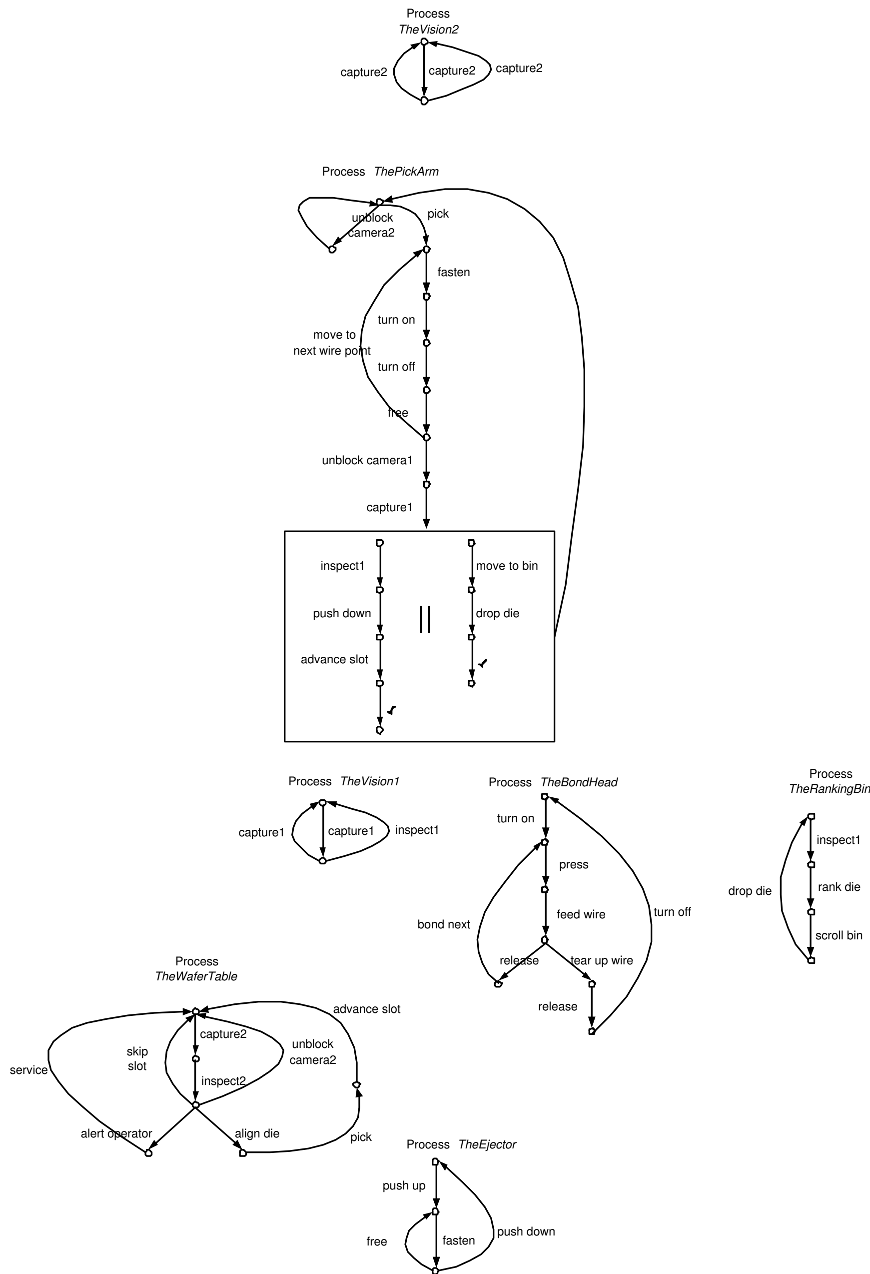


Figure 6.14: The bonding machine in CSP processes

I. *Description of processes:*

$TheBondHead$ The bond head entity is modelled as the process $TheBondHead$. It will

*trun on* the engine, touch the die and *press* on the die gently so as to *feed wire* to seal a wiring point. It may then *release* the pressure and request to *bond next contact*. Since the engine is still working, what it needs to do is to press on the die at the required contact location again and so on. Finally, after a bond has been processed, the process *TheBondHead* will *tear up the wire*, release itself and turn off the engine. It will wait until another bonding is required.

$$
\begin{aligned}
TheBondHead = \; & turn\ on \rightarrow \mu X.(press \rightarrow feed\ wire \rightarrow ( \\
& (release \rightarrow bond\ next \rightarrow X) \\
& \Box (tear\ up\ wire \rightarrow release \rightarrow turn\ off \rightarrow TheBondHead)))
\end{aligned}
$$

*ThePickArm* The pick arm is modelled as the process *ThePickArm*. It tries to *pick* up the die, and then *fasten* it with the help from the ejector. It will start the bonding process by requesting to turn on the engine of the bond head. It then waits until the bonding process is finished and informs the ejector to *free* the die from the bottom. If an additional bonding is required, it will *move to the next wire point* and require another fastening of the die and so on. On the other hand, if no additional bonding is required, it will move off the bond head a little to *unblock* the camera attached to Vision System Number One and request a photo of the die to be taken. As described above, it will try to accomplish two things simultaneously. First, it will signal other entities. The ranking bin is requested to *inspect* the photo captured by Vision System Number One. The ejector is requested to *push down* to return to its ready position. The wafer table is requested to *advance slot*. Secondly, it will *move to* the ranking bin as quickly as it can, and *drop the die* if the ranking bin is ready to receive it. After these two activities, it will return to its ready position and prepare to perform another cycle. Sometimes, because it moves so quickly, it may block Vision System Number Two right over the wafer table and prevent the camera from capturing the image of the die required by the latter. In this case, it will move out a little to *unblock* the camera and then return to its ready position.

$$
\begin{aligned}
ThePickArm = \; & pick \rightarrow \mu X.(fasten \rightarrow turn\ on \rightarrow turn\ off \rightarrow free \rightarrow ( \\
& (move\ to\ next\ wire\ point \rightarrow X) \\
& \Box(unblock\ camera1 \rightarrow capture1 \rightarrow \\
& \quad ((inspect \rightarrow push\ down \rightarrow advance\ slot \rightarrow SKIP) \\
& \qquad \| \; (move\ to\ bin \rightarrow drop\ die \rightarrow SKIP)));ThePickArm))) \\
& \qquad \Box unblock\ camera2 \rightarrow ThePickArm
\end{aligned}
$$

***TheVision1* and *TheVision2*** The vision systems are modelled as simple capturing activities. After the capturing, it will place the image openly to a memory location to be used by any entity. However, it will not keep track of whether the image is used adequately. If there is another capturing, the new image will override the old image. In other words, the old image is lost. The vision systems are passive entities — they will not capture images intelligently. Hence, other entities should coordinate properly to avoid unnecessary re-capturing.

$$TheVision1 = capture1 \rightarrow TheVision1$$

$$TheVision2 = capture2 \rightarrow TheVision2$$

*TheRankingBin* The ranking bin is modelled by the process *TheRankingBin*. Upon receiving the request from the pick arm, it will *inspect* the image from Vision System Number One. Then, it will *rank* the die according to the image catured. There are a number of bins attached. Each bin represents a place for a certain quality of die. It will *scroll* to the required bin and signal the pick arm that it is ready to *drop* the die into the desirable bin. After that, it is ready to serve another request.

$$TheRankingBin = inspect1 \rightarrow rank\ die \rightarrow scroll\ bin \\ \rightarrow drop\ die \rightarrow TheRankingBin$$

***TheEjector*** The ejector is modelled by the process *TheEjector*. Its functionality is self-explanatory. It pushes up the die a little so that the pick arm can pick it more easily. Then, it will fasten and free the position repeatedly. If a bonding process is finished, it will push down so that the wafer table can advance to a subsequent die to be bonded.

$$TheEjector = push\ up \rightarrow \mu X.(fasten \rightarrow free \rightarrow \\ (X \Box push\ down \rightarrow TheEjector)$$

***TheWaferTable*** The functionaltiy of the process *TheWaferTable*, which models the wafer table, has already been explained above.

$$\begin{aligned} TheWaferTable = capture2 \rightarrow inspect2 \rightarrow ( \\ \Box align\ die \rightarrow pick \rightarrow advance\ slot \rightarrow TheWaferTable \\ \Box skip\ slot \rightarrow TheWaferTable \\ \Box alert\ operator \rightarrow service \rightarrow TheWaferTable \\ \Box unblock\ camera2 \rightarrow TheWaferTable) \end{aligned}$$

(*) We observe that all the processes are deterministic. According to the Theorem of Decomposition of Concurrent Deterministic Processes (Theorem 6.5.4), we can split these processes into sequential compositions[20]. Moreover, by applying this theorem, the original processes will σ-extend the preceding components of the decomposition.

First, we rewrite these processes to replace the recursion constructs (such as $\mu X.(a \rightarrow X)$) by some successfully terminated process followed by the process identifier (such as $\mu X.(a \rightarrow$

[20] If their decomposed processes are also deterministic and the corresponding converging processes are not deadlock processes.

$SKIP;X$)). For instance, the process *TheWaferTable* can be rewritten as follows:

$$\begin{aligned}
&\mathit{TheWaferTable} = capture2 \rightarrow inspect2 \rightarrow (\\
&\quad align\ die \rightarrow pick \rightarrow advance\ slot \rightarrow \mathit{TheWaferTable}\\
&\quad \Box skip\ slot \rightarrow \mathit{TheWaferTable}\\
&\quad \Box alert\ operator \rightarrow service \rightarrow \mathit{TheWaferTable}\\
&\quad \Box unblock\ camera2 \rightarrow \mathit{TheWaferTable})
\end{aligned}$$

$$=$$

$$\begin{aligned}
&\mathit{TheWaferTable} = capture2 \rightarrow inspect2 \rightarrow SKIP;(\\
&\quad align\ die \rightarrow pick \rightarrow advance\ slot \rightarrow SKIP;\mathit{TheWaferTable}\\
&\quad \Box skip\ slot \rightarrow SKIP;\mathit{TheWaferTable}\\
&\quad \Box alert\ operator \rightarrow service \rightarrow SKIP;\mathit{TheWaferTable}\\
&\quad \Box unblock\ camera2 \rightarrow SKIP;\mathit{TheWaferTable})
\end{aligned}$$

We can then partition each process according to some criteria such as control flow. In the rest of the example, we use control paths as the partitioning criteria. Consider the wafer table process.[21] We can identify four sub-processes syntactically within the outermost brackets.

$$\begin{aligned}
&\mathit{TheWaferTable} =\\
&\quad WT_{vision};(WT_{adv}\Box WT_{skip}\Box WT_{error}\Box WT_{unblock});\mathit{TheWaferTable}
\end{aligned}$$

$$\begin{aligned}
&WT_{vision} = capture2 \rightarrow inspect2 \rightarrow SKIP\\
&WT_{adv} = align\ die \rightarrow pick \rightarrow advance\ slot \rightarrow SKIP\\
&WT_{skip} = skip\ slot \rightarrow SKIP\\
&WT_{error} = alert\ operator \rightarrow service \rightarrow SKIP\\
&WT_{unblock} = unblock\ camera2 \rightarrow SKIP
\end{aligned}$$

We observe that the converging process $\mathit{TheWaferTable}/\triangle_{WT_{vision}}$ is

$$(WT_{adv}\Box WT_{skip}\Box WT_{error}\Box WT_{unblock});\mathit{TheWaferTable}$$

Similarly, the converging processes for other sub-processes (that is, $WT_{adv}$, $WT_{skip}$, $WT_{error}$ and $WT_{unblock}$ ) are the process *TheWaferTable*. Hence, we can decompose these sub-processes in exactly the same way. Figure 6.15 shows their decompositions.

We observe that none of the converging processes for all the $BH$s, $PA$s, $V1$, $V2$ and $EJ$s behaves like the process $STOP$. We note also that the condition for the Decomposition Theorem (both for a single process and for concurrent processes) are fulfilled automatically. In addition to these, take the $WTs$ as an example. All the $WT$s have different initial events. Hence, one can *anti-extend* the $\Box$-composition by its constitutive processes separately. Let $W$ denote the $\Box$-composition.

$$W = (WT_{adv}\Box WT_{skip}\Box WT_{error}\Box WT_{unblock})$$

[21] We may either unwind the recursion a number of times according to some user parameters. For simplicity, we do not unwind the process *TheWaferTable* in the example.

$$TheBondHead = BH_{start};\mu X.((BH_{next};X)\Box(BH_{done};TheBondHead))$$

$$BH_{start} = turn\ on \rightarrow SKIP$$
$$BH_{next} = press \rightarrow feed\ wire \rightarrow release \rightarrow bond\ next \rightarrow SKIP$$
$$BH_{done} = press \rightarrow feed\ wire \rightarrow tear\ up\ wire \rightarrow release \rightarrow turn\ off \rightarrow SKIP$$

$$ThePickArm = PA_{pick};\mu X.(PA_{bond}; ((PA_{nextbond};X)\Box(PA_{transfer};ThePickArm)) \Box PA_{unblock};ThePickArm$$

$$PA_{pick} = pick \rightarrow SKIP$$
$$PA_{bond} = fasten \rightarrow turn\ on \rightarrow turn\ off \rightarrow free \rightarrow SKIP$$
$$PA_{nextbond} = move\ to\ next\ wire\ point \rightarrow SKIP$$
$$PA_{transfer} = PA_{capture};(PA_{signal} \parallel PA_{bin})$$
$$PA_{capture} = unblock\ camera1 \rightarrow capture1 \rightarrow SKIP$$
$$PA_{signal} = inspect \rightarrow push\ down \rightarrow advance\ slot \rightarrow SKIP$$
$$PA_{bin} = move\ to\ bin \rightarrow drop\ die \rightarrow SKIP$$
$$PA_{unblock} = unblock\ camera2 \rightarrow SKIP$$

$$TheVision1 = V1_{capture};TheVision1$$

$$V1_{capture} = capture1 \rightarrow SKIP$$

$$TheVision2 = V2_{capture};TheVision2$$

$$V2_{capture} = capture2 \rightarrow SKIP$$

$$TheRankingBin = RB_{bin};TheRankingBin$$

$$RB_{bin} = inspect1 \rightarrow rank\ die \rightarrow scroll\ bin \rightarrow drop\ die \rightarrow SKIP$$

$$TheEjector = EJ_{push};\mu X.(EJ_{hold}; (X\Box(EJ_{release};TheEjector)))$$

$$EJ_{push} = push\ up \rightarrow SKIP$$
$$EJ_{hold} = fasten \rightarrow free \rightarrow SKIP$$
$$EJ_{release} = push\ down \rightarrow SKIP$$

Figure 6.15: Components of other processes of the bond machine

In other words, we have

$WT_{adv}$ anti-extends $W$,
$WT_{skip}$ anti-extends $W$,
$WT_{error}$ anti-extends $W$ and
$WT_{unblock}$ anti-extends $W$.

By the Head Anti-Extension Theorem (Theorem 6.4.2),

$$WT_{adv};\mathit{TheWaferTable} \text{ anti-extends } W;\mathit{TheWaferTable}, \text{and}$$
$$WT_{unblock};\mathit{TheWaferTable} \text{ anti-extends } W;\mathit{TheWaferTable}$$

and so on. Similarly, we should have

$$\begin{aligned} &PA_{unblock};\mathit{ThePickArm} \text{ anti-extends } \mathit{ThePickArm} \\ &V2_{capture};\mathit{TheVision2} \text{ anti-extends } \mathit{TheVision2} \end{aligned} \tag{6.10}$$

By Serial Anti-Extension on $WT$s, we can conclude that

$$WT_{vision};WT_{unblock};\mathit{TheWaferTable} \text{ anti-extends } \mathit{TheWaferTable}. \tag{6.11}$$

The whole concurrent system (the process $Sys$) should be represented by the parallel composition of those specification processes.

$$\begin{aligned} Sys = &\mathit{TheBondHead} \parallel \mathit{ThePickArm} \parallel \mathit{TheVision1} \parallel \mathit{TheVision2} \\ &\parallel \mathit{TheRankingBin} \parallel \mathit{TheEjector} \parallel \mathit{TheWaferTable} \end{aligned} \tag{6.12}$$

Since parallel composition in CSP is associative, the system (6.12) can be rewritten as follows:

$$\begin{aligned} Sys = &(\mathit{ThePickArm} \parallel \mathit{TheWaferTable} \parallel \mathit{TheVision2}) \\ &(\mathit{TheVision1} \parallel \mathit{TheBondHead} \parallel \mathit{TheRankingBin} \parallel \mathit{TheEjector}) \end{aligned} \tag{6.13}$$

Consider the concurrent component $\mathit{ThePickArm} \parallel \mathit{TheWaferTable} \parallel \mathit{TheVision2}$. Using the above anti-extended components (that is, processes in (6.10), (6.11)), we apply the Decomposition of Concurrent Deterministic Theorem (Theorem 6.5.4) and conclude that

$$\begin{aligned} &(V2_{capture} \parallel WT_{vision}); \\ &\quad((PA_{unblock} \parallel WT_{unblock}); \\ &\qquad(\mathit{ThePickArm} \parallel \mathit{TheVision2} \parallel \mathit{TheWaferTable})) \\ &\text{anti-extends} \\ &\mathit{ThePickArm} \parallel \mathit{TheWaferTable} \parallel \mathit{TheVision2}. \end{aligned} \tag{6.14}$$

**Discussions** In formal terms, we apply a standard result of extension ($P \parallel T \wedge P \text{ extends } Q \implies P \parallel T \text{ extends } P \parallel Q$) so that the composition

$$\mathit{ThePickArm} \parallel \mathit{TheWaferTable} \parallel \mathit{TheVision2} \tag{6.15}$$

is anti-extended into the composition

$$\begin{aligned} &PA_{unblock};\mathit{ThePickArm} \\ &\quad\parallel (V2_{capture};\mathit{TheVision2} \\ &\qquad\parallel WT_{vision};(WT_{unblock};\mathit{TheWaferTable})) \end{aligned}$$

, and then apply the theorem discussed to the right operand of the first parallel composition

$$\begin{aligned} &(V2_{capture};\mathit{TheVision2} \\ &\parallel WT_{vision};(WT_{unblock};\mathit{TheWaferTable})) \end{aligned}$$

which results in the parallel composition as

$$(V2_{capture} \parallel WT_{vision});(\mathit{TheVision2} \parallel WT_{unblock};\mathit{TheWaferTable})$$

By standard extension rule, the sub-component being discussed (that is, the process in Equation (6.15)) should extend

$$\begin{array}{l} PA_{unblock};\mathit{ThePickArm} \\ \quad \parallel (V2_{capture} \parallel WT_{vision}); \\ \quad\quad (\mathit{TheVision2} \parallel WT_{unblock};\mathit{TheWaferTable}) \end{array} \tag{6.16}$$

We observe that the (observable) alphabets in both the processes $V2_{capture}$ and $WT_{vision}$ are irrelevant to the process $\mathit{ThePickArm}$ and every initial event of the process $PA_{unblock}$ is an initial event of the process $\mathit{TheWaferTable}$. Hence, a further anti-extension to process composition in Equation (6.16) is to "move" the process $V2_{capture} \parallel WT_{vision}$ (according to standard CSP law for parallel operator) to become the extending process in Equation 6.17.

$$\begin{array}{l} (V2_{capture} \parallel WT_{vision}); \\ \quad (PA_{unblock};\mathit{ThePickArm} \parallel \mathit{TheVision2} \parallel WT_{unblock};\mathit{TheWaferTable}) \\ \\ \text{extends} \\ \\ (V2_{capture} \parallel WT_{vision}); \\ \quad ((PA_{unblock} \parallel WT_{unblock}); \\ \quad\quad (\mathit{ThePickArm} \parallel \mathit{TheVision2} \parallel \mathit{TheWaferTable})) \end{array} \tag{6.17}$$

♦

Applying the anti-extension relation (6.14) to the whole system (6.12), and following the same arguments as in the discussion above, the system will extend the following composition:

$$\begin{array}{l} (V2_{capture} \parallel WT_{vision}); \\ \quad ((PA_{unblock} \parallel WT_{unblock}); \\ \quad\quad (\mathit{ThePickArm} \parallel \mathit{TheVision2} \parallel \mathit{ThePickArm} \\ \quad\quad\quad \parallel \mathit{TheVision1} \parallel \mathit{TheBondHead} \parallel \mathit{TheRankingBin} \parallel \mathit{TheEjector})) \end{array} \tag{6.18}$$

We repeat the abstraction and can conclude that the original system $Sys$ can be anti-extended into

$$\begin{array}{l} (V2_{capture} \parallel WT_{vision}); \parallel ((PA_{unblock} \parallel WT_{unblock}); \\ \quad (V2_{capture} \parallel WT_{vision}); \\ \quad\quad (WT_{adv} \parallel ( \\ \quad\quad\quad (PA_{pick} \parallel EJ_{push}); \\ \quad\quad\quad\quad (PA_{bond} \parallel EJ_{hold} \\ \quad\quad\quad\quad \parallel BH_{start};BH_{next};BH_{done}); \\ \quad\quad\quad\quad\quad (PA_{capture} \parallel V1_{capture}); \\ \quad\quad\quad\quad\quad\quad (PA_{signal} \parallel PA_{bin} \\ \quad\quad\quad\quad\quad\quad\quad \parallel RB_{bin} \parallel EJ_{release}));Sys) \end{array}$$

Since the process *Sys* extends the process $STOP$,

$$\begin{array}{l}
(V2_{capture} \parallel WT_{vision}); \parallel ((PA_{unblock} \parallel WT_{unblock}); \\
\quad (V2_{capture} \parallel WT_{vision}); \\
\quad\quad (WT_{adv} \parallel ( \\
\quad\quad\quad (PA_{pick} \parallel EJ_{push}); \\
\quad\quad\quad\quad (PA_{bond} \parallel EJ_{hold} \\
\quad\quad\quad\quad \parallel BH_{start};BH_{next};BH_{done}); \\
\quad\quad\quad\quad\quad (PA_{capture} \parallel V1_{capture}); \\
\quad\quad\quad\quad\quad\quad (PA_{signal} \parallel PA_{bin} \\
\quad\quad\quad\quad\quad\quad\quad \parallel RB_{bin} \parallel EJ_{release}));STOP)
\end{array}$$

anti-extends

*Sys*

Test cases can be constructed from this process (such as using a method similar to that in Example 6.4.6). ■

## 6.6 Chapter Summary

To resolve identified weaknesses of extension in a compositional and abstraction-oriented approach, we have formulated σ-extension and identified the necessary and sufficient conditions. We have also discussed a restricted sub-class of process specification so that these conditions can easily be satisfied. The restricted sub-class is that the successful terminations of processes should be deterministic.

Under these conditions, we achieve a few desirable properties. First, both the preceding and succeeding components of a given sequential composition can be anti-extended. The given composition should extend the sequential composition of their corresponding anti-extended processes. Secondly, for any given parallel composition whose concurrent components are sequential compositions, it should extend a parallel composition of anti-extensions of preceding components of corresponding sequential compositions, followed by a parallel composition of anti-extensions of corresponding succeeding components. In addition, any given sequential composition should σ-extend the preceding component of a resultant sequential composition. We shall further discuss future research directions in the next chapter.

# Chapter 7

# Discussions

In Chapter 6, we have discussed the details related to our technical work. In this chapter, we discuss the possible future directions and compare our results against related work.

## 7.1 The Chosen Semantic Model

We use the failures-divergences model of CSP proposed by Hoare [56] in this thesis. Over years, there are other variants of CSP published. The most closely related one is [87]. Roscoe [87] imposes additional restrictions on the original failures-divergences model and allows the process $SKIP$ to be an operand of $\Box$. He introduces an algebraic law to deal with the successful termination event. This new law is referred to as "$\Box - SKIP$ resolve". Informally, the law allows a process to opt to terminate while it can refuse any other events.

$$P \Box SKIP = (P \sqcap STOP) \Box SKIP$$

To accomplish this change, he also enforces the failures-divergences model with two new conditions. One is to ensure that any successful termination event can be included in divergences of a process only if a prefix of the trace in question is already a divergence trace. Another one is to explicitly state that the successful termination event should be associated with the whole alphabet set. Both of them are reasonable assumptions.

However, as discussed in Chapter 6, the conditions of the two major decomposition theorems are in general not very easy to verify. One reason is that it requires the checking of all refusals of related traces that defies the purpose of using software testing techniques instead of proving techniques to assure the software quality. Consequently, we propose in Chapter 6 a restricted class of CSP specifications so that difficult conditions can be automatically fulfilled. The restricted class is to limit successful terminations to be deterministic on the ground that software engineers would like systems to terminate successfully and deterministically in most circumstances. This proposal helps to degenerate some of the conditions into trivial cases. Nevertheless, if we take Roscoe's proposal into account, a deterministic successful termination may entail a non-deterministic progress of other events. On the other hand, excluding this rule will defy the purpose of Roscoe's enhancement of CSP. The potential of applying our results to Roscoe's work is open.

## 7.2 Applicability to Other Process Algebra

Both sequential composition ($P;Q$ in CSP sense) and the representation of successful termination (*SKIP* in CSP sense) are not supported in the original CCS or π-calculus. There are, however, projects that study termination in similar languages such as [2]. Both LOTOS and ACP, for instance, allow sequential composition and termination and hence our results can be translated into these two calculi. The unification of different branches of process algebras and Petri Net are examined by [11]. The identification of applicable sub-classes of our work using other process algebras should warrant more research.

In Chapter 5, we have identified five weaknesses of extension in terms of its ability for parallel and sequential compositions. In Chapter 6, we have refurbished extension by formulating σ-extension. We prove and demonstrate that one can construct an anti-extension hierarchy by the combination of applications of the Head, Tail and Decomposition of (Concurrent) Processes Theorems. Since it breaches various definitions and theorems, it corresponds to a weakness in extension. Investigations on the weaknesses of extension with respect to other CSP operators or operators in other calculi are open research.

## 7.3 Anti-Extension and Testing Criteria

We have developed an architecture that allows anti-extension to simplify the processes. A simplified process can be used to generate test cases based on some testing criteria. We impose no constraint on the kinds of testing criterion or anti-extension to be used so that the faults can be more easily revealed. More theoretical and empirical studies are worth conducting.

One of the ideas is to share the same test suites amongst the same process definitions used in the specification. For instance, the process *MultiplePick* below is an infinite state process[1] which requires any number of picks be followed by the same number of *post*. And during a transferring process, it may perform an *x-ray* check.

$$\begin{aligned}
&MultiplePick = (clear\ queue \rightarrow SKIP)\\
&\qquad \Box(pick \rightarrow (MultiplePick \Box XRay); Post\\
&XRay = transfer \rightarrow ((pack \rightarrow SKIP) \Box (x\text{-}ray \rightarrow check \rightarrow pack \rightarrow SKIP)\\
&Post = post \rightarrow (MultiplePick \Box XRay)
\end{aligned}$$

Since it is an infinite state process, covering all-state-transition is not a viable testing criterion. We observe, however, that the same process definitions *MultiplePick* and *XRay* are used. We may apply testing criteria applicable to these processes such as trace partitioning (that is, partition testing) to slice out a finite process which covers the test suites obtained. Suppose we use a partitioning and select 4 test cases whose trace lengths are at most 4 (ignoring the termination event). We can obtain the following test suites:

$$TestSuitePartitioning(MultiplePick) = \left\{ \begin{array}{l} \langle clear\ queue, \surd \rangle, \\ \langle pick, clear\ queue, transfer, x\text{-}ray \rangle, \\ \langle pick, transfer, pack, post \rangle, \\ \langle pick, transfer, x\text{-}ray, check \rangle \end{array} \right\}$$

[1] One can observe that after each *pick*, it may choose to behave like *MultiplePick* again. However, there is always a *Post* process stacked up by the sequential composition.

$$TestSuitePartitioning(XRay) = \left\{ \begin{array}{l} \langle transfer, pack, \surd\rangle, \\ \langle transfer, x\text{-}ray, check, pack, \surd\rangle \end{array} \right\}$$

Since there are two locations in the above specification that uses each of the processes *MultplePick* and *XRay*, we divide the two test suites into two parts separately. Based on the partitioned test suites, we can then construct the following test processes.

$$\begin{array}{l} MultiplePick_1 = (clear\ queue \rightarrow SKIP) \\ \quad \Box(pick \rightarrow clear\ queue \rightarrow transfer \rightarrow x\text{-}ray) \end{array}$$

$$\begin{array}{l} MultiplePick_2 = (pick \rightarrow transfer \rightarrow \\ \quad (pack \rightarrow post \rightarrow STOP)\Box(x\text{-}ray \rightarrow check \rightarrow STOP) \end{array}$$

$$XRay_1 = transfer \rightarrow pack \rightarrow SKIP$$

$$XRay_2 = transfer \rightarrow x\text{-}ray \rightarrow check \rightarrow pack \rightarrow SKIP$$

By the Decomposition of Processes Theorem, we can substitute the processes *MultiplePick* and *XRay* by the test processes accordingly, since the test processes are their corresponding anti-extensions. The aggregated anti-extension will become the process *System* as follows:

$$\begin{array}{l} System = MultiplePick' \\ MultiplePick' = (clear\ queue \rightarrow SKIP) \\ \quad \Box(pick \rightarrow (MultiplePick_1 \Box XRay_2);Post' \\ Post' = post \rightarrow (MultiplePick_2 \Box XRay_1) \end{array}$$

Hence, we generate test cases using all-state-transition-coverage criteria to produce the following test suite:

$$\left\{ \begin{array}{l} \langle clear\ queue, \surd\rangle, \\ \langle pick, clear\ queue, transfer, x\text{-}ray\rangle, \\ \langle pick, pick, clear\ queue, transfer, x\text{-}ray\rangle, \\ \langle pick, transfer, x\text{-}ray, check, pack, post, pick, transfer, pack, post\rangle, \\ \langle pick, transfer, x\text{-}ray, check, pack, post, pick, transfer, x\text{-}ray, check\rangle, \\ \langle pick, transfer, x\text{-}ray, check, pack, post, transfer, pack, \surd\rangle \end{array} \right\}$$

## 7.4 Our Framework and Test Case Generation Approaches

Our work consists of a compositional and abstract-oriented approach. Hence, on one hand, we benefit from using the compositional approach [108, 34] to address the state explosion problem. On the other, abstraction via anti-extension allows different process definitions in a specification to be "unified" into the same process definition. In this connection, via our framework, test case generation from this common abstract process can be applied to all corresponding process definitions in the specification. In other words, it provides opportunities to share testing resources (such as test cases) at an abstraction level. This advantage exists even for systems where state explosion is not a problem.

## 7.5 Modules with Manageable Number of States

There are non-trivial software components and programs whose explosion of states is manageably small. Let us discuss the following three scenarios.

I. A simple Java-based applet communicating with a web server and displaying banner advertisements repeatedly. It may consist of a few abstract states such as connection and disconnection, advertisement fetched from the web server and refreshing user-interfaces in a web-based application, although each of these states may be further refined into a number of implementation-oriented sub-states.

II. A simple and short method of classes in an object-oriented system. Such a method (whose purpose may be to provide a wrapping interface to add an object as an element of a container object) is usually trivial in terms of flow control and manipulation of object states.

III. A small, closed, simple and embedded micro-program inside a small device. These programs are usually fully optimized via hardware-software co-design to cope with the limited resources available.

Superficially, the analysis of compositional support and the construction of anti-extension do not seem to be absolutely necessary for these three scenarios. However, the Java-based applet example and the simple method in object-oriented system example are respective components of their enclosing systems. In the former case, the enclosing system is the web application, and in the latter case, it is the object-oriented system. In typical applications, the number of classes and number of components are non-trivially large and hence the problem of state explosion is serious. In such circumstances, our techniques will show their advantages in tackling state explosion and abstraction support.

On the other hand, there are situations, such as the embedded program example above, where the whole application is still manageably small. Our technique can be regarded as a pure abstraction-oriented technique based on anti-extension. As discussed in Sections 7.3 and 7.4, our technique provides an opportunity for sharing test resources. The sharing of test resources is feasible provided that processes in the specification can be abstracted into the same anti-extension. This benefit may not easily be obtained from the process components in the system directly.

## 7.6 Method Integration

There are a number of concurrent models in the literature and in practice. These notions may be formal (such as $OZ-CSP$ [96]) or informal (such as UML [88] and Java [73]). The integration of methods is one of the active research areas these days. In this section, we select three of the CSP-related approaches for discussion.

For Java, CSP has been proposed to be a concurrency model. In [104, 78], the CSP class library for Java, known as JCSP, provides an OCCAM3-like communicating process model for Java. Applications programs can make use of this library to implement necessary communications so as to manage thread execution and synchronization. Hence, for a program defining its communication behaviour using this kind of library, the finite state model for

concurrency can be extracted from the Java program [38]. One may apply our proposal on the extracted CSP model and generate test cases for this extracted model. Each generated test case will be used to construct a corresponding Java class. Since every communication alphabet[2] is extracted from the original Java program, testers can then initialize an object from the class for a test case and include the initialized object to participate into the communication channel (that is, the alphabet). There is other research work in this direction [55].

Besides programming languages, CSP has been used as building block in architectural modelling approaches [5]. [71] agrees that the UML state machine model [50, 51] is a powerful approach to define interactions in their own right. Within their modelling framework, they propose a subset of UML model to which CSP can be applied. Under this proposed subset of UML model, one can apply the techniques presented in our thesis.

In [96], the integration of CSP and Object-Z [95] has been studied. It shows that the notions of refinement in these two formal methods will agree with each other when CSP processes are associated with a restricted sub-class of the failures-divergences model. Under this restricted sub-class of semantic model, incremental development via refinement can be proved. Our work is based on the failures-divergences semantics, which is more general than the model required in [96]. This fact is encouraging. Object-Z is strong in modelling data. Software testers can use the CSP portion to generate abstract test cases and then use the Object-Z schema to find out the data partition where these test cases are indeed feasible.

## 7.7 Comparisons to Selected Work

To our best knowledge, our work is innovative in at least two aspects. They are (i) the use of the (anti-)extension hierarchy as the mean of abstraction to generate test cases for concurrent processes and (ii) the use of an enhanced form of extension (namely $\sigma$-extension) to support process composition in sequential and concurrent ways. Moreover, these two aspects are integrated together. For instance, without $\sigma$-extension, as illustrated in Problems Preceding and Succeeding, the anti-extension hierarchy cannot be built unless we always take the whole process into account. On the other hand, without anti-extension, $\sigma$-extension will serve no purpose unless the only interest is in the anti-$\sigma$-extension hierarchy. In fact, it is not difficult to observe that, when a test case is $\sigma$-extended by a specification, this test case may not conform to an implementation of the specification. We are both sorry and pleased to report that there is no closely comparable work in the literature.

### 7.7.1 Abstraction as Extension and Refinement

Stepney [97] observes that test cases should satisfy the pre-conditions of operation schema in Object-Z [45] and meets the post-conditions of the schema. She proposes an intuition-based approach. She partitions the pre-conditions of an operation in Object-Z schema and strengthens the corresponding post-conditions in such a way that post-conditions do not violate the original post-conditions in the specification for all input points in the partitioned input state. The steps can be recursively applied. The resultant schema is claimed to be an abstract test case schema for an operation schema to an implementation. She does not, however, investigate two important notions. First, she does not examine and formalize the testing relationship

[2] In JCSP, each alphabet is an instance of the communication channel class.

amongst test case schemas, specification schemas and their corresponding implementations for the software under test. Secondly, the way to link up different test case schema from different operations are not addressed. Despite all these weaknesses, the observation of testing as an abstraction is invaluable.

Derrick and Boiten [44] consider the data refinement relation in model based language between a specification and its refinement. They show two results. First, if there is an upward simulation from a concrete operation schema to an abstraction operation schema and all test case schemas are disjoint, then the disjunctive normal form of all test case schemas for the concrete operation will be upwardly refined by the concrete operation. Similarly, if there is a downward simulation from an abstract operation schema to a concrete operation schema, then the disjunctive normal form of all test case schemas for the concrete operation will be downwardly refined by the concrete operation. They also admit that, in general, the refined tests are not disjointed unless the data refinement relation is functional. Hence, their calculation approach cannot generally be applied recursively to different levels of refined specifications. They do not establish a formal relation (refinement or others) between a specification and a test case derived from the specification. In contrast, they show through counter-examples that a state-partitioned test case schema derived from a specification is not an anti-refinement[3] of the specification. Aichernig [4], on the other hand, shows that a few special forms of test case contract can be anti-refinements of its specification in refinement calculus.

Our work identifies that an anti-extension (Definition 4.1.4) from a specification should be a conformance test case to any implementation of the specification. Moreover, we have shown in Chapter 6 that our proposal is compositional and can be applied recursively. In addition to these, when an anti-extension from the specification includes every trace of the original specification, the anti-extension will satisfy the anti-refinement relation. It is because, in this case, the trace difference disappears and hence by definition, they unify to the same definition. This result is a direct consequence of Definitions 4.1.4 and 4.1.3. In other words, the test cases via anti-extensions are also anti-refinements under this restricted situation. In Chapter 5, like [44], we have illustrated via a counter-example that the conformance relation in the parallel composition approach (Section 2.2) will not be retained in general when the relationship between a test case and the specification is the refinement relation.

### 7.7.2 Parallel Composition

Bijl, Rensink and Tretmans [101] investigate the conformance problem for parallel composition of input-output labelled transition systems. They would like to find out the conditions to eliminate all integration testing of concurrent components provided that all these components can be tested to be conformant to the specification in isolation. General speaking, it is based on the assumption that an implementation of any concurrent component will not affect an implementation of another concurrent component, except using the stated input and output synchronizations. They also make a conclusive comment that omitting an input event in an implementation is not compatible with parallel composition. There is no further investigation to resolve this undesirable obstacle.

For the purpose of comparison in this thesis, an input-output labelled transition system can be considered as a CSP process without sequential composition in the process definition.

[3]The refinement relationship considered in that counter-example is a downward simulation, which is the common notion of refinement in model-based languages.

Events for input and output are synchronization alphabets. A program is conformant to a specification provided that, for every sequence of events common to a program and its specification, any feasible and immediate outputs from the program should be an output from its
specification. In other words, their work is based on a trace model[4] rather than in a more general failures-divergences model.

They show that an implementation of a parallel composition of concurrent components will conform to its specification provided that any pair of concurrent processes has to have distinct inputs and distinct outputs. In other words, (i) an environmental input or output has to synchronize with at most one process; (ii) an output event for a process cannot be synchronized with more than one process in the system; and (iii) an input event for a process cannot be synchronized with more than one process in the system.

For a fair comparison, we compare our special case for concurrent processes, namely Theorem for Decomposition of Concurrent Deterministic Processes, against their work.

I. First, our formalism does not restrict the relationship of input and output events amongst the concurrent processes in a system. We show that concurrent processes can make arbitrary synchronizations among themselves providing that the corresponding converging process for $\sigma$-extension will not deadlock immediately. In practice, it is not difficult to check whether a process will deadlock immediately.

II. Secondly, we find out the conditions on the anti-extension of concurrent processes so that whenever these concurrent processes form a parallel composition, the parallel composition of their corresponding anti-extensions should anti-extend the original parallel composition.

III. Finally, a more distinct feature of our approach compared with theirs is that their approach does not take into account the sequential composition of processes and considers each concurrent component as an input-output transition system. Our approach makes use of the compositional property of process algebra and allows abstraction to replace each component process in a sequential composition. In this way, our approach can keep the abstracted form of specification close to the original compositional structure as far as the desired behaviours are concerned. Hence, our approach is more intuitive.

## 7.8 Our Inspiration, Real-Life Examples and Testbed

Our inspiration, real-life examples and testbed of the theoretical results presented in this thesis are in the bonding machine domain when we are working for a University-Industry Collaboration Project. Our theoretical results, on the other hand, are general.

We suggest to conduct further industrial applications and experiments in other domains. In this way, the results presented in this thesis can be reused and the special attributes for the chosen domains may be uncovered and integrated to the results. It should further improve the notion of generating test cases via a compositional and abstraction-oriented mechanism. Conformity between specification and implementation is just one of many desirable properties

[4] A trace model is sufficient for deterministic specification.

that a software system exhibits. The ways to extend the relationship for composition abstraction to cover other desirable properties should be worth studying.

# Chapter 8

# Conclusions

Software concurrent systems such as electronic financial services, mobile games, network protocols, embedded control software and databases are very popular today. Rigorous testing of these systems is indispensable. Nevertheless, the state-explosion problem, in which the size of a system grows exponentially with the number of concurrent components, is one of the most severe obstacles in the generation of test cases.

In this research, we propose a new relation to assure the conformity of test cases for concurrent systems. We stipulate it in Communicating Sequential Processes (CSP), which is an excellent tool to study concurrent systems. Specification of concurrent systems is expressed as processes that are composed sequentially and concurrently. Our proposal supports both abstraction and process composition.

We review the related testing techniques in Chapter 2, and point out their weaknesses in addressing this problem. Some techniques generate test cases from a global (hierarchical and/or compositional) structure of concurrent systems. These global structures of the whole system are often too large to be used effectively. Some techniques use testing criteria applicable to individual concurrent components and heuristically compose local test suites together to form global test suites. The global test suites have to be further checked against all concurrent components to determine whether they may represent feasible synchronization sequences.

A formal framework is developed in Chapters 3, 4, 5 and 6. Chapter 3 introduces the CSP language. Chapter 4 defines formally the notion of conformance testing on CSP and justifies the use of an anti-extension hierarchy as the abstraction means over conformance and reduction. Chapter 5 investigates the weaknesses of anti-extension in a compositional approach. Chapter 6 presents our solutions to address these weaknesses and our recommendation on the type of specification to use the results.

Our approach decomposes given processes into sequential compositions of component processes. Each component can be combinations of abstract forms that substitute their corresponding extending components in a process to form aggregated abstract forms. Since the given processes, components and abstract forms are all processes, this approach can be applied recursively and compositionally. We prove that these aggregated abstract processes should be anti-extensions of the corresponding original processes under a few sufficient and necessary conditions. Hence, test cases generated from these processes will be conformance test cases for the implementations.

In Chapter 3, we have introduced the CSP language. CSP is excellent in modelling concurrency and composition. Besides, it shares the same underlying labelled transition system

model with other process algebras and Petri Net. Hence, our result can be readily translated to other formalisms.

In Chapter 4, we have examined the notion of conformance proposed by Brinksma [15, 13] to formally define conformance (Definition 4.1.2), reduction (Definition 4.1.3) and extension (Definition 4.1.4). The latter two are special kinds of conformance, which are partially ordered. Mutual conformance is shown to be testing equivalent and, in CSP, they are the same as process equivalence under the failures-divergences model [83, 102]. Moreover, refinement is equivalent to reduction in CSP [102]. The perfect alignment between CSP and the notion of conformance let us focus on testing rather than on handling irregularities of the underlying model.

We also justify extension amongst these three as the building blocks of our framework. Conformance is intransitive and hence conformance property cannot be secured in a hierarchy. We use Example 6.4.8 to illustrate that anti-reduction hierarchy is inappropriate. It suffers from the problems that it must include all failures and traces of the original process and, at the same time, it may introduce irrelevant operation sequences. All these are burdens from the testing point of view. Anti-extension, on the other hand, does not allow the introduction of operation sequences or actions, which is the basis for the generation of reliable test cases from anti-extended processes in our formalism. In short, only extension from a specification can provide abstract forms that are guaranteed to be conformed by an implementation of the specification.

Having selected extension, we further investigate it from the perspective of a compositional approach in Chapter 5. We identify five weaknesses of extension in this area. Extension has problems in both direct sequential composition (Problem Preceding and Problem Succeeding) and indirect sequential composition (Problem Decomposition). It also allows splitting of tasks (Problem Prefix). Parallel composition in general does not preserve extension (Problem Paralleling).

In Chapter 6, we have presented our solutions to resolve the identified weaknesses of extension in a compositional and abstraction-oriented approach. We identify the sufficient and necessary conditions for a restricted sub-class of process specifications to be compositionally for testing purposes. We also show that the condition for decomposition can be simplified for deterministic specifications.

We first generalize extension into sequential extension (Definition 6.2.1), which is shown to be partially ordered (Lemma 6.2.1), as a solution to Problem Prefix. We also show that extension is a special kind of sequential extension (Proposition 6.2.2) and testing equivalence is preserved (Proposition 6.2.3).

Furthermore, we restrict sequential extension in order to solve Problem Preceding and Problem Succeeding. We formulate the notions of $\sigma$-extension (Definiton 6.3.1) and converging process for $\sigma$-extension (Definition 6.3.4). Head Anti-Extension Theorem (Theorem 6.4.2) and Tail Anti-Extension Theorem (Theorem 6.4.1) show that extension properties are preserved and hence they solve Problem Preceding and Problem Succeeding, respectively. It is also possible to serialize processes by means of the application of Serial Anti-Extension Theorem (Theorem 6.4.3). Besides, we identify the sufficient and necessary conditions to assure the composition property to tackle Problem Decomposition and Problem Paralleling (Theorem 6.4.4 and Theorem 6.5.3, respectively). Based on the conditions, we propose a restricted sub-class of process specification so that these conditions can easily be satisfied. The restricted sub-class is that the successful terminations of processes should be

deterministic. We also show the theorems (Theorem 6.4.5 and Theorem 6.5.4) for deterministic specification.

Our contributions are three-fold.

- First, we identify extension as a superior tool for abstraction over conformance and reduction from the testing perspective. We also identify its weaknesses in a compositional approach.
- Secondly, we propose an abstraction-oriented compositional approach to generate test cases.
- Finally, we recommend restricting CSP-based specifications to a sub-class for better quality control.

It is a promising approach. We suggest conducting more empirical studies and industrial applications to further confirm and to take advantages of the usefulness.

# Appendix A

## A.1 A Few Results on Conformance, Extension and Reduction

In this section, we prove some standard results to be in this research about conformance, reduction and extension. We first summarize them in Table A.1.

| | | |
|---|---|---|
| $P$ conforms to $Q \wedge Q$ extends $R$ | $\Longrightarrow$ | $P$ conforms to $R$ |
| $P$ conforms to $Q \wedge Q$ reduces$R$ | $\not\Rightarrow$ | $P$ conforms to $R$ |
| $P$ extends $T_1 \wedge Q$ extends $T_2$ | $\Longrightarrow$ | $P \parallel Q$ extends $T_1 \parallel T_2$ |
| $P$ extends $T \wedge Q$ extends $T$ | $\Longrightarrow$ | $P \parallel Q$ extends $T$ |
| $P$ extends $T_1 \wedge P$ extends $T_2$ | $\Longrightarrow$ | $P$ extends $T_1 \parallel T_2$ |
| | | $P$ extends $P \parallel T$ |
| $P \parallel T \wedge P$ extends $Q$ | $\Longrightarrow$ | $P \parallel T$ extends $Q \parallel T$ |
| $P$ extends $T_1$ $\wedge$ $P$ conforms to $T_2$ $\wedge$ $Q$ extends $T_2$ $\wedge$ $Q$ conforms to $T_1$ | $\Longrightarrow$ | $P \Box Q$ extends $T_1 \Box T_2$ |

Table A.1: A summary of a few useful results in conformance, extension and reduction

First, we shall show the (in)transitivity amongst conformance, reduction and extension. In particular, we will show the following. Extension followed by conformance preserves the conformance. However, neither reduction nor conformance followed by conformance will in general preserve the indirect conformance.

**Proposition A.1.1** *Let P, Q and R be processes sharing the same alphabet sets. Suppose that P conforms to Q and Q extends R. P will conform to R.*

*Proof:* Consider a failure $(s,X)$ of $P$. Suppose that $s$ is a trace of $R$. Since $Q$ extends $R$, $s$ should be a trace of $Q$. Moreover, as $P$ conforms to $Q$, by definition of conformance, $(s,X)$ should be a failure of $Q$. In a similar manner, by definition of extension, $(s,X)$ should be a failure of $R$. The divergences condition follows directly from the definitions. $\Box$

**Proposition A.1.2** *Let P, Q and R be processes sharing the same alphabet sets. Suppose that P conforms to Q and Q reduces R. P, in general, will not conform to R.*

*Proof:* We prove this using a counter-example. Consider three processes $P$, $Q$ and $R$ as follows:

$$P = (a \rightarrow b \rightarrow STOP)\Box(b \rightarrow c \rightarrow STOP)$$
$$Q = (a \rightarrow b \rightarrow STOP)$$
$$R = (a \rightarrow b \rightarrow STOP) \sqcap (b \rightarrow a \rightarrow STOP)$$

We observe that $P$ conforms to $Q$ and $Q$ reduces $R$. However, $P$ refuses to proceed to event $a$ after the trace $\langle b \rangle$; whereas $\langle b, a \rangle$ is a trace of $R$. Hence, $(\langle b \rangle, \{a\})$ is a failure of $P$, but it is not a failure of $R$. $\Box$

**Proposition A.1.3** *Let P, Q and R be processes sharing the same alphabet sets. Suppose that P conforms to Q and Q conforms to R. P, in general, will not conform to R.*

*Proof:* It follows directly from Definition 4.1.2, Definition 4.1.3 and Proposition A.1.2. $\Box$

We further show some properties of extension in parallel composition. We show that extension can be retained in parallel composition.

**Proposition A.1.4** *Suppose a process $P$* extends *a process $T_1$ and a process $Q$* extends *a process $T_2$. Then the process $P \parallel Q$* extends *the process $T_1 \parallel T_2$*

$$P \text{ extends } T_1 \wedge Q \text{ extends } T_2 \wedge \alpha P = \alpha Q \implies P \parallel Q \text{ extends } T_1 \parallel T_2$$

*if P and Q carry identical alphabet sets.*

*Proof:* For extension to hold, $P$ and $T_1$, as well as $Q$ and $T_2$, should carry identical alphabet sets, respectively. Given that the alphabet sets of $P$ and $Q$ are identical, all these four processes will carry identical alphabet sets. By the definition of parallel operator, the alphabet set of $P \parallel Q$ should be the same as the alphabet set of $T_1 \parallel T_2$. Moreover, because of the equivalence of alphabet sets, any trace of $T_1 \parallel T_2$ should be a trace of $T_1$ and that of $T_2$ at the same time, which, in turn, should be a trace of $P$ and that of $Q$, respectively. By definition, it should be a trace of $P \parallel Q$. For the failure relationship between $P \parallel Q$ and $T_1 \parallel T_2$, there are three sub-cases. Suppose that $(s, X)$ is a failure of $P \parallel Q$. (1) Suppose that $s$ is neither a trace of $T_1$ nor that of $T_2$, it will be irrelevant to the extension relation between $P \parallel Q$ and $T_1 \parallel T_2$. (2) Suppose that $s$ is both a trace of $T_1$ and that of $T_2$. By the definition of parallel operator, the refusal set $X$ of $P \parallel Q$ should not exceed the pairwise union of corresponding refusal sets from $P$ and $Q$ associated with the trace $s$. Given that $P$ and $Q$ extend $T_1$ and $T_2$ respectively. By the definition of extension (Def. 4.1.4), all failures associated with the trace $s$ from each of them should be included in $T_1 \parallel T_2$ if it is a trace of $T_1 \parallel T_2$. And so, any pairwise union of corresponding refusal sets from $P$ and $Q$ for trace $s$ should be a refusal set of $T_1 \parallel T_2$ for trace $s$. Consequently, $(s, X)$ is a failure of $T_1 \parallel T_2$. (3) The last sub-case is that $s$ is either a trace of $T_1$ or a trace of $T_2$, but not both. Hence, $s$ will not be a trace of $T_1 \parallel T_2$ (which is irrelevant to extension). The proof for divergence condition is trivial. $\Box$

By setting both $T_1$ and $T_2$ as $T$, we have the following corollary.

**Corollary A.1.1** *If processes P and Q* extends *a process T, then the process* $P \parallel Q$ extends *the process T. Mathematically,*

$$P \text{ extends } T \wedge Q \text{ extends } T \implies P \parallel Q \text{ extends } T$$

*Proof:* It is obvious that $\alpha T$, $\alpha P$ and $\alpha Q$ are the same; otherwise the two given extension relations will not hold. And the result follows. [1] □

Another setting is to put both $P$ and $Q$ as the same process. In this case, the alphabet equality constraint will be satisfied, and by property that $P \parallel P = P$, we have the following corollary.

**Corollary A.1.2** *If a process P* extends *processes* $T_1$ *and* $T_2$*, then the process P* extends *the process* $T_1 \parallel T_2$*.*

$$P \text{ extends } T_1 \wedge P \text{ extends } T_2 \implies P \text{ extends } T_1 \parallel T_2$$

**Proposition A.1.5** *If the alphabet sets of processes P and Q are equal, then P extends* $P \parallel Q$*.*

*Proof:* It is obvious. For process $P \parallel Q$, every event should be symphonized; or else it would not be observable. Hence, every trace of $P \parallel Q$ should be a trace of $P$. From the definition of failures for $\parallel$-operator, all failures of $P$ should be failures of $P \parallel Q$, and divergences alike. And the result follows. □

Setting $P$ as $P_1 \parallel Q_1$, and $Q$ as $P_2 \parallel Q_2$ in proposition A.1.5, we have the following corollary.

**Corollary A.1.3** *If the alphabet sets of processes* $P_1 \parallel Q_1$ *and* $P_2 \parallel Q_2$ *are equal, then* $P_1 \parallel Q_1$ *extends* $P_1 \parallel P_1 \parallel Q_1 \parallel Q_2$*.*

**Proposition A.1.6** *If a process P* extends *a process Q, then the process* $P \parallel R$ extends *the process* $Q \parallel R$ *for some process R providing that P and Q shares the same alphabet sets.*

*Proof:* Trivial. □

**Corollary A.1.4** *If a process P* extends *processes* $P_1$ *and* $P_2$*, and a process Q* extends *processes* $Q_1$ *and* $Q_2$*, then both P and Q* extend *the process* $(P_1 \parallel P_2 \parallel Q_1 \parallel Q_2)$ *providing that P and Q share the same alphabet sets.*

[1] A standalone proof. It is obvious that $\alpha T$, $\alpha P$ and $\alpha Q$ are the same; otherwise the two given extension relations will not hold. Moreover, any trace of $P \parallel Q$ should be either a trace of $P$ or a trace of $Q$ which, by the definition of extension (Def. 4.1.4), should be a trace of $T$. Furthermore, supposed that $(s,X)$ is a failure of $P \parallel Q$. Since both $P$ and $Q$ carry the same alphabet sets, trace $s$ should be both a trace of $P$ and a trace of $Q$. By definition of the parallel operator, the refusal set $X$ of $P \parallel Q$ should not exceed the pairwise union of corresponding refusal sets from $P$ and $Q$ associated with the trace $s$. Given that both $P$ and $Q$ extend $T$. By the definition of extension (Def. 4.1.4), all failures associated with the trace $s$ from each of them should be included in $T$ if it is a trace of $T$. And so, the pairwise union of corresponding refusal sets from $P$ and $Q$ for trace $s$ should be a refusal set of $T$ for trace $s$. Consequently, either $(s,X)$ is a failure of $T$ or trace $s$ is irrelevant to $T$, and the result follows.

*Proof:* Process $P \parallel Q$, by Proposition A.1.6, *extends* $P_1 \parallel Q$ which by the same proposition, *extends* $P_1 \parallel Q_1$. Similarly, $P \parallel Q$ also *extends* $P_2 \parallel Q_2$. Since $P \parallel Q = (P \parallel Q) \parallel (P \parallel Q)$, by Proposition A.1.4 and A.1.6, the result follows. □

An interpretation of this corollary is that it is an incremental way of test process construction for a set of processes running in parallel. That is, if we construct a set $\mathbb{U}$ of test processes $U_1, \cdots, U_n$ for process $P$ and a set $\mathbb{V}$ of test processes $V_1, \cdots, V_m$ for process $Q$ such that the $P$ extends all $U_i$ and $Q$ extends all $V_j$, then we can select one or more processes from $\mathbb{U}$ and one or more processes from $\mathbb{V}$ and put all these selected processes in parallel to form a test process for process $P \parallel Q$. Suppose that $\mathbb{W}$ is the set of all possible test processes for process $P \parallel Q$ constructed from $\mathbb{U}$ and $\mathbb{V}$. Composing any number of processes from $\mathbb{W}$ in parallel will form a process which should be in set $\mathbb{W}$ too. Hence, it should extend process $P \parallel Q$.

**Proposition A.1.7** *If a process $P$* extends *a process $T_1$ and a process $Q$* extends *the process $T_2$, then the process $P \Box Q$* extends *process $T_1 \Box T_2$, providing that $Q$ conforms to $T_1$ and $P$ conforms to $T_2$.*

$$P \textit{ extends } T_1 \wedge P \textit{ conforms to } T_2 \wedge Q \textit{ extends } T_2 \wedge Q \textit{ conforms to } T_1 \\ \implies P \Box Q \textit{ extends } T_1 \Box T_2$$

*Proof:* It is obvious that the alphabet set of $P \parallel Q$ is the same as that of $T_1 \parallel T_2$. Moreover, by the definition of the choice operator, any trace of $T_1 \parallel T_2$ should be either a trace of $T_1$ or that of $T_2$, and hence, either a trace of $P$ or a trace of $Q$ respectively. Hence, such traces should also be traces of $P \parallel Q$. For a trace of $P \parallel Q$ which is unique to either $P$ or $Q$, the related failures should be come from either $P$ or $Q$, but not both. Since the extension holds, those failures should be failures of $T_1$ or $T_2$ if the trace is also a trace of $T_1$ or $T_2$ respectively. In case that a trace of $P \parallel Q$ is common to both $P$ and $Q$. The corresponding failures for $P \parallel Q$ would be the pairwise union of the failures of $P$ and $Q$ for that trace. It would also be failures of $T_1 \parallel T_2$ if it is also a common trace of $T_1$ and $T_2$ owing to given extension relations. On the other hand, supposed that only one, say $T_1$, of $T_1$ and $T_2$ contains such a trace. Since $Q$ conforms to $T_1$, the failures of $Q$ for that trace should be a part of failures of $T_1$ for that trace. By the same token, it is also the case if the trace only appears in $T_2$ rather than in $T_1$. Consequently, the failures related to that trace for $P \parallel Q$ should also be failures of $T_1 \parallel T_2$. The case, that the trace is neither a trace of $T_1$ nor a trace of $T_2$, is irrelevant. And the result follows. □

Since extension is a special case of conformance, by rewriting $T_1$ and $T_2$ as $T$, we have the following corollary.

**Corollary A.1.5** *If a process $P$* extends *a process $T$ and a process $Q$ also* extends *the process $T$, then the process $P \Box Q$* extends *the process $T$.*

$$P \textit{ extends } T \wedge Q \textit{ extends } T \implies P \Box Q \textit{ extends } T$$

**Proposition A.1.8** *Let $P$, $Q$ and $T$ be processes. If the process $P$ extends the process $Q$, then the process $P \parallel T$ extends the process $Q \parallel T$.*

*Proof:* Consider a failure $(s, X \cup Y)$ of the process $P \parallel T$. Let $u = s \upharpoonright \alpha P$ and $v = s \upharpoonright \alpha T$. Without the loss of the generality, assume $(u, X)$ be a failure of the process $P$ and $(v, Y)$ be a

failure of the process $T$. Suppose that $s$ is a trace of the process $Q \parallel T$. Given that the process $P$ extends the process $Q$. $(u, X)$ should be a failure of the process $Q$. According to the definition of the parallel operator ($\parallel$), $(s, X cup Y)$ should be a failure of the process $Q \parallel T$. The traces and divergences inclusion followed directly from the definition. $\square$

# Bibliography


[1] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2-3):225–241, 1987.

[2] L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *Journal of the ACM*, 39(1):147–187, January 1992.

[3] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991. An earlier version with a same title appeared in *Proc. of the IFIP WG 6.18th International Symposium on Protocol Specification, Testing, and Verification, June 1988.*

[4] B.K. Aichernig. Test-design through abstraction – a systematic approach based on the refinement calculus. *Journal of Universal Computer Science*, 7(8):710–735, August 2001.

[5] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[6] J.C.M. Baeten and W.P. Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 1990.

[7] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 1990.

[8] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Control*, 60:109–137, 1984.

[9] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, Amsterdam, 2001.

[10] Philip J. Bernhard. A reduced test suite for protocol conformance testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 3(3):201–220, 1994.

[11] Eike Best, Raymond Devillers, and Maciej Koutny. A unified model for nets and process algebra. *Handbook of Process Algebra*, pages 873–944, 2001.

[12] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[13] E. Brinksma. A theory of the derivation of tests. In *Protocol Specification, Testing and Verification VIII: Proceedings of the 8th IFIP WG 6.1 International Symposium*, pages 63–74, Amsterdam, 1988. North-Holland.

[14] E. Brinksma, L. Heerink, and J. Tretmans. Factorized test generation for multi input-output transition systems. In A. Petrenko and N. Yevtushenko, editors, *Int. Workshop on Testing of Communicating Systems 11*, pages 67–82. Kluwer Academic Publishers, 1998.

[15] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. *Protocol Specification, Testing and Verification VI, IFIP 1987*, pages 349–360, 1987.

[16] Ed Brinksma and Jan Tretmans. Testing transition systems: an annotated bibliography. pages 187–195, 2001.

[17] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(7):560–599, 1984.

[18] Peter Buchholz. Hierarchical structuring of superposed GSPNs. *IEEE Transactions on Software Engineering*, 25(2):166–181, March/April 1999. Special Section: Seventh International Workshop on Petri Nets and Performance Models (PNPM'97).

[19] Peter Buchholz and Peter Kemper. Hierarchical reachability graph generation for Petri Nets. *Formal Methods in Systems Design*, 18(3):281–315, 3 2002.

[20] Giacomo Buonanno, Franco Fummi, and Donatella Sciuto. An extended-UIO-based method for protocol conformance. *Journal of Systems Architecture*, 46(3):225–242, jan 2000.

[21] D.C. Kung C.-H. Liu and P. Hsia. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, 2001.

[22] R. H. Carver and K.C. Tai. Static analysis of concurrent software for deriving synchronization constraints. In *11th International Conference on Distributed Computing Systems*, pages 544–551, Washington, D.C., USA, May 1991. IEEE Computer Society Press.

[23] R. H. Carver and K.C. Tai. Test sequence generation from formal specifications of distributed programs. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 360–367, Los Alamitos, CA, USA, May 30–June 2 1995. IEEE Computer Society Press.

[24] R.H. Carver and K.C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*, 24(6):471–490, 1998.

[25] Richard H. Carver and Ronnie Durham. Integrating formal methods and testing for concurrent programs. In *Compass '95: 10th Annual Conference on Computer*

*Assurance*, pages 25–34, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[26] W. K. Chan, T. Y. Chen, and T. H. Tse. An overview of integration testing techniques for object-oriented programs. In *Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science (ICIS 2002)*, pages 696–701. International Association for Computer and Information Science, Mt. Pleasant, Michigan, 2002.

[27] K.H. Chang, S.S. Liao, S.B. Seidman, and R. Chapman. Testing object-oriented programs: from formal specification to test scenario generation. *The Journal of Systems and Software*, 42:141–151, 1998.

[28] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10(1):56–109, 2001.

[29] H.Y. Chen, T.H. Tse, F.T. Chan, and T.Y. Chen. In Black and White: an integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.

[30] T.Y. Chen and M.F. Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, 60(3):135–141, 1996.

[31] T.Y. Chen and C.K. Low. Error detection in C++ through dynamic data flow analysis. *Software: Concepts and Tools*, 18(1):1–13, 1997.

[32] T.Y. Chen and P.C. Poole. Dynamic dataflow analysis. *Information and Software Technology*, 30:497–505, 1988.

[33] T.Y. Chen, T.H. Tse, and Y.T. Yu. Proportional sampling strategy: a compendium and some insights. *Journal of Systems and Software*, 58(1):63–79, 2001.

[34] S.C. Cheung and J. Kramer. Context constraints for compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology*, 5(4):334–376, 1996.

[35] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM sigplan 2002 Conference on Programming language design and implementation*, pages 258–269, Berlin, Germany, Jun 2002. ACM Press New York, NY, USA.

[36] T.S. Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[37] J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, 1996.

[38] J.C. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Transactions on Software Engineering and Methodology*, 9(1):51–93, 2000.

[39] E. Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3(2):129–141, 1991.

[40] A. T. Dahbura, K. K. Sabnani, and M.U. Uyar. Formal methods for generating protocol conformance test sequences. *Proceedings of the IEEE*, 78(8):1317–1325, 1990.

[41] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming*. Academic Press, London, 1972.

[42] M.E. Delamaro, J.C. Maldonado, and A.P. Mathur. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

[43] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[44] J. Derrick and E. Boiten. Testing refinements of state-based formal specifications. *Software Testing, Verification and Reliability*, 1:27–50, 1999.

[45] R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z.* Palgrave Publishers, Basingstoke, UK, 2000.

[46] Alain Finkel and Pierre McKenzie. Verifying identical communicating processes is undecidable. *Theoretical Computer Science (TCS)*, 174:217–230, 1997.

[47] Riccardo Focardi and Roberto Gorrieri. The Compositional Security Checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9):550–571, September 1997.

[48] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988.

[49] P.G. Frankl and E.J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, 1993.

[50] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[51] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[52] Jean Hartmann, Claudio Imoberdorf, and Michael Meisinger. UML-based integration testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 60–70. ACM Press New York, NY, USA, 2000.

[53] M. Hennessy. *Algebraic Theory of Processes*. MIT Press, Cambridge, Massachusetts, 1988.

[54] F. C. Hennie. Fault detecting experiments for sequential circuits. In *Proceedings of 5th Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110. IEEE, New York, Nov 1964.

[55] Gerald Hilderink, Jan Broenik, Wiek Vervoort, and Andre Bakkers. Communicating Java Thread (CJT). http://www.ce.utwente.nl/javacpp/, 2004.

[56] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice-Hall, London, 1985.

[57] G.H. Hwang, K.C. Tai, and T.L. Hunag. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 15(4):493–510, 1995.

[58] ISO 8807. LOTOS: a formal description technique based on the temporal ordering of observed behaviour. International Standard ISO 8807, International Organization for Standardization, Geneva, 1988.

[59] ISO 9646. OSI conformance testing methodology and framework (ISO DP 9646). International Standard ISO, Open Systems Interconnection, 9464, International Organization for Standardization, Geneva, 1988.

[60] Eric Y. T. Juan, Jeffrey J. P. Tsai, and Tadao Murata. Compositional verification of concurrent systems using petri-net-based condensation rules. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(5):917–979, 1998.

[61] B. Karacali and K.C. Tai. Automated test sequence generation using sequencing constraints for concurrent programs. In *Proceedings of 4th International Symposium on the Software Engineering for Parallel and Distributed Systems (PDSE '99)*, pages 97–108. IEEE Computer Society, Los Alamitos, California, USA, 1999.

[62] T.H. Kim, I.S. Hwang, M.S. Jang, and J.Y. Lee. Test case generation of a communication protocol by an adaptive state exploration. *Computer Communications*, 24(13):1242–1255, 2001.

[63] Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings: Software*, 146(4):187–192, 1999.

[64] James D. Kindrick, John A. Sauter, and Robert S. Matthews. Improving conformance and interoperability testing. *StandardView*, 4(1):61–68, 1996.

[65] Z. Kohavi. *Switching and Finite Automata Theory (2nd ed.)*. MsGraw-Hill, 1978.

[66] Pramod V. Koppol, Richard H. Carver, and K.C. Tai. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering*, 28(6):607–623, 2002.

[67] Guy Leduc. Failure-based congruences, unfair divergences and new testing theory. In Son T. Vuong and Samuel T. Chanson, editors, *Protocol Specification, Testing and Verification XIV, Proceedings of the Fourteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Vancouver, BC, Canada, 1994*, volume 1 of *IFIP Conference Proceedings*, pages 252–267. Chapman & Hall, 1995.

[68] David Lee, Krishan K. Sabnani, David M. Kristol, and Sanjoy Paul. Conformance testing of protocls specified as communicating finite state machines — a guided random walk based approach. *IEEE Transactions on Communications*, 44(5):631–640, 1996.

[69] Jeeny J. Li and Eric W. Wong. Automatic test generation from communicating extended finite state machine (CEFSM)-based models. In *Proceedings of the Fifth IEEE International symposium on object oriented real time distributed computing (ISORC'02)*, pages 181–185. IEEE Computer Society, Los Alamitos, California, USA, 2002.

[70] Gang Luo, Gregor von Bochmann, and Alexandre Petrenko. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.

[71] Nenad Medvidovic, David S. Rosenblum, Dvaid F. Redmiles, and Jason E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, 2002.

[72] Microsoft. Introduction to conformance testing. http://msdn.microsoft.com /library/default.asp?url=/library/en-us/oledb/htm/conformancetesting_3.asp, July 2003.

[73] Sun Microsystems. The java language specification. http://java.sun.com/docs/books/jls, 2000.

[74] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[75] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1989.

[76] R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1):1–40,41–77, September 1992.

[77] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition tours. *Proc. of IEEE Fault Tolerant Computing Conference*, pages 238–243, 1981.

[78] Christopher H. Nevison. Seminar: safe concurrent programming in Java with CSP. In *The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, page 367. ACM Press, 1999.

[79] S.C. Ntafos. On comparisons of random, partition, and proportional partion testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, 2001.

[80] C. O'halloran. A calculus of information flow. In *Acte de ESORICS 90, Toulouse*, pages 147–159, October 1990.

[81] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241–284, 1987.

[82] Benjamin C. Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-calculus. *Journal of the ACM (JACM)*, 47(3):531–584, 2000.

[83] David H. Pitt and David Freestone. The derivation of conformance tests from LOTOS specifications. *IEEE Transactions on Software Engineering*, 16(12):1337–1343, Dec 1990.

[84] Roy Rada. Who will test conformance? *Communications of the ACM*, 39(1):19–22, 1996.

[85] T. Ramalingom, A. Das, and K. Thulasiraman. A unified test case generation method for the EFSM model using context independent unique sequences. In *Proceedings of 8th International Workshop on Protocol Test Systems (IWPTS'95)*, pages 289–306, Evry,France, Sep 1995.

[86] Joylyn Reed and Raymond T. Yeh. Specification and verification of liveness properties of cyclic, concurrent processes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(1):156–177, 1988.

[87] A.W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1997.

[88] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, Reading, Massachusetts, 1998.

[89] K. Saleh, A.A. Boujarwah, and J. Al-Dallal. Anaomaly detection in concurrent java programs using dynamic data flow analysis. *Information and Software Technology*, 43:973–981, 2001.

[90] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[91] Steve Schneider. May testing, non-interference, and compositionality. Technical Report CSD-TR-00-02, Department of Computer Science, Royal Holloway, University of London, Department of Computer Science, Egham, Surrey TW20 0EX, England, January 2001.

[92] Burak Serdar and Kuo-Chung Tai. A new approach to checking sequence generation for finite state machines. In *Testing of Communcating Systems XIV: Proceedings of IFIP TC6/WG6.1 14th International Conference on Testing of Communicating Systems*, Ina Schieferdecker and Hartmut Konig and Adam Wolisx (eds.), pages 391–404, Boston, 2002. Kluwer Academic Publishers.

[93] Y.-N. Shen, F. Lombardi, and A. T. Dahbura. Protocol conformance testing using multiple UIO sequences. In *Proc. 9th Intl. Symp. on Protocol Specification, Testing and Verification (IFIP WG 6.1)*, pages 131–143, Enschede, The Netherlands, June 1989.

[94] D. P. Sidhu and T.-K. Leung. Formal methods for protocol testing: A detailed study. *IEEE Transactions on Software Engineering*, 15(4):413–426, April 1989.

[95] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Boston, 2000.

[96] G. Smith and J.M. Derrick. Specification, refinement and verification of concurrent systems. an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[97] S. Stepney. Testing as abstraction. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation, 9th International Conference of Z Users, Limerick, Ireland, September 7–9, 1995, Proceedings*, volume 967 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1995.

[98] Q. Tan, A. Petrenko, and G. von Bochmann. Testing trace equivalence for labeled transition systems. Technical report, 1995.

[99] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[100] Piyu Tripathy and Behcet Sarikaya. Test generation from LOTOS specifications. *IEEE Transactions on Computers*, 40(4):543–552, Apr 1991.

[101] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with IOCO. In *Third International Workshop on Formal Approaches to Testing of Software, (FATES 2003)*, volume 2931 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, Berlin, Germany, 2004.

[102] R.J. van Glabbeek. The linear time — branching time spectrum I. *Handbook of Process Algebra*, pages 1–99, 2001.

[103] Gregor von Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 international symposium on Software testing and analysis*, pages 109–124. ACM Press, 1994.

[104] Peter Welch. Communicating Sequential Processes for Java$^{TM}$ (JCSP). http://www.cs.kent.ac.uk/projects/ofa/jcsp/, 2004.

[105] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineer ing (ICSE '95)*, pages 41–50, Los Alamitos, California, 1995. IEEE Computer Society Press.

[106] W.E. Wong, J.R. Horgan, A.P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 522–528, Los Alamitos, California, 1997. IEEE Computer Society Press.

[107] M. Yao, A. Petrenko, and G. von Bochmann. Conformace testing of protocol machines without reset. In *Protocol Specification, Testing and Verification XIII: Proceedings of the 13th IFIP WG 6.1 International Symposium*, A.A.S. Danthine, G. Leduc, and P. Wolper (eds.), pages 241–256, Amsterdam, 1993. North-Holland.

[108] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 49–59. ACM Press, 1991.

[109] H. Yoon, B. Choi, and J.O. Jeon. Mutation-based inter-class testing. In *Proceedings of Asia Pacific Software Engineering Conference (APSEC '98)*, pages 174–181. IEEE Computer Society, Los Alamitos, California, USA, 1998.

*All examples on bonding machines presented in this thesis are extracted from the project entitled "VITAMIN: a VIsual authoring Toolset with Automatic code generation capability for ManufacturIng automatioN" supported in part by a funding of ASM Assembly Automation Ltd. and a grant of the Innovation and Technology Commission (Project No. UIM/77). The examples have been modified so as not to disclose any confidential information of the company.* ♡