JAIST Repository
https://dspace.jaist.ac.jp/

Title	フィーチャー指向ソフトウェアにおける増分的一貫性検証
Author(s)	Nguyen, Truong Thang
Citation	
Issue Date	2005-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/984
Rights	
Description	Supervisor:片山　卓也，情報科学研究科，博士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Incremental Verification of Consistency in Feature-Oriented Software

by

Nguyen Truong Thang

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

Supervisor: Professor Takuya Katayama

School of Information Science
Japan Advanced Institute of Science and Technology

September, 2005

Abstract

Object-oriented (OO) software technology is by far the most effective approach for software development. Although, it contributes greatly to the advance of software industry, it still suffers from several drawbacks, especially its well-known fragility for capturing *concerns* [37] crosscutting multiple objects. The crosscutting factor is due to the tangled code implementing the concern among those objects. As a result, the ultimate result is high cost.

As an extension of the OO technology, aspect-oriented software development (AOSD) comes to solve this problem. It emerges as a promising future software paradigm with high reusability, openness among many other advanced characteristics. Throughout this dissertation, the terms concern [37], aspect [17], feature [28, 29], collaboration [10, 27, 35], hyperslice [36], component are interchangeably used. Even though these terms address different levels of concern granularity at various stages in the software lifecycle, they can be equivalently treated. More specifically, the dissertation inclines toward the standard definition of hyperslices which are functions or services offered by the system through the collaboration of several objects.

Feature-oriented software is a special case of AOSD in which a feature corresponds to a functional requirement of the system. In essence, a system is constructed by composing several features. Each feature is independently specified, implemented and maintained. Those features communicate with each other via well-defined interfaces. Based on this design style, a system can be flexibly built depending on the decision of whether or not to include a specific feature.

However, to realize such an ideal software paradigm, one of the key issues is to ensure that those separately specified and implemented features do not conflict to each other when composed - the *feature consistency* issue. This dissertation takes a formal approach toward this problem and its related issues. Basically, the dissertation consists of three parts.

First, a formal model of feature-oriented software is introduced in which each feature is separately encapsulated by a state transition model. The formal model of a feature also provides the interface at which other features are plugged with. With respect to some rules, interface states among different features are mapped to each other so that member features can be composed together to form the target composite system.

Second, the core of this dissertation is about the theoretical foundation of consistency verification among features. A feature is *consistent* with another if during its execution, it does not violate the inherent CTL properties of the latter feature. The verification approach - open incremental model checking (OIMC) - is done via assumption model checking [6, 20]. The difference of OIMC when compared with the other modular model checking techniques [13, 18, 20, 31] is in its *incremental openness*. Under the approach, systems are considered as ever-evolving. A typical evolution scenario is to incorporate new modules (*extensions*) to an existing system (*base*) consistently. In such a circumstance, the new method is executed only in the extension. On the contrary, traditional model checking

methods are required to re-run over the whole system, even though possibly in modular fashion. This is the most prominent feature of this challenging area. A sound theoretical foundation of OIMC is proposed in the dissertation. Its goal is to provide necessary conditions under which the extension components do not violate key properties of the base component. In addition, the soundness and scalability of this incremental verification approach are also discussed. Except in some extreme cases of feature composition, the result delivered by OIMC is sound. Moreover, under the consistency condition, OIMC also preserves its complexity for any subsequent extensions, i.e. scalable.

The final part of the dissertation briefs some possible applications of the proposed theory such as the component specification and composition. In addition, it shows the relationship between the formal specification model of feature-oriented software and its implementation. More specifically, this part shows a basic code-transforming mechanism from the formal model into a specific programming language, e.g. Java. Because features are independently specified, their respective codes are independently generated according to some transformation rules. At the end, the corresponding codes of the features are composed [36] or woven [38] together in a well-defined manner.

Acknowledgments

This thesis has been prepared according to TeX version 3.14 [19] and compiled with LaTeX2e. This software makes the typesetting for scientific report much easier. The illustrated example is tested with gcc 3.3.2 and Java 2 Standard Edition 1.4.2 on Fedora Core 1 Linux operating system.

This thesis is a three-year work. It can not be completed without the help and inspiration of so many people. First of all, the author wants to thank all colleagues at the Foundations of Software Lab - JAIST. Without their excellent cooperation and facility maintenance, this work could have not been completed as it is. In particular, the author wishes to express his sincere gratitude to his principal adviser - Professor Takuya Katayama of Japan Advanced Institute of Science and Technology - for his constant encouragement and kind guidance. In addition, the LAN and computing facility served during the thesis compilation are very well managed and constantly upgraded to the best equipment available by Associate Toshiaki Aoki and many others such as Hayato Kawashima, Mitsutaka Okazaki just to name a few.

During the PhD course, the author has received much support from several people from industry especially Tokyo Research Lab. (TRL) of IBM Japan Corp; R&D Center (RDC) of Toshiba Corp. In particular, his sincere gratitude is sent to Autonomic Computing group at TRL and Software Engineering Center (SEC) of Toshiba Corp. They are Takeshi Fukuda, Sei Kato, Ai Takizawa (TRL) and Masayuki Hirayama, Shigeo Baba, Yoshio Kataoka (RDC and SEC).

Finally, above all, the author's enthusiasm would not be very high without the faith, encouragement and support of his family for so long. Neither do his achievements so far.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**
1.1 Incremental Changes - The Typical Path of Software Evolution 1
1.2 Some Fundamental Issues of AOSD 2
1.3 Scope of the Research 4

2 Background **6**
2.1 Feature-Oriented Evolutionary Domains 6
2.2 Aspect-Oriented Software Development 7
2.3 An Initial Attempt on the Formal Model of Feature-Oriented Software . . . 11
2.4 Model Checking 11
2.5 OIMC - The Basics 12

3 Formal Models of Feature-Oriented Software **14**
3.1 Basic Definitions 14
3.2 Fundamental Issues of OIMC 19
3.3 Basic Notations 19

4 Fundamental Theoretical Foundation of OIMC **21**
4.1 Properties Preservation at Base States 21
4.1.1 A Theorem on Additive-Only Composition 21
4.1.2 A Theorem on Limited-Overriding Composition 27
4.1.3 The Feature Consistency Issue 31
4.1.4 Open Incremental Model Checking 31
4.2 The Soundness of Incremental Verification 32
4.2.1 Criticality of Transitions 32
4.2.2 Dependency Structure Among Base States 33
4.2.3 Cyclic Dependency with Base Links Only 36
4.2.4 Cyclic Dependency with Extension Links Only 40
4.2.5 Cyclic Dependency with Both Base and Extension Links 43
4.3 Properties Preservation at Extension States 48
4.4 The Scalability of Incremental Verification 50
4.5 An Example about Consistency among Single-Object Features 52

5 OIMC for Consistency among Multi-Object Features 56
5.1 The Fundamental Approach . . . 56
5.1.1 A Formal Model of Features Crosscutting Multiple Objects . . . 56
5.1.2 Transforming Multi-Object Models into a Global Model . . . 57
5.1.3 Incremental Verification Within Global Model . . . 59
5.2 An Example of Consistency among Multi-Object Features . . . 60

6 OIMC Improvements 63
6.1 Relaxing the Conformance Condition . . . 63
6.2 The Soundness Issue . . . 66
6.3 The Scalability Issue . . . 66
6.4 Parallelizing the OIMC Algorithm . . . 66

7 Model-Based Feature Implementation and Application 69
7.1 Component Specification and Consistency Verification . . . 69
7.1.1 Interface Signature . . . 70
7.1.2 Interface Constraints . . . 70
7.1.3 Component Specification and Composition . . . 71
7.2 Layered Architecture for Feature-Oriented Software . . . 77
7.3 Transforming Formal Feature Model to Codes . . . 78
7.4 Composing/Weaving Features via Hyper/J and AspectJ . . . 79
7.5 NuSMV2 . . . 80

8 Related Work and Conclusion 81

Bibliography 84

Publications 87

List of Figures

1.1 The coverage description of the dissertation. 4

2.1 The evolution of a complex data structure via handling some extra features. 8

3.1 The extension feature (E) and the composition (C) of both base and extension features in the library system. The base feature (B) is obvious from the composition model. 16

4.1 An illustration of B and E conformance in case of additive-only composition. The property $p = \mathbf{EG}\, f$ is preserved at ex and all states in B. 22
4.2 An illustration of B and E conformance in case of overriding composition. The property $p = \mathbf{EG}\, f$ is preserved at ex and all states in B. 27
4.3 An example of B and E composition in which the verification result is sound. E overrides the transition in B associated with event e. 35
4.4 Another example of B and E composition in which a sound verification result can be delivered. 35
4.5 An example of B and E composition in which the incremental verification may not be sound. 36
4.6 An example of cyclic dependency due to base links only. 37
4.7 An illustration for composition with base-only cyclic dependency which still preserves $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ at exit states (pp_1 and pp_2 can not exist at the same time). 39
4.8 An example of cyclic dependency due to extension links only. 39
4.9 An illustration for the additive-only composition with extension-only cyclic dependency which still preserves $\mathbf{A}\,[f\,\mathbf{U}\,g]$ at exit states (pp_1 and pp_2 can not exist at the same time). 42
4.10 An example of composition failing to preserve $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ in case of extension-only cyclic dependency. 43
4.11 An example of cyclic dependency due to both base and extension links. . . 43
4.12 The difference between assumption model checking in E and regular model checking in C, in terms of the computation tree at ex within E, in the base-extension circular dependency. 45
4.13 Strict conformance condition: The need of extra checking for any f^* loop between ex and re at which $\neg\mathbf{EG}\, f$ holds. 46
4.14 Strict conformance condition: The need of extra checking for any $(f \wedge \neg g)^*$ loop between ex and re at which $\mathbf{A}\,[f\,\mathbf{U}\,g]$ holds. 47
4.15 An example of failing composition to preserve $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ in case of base-extension cyclic dependency. 48
4.16 A base component and its refining components for a user account. 53

4.17 The state transition chart of a user account in the library management system . . . 54

5.1 Composing a full extension model with a partial base feature. . . . 57
5.2 A simple library with two features: book reservation and loss-handling. . . 60
5.3 The global extension model of the library example. . . . 62

6.1 An illustration of B and E conformance in case of overriding composition. The truth value with respect to the property $p = \mathbf{AG}\, f$ is preserved at ex and all states in B. . . . 65

7.1 The dynamic behavior model of the "black" component. . . . 72
7.2 Component refinements and component composition via class aggregation. 72

Chapter 1

Introduction

1.1 Incremental Changes - The Typical Path of Software Evolution

Changes are essential substance of successful software. A long-lived software system must continue to evolve and adapt with ever-changing environment. To a large extent of software technology, that process is called software evolution. With respect to a broader context, software evolution involves all periods in the software lifecycle, namely development, maintenance and evolution. A desirable software technique should take minimum costs and efforts during software evolution, even in the dynamic environment. This dissertation focuses on the third period, i.e. evolution. A working system must evolve when new requirements come in or the operating environment changes. Handling the changes and integrating them into the existing system *consistently* with the *minimum costs* are the main goal of the dissertation.

Requirement changes are nature of software evolution. Most often are small *incremental changes* in comparison with existing systems. In our view, incremental changes are a major source of evolution in practical systems. For *feature-oriented* software in which features are the basic building units, incremental changes could be either a new feature to be added to many existing features or just simply a more refined version of a current feature supported by the system. The term *feature* means an abstract description of a system function or service. For example, current advanced mobile phones provide the traditional talking function. This feature enables two persons to communicate with each other. Besides that base feature, depending on the model, a mobile phone can provide extra features such as photo-shooting, video conferencing, Internet browsing etc. Even though incremental changes are small with respect to overall system, they come so frequently that system engineers find them difficult. Particularly, the difficulty is on how to minimize the effects, possibly errors or bugs, when introducing changes to systems. Therefore, evolving system in the events of such changes are quite challenging and requires an effective approach.

There has been a clear shift from traditional, such as object-oriented (OO), software development paradigms in dealing with incremental changes. In those old days, changes are captured and immediately integrated to the existing system's artifacts in early phases of the development process. After integration, changes are manually or semi-automatically interleaved with the existing system artifacts during subsequent phases so that their

effects to the system are propagated to system-wide level. The system together with the changes is then certainly very difficult to comprehend and manage. Furthermore, if changes are integrated so early in the software lifecycle, it is difficult to undo changes and to backtrack errors discovered in subsequent stages. As a general consensus between software practitioners, manual changes made to such artifacts are dangerous and likely introduce errors.

A new approach tries to delay the change-integrating stage as late as possible. As before, changes are captured in early phases but kept separate from the system until the very last phases of the software development process. As the software process proceeds, the corresponding codes for those changes are separately implemented. The final step involves a composer/weaver which automatically weave the corresponding codes of changes into the existing system.

The new software paradigm, called *aspect-oriented software development* (AOSD), relieves people from many hard works in manipulating source codes and checking for program correctness manually. This approach is clearly promising to software development and evolution. However, current practice in AOSD stops short of that target. Currently, there are some excellent automatic mechanisms for code weaving/composing process in the implementation phase, notably Hyper/J [36] and AspectJ [38]. In contrast, there is clearly a lack of AOSD research and respective tool supports in early stages of development process such as requirement modeling, analysis and design. From that viewpoint, this work focuses on the analysis phase. More specifically, aspects can be formally analyzed for their *consistency*.

In the analysis phase, system behavior is expressed by state and use case diagrams in Unified Modeling Language (UML) [33]. Requirements can be modeled by use case diagrams [1, 32]. So are incremental requirement changes. Refining use case diagrams by adding new scenarios is a way to make system behavior richer. An incremental change in a use case (*base*) is usually an *extension* to that use case. The extension is quite small compared with the overall complexity of the base system. This work focuses on this typical software evolution so that the evolved system is still consistent.

1.2 Some Fundamental Issues of AOSD

Separation of concerns is the core of successful software [17, 37]. This concept is realized in several different forms within aspect-oriented software community. A promising software paradigm has emerged from the idea to capture concerns crosscutting multiple objects in modular fashion. It is AOSD. This paradigm relieves object-oriented software development from its most severe weakness of handling multi-object concerns. The two most prominent forms of concerns are aspects [17] and hyperslices [37] among many others. Hyperslices [37] in general address crosscutting features at high level of abstraction. On the other hand, aspects [17] are usually for addressing programming-level concerns. This work inclines towards hyperslices which are essentially system-wide functional features presented by the collaboration between several objects in the system. As mentioned, in this dissertation, a *feature* means an abstract description of a system function or service. For example, a modern mobile phone, in addition to its traditional talking function, also provides several extra advanced features such as photo-shooting, video conferencing, Internet browsing, song downloading etc. These features interleave with each other even at

run time. They interacts with each other and may affect working functions of others.

In feature-oriented software - the focus of this thesis, a system is viewed as a structure of several features it serves the environment. This system view is particularly useful when designing system architecture as the features can be layered on top of each other [35]. In terms of design, each feature is implemented by a collaboration of several objects, i.e. the feature crosscuts those objects. The system is called *feature-oriented* in specification and *collaboration-based* or *role-based* [10, 27, 35] in design. Henceforth, the terms component, feature, hyperslice, aspect and collaboration are considered as equivalent as they address the same thing - the high-level description of multi-object crosscutting concerns.

Features have well-defined *interfaces* allowing their composition to construct bigger systems. The advantage of this design style is that by specifying each feature independently, a system can be flexibly constructed based on the decision of whether or not to include a particular feature. On the other hand, they communicate to each other via interfaces. Therefore, from software verification perspective [6], the *compositional reasoning* for entire system validation can be applied by separately verifying each feature with respect to the rest of system. That is the motivation of the so-called *modular model checking* [13, 18].

Unlike traditional modular model checking approaches which treat systems as static, a new method of model checking to be presented by this work, called *open incremental model checking* (OIMC) in our opinion, is proposed to address the changes to an existing system as the system continues to evolve [10, 27]. In the typical evolution scenario, a system enriches its feature set by some extra features. In the basic case, given a *base* feature, an *extension* feature is attached with the base. The most fundamental issue is on how to verify that the extension does not violate some properties inherent to the base. The verification approach is required to be *open* because future extensions are not known in advance. Further, it should be *efficient* instead of verifying whole the base plus the extension together. The proposed approach is quite open since the interface between features is very flexible for change. The model checking itself is executed in an *incremental* manner as it only executes within the extension, i.e. very efficient.

The theoretical foundation of the proposed verification technique is built with respect to the generalized model of feature-oriented software. Initially, fundamental theorems about base and extension consistency are presented. Provided some assumptions about the base at *reentry states*, the incremental verification is only required to check at all *exit states* to confirm the consistency between the base and the extension. Ensuring those assumptions at reentry states is actually a major part for the soundness of OIMC.

Moreover, as the system continues to evolve, the scalability of OIMC is in fact concerned with not only one-step extension, but also many subsequent extension features. That is another issue of OIMC to be addressed. By a solid theoretical proof, OIMC maintains its scalability to ensure the preservation of base property for many subsequent extensions as long as subsequent extensions *conform* with their respective bases at all exit states. Some difficulties in applying OIMC to multi-object features and the solutions are also addressed.

Finally, some applications of the above theory are briefed. Specifically, the most challenging issue of component-based software in general, especially AOSD, is about component specification. Current component technologies such as OMG's CORBA (Object Management Group's Common Object Request Broker Architecture), Microsoft's COM/DCOM (Distributed Component Object Model), Sun's Java or UML/OCL (Ob-

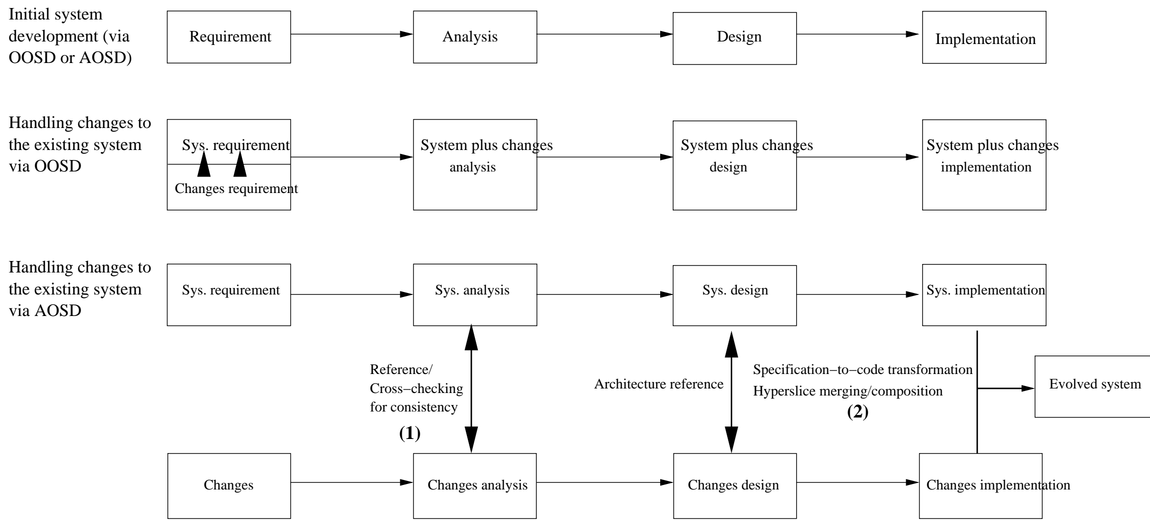

Figure 1.1: The coverage description of the dissertation.

ject Constraint Language) face the same difficulty in composing components. They only specify component interface in syntactical terms. Hence, the components can be syntactically *plugged*. However, as the semantic constraints among components are not detailed enough, once composed, the components do not *play* correctly. That is the phenomenon called *plug-but-not-play* within component-based software. The above theoretical foundation on feature consistency can solve the problem as it suggests how to explicitly specify semantic constraints for consistency among components in the interface.

1.3 Scope of the Research

Figure 1.1 shows the coverage of this work in a typical software development process. Compared with the traditional OO software development (OOSD), there is a significant difference between OOSD and AOSD. Figure 1.1 depicts the typical evolution scenarios in both approaches. In the OOSD approach, changes are integrated into the system at the beginning of the development process. As a result, subsequent stages of the process need to deal with the whole bulk of changes and existing system artifacts together. Even with the help of the OO technology, the development cost to handle the changes are high because the changes are propagated to many system modules, i.e. objects. That is a bad evolution process. On the contrary, within the AOSD paradigm, changes and system artifacts are kept separate during the development process. They are only composed together at the end of the development lifecycle. The separation of existing artifacts and changes is the core of the successful software evolution in this case. "Separation of concerns" - the key of successful software - is achieved. It minimizes efforts and costs as the *concerns*, i.e. changes and existing artifacts, are separately managed and implemented.

The dissertation can be divided into two major parts numbered accordingly in the figure. Those are:

1. Capturing incremental changes in the analysis stage and integrating them consistently. In particular, we check for consistency between the changes and the base system. To deal with this issue, a formal model to express feature-oriented system

and changes is introduced. The theoretical foundation for feature consistency is presented with regards to the formal model.

2. Presenting some typical applications of the theory in Part 1. A notable application is about component specification and composition in terms of semantic constraints. Besides, from the formal feature-based model, a formal language to express system and changes in a modular fashion is suggested. The language is graphical-based like that in [2]. From the specification written in that language, corresponding codes can be automatically generated via some context-free transformation rules [8, 39]. After transforming feature specifications into a specific target programming language, e.g. Java as of this work due to the available code composing/weaving support tools like Hyper/J [36] or AspectJ [38], the final step is simple in terms of making the actual composition by activating those programming tools to obtain the target system in the evolution process.

In this thesis, the first task in Figure 1.1 is the core, whereas the topics in the second part are just briefed in principle. The core topic covers Chapters 3-6. It can be divided into two related sub-parts: a formal model of feature-oriented software (Chapter 3) and the verification of feature consistency with regards to the model (Chapters 4-6).

In details, the structure of the thesis is presented in the following. Chapter 2 reviews the basis of Computation Tree Logic (CTL) and incremental model checking. Chapter 3 is concerned with a formal approach to verify CTL properties in incremental manner. In particular, Chapter 3 initially provides basic definitions to be used for the subsequent discussion. Chapter 4 details the OIMC with respect to CTL properties. Later, Section 4.2 devotes to the soundness of open incremental verification, i.e. proving the assumption at the interface instead of assuming them (i) and given the conformance, proving the preservation of base properties at exit states (ii). Section 4.4 proves the scalability of OIMC for future extensions. Chapter 5 describes the difficulties and guidelines when dealing with multi-object features. Sections 4.5 and 5.2 present the application of OIMC to single-object and multi-object features respectively. Chapter 6 improves the theoretical foundation a bit by relaxing the conformance condition between base and extension features. Chapter 7 deals with the 2nd task depicted in Figure 1.1, i.e. the application of the theory to component specification and composition in terms of semantic constraints among components; the transformation of features from formal specification into codes; and then the composition of those codes into the target system. Finally, the contribution of this paper, its future and related work are presented in Chapter 8.

Chapter 2

Background

2.1 Feature-Oriented Evolutionary Domains

Evolutionary domain is a reference framework for software evolution [16]. In brief, it describes software evolution as a process of changing specification. In essence, it defines an evolution relation $\sqsubseteq$ on which software are related to each other according to degree of maturity. The typical scenario in software evolution is that the new specification is more evolved than the previous version. Formally, the evolution problem can be expressed as below:

- Let s, p be the current specification of the system and its implementation respectively. A new specification s' is formed by adding a change Δs to s, i.e. $s' = s + \Delta s$. In this context, we define $s \sqsubseteq s'$, namely s' is more evolved than s.

- We need to find a program p' implementing s' such as p' is constructed from the composition of p and Δp - the program fragment implementing Δs.

The process realizing specifications like s, s' or Δs into respective concrete programs such as p, p' and Δp is named *evolutionary development process* [16]. Basically, evolutionary development process is specific to the software development paradigm in use. AOSD paradigm is very promising in terms of software development and evolution because the separation of concerns is achieved. s and Δs can be separately implemented by p and Δp. Here, concerns (possibly composite) are s and Δs. The feature-oriented approach is a particular case of AOSD in the sense that concerns are features the system offers its clients. A key issue of software evolution relates to the consistency between p and Δp, namely Δp does not conflict to any inherent properties of p. This dissertation is addressing the issue in the context of feature-oriented software. It also justifies the proposed approach to be efficient.

This section describes an external view to feature-based systems from the perspective of evolutionary domain. A system is structured by composing several features. Let PF be the set of *primitive* features. The specification s of a feature-based system is a set of features, $fs \subset PF$, it serves the environment. As more requirements come in, the system evolves to a new version s' which provides a richer set of features fs'. In this circumstance, we define s' to be more evolved than s. In a reference to the evolutionary domain framework [16], we define:

- $s \sqsubseteq s' \Leftrightarrow fs \subseteq fs'$.

- The change to the system is $\Delta s = fs' \setminus fs$.

The above defines the evolution relation $\sqsubseteq$ in the feature-based specification domain. By a feature-based software development process, each feature is separately mapped into a respective hyperslice [37]; and the programs p, p' and Δp corresponding to s, s' and Δs are formed by composing hyperslices in a well-ordered manner.

s, s' are represented by set of primitive features. Composing primitive features together results in a composite feature implemented by some program. In essence, s and s' are considered as *complete* features because their respective programs p and p' can run on its own. On the contrary, though Δs is also a set of primitive features, it is not regarded as a complete composite feature since its program can not stand on its own. Rather it refines or extends s. Its implementation is usually a partial program. Δs is called *partial* feature, either composite or primitive. From the perspective on the formal model defined in Chapter 3, a partial feature does not have its own initial state. Rather, that state is defined as no-care value ($\perp$) because the initial state is supplied by some complete base feature on which the extension feature refines.

Subsequent discussion in the thesis treats the typical case in which the *base* is a complete feature, while the *extension* is partial. The arguments can be equally applied to both composite and primitive features. More specifically, we focus on the formal specification structure of each feature based on which consistency between features is formally defined and verified.

2.2 Aspect-Oriented Software Development

Separation of concerns is at the core of software engineering, and all developers do it. In its most general form, it refers to the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal or purpose. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. There are many kinds of concerns. For example, OOSD focuses on *classes*, functional languages are interested in *functions* or this thesis only considers *features* crosscutting multiple objects. A kind of concern like class, function or feature is referred as a *dimension* of concern. Separating concerns in a software according to a particular dimension may result in a contrasting effectiveness in software evolution. Achieving a "clean" separation of concerns via a proper dimension can reduce software complexity, improve comprehensibility, promote traceability throughout the software lifecycle, facilitate evolution and reuse etc. This thesis advocates the separation of concerns according to feature dimension. Software developed in this manner is called *feature-oriented*.

AOSD is an emerging software technology and has attracted great attention from the software engineering community. It is an improvement from OOSD. Even though OO technology brings about a great change in effectiveness and efficiency during software development and evolution via its three main mechanisms, namely encapsulation, inheritance and polymorphism, it still suffers from a problem called the *tyranny of the dominant decomposition* [36]. It only permits the separation and encapsulation of classes - the dominant dimension in the OO world. The same thing also happens in feature-oriented software. However, from system evolution perspective as shown in Section 2.1 above, the requirement changes often occur in terms of new system services or features

instead of classes. Further, these changes of features very often crosscuts many member classes in the system - separating system along the class dimension would be a very bad idea. Hence, capturing the changes by separating concerns according to the feature dimension is the best way to limit the effect of the changes and to minimize the development efforts. Specifically, changes are usually localized at a single feature or at most some neighbor features. That is the reason for feature-oriented software to be in focus of the dissertation.

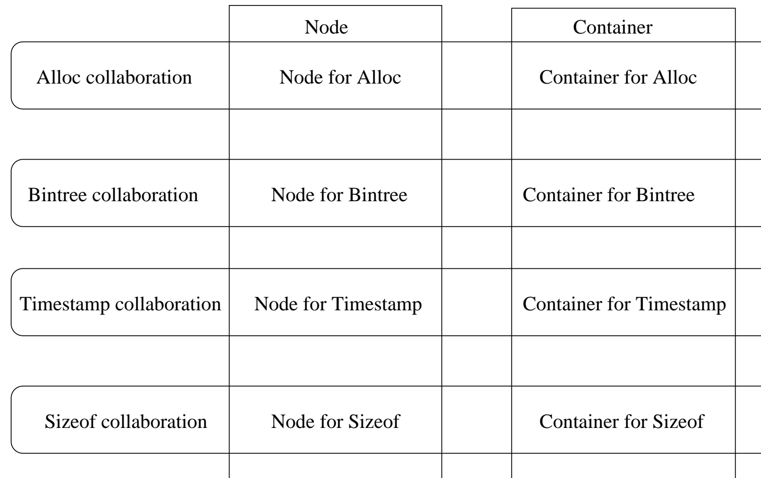

Figure 2.1: The evolution of a complex data structure via handling some extra features.

Below is a simple example showing the advantages of AOSD by separating concerns via the dominant feature dimension over OOSD. The example about a data structure is shown in Figure 2.1. That is a specific example illustrated in [35]. The example is about a data structure design. In this simplified example, there are two participating classes, namely a *node* class and a *container* class. All data nodes in this data structure are instances of node class, while there is only one instance of container class per data structure. The target data structure consists of four different collaborations or features: `BinTree`, `Alloc`, `TimeStamp` and `Sizeof`. `BinTree` captures the functionality of a binary tree. `Alloc` is in charge of memory allocation. `TimeStamp` is responsible for maintaining timestamps for data structure and element updates. `Sizeof` simply keeps track of the data structure size. In the simplest way, the system has evolved from the simplest structure of binary tree only (two top layers) into a more complex structure with additional timestamp and size features. Though AOSD via proper concern dimension outperforms OOSD in all stages of software lifecycle, due to the small example, its advantage over OOSD is presented in the implementation phase only. In OOSD, surely all code fragments of the same class below must be merged into a single bulk. Any new feature, if added to the system, simultaneously affect the two classes. That is certainly very hard to maintain and evolve in the event of changes. On the contrary, the situation is different in AOSD because each feature is managed and implemented separately in its own layer or at most in a few neighbor related layers. The codes in C++ and Java are shown below.

In C++, each collaboration is implemented in a layer via template.

```
template <class EleType> class ALLOC{
public:
   class Node {
```

```
      EleType element; // The actual stored data
   public:
      etc...// methods definition
   };

   class Container {
   protected:
      typedef Super::EleType EleType;
      // The actual type of stored data
      void* node_alloc();//memory allocation
   };
};

template <class Super> class BINTREE: public Super{
public:
   class Node:  public Super::Node {
   // Class name is hard-coded into definition.
   // Upper layer must consist of the class Node.
   // Otherwise, error - super-class not found.
      Node* plink, llink, rlink;// Node data members
   public:
      etc...// node interface
   };

   class Container:  public Super::Container {
      Node* header;
   public:
      void insert (EleType el) {...}
      void erase (Node* node) {...}
      bool find (EleType* el) {...}
   };
};
```

The template-based implementation in C++ for the other two collaborations, namely `TimeStamp` and `Sizeof` are skipped. The target data structure is formed by composing layers. Depending on the desired functions of the target data structure, a layer can be flexibly decided on whether or not to be included. For example, a binary tree structure storing integers, maintaining time-related access information and size can be formed as:

```
typedef SIZEOF < TIMESTAMP < BINTREE < ALLOC < int > > > > Tree;
```

On the other hand, the Java codes and the composition via Hyper/J are shown below. Each feature is mapped into a separate package `collab.*`.

```
// Alloc layer
package collab.Alloc;
public class Container{// Container definition
   public Node AllocNode(int n, String s){
      Node res = new Node(n, s);
```

```
      return res;
   };
   void Process(Node n){
      // process node appropriately in this layer
   };
}; // END class Container

package collab.Alloc;
public class Node{// Node definition
   public Node(int n, String s){
      // create proper elem data member by option s.
   };
   ...
   private Element elem;
}; // END class Node
```

The Java code for the other three layers, namely `BinTree`, `TimeStamp` and `Sizeof`, are quite similar to that of `Alloc` layer. The source code of those layers are skipped. Once a layer implementation is completed, we can compile Container and Node classes separately into Java's `.class` files. When the compilation is finished, Hyper/J compositor comes in to deal with those .class files. It looks for its three component files in the project.

```
// Hyperspace specification file:  OpenDS.hs
// It instructs Hyper/J to compose all Java classes in provided packages.
hyperspace CollabHyperspace
   composable class collab.Element.*;
   composable class collab.Alloc.*;
   composable class collab.BinTree.*;
   composable class collab.TimeStamp.*;
   composable class collab.SizeOf.*;

// Concern mapping file:  concerns.cm
// It specifies the feature corresponding to each package.
package collab.Element :  Feature.Alloc
package collab.Alloc :  Feature.Alloc
package collab.BinTree :  Feature.BinTree
package collab.TimeStamp :  Feature.TimeStamp
package collab.SizeOf :  Feature.SizeOf

// Hypermodule specification file:  DS.hm
// It instructs Hyper/J to merge classes, methods sharing the same name.
hypermodule DS
   hyperslices:
      Feature.Alloc,
      Feature.BinTree,
      Feature.TimeStamp,
      Feature.SizeOf;
   relationships:
```

```
    mergeByName;
    override action Feature.BinTree.Node.Value with action
       Feature.Alloc.Node.Value;
end hypermodule;
```

2.3 An Initial Attempt on the Formal Model of Feature-Oriented Software

An initial effort towards the formal model of feature-oriented software has been attempted [25]. However, in that work, only class diagrams are considered. Hence, the model is static. The formal model presented in this thesis takes a step further in defining the dynamic behavior within classes. Comparing the model in [25] and the multi-object model in Section 5.1, there is a great similarity. Models are all expressed by layers of collaborations [25] and features in this work. Within each layer are many member objects. This work goes further into the details of each object in any layer by defining its behavior via a state transition model. Each state model provides interface states at which other models of the same object belonging to other layers can be plugged together. To some extent, this dissertation continues the work in [25].

2.4 Model Checking

CTL* is the logic for specifying properties of state transition systems. A tree is formed by designating a state in the system as the initial state and then expanding the structure into an possibly infinite tree with the designated state at the root. Basically, this logic is formally expressed via two quantifiers **A** ("for all paths") and **E** ("for some path") together with five operators **X** ("next"), **F** ("eventually"), **G** ("always"), **U** ("until") and **R** ("release").

This dissertation is only concerned with the verification of CTL properties. CTL is a restricted subset of CTL* in which each temporal operator must be preceded by a quantifier. Basically, we have ten basic CTL operators. Those are: $\mathbf{EX}\, f$, $\mathbf{AX}\, f$, $\mathbf{EG}\, f$, $\mathbf{AG}\, f$, $\mathbf{EF}\, f$, $\mathbf{AF}\, f$, $\mathbf{E}\,[f\,\mathbf{U}\,g]$, $\mathbf{A}\,[f\,\mathbf{U}\,g]$, $\mathbf{E}\,[f\,\mathbf{R}\,g]$, $\mathbf{A}\,[f\,\mathbf{R}\,g]$ where f and g are CTL properties or atomic propositions. According to [6], these ten operators can be transformed into three basics, namely $\mathbf{EX}\, f$, $\mathbf{EG}\, f$, $\mathbf{E}\,[f\,\mathbf{U}\,g]$. In general, all CTL formulae can be transformed into a form expressed by these three operators, a negation and a union operators.

- $\mathbf{AX}\, f = \neg\mathbf{EX}\,(\neg f)$
- $\mathbf{EF}\, f = \mathbf{E}\,[True\,\mathbf{U}\,f]$
- $\mathbf{AG}\, f = \neg\mathbf{EF}\,(\neg f)$
- $\mathbf{AF}\, f = \neg\mathbf{EG}\,(\neg f)$
- $\mathbf{A}\,[f\,\mathbf{U}\,g] \equiv \neg\mathbf{E}\,[\neg g\,\mathbf{U}\,(\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\,(\neg g) = \neg(\mathbf{E}\,[\neg g\,\mathbf{U}\,\neg(f \vee g)] \vee \mathbf{EG}\,(\neg g))$
- $\mathbf{A}\,[f\,\mathbf{R}\,g] \equiv \neg\mathbf{E}\,[\neg f\,\mathbf{U}\,\neg g]$

- $\mathbf{E}\,[f\,\mathbf{R}\,g] \equiv \neg\mathbf{A}\,[\neg f\,\mathbf{U}\,\neg g]$

Model checking field faces a common problem, namely state explosion, in which all reachable states in the model must be checked. There are some notable approaches to deal with the problem. The first is via *modular model checking* [13, 18, 20, 31]. However, these methods are rather closed, though modular. From the software evolution perspective, they are not very helpful since future changes are unanticipated. OIMC proposed in the dissertation addresses exactly the weakness. Besides, to tackle the number of states to be verified, parallel model checking is suggested. The parallel version of OIMC is proposed in Section 6.4.

The theoretical foundation on software verification in the thesis focuses on the infinite model by considering only infinite paths starting from the initial state. The finite model differs only in the fact that those paths must end at an implicitly designated final state. Certainly, the infinite model is stronger, even though the two are quite identical in many aspects. Subsequently, upon any conclusion or theorem drawn with respect to the infinite model, a corresponding conclusion with respect to the finite model is presented without going into details.

2.5 OIMC - The Basics

The initial incremental verification methodology of feature-based software has been attempted by [10]. The approach consists of the following activities:

1. Proving a CTL property of a system.

2. Deriving a set of constraints in the interface states of the system such that if those constraints are preserved, the corresponding property is guaranteed. According to [10], those constraints are named *preservation constraints*.

3. A separate feature does not violate the above property of the first feature if, during its execution, the preservation constraints are preserved. In this activity, rather than entire state space of the first feature, only interface states are considered during verifying the second feature.

The following list are the key ideas presented by [10] to accomplish activities above.

- Proposing a formal model of features of which interfaces are fixed with a single *exit* and a single *reentry* states. Implicitly, this model is additive, i.e. the extension is not allowed to override any behavior of the base.

- Presenting a verification algorithm to check whether a CTL property p continues to hold at those exit states. Fundamentally, the algorithm is based on an assumption that all reentry states are proper. From that assumption, conclusion about the extension with respect to the property is drawn.

We believe that [10] is based on a rather strict model and too relaxed assumptions. The formal interface is generalized to accommodate multiple exit and reentry points (Chapter 3) with base-behavior overriding capability. Due to that generalization, fundamental foundation of OIMC is rebuilt (Section 4.1). Unlike [10], a sound theoretical foundation

of OIMC is introduced. More specifically, the conformance condition, soundness and scalability of OIMC with respect to this generalized model are investigated. Initially, the theoretical foundation and related issues are constructed for single-object features. Issues related to multi-object features, especially state explosion due to cross-product of member states, are discussed in Chapter 5.

Chapter 3

Formal Models of Feature-Oriented Software

3.1 Basic Definitions

In feature-oriented software, each feature is packaged by one or more components as shown in the example in Section 2.2. In general, each feature can be regarded as a component in the most general sense (either composite or primitive). A composite component is formed from some sub-components, whereas a primitive component is the most basic building block.

The most common form of components in practice is Commercial-Off-The-Shelf (COTS) on very independent components. The computation paths of these components rarely interleave with each other. The relationship between components can be named *functional addition*. Besides COTS, there is another aspect of component evolution - *functional refinement* - in which a feature refines another. Feature refinements are fairly coupled because they interleave execution paths. Even though the discussion in this work mainly focuses on refinement, the results can be well applied to COTS because analyzing COTS is obviously simpler.

Typically, there are two features under consideration: a base and an extension. Between the base and its extension, on the base side, is an interface consisting of *exit* and *reentry* states. An exit state is the state where control is passed to the extension. On the other hand, a reentry state is the point at which the base regains control. Correspondingly, the extension interface contains *in-* and *out*-states at which the refinement component receives and releases system control. Let AP be a set of atomic propositions. Each feature is separately specified by a state transition model.

Definition 1 *A state transition model M is a tuple $\langle S, \Sigma, s_0, R, L\rangle$ where S is a set of states, Σ is the set of input events, $s_0 \in S$ is the initial state, $R \subseteq S \times PL(\Sigma) \rightarrow S$ is the transition function (where $PL(\Sigma)$ denotes the set of guarded events in Σ whose conditions are propositional logic expressions), and $L : S \rightarrow 2^{AP}$ labels each state with the set of atomic propositions true in that state.*

Definition 2 *A* base *is expressed by a transition model B and an* interface *I, where $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B\rangle$. An interface is a tuple of two state sets $I = \langle exit, reentry\rangle$, where $exit, reentry \subseteq S_B$ and $exit, reentry \neq \emptyset$.*

Definition 3 *An* extension *is represented by a model $E = \langle S_E, \Sigma_E, \perp, R_E, L_E \rangle$. $\perp$ denotes no-care value. The interface of E is $J = \langle in, out \rangle$.*

Above base and extension models are all intended for single-object features. The model of features crosscutting multiple objects is introduced in Section 5.1.

Unlike the simple model in [10], the interface I could have multiple points. Moreover, it is possible that $exit \cap reentry \neq \emptyset$, i.e. *dual* states are allowed. Note that, unlike the interface $I = \langle exit, reentry \rangle$ of the base, the in- and out-states are *terminal* in E because at those points, E starts and ends its execution trace. Formally,

- $\forall i \in J.in, \nexists s \in S_E : (s, \perp, i) \in R_E$.
- Similarly, $\forall o \in J.out, \nexists s \in S_E : (o, \perp, s) \in R_E$.

E can be syntactically plugged with B via *compatible* interface states. Logically, along the computation flow, when the system is in an exit state $ex \in I.exit$ of B matched with an in-state $i \in J.in$ of E, denoted as $ex \leftrightarrow i$, it can enter E if the conditions to accept extension events, namely the set of atomic propositions at i, are satisfied. Similar arguments are made for the matching of a reentry state $re \in I.reentry$ and an out-state $o \in J.out$. The conditions resembles to pre- and post-conditions in *design by contract* [24].

Definition 4 *Within interfaces I and J of B and E, the pairs $\langle ex, i \rangle$ and $\langle re, o \rangle$ can be respectively mapped according to the following conditions.*

- *$ex \leftrightarrow i$ if $L_B(ex) \Rightarrow L_E(i)$.*
- *$re \leftrightarrow o$ if $L_E(o) \Rightarrow L_B(re)$.*

Given two features B and E, the above definition provides the syntactical condition for two interface states to be mapped together. Unlike [10, 28], features specified in this model are clearly independent with each other. The decision for actual matching is left to the modeler. Along the flow of computation paths, if an interface state is mapped to more than one interface states of the other component, the mapping is a *fork*. Conversely, the mapping is a *join*. During composition, the transitions (states, events and guards) at interface states will be adapted according to the mapping. Fork mapping is illustrated Figure 3.1 in which the out-state o_2 of E is mapped to two reentry states $u.clr$ and $u.crd$ of B. Correspondingly, the transition $paid_u$ between $u.chg$ and o_2 in the component E is represented by two complementary transitions $paid_u$ with `[ontime]` and `[overdue]` guards to the two reentry states respectively. On the other hand, other interface state mappings are simple such as $i_1 \leftrightarrow u.wait$, $i_2 \leftrightarrow u.borr$ and $o_1 \leftrightarrow u.init$. Details are presented in the example in Chapter 4.

The actual mapping configuration is decided by the modeler at composition time. Subsequently, states ex and re will be used in place of i and o whenever interface states are referred. Identically, J is replaced by I.

The composition of B and E is defined as:

Definition 5 *Composing the base B with the extension E, through the interface I produces a composition model $C = \langle S_C, \Sigma_C, s_{0_C}, R_C, L_C \rangle$. C is defined from $B = \langle S_B, \Sigma_B, s_{0_B}, R_B, L_B \rangle$ and $E = \langle S_E, \Sigma_E, \perp, R_E, L_E \rangle$.*

- *$S_C = S_B \cup S_E$;*

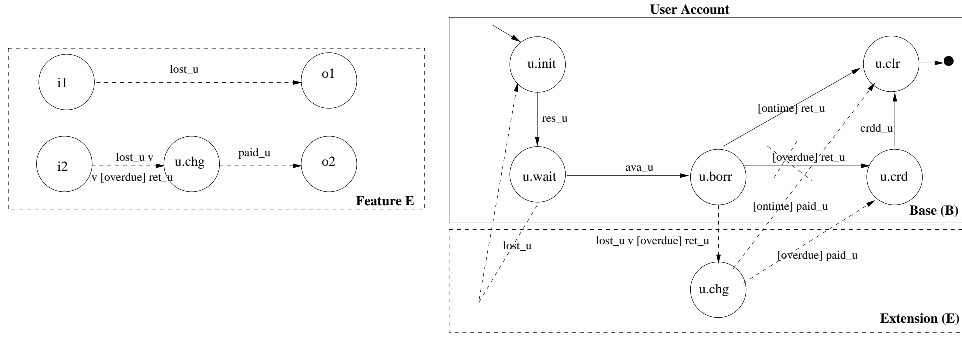

Figure 3.1: The extension feature (E) and the composition (C) of both base and extension features in the library system. The base feature (B) is obvious from the composition model.

- $\Sigma_C = \Sigma_B \cup \Sigma_E$;
- $s_{0_C} = s_{0_B}$;
- *R_C is defined from R_B and R_E. For each $s \in S_C$, let $\vee_s^E = \bigvee pl_i$ where $(s, [pl_i]\ e) \in Dom(R_E)$,*
  - $\forall (s, [pl_i]\ e) \in Dom(R_E)$: $R_C(s, [pl_i]\ e) = R_E(s, [pl_i]\ e)$
  - $\forall (s, [pl_B]\ e) \in Dom(R_B)$: $R_C(s, [pl_B \wedge \neg\vee_s^E]\ e) = R_B(s, [pl_B \wedge \neg\vee_s^E]\ e)$
- $\forall s \in S_B, s \notin I.exit \cup I.reentry : L_C(s) = L_B(s)$;
- $\forall s \in S_E, s \notin J.in \cup J.out : L_C(s) = L_E(s)$;
- $\forall s \in I.exit \cup I.reentry : L_C(s) = L_B(s)$;

The atomic proposition produced at the interface states can be explained in the following. Due to Definition 4, if an exit state ex is mapped with an in-state i, $L_B(ex) \Rightarrow L_E(i)$. After composition, $ex \leftrightarrow i$, the set of atomic propositions is the union from $L_B(ex) \cup L_E(ex)$. However, as $L_B(ex) \Rightarrow L_E(i)$, we have: $L_B(ex) \cup L_E(ex) \equiv L_B(ex)$. On the other hand, if a reentry state re is mapped with an out-state o, according to Definition 4, $L_E(o) \Rightarrow L_B(re)$. By similar arguments, the union $L_E(o) \cup L_B(re) \equiv L_E(o)$. However, besides computation paths through E to re, there exist some old paths completely lying in B. In such a situation, $L_E(re)$ does not contribute to the atomic proposition set at re. By unifying the two different routes, the weakest condition, namely $L_B(re)$, is taken as in Definition 5.

In the formal model, the expressions of guarded events between different transitions from the same state are disjoint [1]. $\vee_s^E$ represents the union of all events directing to the extension from a state s. In the composition definition above, a transition (s, pl_B, s') in the base can be partially or completely overridden by the extension. In this circumstance, s is certainly an exit state. It is completely removed from C if $pl_B \wedge \neg\vee_s^E = false$. Otherwise, it is partially overridden. This is another key difference between our model and the former [10] in which the extension is not allowed to override any transition in the base. The former is called *additive-only* composition, while ours is *overriding*. Further,

[1] The formal model is deterministic.

there is no inconsistent interface state due to atomic proposition composition. That is, $\forall s \in exit \cup reentry : \nexists a \in AP, a \in L_B(s) \wedge \neg a \in L_E(s)$.

This work concentrates on the preservation of CTL properties when features are composed. A CTL property is *normal* if it does not contain any logical operators such as $\neg$, $\vee$, $\wedge$ at the outermost level. It is a *negation single* property if it is in negation form of a normal property. It is *single* if it is either a normal or a negation property. Otherwise, the property is *composite*. For example, $\mathbf{EX}\,(\mathbf{AG}\,f \vee \mathbf{E}\,[f\,\mathbf{U}\,g])$ is a normal single property, while $\mathbf{EX}\,f \vee \mathbf{A}\,[f\,\mathbf{U}\,g]$ is composite.

Definition 6 *The order of a CTL property p, $|p|$, is defined as follows:*

- $p \in AP : |p| = 0$
- *p is one of* $\mathbf{AX}\,f, \mathbf{EX}\,f, \mathbf{AF}\,f, \mathbf{EF}\,f, \mathbf{AG}\,f, \mathbf{EG}\,f : |p| = 1 + |f|$
- *p is one of* $\mathbf{A}\,[f\,\mathbf{U}\,g], \mathbf{E}\,[f\,\mathbf{U}\,g], \mathbf{A}\,[f\,\mathbf{R}\,g], \mathbf{E}\,[f\,\mathbf{R}\,g] : |p| = 1 + max(|f|, |g|)$
- $p = \neg f : |p| = |f|$
- $p = f \vee g$ *or* $p = f \wedge g$: $|p| = max(|f|, |g|)$

Definition 7 *The* closure *of a property p, $cl(p)$, is the set of all sub-formulae of p including itself. $cl(p)$ is defined as:*

- $p \in AP : cl(p) = \{p\}$
- *p is one of* $\mathbf{AX}\,f, \mathbf{EX}\,f, \mathbf{AF}\,f, \mathbf{EF}\,f, \mathbf{AG}\,f, \mathbf{EG}\,f : cl(p) = \{p\} \cup cl(f)$
- *p is one of* $\mathbf{A}\,[f\,\mathbf{U}\,g], \mathbf{E}\,[f\,\mathbf{U}\,g], \mathbf{A}\,[f\,\mathbf{R}\,g], \mathbf{E}\,[f\,\mathbf{R}\,g] : cl(p) = \{p\} \cup cl(f) \cup cl(g)$
- $p = \neg f : cl(p) = cl(f)$
- $p = f \vee g$ *or* $p = f \wedge g$: $cl(p) = cl(f) \cup cl(g)$

Note that, if two properties p and q are logically equivalent, their respective closure sets may not. For example: $\mathbf{A}\,[f\,\mathbf{U}\,g] = \neg\mathbf{E}\,[\neg g\,\mathbf{U}\,(\neg f \wedge \neg g)] \wedge \neg\mathbf{EG}\,(\neg g)$. The closure set of the left-hand side is:

$$cl(LHS) = \{\mathbf{A}\,[f\,\mathbf{U}\,g]\} \cup cl(f) \cup cl(g)$$

On the other hand, the closure set of the right-hand side is:

$$\begin{aligned} cl(RHS) &= \{cl(\neg\mathbf{E}\,[\neg g\,\mathbf{U}\,(\neg f \wedge \neg g)]), cl(\neg\mathbf{EG}\,(\neg g))\} \\ &= \{\neg\mathbf{E}\,[\neg g\,\mathbf{U}\,(\neg f \wedge \neg g)], \neg\mathbf{EG}\,(\neg g)\} \cup cl(f) \cup cl(g). \end{aligned}$$

Surely, the later is “stronger” than the former in the sense that the first two properties in $cl(RHS)$ can derive their union property which is exactly $\mathbf{A}\,[f\,\mathbf{U}\,g]$. In terms of the preservation of closure set to be discussed in subsequent sections, obviously preserving $cl(RHS)$ requires a more restricted condition than that of $cl(LHS)$. Specifically, the preservation of the first two properties of $cl(RHS)$ can derive the preservation of $\mathbf{A}\,[f\,\mathbf{U}\,g]$ but not the other way around. This is the reason why $\mathbf{A}\,[f\,\mathbf{U}\,g]$ requires a complete proof in all theorems in this thesis with respect to the preservation of closure set at exit states in the later proofs. We can not employ the equivalent form of union property for $\mathbf{A}\,[f\,\mathbf{U}\,g]$ because the condition to preserve the closure set of the union property is stronger than the counterpart of $cl(\mathbf{A}\,[f\,\mathbf{U}\,g])$. Proof details are shown in Chapter 4.

Definition 8 *Given two properties p and q, two respective closure sets $cl(p)$ and $cl(q)$ are logically equivalent, $cl(p) \equiv cl(q)$, in terms of preservation if a tuple of truth values at a state with respect to the former can derive directly the corresponding tuple of truth values of that state with respect to the latter and vice versa.*

From this definition, except $\mathbf{A}\,[f\;\mathbf{U}\;g]$, we have the following equivalent closure sets from the equivalent properties pairs in Section 2.4:

- $cl(\mathbf{AX}\,f) \equiv cl(\neg\mathbf{EX}\,(\neg f))$
- $cl(\mathbf{EF}\,f) \equiv cl(\mathbf{E}\,[True\;\mathbf{U}\;f])$
- $cl(\mathbf{AG}\,f) \equiv cl(\neg\mathbf{EF}\,(\neg f)) \equiv cl(\mathbf{E}\,[True\;\mathbf{U}\,(\neg f)])$
- $cl(\mathbf{AF}\,f) \equiv cl(\neg\mathbf{EG}\,(\neg f))$
- $cl(\mathbf{A}\,[f\;\mathbf{R}\;g]) \equiv cl(\neg\mathbf{E}\,[\neg f\;\mathbf{U}\;\neg g])$
- $cl(\mathbf{E}\,[f\;\mathbf{R}\;g]) \equiv cl(\neg\mathbf{A}\,[\neg f\;\mathbf{U}\;\neg g])$

Definition 9 *For each model $M = \langle S, \Sigma, s_0, R, L\rangle$, the set of* ending states*, $end(M)$, consists of all states without any out-going transitions, i.e. $end(M) = \{s \in S \mid \nexists\, pl, e, s' : (s, [pl]\, e, s') \in R\}$.*

Definition 10 *An* assumption function *for a transition model $M = \langle S, \Sigma, s_0, R, L\rangle$ is a function $As : end(M) \rightarrow 2^{CTL}$.*

CTL denotes the set of all CTL properties. For every $f \in cl(p)$, if $f \in As(e)$, we assume that f holds at the ending state e. A property f is said to hold under *assumption model checking* [20] at a state $s \in S$, $M, s \models_{as} f$, if either $M, s \models f$ directly under regular model checking or $M, s \models f$ if assumption truth values at some ending state are required.

Definition 11 *The truth values of state s with respect to a set of CTL properties ps within a model M, denoted $\mathcal{V}_M(s, ps)$, is a function: $S \times 2^{CTL} \rightarrow 2^{CTL}$.*

- $\mathcal{V}_M(s, \emptyset) = \emptyset$
- $\mathcal{V}_M(s, \{p\} \cup ps) = \mathcal{V}_M(s, \{p\}) \cup \mathcal{V}_M(s, ps)$
- $\mathcal{V}_M(s, \{p\}) = \begin{cases} \{p\} & \textit{if } M, s \models p \\ \{\neg p\} & \textit{otherwise} \end{cases}$

Hereafter, $\mathcal{V}_M(s, \{p\}) = \{p\}$ (or $\{\neg p\}$) is written in the shorthand form as $\mathcal{V}_M(s, p) = p$ (or $\neg p$) for individual property p.

In the subsequent discussion, OIMC is represented by an assumption model checking in E only rather than in C. In such a situation, the ending states in E, i.e. reentry states re, are assumed with some truth values seeded from B, $\mathcal{V}_B(re, cl(p))$. In other words, the assumption function As is defined as $As(re) = \mathcal{V}_B(re, cl(p))$.

Definition 12 *The assumption function As of an assumption model checking in E is* proper *at an ending state re if the assumed truth values are exactly those resulted at re from the standard model checking in C, i.e. $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$.*

OIMC is particularly useful for open systems - future extensions are not known in advance. In the typical case of component refinement, the composite model C is regarded as the combination of two sequential components B and E. Besides execution paths already defined in B, a typical execution path in C consists of three parts: initially in B, next in E and then back to B. Associated with each reentry state re of E is a computation tree in B represented by a set of temporal properties. If these properties at re are known, without loss of correctness, we can efficiently derive the properties at the upstream states in E by ignoring model checking in B to find the properties at re. Instead, we start from these reentry states with the associated properties; check the upstream of the extension component, and then the base component if needed [2]. The properties associated with a reentry state re are assumed with truth values from B, $As(re) = \mathcal{V}_B(re, cl(p))$. Of course, this method is reliable if $As(re)$ is proper.

3.2 Fundamental Issues of OIMC

Given a structure $B = \langle S, \Sigma, s_0, R, L\rangle$ defined upon AP as in Definition 1, a property p is inherent to B if $B, s_0 \models p$. The base B is then composed with an extension E. We propose the way in which E does not violate p in B.

Definition 13 *Given the property p inherent to the base B, an extension E is* consistent *with B regarding p if $C, s_0 \models p$, where C is the composition of B and E.*

From Definition 13, if B and E are consistent, E does not violate the property p in B. To this stage, the most fundamental issue of OIMC with can be described as follows:

Given B and p, what are the necessary conditions for E so that B and E are consistent with respect to p?

Because the formal model allows the extension to override the base as in Definition 5, that fact complicates the matter to a great extent. There is another question. Which overriding manner can still maintain the soundness of the verification task? In addition, this work also examines the scalability of OIMC. We are concerned with the preservation of p not only for the addition of E but also for many subsequent extensions to C.

3.3 Basic Notations

In the subsequent discussion, the following notations are used.

- If a state d is reachable from a state a, a is the ascendant while d is called the descendant. If $(a, \perp, d) \in R$, d is an immediate descendant of a.
- b_ϕ, e_ϕ, c_ϕ are sets of states in the base B, the extension E and the composition C at which the property ϕ holds. Formally, $b_\phi = \{s \in S_B | \, B, s \models \phi\}$. e_ϕ, c_ϕ are similarly defined.

[2]There is no need to check the base again if the consistency constraints associated with the exit states of B are preserved at the corresponding in-states of E as of Theorem 17 later.

- B' is the model resulted from B after removing all overridden transitions, if any. Surely, in case of additive composition, B and B' are exactly identical. In case of overriding composition, B' is very much similar to B. In fact, the labels at corresponding states between the two can be derived rather quickly.

- $s_1 \xrightarrow{dep.} s_2$ - verification dependency due to model checking, i.e. from an ascendant state s_1 to a descendant state s_2. The truth values of s_1 with respect to a set of properties can be derived only if the counterparts of s_2 are already determined.

- Also $c_1 \xrightarrow{dep.} c_2$ - verification dependency due to model checking between components c_1 and c_2. In terms of OIMC, model checking c_2 requires input from the result of OIMC within c_1. Hence, c_2 can only be checked after OIMC has been completed in c_1. This notation is used only in Section 6.4.

- A computation path can be represented by two ways: state names or respective labels at those states. For example, a path $\pi = s_0 s_1 s_2 \ldots$ is shown in the first form via states s_i. In the second form, π can be represented by the sequence of relevant truth values (i.e. *formula pattern*) at those states on the path. Typically, we consider two patterns:

  1. $\pi = f^*$ - the pattern for the path π in which $\mathbf{EG}\, f$ holds. This pattern means f holds at all states along the path.
  2. $\pi = (f, \bot)^* \, (\bot, g) \, (\bot, \bot)^*$ - the pattern for any path in which $\mathbf{E}\,[f \, \mathbf{U} \, g]$ holds. This pattern requires the first property f to hold until the second property g is found. Each 2-tuple represents the pair of truth values with respect to f and g respectively.

For comprehensibility, we start the presentation by considering the simple case of composition: the interface consists of a single exit state ex and a single reentry state re. Further, we assume that re is not an ascendant of ex. E does not have a cycle containing ex and re either. The generalized interface will be dealt with in the soundness issue after the theoretical foundation has been established. In Section 4.2 about the soundness of OIMC, the previous assumption among ex and re is dropped.

As a first step in verifying properties in the extension, some observations have been made. All properties under consideration are CTL. Regarding to a CTL property p, the labels of a state only depends on the labels of its descendants in the computation paths. Therefore, in the event of an extension addition, if the computation tree (i.e. shape and labels at nodes) associated with a given state is not changed, its labels with respect to $cl(p)$ are unchanged. From that observation, it is easy to conclude that the truth values of re and its descendant states are preserved regardless of the extension. The states of interest are the rest, namely ex and its ascendants.

Chapter 4

Fundamental Theoretical Foundation of OIMC

4.1 Properties Preservation at Base States

4.1.1 A Theorem on Additive-Only Composition

Due to the inherently inside-out characteristic of model checking, at the end of a task checking a property p, at each state s in B, besides the atomic proposition labels provided by L_B, $\mathcal{V}_B(s, cl(p))$ are also recorded. The truth values $\mathcal{V}_B(ex, cl(p))$ at the exit state ex serve as the conformance for the composition between B and E according to the theorems below.

Definition 14 *B and E are in* conformance *at an exit state ex (with respect to $cl(p)$) if $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$. They are in* strict conformance *if besides the regular conformance condition, with regards to any property in $cl(p)$ of the form $\mathbf{A}\,[f\,\mathbf{U}\,g]$ (or $\neg\mathbf{EG}\,f$) holding at ex, E does not make $\mathbf{A}\,[f\,\mathbf{U}\,g]$ false (or $\mathbf{EG}\,f$ true) at ex by patching a path of $(f \wedge \neg g)^*$ (or f^*) with another existing $(f \wedge \neg g)^*$ (or f^*) path in B to make a complete cycle $(f \wedge \neg g)^*$ (or f^*) through ex in C.*[1]

Theorem 15 *Given a model B and a CTL property p, an extension E is attached to the model B via an exit state ex. $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$, if B and E conform each other at ex.*

Intuitively, E does not affect the truth values with respect to $cl(p)$ at all base states if it conforms B at exit states. Formally, if there is a conformance between B and E, $\forall\phi \in cl(p) : b_\phi = c_\phi \cap S_B$. Figure 4.1 only shows that B and E are composable with respect to single property $p = \mathbf{EG}\,f$. In this figure, we do not care about the descendant states in E. Thus, E is intentionally left end-open so that the reentry state re is not explicitly displayed. In this part, what E can deliver at ex is important regardless of its descendants in E. We actually want $\forall\phi \in cl(p) : E, ex \models \mathcal{V}_B(re, \phi)$ no matter via regular model checking (the assumption at re is not needed) or assumption model checking (the assumption at re is required). The subsequent arguments on the theorem are still valid when the downstream of E converges to re.

[1]Due to the partial model in assumption model checking, there is a danger of mistakenly claiming $\mathbf{A}\,[f\,\mathbf{U}\,g]$ or $\neg\mathbf{EG}\,f$ to be true at interface states. How to check for these extra requirements is shown in Section 4.1.4. Regarding to the finite model, the regular conformance condition is sufficient.

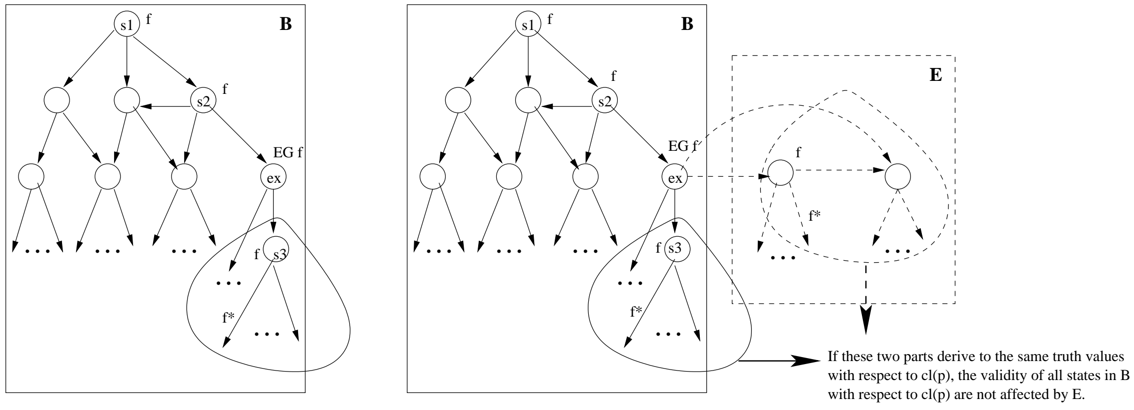

Figure 4.1: An illustration of B and E conformance in case of additive-only composition. The property $p = \mathbf{EG}\ f$ is preserved at ex and all states in B.

Most theorem proofs in this paper are essentially structured into two mutually related parts to prove the theorems with respect to the CTL set. It is easy to see that if part 1 (dealing with normal properties, namely the ten basic operators) and part 2 (dealing with the rest of CTL properties formed by the combination of normal properties with negation and union operators) are both justified, the theorems are valid with respect to the whole CTL set.

- Proving the theorems for the basic case - atomic propositions, i.e. the order $n = 0$. This initial proof claims that the theorems are true with respect to all normal properties with zero-order.
- The main proof consists of two parts:
  1. Proving the theorems for the ten normal temporal properties of an arbitrary order n by induction. By assuming the theorems for all CTL properties with the order $n \leq k$, the theorems are confirmed for normal properties with the order $n = (k+1)$. Basically, we only need to prove four basic cases: $\mathbf{EX}f$, $\mathbf{EG}f$, $\mathbf{E}\,[f\ \mathbf{U}\ g]$ and $\mathbf{A}\,[f\ \mathbf{U}\ g]$ [2]. The other six operators are rather straightforward derivatives from the four.
  2. Proving the theorems for any other CTL property with order less-than-equal to k, i.e. a property formed by normal properties via negation ($\neg$) and union ($\vee$) operators.

The justification for the induction proof structure with respect to property order is described below:

K1. Assuming that the theorems are valid for any CTL property with order $\leq k$.

K2. Part 1 of the proof: based on (K1), proving the theorems with respect to four basic operators $\mathbf{EX}\ f$, $\mathbf{EG}\ f$, $\mathbf{E}\,[f\ \mathbf{U}\ g]$ and $\mathbf{A}\,[f\ \mathbf{U}\ g]$ with order $(k+1)$.

[2]The operator $\mathbf{A}\,[f\ \mathbf{U}\ g]$ is exceptional, even though it can be expressed via two operators $\mathbf{E}\,[f\ \mathbf{U}\ g]$ and $\mathbf{EG}\ f$. That is because in the equivalent form expressed via the two operators, there is a union operator strengthening the closure set than the closure set of $\mathbf{A}\,[f\ \mathbf{U}\ g]$ itself (see Definitions 7 and 8 in Section 3.1). Hence, $\mathbf{A}\,[f\ \mathbf{U}\ g]$ takes a separate part in the proofs.

K3. By part 2 of the proof with respect to the negation operator, the theorems are valid with regards to the corresponding negation forms of the above four operators, namely $\neg\mathbf{EX}\, f$, $\neg\mathbf{EG}\, f$, $\neg\mathbf{E}\,[f\ \mathbf{U}\ g]$, $\neg\mathbf{A}\,[f\ \mathbf{U}\ g]$.

K4. From (K2) and (K3), the theorems can be proved regarding the remaining six normal CTL properties of order $(k+1)$ since those properties can be expressed via negation forms of the four basic properties above (Refer to Definition 8 in Section 3.1).

K5. Similar to (K3), via the negation operator, the theorems are proved with respect to the negation forms of the six remaining basic operators in (K4). Up to this stage, all ten normal CTL properties of order (k+1) and their negation forms are proved. That is, the theorems hold for all single CTL properties with order $(k+1)$.

K6. By part 2 on the union operator and (K5), the theorems are proved with respect to any composite CTL property with order $(k+1)$. To this point, the theorems are justified for all CTL properties with order $(k+1)$. The induction proof is done.

Due to the assumption on the reentry state re for presentation comprehensibility at the end of Section 3.3, namely re is not an ascendant of ex in B, there is surely no loop between re and ex formed by B and E in which each feature contains a half of the loop. Hence, the extra loop-checking in Definition 14 for $\mathbf{A}\,[f\ \mathbf{U}\ g]$ and $\neg\mathbf{EG}\, f$ is not needed. We are only concerned with the regular conformance condition at this stage. Below is the proof for Theorem 15.

Proof

i). $p \in AP$: $cl(p) = \{p\}$. All atomic proposition labels for states in B are preserved after composition, according to the Definition 5. In other words, for any $s \in S_B$: $\mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ in case of $|p| = 0$, .

Suppose the theorem is true for all CTL properties whose order is less than or equal to k. We prove the theorem is also valid for all single CTL properties whose order is $(k+1)$. We only need to prove in four basic cases, namely $\mathbf{EX}\, f$, $\mathbf{EG}\, f$ where $|f| = k$; and $\mathbf{E}\,[f\ \mathbf{U}\ g]$, $\mathbf{A}\,[f\ \mathbf{U}\ g]$ where $max(|f|, |g|) = k$, i.e. (K1).

ii). $p = \mathbf{EX}\, f$ where $|f| = k$: Let s be a state in S_B.

If E conforms B at ex with respect to $cl(p)$ then certainly E conforms B with respect to $cl(f) \subset cl(p)$. Due to the conformance B and E on $cl(f)$ and $|f| = k$, according to the inductive hypothesis above, the set b_f is not affected by E, i.e. $b_f = c_f \cap S_B$. From the perspective of $p = \mathbf{EX}\, f$, the portion of c_p in B certainly includes b_p: $b_p \subseteq c_p \cap S_B$ by the following arguments.

Given a state $s \in b_p$, at least one of its immediate descendants must possess f label. Because $b_f = c_f \cap S_B$, that descendant is still a member of c_f and hence it causes s to satisfy $p = \mathbf{EX}\, f$, i.e. $s \in c_p$. That is, $\forall s \in b_p : s \in c_p$. Hence, $b_p \subseteq c_p \cap S_B$.

We now prove that $b_p = c_p \cap S_B$ by showing that E does not re-label any state in $b_{\neg p}$ into c_p. In terms of $p = \mathbf{EX}\, f$, the only base state possibly under E's relabeling is ex. However, because E and B agree at ex with respect to p, there is no new state to be added by E to the set $c_p \cap S_B$ other than those in b_p.

In brief, b_p does not change due to E. We have: $\forall s \in S_B, \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f)$ and

- $\mathcal{V}_B(s, p) = \mathcal{V}_C(s, p)$ (by above arguments).

- $\mathcal{V}_B(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the inductive hypothesis for $|f| = k$.

The theorem is valid for $p = \mathbf{EX}\, f$ whose order is $(k+1)$.

iii). $p = \mathbf{EG}\, f$ where $|f| = k$: Similar to the above arguments, since f is a sub-formula of p, if B and E conform at ex with respect to $cl(p)$ then they do so with respect to $cl(f)$. By the hypothesis, the set b_f is not affected by E, i.e. $b_f = c_f \cap S_B$.

Because C contains B completely, any path in B with the pattern f^* is certainly in C. If $s \in b_p$ then $s \in c_p$. That is, $b_p \subseteq c_p \cap S_B$. We now prove that the contribution of E to the $c_p \cap S_B$ is not new from that of b_p, i.e. $b_p = c_p \cap S_B$. We are only concerned with the ascendants of ex because E only affects to ex directly and its ascendants indirectly via ex. There are two cases to consider.

First, if $B, ex \not\models p$, then $E, ex \not\models p$ due to the conformance, i.e. for all computation paths from ex to E, their languages are not of the pattern f^*. In such a case, there is surely no path from an ascendant in B through ex to E with the language of f^*. Hence, E does not relabel any state in $b_{\neg p}$ into c_p. The labels in base states with respect to p are preserved.

Second, $B, ex \models p$ and so does $E, ex \models p$. That means, at ex there are at least two paths. Each lies completely in either B or E. From the perspective of $\mathbf{EG}\, f$, there is no difference between having two or one paths. If an ascendant $s \notin b_p$, there is no path from s to ex such that f holds along the path. As a result, the extra path in E does not help to assign p to s. If $s \in b_p$, with the occurrence of E, s has at least one more path of f^* via E. Overall, with respect to p, the truth values of base states are preserved after composing E: $b_p = c_p \cap S_B$.

In brief, $\forall s \in S_B, \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f)$ and

- $\mathcal{V}_B(s, p) = \mathcal{V}_C(s, p)$ (by above arguments).
- $\mathcal{V}_B(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the inductive hypothesis for $|f| = k$.

The theorem is valid for $p = \mathbf{EG}\, f$ whose order is $(k+1)$.

iv). $p = \mathbf{E}\,[f\, \mathbf{U}\, g]$ where $max(|f|, |g|) = k$: We have $cl(f), cl(g) \subset cl(p)$. If B and E conform then B and E are certainly in agreement for $cl(f)$ and $cl(g)$. Of course, $|f|, |g| \leq k$, by the hypothesis, $b_f = c_f \cap S_B$ and $b_g = c_g \cap S_B$.

Because C contains B completely, any path in B with pattern $(f, \bot)^*\ (\bot, g)\ (\bot, \bot)^*$ is certainly in C. In other words, $b_p \subseteq c_p \cap S_B$. We now prove that the contribution of E to the $c_p \cap S_B$ is not new from that of b_p. As above, we are only concerned with ex and its ascendants. There are two cases to consider.

First, if $B, ex \not\models p$, then $E, ex \not\models p$, i.e. for all computation paths from ex to E, their languages are not that of $(f, \bot)^*\ (\bot, g)\ (\bot, \bot)^*$. In such a case, E could not relabel any state in $b_{\neg p}$ to be a member of c_p either. Hence, $b_p = c_p \cap S_B$.

Second, if $B, ex \models p$ and so does $E, ex \models p$. That means, at ex there are at least two "suffix" paths with the pattern $(f, \bot)^*\ (\bot, g)\ (\bot, \bot)^*$. Each lies completely in either B or E. From the perspective of $\mathbf{E}\,[f\, \mathbf{U}\, g]$, there is no difference between having two or one paths. If an ascendant s is not in b_p then there is no prefix path from s to ex such that either $(f, \bot)^*$ or $(f, \bot)^*\ (\bot, g)\ (\bot, \bot)^*$ hold along this prefix path. This prefix pattern is needed to concatenate with the suffix path from ex to form a p-satisfying path rooted at s. In brief, E does not relabel p to any $s \in b_{\neg p}$.

Finally, we have: $\forall s \in S_B, \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f) \cup cl(g)$ and

- $\mathcal{V}_B(s, p) = \mathcal{V}_C(s, p)$ (by above arguments).
- $\mathcal{V}_B(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the inductive hypothesis for $|f|, |g| \leq k$.

The theorem is valid for $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ whose order is $(k+1)$.

v). $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ where $max(|f|, |g|) = k$: As above, we are only concerned with ex and its ascendants. There are two cases to consider.

First, if $B, ex \not\models p$, then $E, ex \not\models p$. There exists some computation path from ex to E, whose pattern is not $(f,\ \bot)^*\ (\bot,\ g)\ (\bot,\ \bot)^*$. Thus, E could not relabel p at ex or any ex's ascendants in $b_{\neg p}$. That is, $b_p = c_p \cap S_B$.

Second, $B, ex \models p$ and $E, ex \models p$. From ex, all paths - completely lying in B (I); involving some prefix in E and then back to B (II) - satisfy $[f\,\mathbf{U}\,g]$. For paths of types (I) and (II), it is obvious about their fulfillment with respect to p.[3]

From the above arguments, $C, ex \models p = \mathbf{A}\,[f\,\mathbf{U}\,g]$. For any ascendant s of ex, because the labels at ex are fixed while all intermediate base states between s and ex are unchanged both in terms of transitions and labels up to f, g (due to the inductive hypothesis), the truth value at s with respect to p is preserved. That is, $b_p = c_p \cap S_B$.

The theorem is valid for $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ whose order is $(k+1)$.

Briefly, to this stage, the theorem holds with regards to four basic normal operators (K2).

vi). For the other normal properties, the proof is simple as they can be directly derived from the above four normal properties - (K3).

First, $p = \mathbf{AX}\,f = \neg\mathbf{EX}\,(\neg f)$, $cl(\mathbf{AX}\,f) \equiv cl(\mathbf{EX}\,(\neg f))$ (Definition 8). From the case **ii**, we have:

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{EX}\,(\neg f))) = \mathcal{V}_E(ex, cl(\mathbf{EX}\,(\neg f))),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{EX}\,(\neg f))) = \mathcal{V}_C(s, cl(\mathbf{EX}\,(\neg f))).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{AX}\,f)) = \mathcal{V}_E(ex, cl(\mathbf{AX}\,f)),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{AX}\,f)) = \mathcal{V}_C(s, cl(\mathbf{AX}\,f)).$$

The theorem holds for the normal property $p = \mathbf{AX}\,f$.

Second, $p = \mathbf{EF}\,f = \mathbf{E}\,[True\,\mathbf{U}\,f]$, $cl(\mathbf{EF}\,f) \equiv cl(\mathbf{E}\,[True\,\mathbf{U}\,f])$ (Definition 8). From the case **iv**, we have:

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{E}\,[True\,\mathbf{U}\,f])) = \mathcal{V}_E(ex, cl(\mathbf{E}\,[True\,\mathbf{U}\,f])),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{E}\,[True\,\mathbf{U}\,f])) = \mathcal{V}_C(s, cl(\mathbf{E}\,[True\,\mathbf{U}\,f])).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{EF}\,f)) = \mathcal{V}_E(ex, cl(\mathbf{EF}\,f)), \forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{EF}\,f)) = \mathcal{V}_C(s, cl(\mathbf{EF}\,f)).$$

The theorem holds for the normal property $p = \mathbf{EF}\,f$.

Third, $p = \mathbf{AG}\,f = \neg\mathbf{EF}\,(\neg f)$, $cl(\mathbf{AG}\,f) \equiv cl(\mathbf{EF}\,(\neg f))$ (Definition 8). From the right above arguments for $\mathbf{EF}\,f$, we have:

[3] For the paths of type (II), since re is not an ascendant of ex in B, there certainly exists no f^* loop between ex and re in C through the connection of two half cycles between re and ex in B and E. Hence, $C, ex \not\models \mathbf{EG}\,f$ and $C, ex \models \mathbf{A}\,[f\,\mathbf{U}\,g]$.

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{EF}\,(\neg f))) = \mathcal{V}_E(ex, cl(\mathbf{EF}\,(\neg f))),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{EF}\,(\neg f))) = \mathcal{V}_C(s, cl(\mathbf{EF}\,(\neg f))).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{AG}\, f)) = \mathcal{V}_E(ex, cl(\mathbf{AG}\, f)),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{AG}\, f)) = \mathcal{V}_C(s, cl(\mathbf{AG}\, f)).$$

The theorem holds for the normal property $p = \mathbf{AG}\, f$.

Fourth, $p = \mathbf{AF}\, f = \neg\mathbf{EG}\,(\neg f)$, $cl(\mathbf{AF}\, f) \equiv cl(\mathbf{EG}\,(\neg f))$ (Definition 8). From the case **iii**, we have:

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{EG}\,(\neg f))) = \mathcal{V}_E(ex, cl(\mathbf{EG}\,(\neg f))),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{EG}\,(\neg f))) = \mathcal{V}_C(s, cl(\mathbf{EG}\,(\neg f))).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{AF}\, f)) = \mathcal{V}_E(ex, cl(\mathbf{AF}\, f)), \forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{AF}\, f)) = \mathcal{V}_C(s, cl(\mathbf{AF}\, f)).$$

The theorem holds for the normal property $p = \mathbf{AF}\, f$.

Fifth, $p = \mathbf{A}[f\mathbf{R}g] = \neg\mathbf{E}[(\neg f)\mathbf{U}(\neg g)]$, $cl(\mathbf{A}[f\mathbf{R}g]) \equiv cl(\mathbf{E}[(\neg f)\mathbf{U}(\neg g)])$ (Definition 8). From the case **iv**, we have:

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{E}\,[(\neg f)\,\mathbf{U}\,(\neg g)])) = \mathcal{V}_E(ex, cl(\mathbf{E}\,[(\neg f)\,\mathbf{U}\,(\neg g)])),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{E}\,[(\neg f)\,\mathbf{U}\,(\neg g)])) = \mathcal{V}_C(s, cl(\mathbf{E}\,[(\neg f)\,\mathbf{U}\,(\neg g)])).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{A}\,[f\,\mathbf{R}\,g])) = \mathcal{V}_E(ex, cl(\mathbf{A}\,[f\,\mathbf{R}\,g])),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{A}\,[f\,\mathbf{R}\,g])) = \mathcal{V}_C(s, cl(\mathbf{A}\,[f\,\mathbf{R}\,g])).$$

The theorem holds for the normal property $p = \mathbf{A}\,[f\,\mathbf{R}\,g]$.

Sixth, $p = \mathbf{E}[f\mathbf{R}g] = \neg\mathbf{A}[(\neg f)\mathbf{U}(\neg g)]$, $cl(\mathbf{E}[f\mathbf{R}g]) \equiv cl(\mathbf{A}[(\neg f)\mathbf{U}(\neg g)])$ (Definition 8). From the case **v**, we have:

$$\text{If } \mathcal{V}_B(ex, cl(\mathbf{A}\,[(\neg f)\,\mathbf{U}\,(\neg g)])) = \mathcal{V}_E(ex, cl(\mathbf{A}\,[(\neg f)\,\mathbf{U}\,(\neg g)])),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{A}\,[(\neg f)\,\mathbf{U}\,(\neg g)])) = \mathcal{V}_C(s, cl(\mathbf{A}\,[(\neg f)\,\mathbf{U}\,(\neg g)])).$$

By a simple substitution between equivalent closure sets:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(\mathbf{E}\,[f\,\mathbf{R}\,g])) = \mathcal{V}_E(ex, cl(\mathbf{E}\,[f\,\mathbf{R}\,g])),$$
$$\forall s \in S_B : \mathcal{V}_B(s, cl(\mathbf{E}\,[f\,\mathbf{R}\,g])) = \mathcal{V}_C(s, cl(\mathbf{E}\,[f\,\mathbf{R}\,g])).$$

The theorem holds for the normal property $p = \mathbf{E}\,[f\,\mathbf{R}\,g]$.

Overall, the induction proof is completed for Part 1, i.e. Theorem 15 is true for all ten normal CTL properties - (K2, K4).

Concerning with Part 2, the proof is carried out for two cases: negation (i.e. $p = \neg f$, where f is normal) and union ($p = f \vee g$ where f, g are single CTL properties). Note that the closure set, $cl(\neg f) = cl(f)$ and $cl(f \vee g) = cl(f) \cup cl(g)$ as defined in Definition 7.

In the first case, the proof is simple via definition: if $p = \neg f$, $cl(p) = cl(f)$. Therefore,

$$\text{If } \mathcal{V}_B(ex, cl(f)) = \mathcal{V}_E(ex, cl(f)), \forall s \in S_B : \mathcal{V}_B(s, cl(f)) = \mathcal{V}_C(s, cl(f)).$$

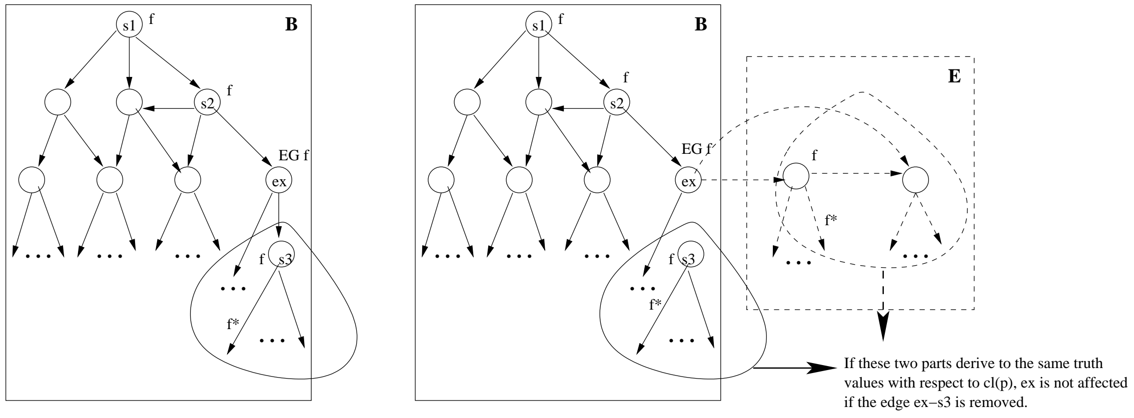

Figure 4.2: An illustration of B and E conformance in case of overriding composition. The property $p = \mathbf{EG}\ f$ is preserved at ex and all states in B.

By a simple substitution between two equal closure sets $cl(p)$ and $cl(f)$:

$$\Rightarrow \text{If } \mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p)),\ \forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p)).$$

Theorem 15 is valid for negation operator - (K3, K5).

In the second case, $p = f \vee g$, $cl(p) = cl(f) \cup cl(g)$ where f and g are single properties. Given the truth values with respect to $cl(p)$ are preserved at ex, the counterparts with respect to $cl(f)$ and $cl(g)$ are certainly preserved. For properties in $cl(f)$, as they are preserved at ex, according to the theorem so far for single properties - (K2, K3, K4, K5), the truth values with respect to $cl(f)$ are preserved at all base states. In case of $cl(g)$, the argument is similar. Because the member labels $cl(f)$, $cl(g)$ are preserved, their union labels $cl(p)$ are certainly preserved at all base states - (K6).

From arguments above, Part 2 of the proof is completed. Together with Part 1, Theorem 15 is proved. □

4.1.2 A Theorem on Limited-Overriding Composition

The previous section addresses the case in which E only introduces new behaviors to B without changing any base behavior. The following theorem is about the case in which the extension overrides base behavior.

Theorem 16 *Let C be a composition model formed by the base B and the extension E as of Theorem 15. Let C' be the new model resulted from C after removing some or even all out-going transitions from ex in B. If B and E conform with each other at ex with respect to a CTL property p, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_{C'}(s, cl(p))$*

Figure 4.2 illustrates this theorem. If B and E conform with each other, B and E can be composed even at the cost of some transitions from ex. The preserved property shown in this figure is $p = \mathbf{EG}\ f$. Like Figure 4.1 in Section 4.1.1, reentry state re is not explicitly shown. Formally, the theorem claims that if there is a conformance between B and E, $\forall \phi \in cl(p) : b_\phi = c'_\phi \cap S_B$.

Proof

The proof is very similar to the previous. It also consists of two parts: the first for the ten normal CTL properties, and another for logical operators ($\neg$ and $\vee$). We prove that $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. Further, due to Theorem 15, $\forall s \in S_B$: $\mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. The result is that: $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_{C'}(s, cl(p))$ for any CTL property p.

i). $p \in AP$: $cl(p) = \{p\}$. All atomic proposition labels for states in B are preserved after composition, according to the Definition 5. Furthermore, any transition removal does not affect those atomic label sets. In other words, for any $s \in S_B, \mathcal{V}_B(s, cl(p)) = \mathcal{V}_{C'}(s, cl(p))$ in case $|p| = 0$.

Suppose the theorem is valid for all CTL properties whose order is less than or equal to k. We prove the theorem is also valid for all normal CTL properties whose order is $(k+1)$. Like the previous proof in Section 4.1.1, only four basic normal properties, namely $\mathbf{EX}\, f$, $\mathbf{EG}\, f$, $\mathbf{E}\,[f\ \mathbf{U}\ g]$ and $\mathbf{A}\,[f\ \mathbf{U}\ g]$, are required. By an observation, if any out-going transitions are removed, the action only affects the truth values with respect to $cl(p)$ of ex directly and its ascendants indirectly via ex. Intuitively, because E patches any change at ex due to transition removal, this patch effectively keeps any ex's ascendant unchanged with regards to $cl(p)$.

ii). $p = \mathbf{EX}\, f$ where $|f| = k$: Due to B and E conformance at ex, though some transitions are removed, $\mathcal{V}_{C'}(ex, p) = \mathcal{V}_C(ex, p)$. There are two cases to consider.

First, $C, ex \models p$: If the removed transition contributes to the only valid path for ex with respect to p, i.e. to a state at which f holds, due to the conformance, there exists another state in E that can fill up this role. As a result, $C', ex \models p$.

Second, $C, ex \not\models p$, all paths from ex do not satisfying p. If some of transitions are removed, surely the computation tree rooted at ex gets smaller and there is no path satisfying p. The result after removal is that $C', ex \not\models p$.

There is no change at ex, $\forall s \in S_B : \mathcal{V}_{C'}(s, p) = \mathcal{V}_C(s, p)$. In addition, for any ascendant s of ex:

- The computation tree rooted at s is fixed in C and C'.
- $\mathcal{V}_C(s, cl(f)) = \mathcal{V}_{C'}(s, cl(f))$ (by the inductive hypothesis).
- For any immediate descendant d of s, $\mathcal{V}_C(d, cl(f)) = \mathcal{V}_{C'}(d, cl(f))$ (by the inductive hypothesis).

$$\Rightarrow \mathcal{V}_C(s, p) = \mathcal{V}_{C'}(s, p), \text{ where } p = \mathbf{EX}\, f$$

In summary, $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f)$ and

- $\forall s \in S_B : \mathcal{V}_{C'}(s, p) = \mathcal{V}_C(s, p)$ (above arguments).
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the inductive hypothesis for $|f| = k$.

From Theorem 15, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. We have: $\forall s \in S_B$: $\mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_B(s, cl(p))$. The theorem is valid for $p = \mathbf{EX}\, f$ whose order is $(k+1)$.

iii). $p = \mathbf{EG}\, f$ where $|f| = k$: Similarly, there are two cases to consider.

First, $C, ex \models p$: If the removed transition is *critical* to ex with respect to p [4], due to the conformance, there exists another path in E that can fill up this role. As a result, $C', ex \models p$.

[4]The criticality of a transition is defined in Section 4.2.1

Second, $C, ex \not\models p$, all paths from ex do not satisfy p. If some of transitions are removed, surely the computation tree rooted at ex gets smaller and there is no path satisfying p. The result after removal is that $C', ex \not\models p$.

There is no change at ex, $\forall s \in S_B : \mathcal{V}_{C'}(s,p) = \mathcal{V}_C(s,p)$. Furthermore, for any ascendant s of ex, we have:

- The computation tree rooted at s is fixed in C and C'.
- $\mathcal{V}_C(s, cl(f)) = \mathcal{V}_{C'}(s, cl(f))$ (by the inductive hypothesis).
- For any base descendant d of s, $\mathcal{V}_C(d, cl(f)) = \mathcal{V}_{C'}(d, cl(f))$ (by the inductive hypothesis).
- In case of ex - the only descendant with changing computation tree, in addition to $cl(f)$, the truth value with respect to p is also preserved.

$$\Rightarrow \mathcal{V}_C(s,p) = \mathcal{V}_{C'}(s,p) \text{ with } p = \mathbf{EG}\ f$$

In summary, $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f)$ and

- $\forall s \in S_B : \mathcal{V}_{C'}(s,p) = \mathcal{V}_C(s,p)$ (above arguments).
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the inductive hypothesis for $|f| = k$.

From Theorem 15, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. We have: $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_B(s, cl(p))$. The theorem is valid for $p = \mathbf{EG}\ f$ whose order is $(k+1)$.

iv). $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$ where $max(|f|, |g|) = k$: Identically, we consider two cases.

First, $C, ex \models p$: If the removed transition is critical to ex with respect to p, due to the conformance, there exists another path in E that can fill up this role. As a result, $C', ex \models p$.

Second, $C, ex \not\models p$, all paths from ex do not satisfy p. If some of transitions are removed, surely the computation tree rooted at ex gets smaller and there is no path satisfying p. The result after removal is that $C', ex \not\models p$.

There is no change at ex, $\forall s \in S_B : \mathcal{V}_{C'}(s,p) = \mathcal{V}_C(s,p)$. Moreover, for any ascendant s of ex, we have:

- The computation tree rooted at s is fixed in C and C'.
- $\mathcal{V}_C(s, cl(f)) = \mathcal{V}_{C'}(s, cl(f))$, $\mathcal{V}_C(s, cl(g)) = \mathcal{V}_{C'}(s, cl(g))$ (by the inductive hypothesis).
- For any base descendant d of s, $\mathcal{V}_C(d, cl(f)) = \mathcal{V}_{C'}(d, cl(f))$, $\mathcal{V}_C(s, cl(g)) = \mathcal{V}_{C'}(s, cl(g))$ (by the inductive hypothesis).
- In case of ex - the only descendant with changing computation tree, in addition to $cl(f)$ and $cl(g)$, the truth value with respect to p is also preserved.

$$\Rightarrow \mathcal{V}_C(s,p) = \mathcal{V}_{C'}(s,p) \text{ in which } p = \mathbf{E}\,[f\ \mathbf{U}\ g]$$

In summary, $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f) \cup cl(g)$ and

- $\forall s \in S_B : \mathcal{V}_{C'}(s, p) = \mathcal{V}_C(s, p)$.
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the hypothesis for $|f| \leq k$.
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(g)) = \mathcal{V}_C(s, cl(g))$ due to the hypothesis for $|g| \leq k$.

By Theorem 15, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. We have: $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_B(s, cl(p))$. The theorem is valid for $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ whose order is $(k+1)$.

v). $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ where $max(|f|, |g|) = k$: By analogy, there are two cases to consider.

First, $C, ex \models p$: If there are some transitions to be removed, the set of computation paths starting at ex in C' will be a true subset of the counterpart in C. Moreover, each computation path is a valid path with respect to $[f\ \mathbf{U}\ g]$. As a result, $C', ex \models p$.

Second, $C, ex \not\models p$, some path from ex does not satisfy p. After composition, surely even the computation tree rooted at ex in E itself violates $\mathbf{A}\ [f\ \mathbf{U}\ g]$. The result after removal is that $C', ex \not\models p$.

There is no change at ex, $\forall s \in S_B : \mathcal{V}_{C'}(s, p) = \mathcal{V}_C(s, p)$. Besides, for any ascendant s of ex, we have:

- The computation tree rooted at s is fixed in C and C'.
- $\mathcal{V}_C(s, cl(f)) = \mathcal{V}_{C'}(s, cl(f))$, $\mathcal{V}_C(s, cl(g)) = \mathcal{V}_{C'}(s, cl(g))$ (by the inductive hypothesis).
- For any descendant d of s, $\mathcal{V}_C(d, cl(f)) = \mathcal{V}_{C'}(d, cl(f))$, $\mathcal{V}_C(s, cl(g)) = \mathcal{V}_{C'}(s, cl(g))$ (by the inductive hypothesis).
- In case of ex - the only descendant with changing computation tree, in addition to $cl(f)$ and $cl(g)$, the truth value with respect to p is also preserved.

$$\Rightarrow \mathcal{V}_C(s, p) = \mathcal{V}_{C'}(s, p) \text{ where } p = \mathbf{A}\ [f\ \mathbf{U}\ g]$$

In summary, $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ because $cl(p) = \{p\} \cup cl(f) \cup cl(g)$ and

- $\forall s \in S_B : \mathcal{V}_{C'}(s, p) = \mathcal{V}_C(s, p)$.
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(f)) = \mathcal{V}_C(s, cl(f))$ due to the hypothesis for $|f| \leq k$.
- $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(g)) = \mathcal{V}_C(s, cl(g))$ due to the hypothesis for $|g| \leq k$.

From Theorem 15, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$. We have: $\forall s \in S_B : \mathcal{V}_{C'}(s, cl(p)) = \mathcal{V}_B(s, cl(p))$. The theorem is valid for $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ whose order is $(k+1)$.

vi). For the other normal properties, the proof is simple as they are derived from the above four normal properties. The proof is exactly as the counterpart in Theorem 15. The rest of six normal properties are proved.

Overall, the induction proof is completed for Part 1, i.e. Theorem 16 is true for all normal CTL properties.

Concerning with Part 2, the proof is carried out for two cases: negation (i.e. $p = \neg f$, where f is normal) and union ($p = f \vee g$, where f, g are single CTL properties) operators. It is identical to that in Theorem 15. Part 2 of the proof is completed. Together with Part 1, Theorem 16 is proved. □

4.1.3 The Feature Consistency Issue

Theorems 15 and 16 above are justified based on two facts: a simple interface (single exit and single reentry states) and the assumption about reentry state and exit state, i.e. the reentry state is not an ascendant of the exit state. The simple interface is used for simplicity. The assumption is used to guarantee that seeded values at re is proper, i.e. the assumption function As is proper. All arguments in the proofs can be well extended to the generalized interface of many exit, reentry and dual states as long as the assumption function As is proper, and there is a conformance between base and extension at all exit states.

The generalized version of Theorems 15 and 16 with respect to generalized interface about the conformance between base and extension, no matter the composition is additive-only or overriding, can be described in the following.

Theorem 17 *Given a base B and a property p, an extension E is composed with B via some interface states (either additive or overriding). Further, suppose that the assumption function As defined during model checking E is proper. If B and E conform with each other at all exit states, $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$.*

Formally, the theorem claims that if the base and the extension conform, $\forall \phi \in cl(p) : b_\phi = c_\phi \cap S_B$. From Theorem 17, given a property p holds on B, it continues to hold in C if B and E conform with each other at exit states. The following corollary is the answer to the problem prescribed in Section 3.2 - the key of the dissertation.

Corollary 18 *Given a model B and a CTL property p adhered to it, an extension E is attached to B at some interface states. Suppose that the assumption function As in model checking E is proper. E does not violate p inherent to B if B and E conform with each other at all exit states.*

The properness of the assumption function As is a major part for the soundness of the incremental verification to be dealt with in Section 4.2. In that section, instead of assuming As's properness, we prove it.

4.1.4 Open Incremental Model Checking

Incremental verification method should consist of the following steps:

1. Proving a CTL property of a base system B.

2. Deriving a set of constraints at the interface states of B such that if those constraints are preserved, the corresponding property is guaranteed. According to [10], those constraints are named *preservation constraints*. From Theorem 17, the preservation constraints are required at exit states only. At each exit state ex, the constraints are exactly $\mathcal{V}_B(ex, cl(p))$.

3. An extension feature E does not violate the above property of B if, during its execution, the constraints are preserved. In this activity, only the state space of the extension feature is verified.

Corresponding to each exit state ex, the algorithm to verify a preservation constraint in E can be briefly described as follows:

i. Seeding each reentry state re with the corresponding $\mathcal{V}_B(re, cl(p))$. They represent the ending states as of Definition 9. These seeded values are related to the soundness of the assumption function As to be discussed in Section 4.2.

ii. Executing the CTL assumption model checking procedure in E to check for ϕ, $\forall \phi \in cl(p)$.

iii. At the end of the model checking task, checking if $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$.

iv. If re is not an ascendant of ex in B', then simply skip to the next step. Otherwise, for the strict conformance, two extra checks for the non-existence of any $(f \wedge \neg g)^*$ or f^* loop to assure properties of the forms $\mathbf{A}\,[f\,\mathbf{U}\,g]$ and $\neg\mathbf{EG}\,f$ are required. Details of the checking are shown separately.

v. Repeating the procedure for other exit states.

At the end of the process, if at all exit states, the truth values with respect to $cl(p)$ are matched respectively. B and E are consistent.

The procedure of loop checking is below:
For any sub-formula: $\phi = \mathbf{A}\,[f\,\mathbf{U}\,g] \in cl(p)$ holding at ex in both B' and E, surely $B', ex \models \neg\mathbf{EG}\,f$; and $B', re \models \mathbf{A}\,[f\,\mathbf{U}\,g]$ (for seeding during assumption model checking within E so that there exists a chance for any f^* circle via ex and re). Due to $B', re \models \mathbf{A}\,[f\,\mathbf{U}\,g]$, $B', re \models \neg\mathbf{EG}\,f$. Because $\neg\mathbf{EG}\,f$ holds in both ex and re with respect to both B' and E via assumption model checking, we are certain that the circle of f^* between ex and re in C, if any, is the only path causes the label $\mathbf{EG}\,f$ to be turned on at these two states. Therefore, the equivalent check for a circle of f^* between ex and re can be carried out as follows:

1. Seeding at re the property $\mathbf{EG}\,(f \wedge \neg g)$ [5] and checking within E to see if $E, ex \models_{as} \mathbf{EG}\,(f \wedge \neg g)$. If so, there is definitely a path of $(f \wedge \neg g)^*$ from ex to re in E.

2. Seeding at ex the property $\mathbf{EG}\,(f \wedge \neg g)$ and checking within B' to see if $B', re \models_{as} \mathbf{EG}\,(f \wedge \neg g)$. If so, there is certainly a path of $(f \wedge \neg g)^*$ from re to ex in B'.

3. When both checks above return $\mathbf{EG}\,(f \wedge \neg g)$ at both ex and re, i.e. there exists a circle of $(f \wedge \neg g)^*$ between the two states, signaling that B and E do not conform with respect to the strict conformance condition.

For any sub-formula: $\phi = \neg\mathbf{EG}\,f \in cl(p)$ holding at ex in both B' and E, the check is similar to the above, except that $(f \wedge \neg g)$ is replaced by f.

4.2 The Soundness of Incremental Verification

4.2.1 Criticality of Transitions

This section focuses on the correctness of Theorem 17 in Section 4.1 after dropping the assumption on the properness of As. The question is that with respect to the generalized

[5] $\neg g$ is added to ensure that g is not turned on at any state in the circle. Otherwise, this circle of f^* still satisfies $[f\,\mathbf{U}\,g]$.

interface, whether the conformance between B and E at exit states can derive the preservation of properties in base states. The answer is yes for any additive-only or non-critical overriding composition, while it may fail for extreme cases of overriding composition where the overridden transition is *critical* to some sub-formula of p at ex.

The criticality of one or more transitions to a property p at state s is informally expressed by the fact that if those transitions are removed, the label for p at s is inverted. Initially, let $M = \langle S, \Sigma, s_0, R, L\rangle$ and $s \in S$. Further, $M' = \langle S, \Sigma, s_0, R \setminus \{(s, \perp, d)\}, L\rangle$ - the remainder of M after removing the transition. The following definition defines the criticality of the transition $(s, \perp, d)$ to the state s. For a group of transitions, the definition is similar.

Definition 19 *A transition* $(s, \perp, d) \in R$ *is* critical *to* s *with respect to* $cl(p)$ *if* $\exists \phi \in cl(p) : M, s \models \phi$ *and* $M', s \not\models \phi$.

In the base model B of Figure 4.10, the transition associated with the event e_1 is critical to the property $\mathbf{E}[f\mathbf{U}g]$ at the state ex_1. If e_1 is removed from B, then $B', ex_1 \models \neg\mathbf{E}[f\mathbf{U}g]$, whereas initially $B, ex_1 \models \mathbf{E}\,[f\ \mathbf{U}\ g]$.

Due to the inherently inside-out characteristic of model checking, at the end of a task checking a property p in $M = \langle S, \Sigma, s_0, R, L\rangle$, at each state s, $\mathcal{V}_M(s, cl(p))$ are recorded. To check the criticality of a transition $(s, \perp, d) \in R$ regarding any sub-property $\phi \in cl(p)$, the algorithm can be described briefly in the following.

1. $\phi \in ACTL$ and $\mathcal{V}_M(s, \phi) = \phi$, the transition is critical only if there is a unique immediate descendant d of s in M.

2. $\phi \in ECTL$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if there is a unique immediate descendant d of s in M.

3. $\phi = \mathbf{AX}\ f$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if $\nexists(s, \perp, d') \in R, d \neq d' : M, d' \not\models f$.

4. $\phi = \mathbf{AF}\ f$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if $\nexists(s, \perp, d') \in R, d \neq d' : M, d' \not\models \phi$.

5. $\phi = \mathbf{AG}\ f$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if $M, s \models f$ and $\nexists(s, \perp, d') \in R, d \neq d' : M, d' \not\models \phi$.

6. $\phi = \mathbf{A}\,[f\ \mathbf{U}\ g]$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if $M, s \models (f \wedge \neg g)$ and $\nexists(s, \perp, d') \in R, d \neq d' : M, d' \not\models \phi$.

7. $\phi = \mathbf{A}\,[f\ \mathbf{R}\ g]$ and $\mathcal{V}_M(s, \phi) = \neg\phi$, the transition is critical only if $M, s \models (\neg f \wedge g)$ and $\nexists(s, \perp, d') \in R, d \neq d' : M, d' \not\models \phi$.

8. For $\phi \in ECTL$ and $\mathcal{V}_M(s, \phi) = \phi$, it is similar to the cases 3-7.

4.2.2 Dependency Structure Among Base States

In Section 4.1, in the assumption model checking within E, the assumption function As is constructed by copying the truth values at reentry states in the model B directly. The copying step implicitly assumes that As is proper at all reentry states. This section

is mainly concerned with proving As's properness instead of assuming it, i.e. checking whether Theorem 17 remains valid if the assumption on the properness of As is dropped.

The CTL model checking procedure defines a dependency from an ascendant to its descendants in the sense that its truth value with respect to any CTL property is derived from the truth values of the descendants. In Section 4.1.4, to model check an exit state ex, we have implicitly assumed that the seeded $\mathcal{V}_B(re, cl(p))$ at reentry states are proper. A conservative thinking would treat the truth values at the reentry states with respect to $cl(p)$ in C to be different from those counterparts in B. As a matter of fact, when E extends B, it more likely causes the truth values of those states to change than to preserve. Our job is to prove that E preserves those truth values, if possible.

For the generalized interface of many exit and reentry states, a reentry state re depends on all base descendant states which also perform exit states of the interface. Those exit states again depend on their reachable reentry states in E. Similarly, this model checking dependency chain among interface states continues towards the end of B's computation paths. Formally, we have the following structure constructed from two basic dependency types due to model checking.

- A base state s depends on a base descendant ex if ex is an exit state: $\mathcal{V}_B(s, p) \xrightarrow{dep.} \mathcal{V}_B(ex, p)$. The state s is called B-dependents on ex.

- Within E, an exit state ex depends on all reachable reentry states re: $\mathcal{V}_E(ex, p) \xrightarrow{dep.} \mathcal{V}_E(re, p)$. Because these dependencies are supplied by E, ex is called E-dependent on re.

By this dependency chain, a structure of nodes is constructed. Each node corresponds to a state in B. Between nodes is a dependency link showing the source node is either a B- or a E-dependent on the destination. This structure is different from the model B in the following manner. At least one end of a link is an interface state: a B-dependent link departs from a base state and finishes at an exit state; a E-dependent link connects an exit state to a reachable reentry state. Moreover, this structure is entirely defined after the addition of E.

From this structure, we can conclude that if there is no cycle in the above dependency chain, the incremental verification is sound (As is surely proper if B and E conform). We start from the reentry states re in the lowest downstream of dependency chain whose descendants are unaffected by E. The assumed truth values at re are proper and exactly $\mathcal{V}_B(re, cl(p))$. $\mathcal{V}_E(ex, cl(p))$ derived at any exit state ex, which is an E-dependent on re, are reliably established. If B and E conform at ex, we can update the set of new reentry states by uniting current re states with those newly verified ex since some of these ex may be dual. The process is then repeated until all exit states are confirmed. Theorem 17 is still valid without the assumption on As's properness.

Figure 4.3 presents a composition which can give a sound incremental verification. In this figure, there is a cycle within base states s_2, s_3 but that cycle does not involve any interface state. From B and E, the dependency structure is essentially acyclic. In details,

- $\mathcal{V}_C(s_1, p)$, $\mathcal{V}_C(s_2, p)$ and $\mathcal{V}_C(s_3, p) \xrightarrow{dep.} \mathcal{V}_C(s_4, p)$ and $\mathcal{V}_C(s_5, p)$ (B-dependents) because within B, s_4, s_5 are reachable exit states from s_1, s_2 and s_3.

- $\mathcal{V}_C(s_4, p) \xrightarrow{dep.} \mathcal{V}_C(s_5, p)$ and $\mathcal{V}_C(s_6, p)$ (E-dependent) since s_5 and s_6 are reachable reentry states from s_4 within E.

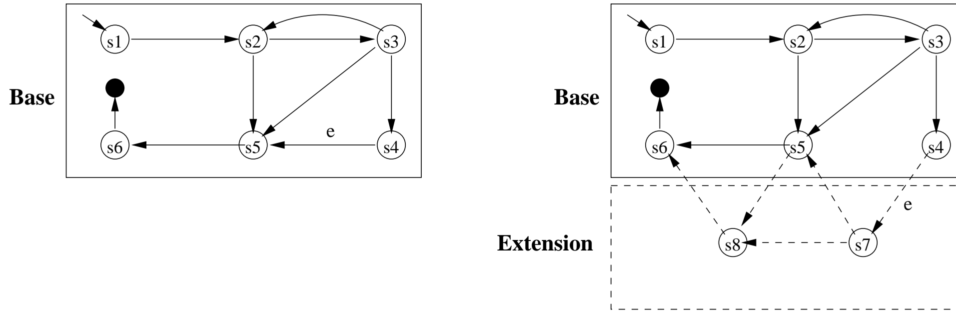

Figure 4.3: An example of B and E composition in which the verification result is sound. E overrides the transition in B associated with event e.

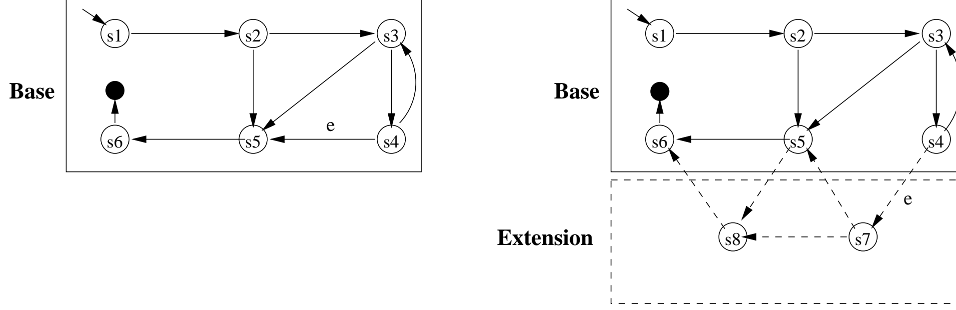

Figure 4.4: Another example of B and E composition in which a sound verification result can be delivered.

- $\mathcal{V}_C(s_5, p) \xrightarrow{dep.} \mathcal{V}_C(s_6, p)$ (E-dependent) as s_6 is a reachable reentry state from s_5 in E.

Initially, p holds in B. After composition, $\mathcal{V}_C(s_6, cl(p)) = \mathcal{V}_B(s_6, cl(p))$ is valid because it is a base state unaffected by E. By an assumption model checking within E whose input is $\mathcal{V}_B(s_6, cl(p))$, we can determine $\mathcal{V}_E(s_5, cl(p))$. More importantly, the result of this verification is well-justified. Suppose B and E are verified to conform at s_5, the verification task is repeated by feeding two inputs $\mathcal{V}_C(s_5, cl(p))$ and $\mathcal{V}_C(s_6, cl(p))$ to verify the conformance at s_4. If s_4 is also confirmed then the rest of states, namely s_1, s_2 and s_3, are automatically qualified without any further model checking.

Figure 4.4 presents another composition which can give a sound incremental verification. This figure is slightly different from the previous figure. There is a cycle in the base involving the interface state s_4. However, the dependency structure is still acyclic. The incremental model checking is still sound. The dependency structure is exactly the same as that of Figure 4.3. The verification procedure is hence identical.

However, if there is a cycle in the dependency structure, ensuring the soundness of incremental verification is more complicated. In fact, in some special cases of B and E, the incremental verification does not deliver sound results since As is not proper. There are three basic cases in which a cyclic dependency could come from:

1. Two exit states ex_1 and ex_2 are B-dependents of each other, namely two B-links form the cycle. Refer to Figure 4.6 for the illustration.[6]

[6] In fact, there are two other simpler cases with this circular dependency style. First, re_1 and re_2 are

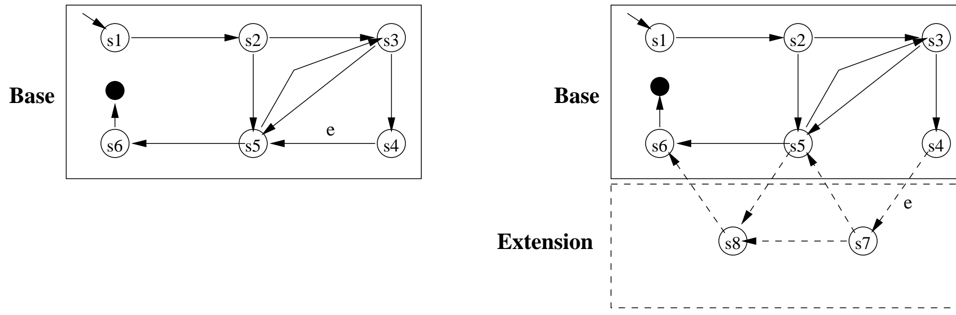

Figure 4.5: An example of B and E composition in which the incremental verification may not be sound.

2. Two exit states ex_1 and ex_2 are reentry states of each other, i.e. two E-links form the cycle. This case is depicted in Figure 4.8.

3. re is actually a base ascendant of ex, i.e. the assumption about the relationship between re and ex is dropped. In this case, a B-link and an E-link form the cycle. The case is shown in Figure 4.11.

Figure 4.5 depicts a composition in which OIMC may not deliver a sound result. Unlike Figures 4.3 and 4.4, there is a base connection from reentry state s_5 back to an ascendant of exit state s_4. This difference causes s_5 to be an ascendant of s_4. Therefore, by incremental model checking, $\mathcal{V}_C(s_5, cl(p))$ is entirely determined by $\mathcal{V}_C(s_4, cl(p))$. On the other hand, within the extension, we have the opposite, namely $\mathcal{V}_E(s_4, cl(p))$ is decided by $\mathcal{V}_C(s_5, cl(p))$. As the labels at these two states mutually affect each other, the result delivered by OIMC - based on the principle of fixing labels at one state to verify the other - may be unsound.

We will examine each circular dependency in turn. An observation is made. If ex still preserves its truth values with respect to $cl(p)$ then its base ascendants preserve their truth values as well. That leaves the work in this section to focus on properties preservation at interface states - exit and reentry - only. The soundness problem in essence consists of two parts:

1. Proving the assumed truth values at reentry states are in fact proper instead of simply assuming them. (Soundness Problem 1)

2. Based on the above properness at reentry states, proving that the truth values with respect to $cl(p)$ are preserved at exit states if B and E conform. (Soundness Problem 2)

If As can be proved to be proper and the preservation constraints at exit states are justified, Theorem 17 can be perfectly applied: properties preservation at all base states.

4.2.3 Cyclic Dependency with Base Links Only

This basic case happens when two exit states ex_1 and ex_2 are mutually ascendants in B. As a result, we have in both B and C: $ex_1 \xrightarrow{dep.} ex_2$ and $ex_2 \xrightarrow{dep.} ex_1$. In this cyclic

in a cycle of B, while ex_1 and ex_2 are distinct exit states. Second, ex_1 and re_2 are in a cycle, whereas re_1 and ex_2 are separate. These two cases are ignored.

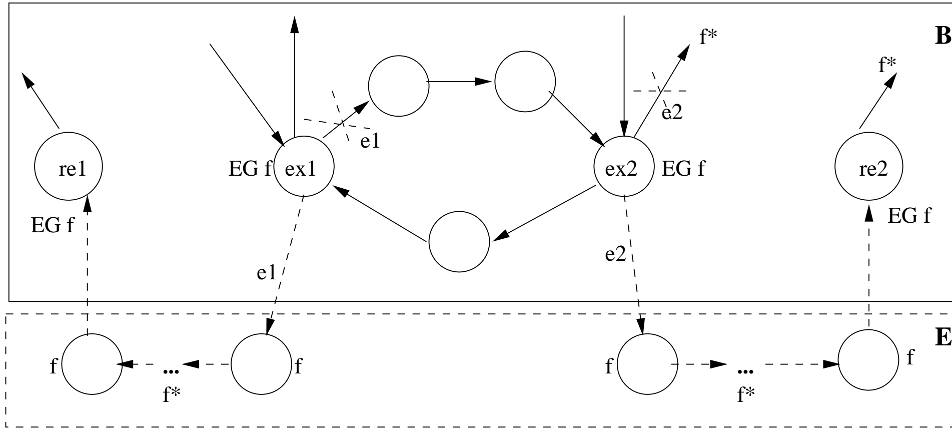

Figure 4.6: An example of cyclic dependency due to base links only.

dependency, we can not verify a state based on the fixed truth values of the other which is indeed derived from the verified truth values of the former state.

Figure 4.6 illustrates the basic case where reentry states re_1, re_2 are distinct from ex_1, ex_2. In the section below, we prove that Theorem 17 is still valid in this structure, after dropping the assumption on As. The proof follows the usual proof structure to show that: As is proper at both reentry states re_1 and re_2 - Soundness Problem 1; (II) truth values with respect to $cl(p)$ at both exit states ex_1 and ex_2 are preserved - Soundness Problem 2.

Proof

In this basic case, re_1 and re_2 are not ascendants of ex_1 and ex_2. The extension does not affect re_1 and re_2. Hence, As is proper at both re_1 and re_2. Now we are left with only Soundness Problem 2 - proving properties preservation at ex_1 and ex_2 if B and E conform.

i). $p \in AP$: the atomic label sets at all states in B do not change. p is preserved at both ex_1 and ex_2. Theorem 17 is valid.

Suppose the theorem is valid for all CTL properties whose order is less-than-equal k. We prove it holds for normal properties p where $|p| = (k+1)$ via basic four operators $\mathbf{EX}\, f, \mathbf{EG}\, f$, $\mathbf{E}\,[f\; \mathbf{U}\; g]$ and $\mathbf{A}\,[f\; \mathbf{U}\; g]$.

ii). $p = \mathbf{EX}\, f$: At ex_1, its truth value with respect to p is not changed, even in case of transition removal due to E. We consider two cases. First, $B, ex_1 \models p$, then $E, ex_1 \models p$. There exists at least an immediate descendant in E causing $C, ex_1 \models p$. Second, $B, ex_1 \not\models p$, hence $E, ex_1 \not\models p$. The result is that all immediate descendants of ex_1 in B and E contain no f label. That is, $C, ex_1 \not\models p$. Overall, the truth value at ex_1 with respect to p is preserved.

In case of ex_2, it is similar. We have: $cl(p) = \{p\} \cup cl(f)$ and

- $\mathcal{V}_B(ex_1, p) = \mathcal{V}_C(ex_1, p)$ and $\mathcal{V}_B(ex_2, p) = \mathcal{V}_C(ex_2, p)$ (due to above arguments).
- $\mathcal{V}_B(ex_1, cl(f)) = \mathcal{V}_C(ex_1, cl(f))$ and $\mathcal{V}_B(ex_2, cl(f)) = \mathcal{V}_C(ex_2, cl(f))$ (due to the hypothesis where $|f| = k$).

In summary, Soundness Problem 2 is proved in case $p = \mathbf{EX}\, f$ with the order $(k+1)$.

iii). $p = \mathbf{EG}\, f$: Similarly, there are two cases to consider. First, $B, ex_1 \models p$ and hence $E, ex_1 \models p$: there exists at least a path in E from ex_1 to re_1 and back to B such

that $C, ex_1 \models p$. At ex_1, its truth value with respect to p is not changed, even in case of transition removal, due to E.

Second, $B, ex_1 \not\models p$ and $E, ex_1 \not\models p$. All paths rooted at ex_1 do not satisfy p. Even after transition removal, the set of computation paths gets smaller and certainly it does not fulfill p. $C, ex_1 \not\models p$.

In case of ex_2, it is similar. In brief, the truth value with respect to p is preserved at both ex_1 and ex_2. We have: $cl(p) = \{p\} \cup cl(f)$ and

- $\mathcal{V}_B(ex_1, p) = \mathcal{V}_C(ex_1, p)$ and $\mathcal{V}_B(ex_2, p) = \mathcal{V}_C(ex_2, p)$ (due to above arguments).
- $\mathcal{V}_B(ex_1, cl(f)) = \mathcal{V}_C(ex_1, cl(f))$ and $\mathcal{V}_B(ex_2, cl(f)) = \mathcal{V}_C(ex_2, cl(f))$ (due to the hypothesis where $|f| = k$).

In summary, Soundness Problem 2 is proved in case $p = \mathbf{EG}\, f$ with the order $(k+1)$.

iv). $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$: Identically, there are two cases to consider. First, $B, ex_1 \models p$ and hence $E, ex_1 \models p$: there exists at least a path in E from ex_1 to re_1 and back to B such that $C, ex_1 \models p$. At ex_1, its truth value with respect to p is not changed, even in case of transition removal, due to E.

Second, $B, ex_1 \not\models p$ and $E, ex_1 \not\models p$. All paths rooted at ex_1 do not satisfy p. Even after transition removal, the set of computation paths gets smaller and certainly it does not fulfill p. $C, ex_1 \not\models p$.

In case of ex_2, it is similar. In brief, the truth value with respect to p is preserved at both ex_1 and ex_2. We have: $cl(p) = \{p\} \cup cl(f) \cup cl(g)$ and

- $\mathcal{V}_B(ex_1, p) = \mathcal{V}_C(ex_1, p)$ and $\mathcal{V}_B(ex_2, p) = \mathcal{V}_C(ex_2, p)$ (due to above arguments).
- $\mathcal{V}_B(ex_1, cl(f)) = \mathcal{V}_C(ex_1, cl(f))$ and $\mathcal{V}_B(ex_2, cl(f)) = \mathcal{V}_C(ex_2, cl(f))$ (due to the inductive hypothesis where $|f| \leq k$).
- $\mathcal{V}_B(ex_1, cl(g)) = \mathcal{V}_C(ex_1, cl(g))$ and $\mathcal{V}_B(ex_2, cl(g)) = \mathcal{V}_C(ex_2, cl(g))$ (due to the inductive hypothesis where $|g| \leq k$).

In summary, Soundness Problem 2 is proved in case $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$ with the order $(k+1)$.

v). $p = \mathbf{A}\ [f\ \mathbf{U}\ g]$ where $max(|f|, |g|) = k$: Identically, at ex_1, its truth value with respect to p is not changed, even in case of transition removal. We can prove the preservation of the truth value with respect to p at ex_1 according to the following.

First, $B, ex_1 \not\models p$, at least a path from ex_1 does not satisfy p. Surely even the computation tree rooted at ex_1 in E itself violates $\mathbf{A}\,[f\,\mathbf{U}\,g]$. The result is that $C, ex_1 \not\models p$.

Second, $B, ex_1 \models p$: Due to the preservation of f and g labels in all states, certainly all paths from ex_1 not through ex_2 are still valid with respect to $[f\ \mathbf{U}\ g]$. The only concerned paths are those via ex_2 (whose truth values may change due to composition). If all those paths reach g before ex_2 then they still satisfy $[f\ \mathbf{U}\ g]$ after composition due to preservation of f and g in the base states. That is, the truth value of ex_1 with respect to p is unchanged. Suppose there exists a path having f along until ex_2 (and only that path may violate $[f\,\mathbf{U}\,g]$ after composition). This "prefix" path will be concatenated with the set of computation paths rooted at ex_2. By the semantic of $\mathbf{A}\ [f\ \mathbf{U}\ g]$ at ex_1, due to this prefix path, $B, ex_2 \models \mathbf{A}\ [f\ \mathbf{U}\ g]$. By similar arguments, potential p-violating paths from ex_2 after composition are those via ex_1 with f to be labeled at all intermediate states. This situation creates a cycle between ex_1 and ex_2 in B such that f holds in all states. Refer to Figure 4.7 for the illustration. However, this f^* cycle between ex_1 and

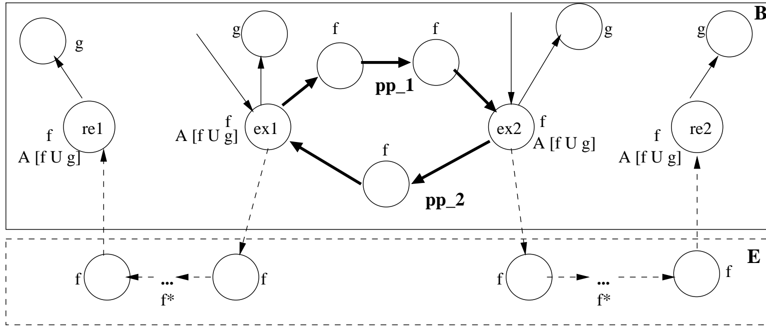

Figure 4.7: An illustration for composition with base-only cyclic dependency which still preserves $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ at exit states (pp_1 and pp_2 can not exist at the same time).

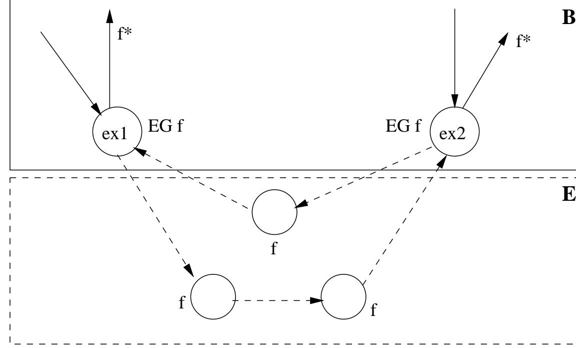

Figure 4.8: An example of cyclic dependency due to extension links only.

ex_2 can not exist. If so, $B, ex_1 \models \mathbf{EG}\, f \Rightarrow B, ex_1 \not\models \mathbf{A}\,[f\,\mathbf{U}\,g]$ which is contrasting with $B, ex_1 \models p$. Thus, we are certain that pp_1 and pp_2 - two halves of the cycle - can not co-exist. $\mathbf{A}\,[f\,\mathbf{U}\,g]$ continues to hold in ex_1 after composition, even with transition removal.

In case of ex_2, it is similar. In brief, the truth value with respect to p is preserved at both ex_1 and ex_2. By analogy, truth values with respect to all lower order sub-formulae of $cl(p)$ are preserved due to the inductive hypothesis, Soundness Problem 2 is proved in case $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ whose order is $(k + 1)$.

vi). For the other normal properties, the proof is simple as they are derived from the above four normal properties. The proof is exactly as the counterpart in Theorem 15. The rest of six normal properties are proved.

Until this stage, part 1 of the proof is completed. We turn to part 2 involving $p = \neg f$ (f is normal) and $p = f \vee g$ (f and g are single CTL properties). In the first case, the complementary set $b_f = S_B \setminus b_p$ of the set b_p does not change due to part 1. So does b_p. Soundness Problem 2 is proved for negation operator. Similar, in the latter case, $b_p = b_f \vee b_g$ is not changed because both b_f and b_g are not changed due to part 1 and the proof in the negation operator.

Overall, the proof of Soundness Problem 2 is completed. Theorem 17 is still valid for this circular dependency style. □

4.2.4 Cyclic Dependency with Extension Links Only

In this case, two exit states ex_1 and ex_2 are mutually reachable reentry states of each other. As a result, we have in both E and C: $ex_1 \xrightarrow{dep.} ex_2$ and $ex_2 \xrightarrow{dep.} ex_1$. These two states affects each other during model checking. Like the previous section, copying directly the labels from B into E for the assumption model checking is not sound as the characteristic of that seeding step is to fix the labels at one state to verify the other. Figure 4.8 illustrates this basic cyclic dependency case. We prove that Soundness Problems 1 and 2 are valid in the additive and non-critical overriding composition. On the contrary, they may not be in the critical-overriding composition because As is not proper, i.e. Soundness Problem 1 fails. Its counter-example is shown in the proof.

Proof

i). $p \in AP$: the atomic label sets at all states in B do not change. Both soundness problems are justified.

Suppose both soundness problems are proved for all CTL properties whose order is less-than-equal k. We prove them for normal properties p for the additive and non-critical overriding composition where $|p| = (k+1)$ via $\mathbf{EX}\, f, \mathbf{EG}\, f$, $\mathbf{E}\,[f\,\mathbf{U}\,g]$ and $\mathbf{A}\,[f\,\mathbf{U}\,g]$.

ii). $p = \mathbf{EX}\, f$: First, if $B, ex_1 \models p$, there exists an immediate descendant s of ex_1 in B possessing f label. After composition, due to the hypothesis, s still preserves f. As a result, $C, ex_1 \models p$.

Second, $B, ex_1 \not\models p$ and hence $E, ex_1 \not\models p$. That is, all immediate descendant states of ex_1 do not satisfy f. By the inductive hypothesis, their truth values with respect to f are preserved after composition. So, $C, ex_1 \not\models p$.

The truth value with respect to p at ex_1 is preserved. By a similar argument, the same thing happens to ex_2. Together with the inductive hypothesis on the preservation of truth values with respect to $cl(f)$ at ex_1 and ex_2, we conclude that the truth values with respect to $cl(p)$ are preserved at all exit states. This conclusion implies two points:

- $\mathcal{V}_B(ex_1, cl(p)) = \mathcal{V}_C(ex_1, cl(p))$ and $\mathcal{V}_B(ex_2, cl(p)) = \mathcal{V}_C(ex_2, cl(p))$: As is proper at both "reentry states" ex_1 and ex_2. That is, Soundness Problem 1 is proved.
- Properties preservation at both exit states ex_1 and ex_2. That is, Soundness Problem 2 is justified.

Both soundness problems are completed for $p = \mathbf{EX}\, f$ of the order $(k+1)$.

iii). $p = \mathbf{EG}\, f$: Similarly, first, consider the case $B, ex_1 \models p$. For additive and non-critical overriding composition, because the computation path satisfying p, rooted at ex_1 and lying completely in B is preserved in C both in terms of states and their labels up to $cl(f)$ (due to the inductive hypothesis for $|f| = k$). Hence ex_1 still meets p's requirement after composition. As a result, $C, ex_1 \models p$.

Second, if $B, ex_1 \not\models p$ and hence $E, ex_1 \not\models p$. That is, all paths rooted at ex_1 lying either in B or E do not satisfy p. Neither do they after composition, $C, ex_1 \not\models p$.

The truth value with respect to p at ex_1 is preserved. By a similar argument, the same thing happens to ex_2. Together with the inductive hypothesis on the preservation of truth values with respect to $cl(f)$ at ex_1 and ex_2, we conclude that the truth values with respect to $cl(p)$ are preserved at all exit states. This conclusion implies two things:

- $\mathcal{V}_B(ex_1, cl(p)) = \mathcal{V}_C(ex_1, cl(p))$ and $\mathcal{V}_B(ex_2, cl(p)) = \mathcal{V}_C(ex_2, cl(p))$: As is proper at both "reentry states" ex_1 and ex_2. That is, Soundness Problem 1 is confirmed.

- Properties preservation at both exit states ex_1 and ex_2. That is, Soundness Problem 2 is solved.

Both soundness problems are completed for $p = \mathbf{EG}\, f$ of the order $(k+1)$.

iv). $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$: Identically, we consider two cases. First, $B, ex_1 \models p$. For additive and non-critical composition, because the computation path satisfying p, rooted at ex_1 and lying completely in B is preserved in C both in terms of states and their labels up to $cl(f)$ and $cl(g)$ (due to the inductive hypothesis for $|f|, |g| \leq k$). Hence ex_1 continues to satisfy p after composition. As a result, $C, ex_1 \models p$.

Second, if $B, ex_1 \not\models p$ and hence $E, ex_1 \not\models p$. That is, all paths rooted at ex_1 lying either in B or E do not satisfy p. Neither do they after composition, $C, ex_1 \not\models p$.

The truth value with respect to p at ex_1 is preserved. By a similar argument, the same thing happens to ex_2. Together with the inductive hypothesis on the preservation of truth values with respect to $cl(f)$ and $cl(g)$ at ex_1 and ex_2, we conclude that the truth values with respect to $cl(p)$ are preserved at all exit states. This conclusion implies two points:

- $\mathcal{V}_B(ex_1, cl(p)) = \mathcal{V}_C(ex_1, cl(p))$ and $\mathcal{V}_B(ex_2, cl(p)) = \mathcal{V}_C(ex_2, cl(p))$: *As* is proper at both "reentry states" ex_1 and ex_2. That is, Soundness Problem 1 is proved.

- Properties preservation at both exit states ex_1 and ex_2. That is, Soundness Problem 2 is established.

Both soundness problems are proved for $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ of the order $(k+1)$.

v). $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ where $max(|f|, |g|) = k$: Identically, for additive and non-critical composition, at ex_1, its truth value with respect to p is not changed. We can prove the preservation of p at ex_1 according to the following.

First, $B, ex_1 \not\models p$, at least a path from ex_1 does not satisfy p. In additive and non-critical composition, certainly in that invalid path with respect to $[f\,\mathbf{U}\,g]$ are preserved all f and g labels along its states. In C, the path is also invalid. The result is $C, ex_1 \not\models p$.

Second, $B, ex_1 \models p$: Due to the preservation of f and g labels in all states, certainly all paths from ex_1 in B are still valid with respect to $[f\,\mathbf{U}\,g]$. Similar, all paths in E but not through ex_2 are still valid in C. The only concerned paths are those from ex_1 in E via ex_2 (whose truth values may change due to composition). If all those paths reach g before ex_2 then they still satisfy $[f\,\mathbf{U}\,g]$ after composition due to preservation of f and g markings in the intermediate states. That is, the truth value of ex_1 with respect to p is unchanged. Suppose there exists a path keeping f along until ex_2 (and only that path may violate $[f\,\mathbf{U}\,g]$ after composition). This prefix path will be concatenated with the set of paths at ex_2. By the semantic of $E, ex_1 \models_{as} \mathbf{A}\,[f\,\mathbf{U}\,g]$, due to this prefix path, $B, ex_2 \models \mathbf{A}\,[f\,\mathbf{U}\,g]$. By similar arguments, potential p-violating paths from ex_2 after composition are those in E via ex_1 with f to be labeled at all intermediate states. This situation creates a cycle between ex_1 and ex_2 completely in E such that f holds in all states. Refer to Figure 4.9 for the illustration. However, this f^* cycle between ex_1 and ex_2 invalidate ex_1 and ex_2 regarding $\mathbf{A}\,[f\,\mathbf{U}\,g]$. If there exists the cycle, $E, ex_1 \models \mathbf{EG}\, f \Rightarrow E, ex_1 \not\models \mathbf{A}\,[f\,\mathbf{U}\,g]$ which is contrasting with the conformance with $B, ex_1 \models p$. We are certain that the cycle does not exist. Thus, $\mathbf{A}\,[f\,\mathbf{U}\,g]$ continues to hold in ex_1 after composition.

In case of ex_2, it is similar. In brief, p is preserved at both ex_1 and ex_2. By analogy, together with the preservation of truth values with respect to $cl(f)$ and $cl(g)$ due to the

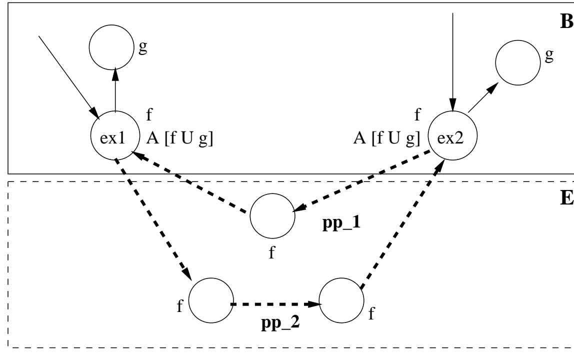

Figure 4.9: An illustration for the additive-only composition with extension-only cyclic dependency which still preserves $\mathbf{A}\,[f\,\mathbf{U}\,g]$ at exit states (pp_1 and pp_2 can not exist at the same time).

inductive hypothesis, the truth values with respect to $cl(p)$ at ex_1 and ex_2 are preserved. This conclusion implies two things:

- $\mathcal{V}_B(ex_1, cl(p)) = \mathcal{V}_C(ex_1, cl(p))$ and $\mathcal{V}_B(ex_2, cl(p)) = \mathcal{V}_C(ex_2, cl(p))$: *As* is proper at both "reentry states" ex_1 and ex_2. That is, Soundness Problem 1 is proved.

- Properties preservation at both exit states ex_1 and ex_2. That is, Soundness Problem 2 is validated.

Both soundness problems are proved for $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$ of the order $(k+1)$.

vi). For the other normal properties, the proof is simple as they are derived from the above four normal properties. The proof is exactly as the counterpart in Theorem 15. The rest of six normal properties are proved.

However, in case of overriding composition, if the descendant is critical to the truth value of ex_1 with respect to $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$, after its removal ($B$ becomes B'), $B', ex_1 \not\models p$. ex_1 has to rely on paths through E to satisfy p. For ex_2, if that scenario also happens, the final result is that ex_1, ex_2 mutually depend on each other. In E, even computation paths from ex_1 to ex_2 and vice versa are of the form $(f,\ \neg g)^*$, the ending states ex_1 and ex_2 do not guarantee any more path in B' with the suffix pattern of $(f,\ \neg g)^*\ (\bot,\ g)\ (\bot,\ \bot)^*$. Overall, $C, ex_1 \not\models p$ and $C, ex_2 \not\models p$, i.e. $\mathcal{V}_C(ex_1, cl(p)) \neq \mathcal{V}_B(ex_1, cl(p))$. *As* is not proper at ex_1 and similarly at ex_2. At states ex_1 and ex_2, initially in B, the property $\mathbf{E}\,[f\,\mathbf{U}\,g]$ is satisfied. But in the composition model C, the property (being crossed), even very weak in this case as $\mathbf{E}\,[f\ \mathbf{U}\ g]$, no longer holds at those states. Two soundness problems could not be proved for this overriding case. This failing case is illustrated in Figure 4.10.

Until this point, we have proved part 1 for the two soundness problems in this cyclic structure with respect to the additive and non-critical overriding composition. OIMC may not be sound in some extreme cases of critical-transition removal. For the part 2, the argument is very similar to the counterpart with respect to negation and union operators of Theorem 15. In short, OIMC is sound for all CTL properties under this kind of circular dependency of the additive and non-critical overriding composition. For the critical overriding composition, it may not. In such a circumstance, OIMC requires an extra checking at removed transitions. Details are discussed in Section 4.2.1. □

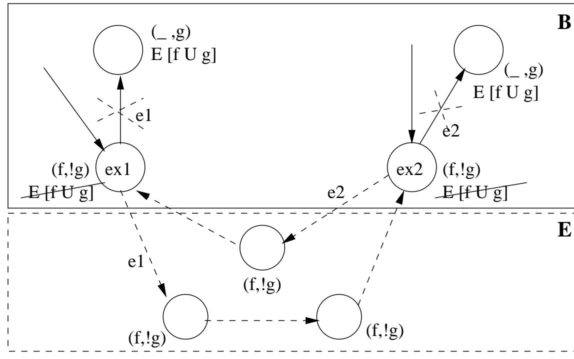

Figure 4.10: An example of composition failing to preserve $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$ in case of extension-only cyclic dependency.

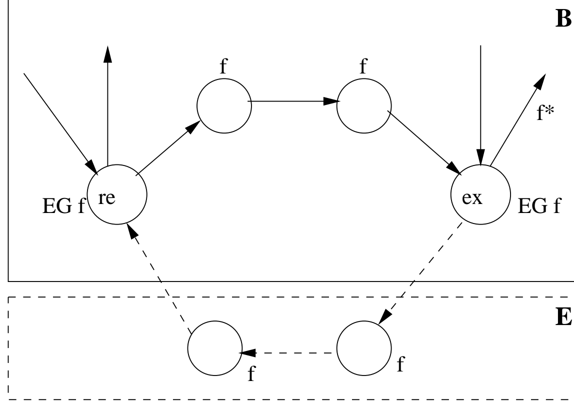

Figure 4.11: An example of cyclic dependency due to both base and extension links.

4.2.5 Cyclic Dependency with Both Base and Extension Links

In this structure, re is actually an ascendant of ex in B. As a result, we have: $re \xrightarrow{dep.} ex$ (in B and C) and $ex \xrightarrow{dep.} re$ (in E and C). By analogy to the previous cyclic dependency cases, these two states mutually determine the truth values of each other with respect to $cl(p)$ during model checking. Simply copying the labels at re from B into E for the assumption model checking is not sound. Figure 4.11 illustrates this basic cyclic dependency case. Like the previous section, we can prove that OIMC is sound in the additive and non-critical overriding composition (Soundness Problems 1 and 2 are proved). In contrast, it may not be sound in the critical overriding composition because Soundness Problem 1 may fail. Its counter-example is also shown in the proof. Importantly, during the proof, the reason for the enforcement of loop-checking in Definition 14 is also explained.

Proof

i). $p \in AP$: the atomic label sets at all states in B do not change. OIMC is sound in the basic case.

Suppose both soundness problems are proved for all CTL properties whose order is less-than-equal k. We prove them for normal properties p for the additive and non-critical overriding composition where $|p| = (k+1)$ via $\mathbf{EX}\,f, \mathbf{EG}\,f$, $\mathbf{E}\,[f\ \mathbf{U}\ g]$ and $\mathbf{A}\,[f\ \mathbf{U}\ g]$.

ii). $p = \mathbf{EX}\,f$: We consider two cases. First, if $B, ex \models p$, there exists an immediate

descendant s of ex labeled with f. After composition, due to the hypothesis, s still preserves f. As a result, $C, ex \models p$.

Second, $B, ex \not\models p$ and hence $E, ex \not\models p$. That is, all immediate descendant states of ex do not satisfy f. By the inductive hypothesis, their truth values with respect to f are preserved after composition. Thus, $C, ex \not\models p$.

The truth value with respect to p at ex is preserved. Together with the inductive hypothesis on the preservation of truth values with respect to $cl(f)$ at ex, we conclude that the truth values with respect to $cl(p)$ are preserved at ex. Consider the computation tree rooted at re. Basically, its shape and labels at states up to $cl(f)$ do not change after composing E. The only change occurs at ex. Fortunately, its labels do not. Overall, the change in the shape of the computation tree rooted at ex does not affect labels at re. In terms of assumption model checking, the truth values at re with respect to $cl(p)$ are also preserved.

In brief, we have:

- $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$: Properties preservation at re, i.e. As is proper at re. Soundness Problem 1 is established.

- $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_C(ex, cl(p))$: That is, Soundness Problem 2 is proved.

Both soundness problems are completed for $p = \mathbf{EX}\, f$ of the order $(k+1)$.

iii). $p = \mathbf{EG}\, f$: Similarly, first, consider the case $B, ex \models p$. Due to additive and non-critical overriding composition, because the computation path satisfying p, rooted at ex and lying completely in B is preserved in C both in terms of states and their labels up to $cl(f)$ (due to the inductive hypothesis for $|f| = k$). Hence ex still meets p's requirement after composition: $C, ex \models p$.

Second, if $B, ex \not\models p$ and hence $E, ex \not\models p$. That is, all paths rooted at ex lying either in B or E do not satisfy p. Now the tricky portion is related with the introduction of the extra checking for f^* loop in Definition 14 when $\neg\mathbf{EG}\, f$ holds at ex and re. What if $B', re \models \neg\mathbf{EG}\, f$ only occurs if $B', ex \models \neg\mathbf{EG}\, f$? That means, there exists at least a path from re to ex in B' in which f^* holds at all intermediate states. Unfortunately, as shown in Figure 4.12, the computation tree at ex in C is a superset of that in the assumption model checking in E. The extra branch causes trouble to $\mathcal{V}_C(ex, p)$. The illustration is shown in Figure 4.13. Initially, in B and B', $\neg\mathbf{EG}\, f$ holds at both states ex and re. If E extends B in the manner that there is a f^* path from ex to re. According to a separate assumption model checking in E, given $B', re \models \neg\mathbf{EG}\, f$, certainly $E, ex \models_{as} \neg\mathbf{EG}\, f$. That is, separate model checking in B and E gives the conformance at ex with respect to $\neg\mathbf{EG}\, f$. However, within C, a f^* loop exists through the patching of two f^* paths between re and ex in B' and E. This loop ensures $C, ex \models \mathbf{EG}\, f$. So does at re. This is the reason for the enforcement of the extra checking of the f^* loop in Definition 14. Once B and E are in *strict conformance*, this loop never arises, $\neg\mathbf{EG}\, f$ is preserved in C at both ex and re. That is, As is proper at re.

The truth value with respect to p at ex is preserved. By a similar argument in the case (ii) above, the truth values at re with respect to $cl(p)$ are also preserved. In brief, we have:

- $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$: Properties preservation at re, i.e. As is proper at re. Soundness Problem 1 is solved.

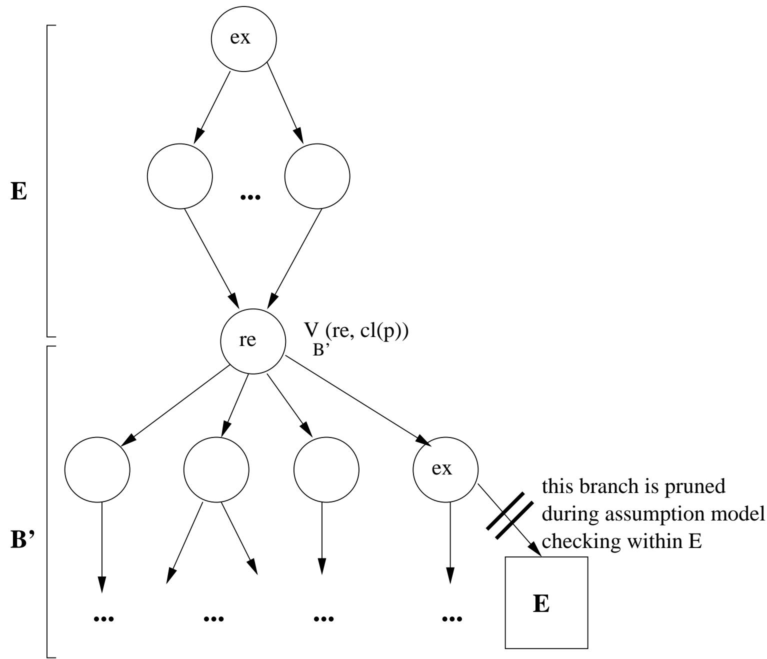

Figure 4.12: The difference between assumption model checking in E and regular model checking in C, in terms of the computation tree at ex within E, in the base-extension circular dependency.

- $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_C(ex, cl(p))$: That is, Soundness Problem 2 is proved.

Both soundness problems are proved for $p = \mathbf{EG}\, f$ of the order $(k + 1)$.

iv). $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$: Identically, we consider two cases for the additive and non-critical overriding composition. First, $B, ex \models p$. Because the computation path satisfying p, rooted at ex and lying completely in B is preserved in C both in terms of states and their labels up to $cl(f)$ and $cl(g)$ (due to the inductive hypothesis for $|f|, |g| \leq k$). Hence ex continues to satisfy p after composition. As a result, $C, ex \models p$.

Second, if $B, ex \not\models p$. So, $E, ex \not\models p$. That is, all paths rooted at ex lying either in B or E do not satisfy p. Neither do they after composition, $C, ex \not\models p$.

The truth value with respect to p at ex is preserved. By a similar argument in the case (ii) above, the truth values at re with respect to $cl(p)$ are also preserved. In brief, we have:

- $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$: Properties preservation at re, i.e. As is proper at re. Soundness Problem 1 is justified.
- $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_C(ex, cl(p))$: That is, Soundness Problem 2 is proved.

Both soundness problems are proved for $p = \mathbf{E}\,[f\,\mathbf{U}\,g]$ of the order $(k + 1)$.

v). $p = \mathbf{A}\,[f\,\mathbf{U}\,g]$ where $max(|f|, |g|) = k$: Identically, for the additive and non-critical overriding composition, at ex, its truth value with respect to p is not changed. We can prove the preservation of p at ex according to the following.

First, $B, ex \not\models p$, at least a path from ex does not satisfy p. In additive and non-critical overriding composition, certainly in that invalid path with respect to $[f\,\mathbf{U}\,g]$ are preserved all f and g labels along its states. In C, the path is also invalid. The result is $C, ex \not\models p$.

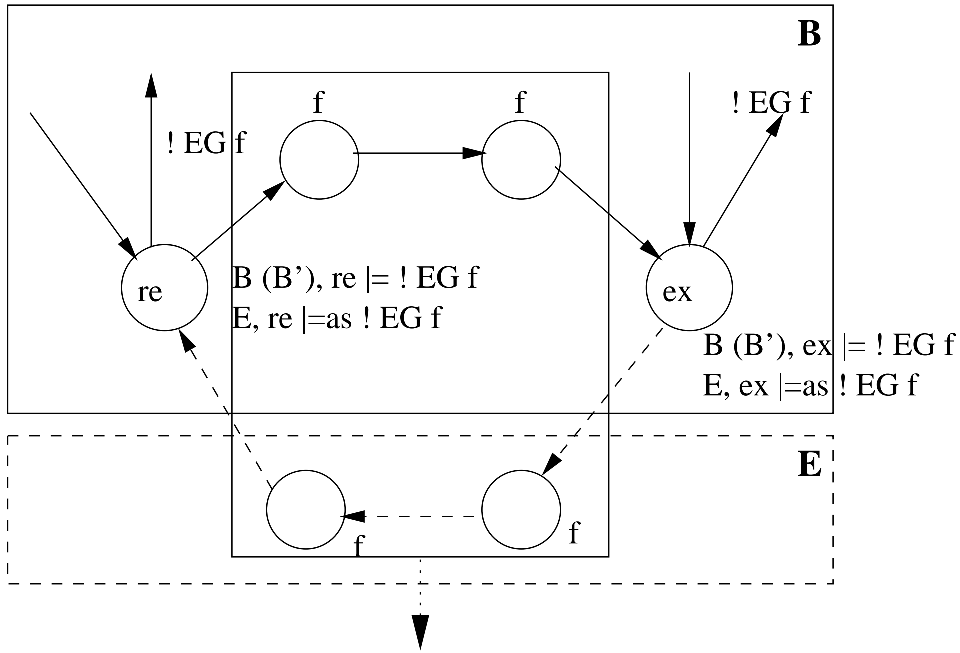

Figure 4.13: Strict conformance condition: The need of extra checking for any f^* loop between ex and re at which $\neg\mathbf{EG}\ f$ holds.

Second, $B, ex \models p$: Due to the preservation of f and g labels in all states, certainly all paths from ex in B are still valid with respect to $[f\ \mathbf{U}\ g]$. Similar, all paths rooted at ex in E but not through re are still valid in C. The only concerned paths are those in E from ex via re (whose truth values may change due to composition). If all those paths reach g before re then they still satisfy $[f\ \mathbf{U}\ g]$ after composition due to preservation of f and g labels in the intermediate states. That is, the truth value of ex with respect to p is unchanged. Suppose there exists a path keeping f along until re (and only that path may violate $[f\ \mathbf{U}\ g]$ after composition). This prefix path will be concatenated with the set of paths at re. From the semantic of $E, ex \models_{as} \mathbf{A}\ [f\ \mathbf{U}\ g]$, due to this prefix path, $B, re \models \mathbf{A}\ [f\ \mathbf{U}\ g]$. By similar arguments, potential p-violating paths from re after composition are those in B via ex with f to be labeled at all intermediate states. This situation creates a cycle between ex_1 and ex_2 completely in C such that f holds in all states. Refer to Figure 4.14 for the illustration. The following explains why the extra checking is enforced in Definition 14.

In terms of infinite model, if $\mathbf{A}\ [f\ \mathbf{U}\ g]$ holds at ex and re then the extra checking for any $(f \wedge \neg g)^*$ loop between ex and re is essential. Figure 4.14 shows the need for extra checking with respect to any $\phi = \mathbf{A}\,[f\,\mathbf{U}\,g] \in cl(p)$ since the regular conformance condition may mistakenly claim the preservation of $\mathbf{A}\ [f\ \mathbf{U}\ g]$ at exit states. By assumption model checking in E after setting $\mathbf{A}\ [f\ \mathbf{U}\ g]$ at re due to $B', re \models \mathbf{A}\ [f\ \mathbf{U}\ g]$, we can easily derive in the figure, $E, ex \models_{as} \mathbf{A}\ [f\ \mathbf{U}\ g]$. Under the regular conformance condition, B and E conform with respect to $cl(p)$. However, the conclusion of property preservation at ex from this regular conformance condition is wrong. In such a case, the *strict conformance* condition is required because it checks for the absence of $(f \wedge \neg g)^*$ loop between ex and re. Initially, before composition, $\mathbf{A}\ [f\ \mathbf{U}\ g]$ holds at both ex and re. Further, there is a path of $(f \wedge \neg g)^*$ from re to ex lying in B'. However, after composition, E patches

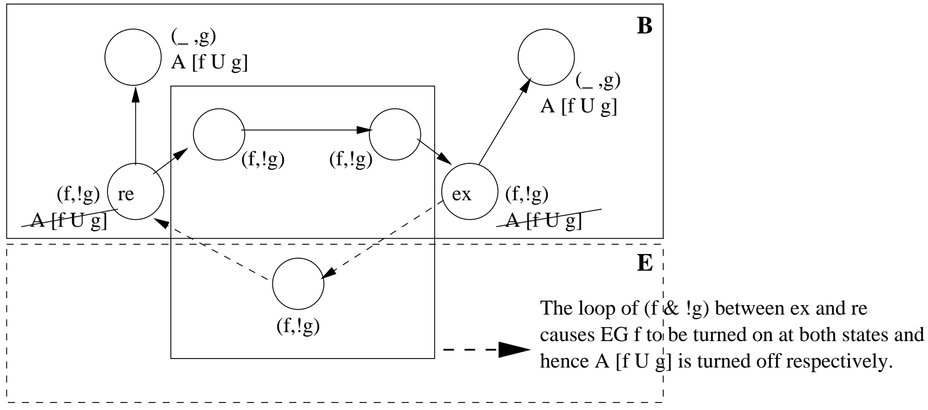

Figure 4.14: Strict conformance condition: The need of extra checking for any $(f \wedge \neg g)^*$ loop between ex and re at which $\mathbf{A}\,[f\ \mathbf{U}\ g]$ holds.

another $(f \wedge \neg g)^*$ path from ex to re. The final result is $C, ex \models \mathbf{EG}\, f$ (the loop $(f \wedge \neg g)^*$ connecting ex and re via the combination of the two above paths), i.e. $C, ex \not\models \mathbf{A}\,[f\,\mathbf{U}\,g]$. This is the reason to enforce another extra loop checking in Definition 14.

Overall, the truth value with respect to p is preserved at ex. By a similar argument in the case (ii), the truth values at ex and re with respect to $cl(p)$ are preserved. In brief, we have:

- $\mathcal{V}_B(re, cl(p)) = \mathcal{V}_C(re, cl(p))$: Properties preservation at re, i.e. As is proper at re. Soundness Problem 1 is established.
- $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_C(ex, cl(p))$: That is, Soundness Problem 2 is proved.

Both soundness problems are proved for $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$ whose order is $(k+1)$.

vi). For the other normal properties, the proof is simple as they are derived from the above four normal properties. The proof is exactly as the counterpart in Theorem 15. The rest of six normal properties are proved. Theorem 17 is valid, with regards to strict conformance condition, for additive and non-critical overriding composition.

However, in case of transition overriding, if the descendant is critical to the truth value of ex with respect to p, after its removal (B becomes B'), $B', ex \not\models p$. ex has to rely on E to satisfy p. For example, from ex to re in the extension, f holds along all states of the path. For re, an extreme scenario happens such that its truth value with respect to p depends solely on ex, namely from re to ex in the base, $(f, \neg g)$ holds along all states of the path. The final result is that there is a cycle between ex, re in C and $(f, \neg g)$ holds at all states. However, if there is no state in the cycle from which there is a path with the suffix pattern of $(f,\ \bot)^*\ (\bot,\ g)\ (\bot,\ \bot)^*$, all states in the cycle, including ex and re, do not satisfy p. That is, $\mathcal{V}_C(re, cl(p) \neq \mathcal{V}_B(re, cl(p)))$. As is not proper. At states ex and re, initially in B, the property $\mathbf{E}\,[f\,\mathbf{U}\,g]$ is satisfied. But in the composition model C, the property (being crossed) no longer holds at those states. Two soundness problems could not be proved for this overriding case. This failing case is illustrated in Figure 4.10, even for a weak property like $\mathbf{E}\,[f\ \mathbf{U}\ g]$.

Until this point, we have proved part 1 for the two soundness problems in this cyclic structure with respect to the additive and non-critical overriding composition. OIMC may not be sound in some extreme cases of critical transition removal. For part 2, the argument is very similar to the counterpart with respect to negation and union operators

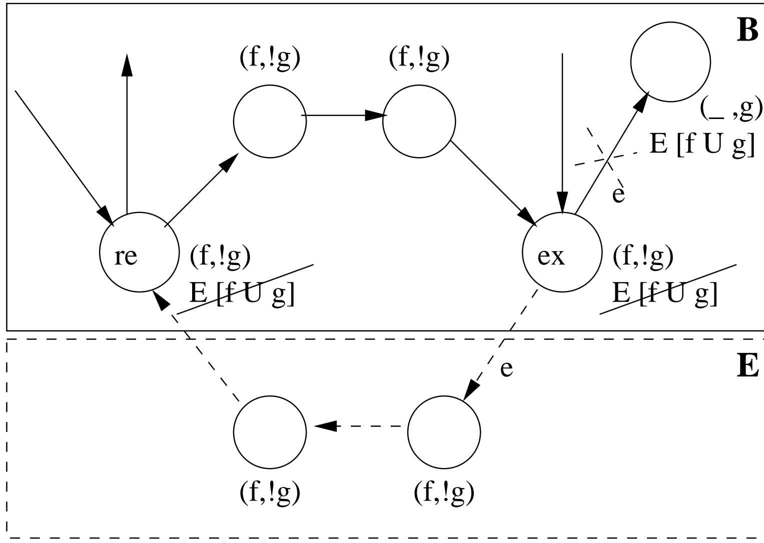

Figure 4.15: An example of failing composition to preserve $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$ in case of base-extension cyclic dependency.

of Theorem 15. In short, OIMC is sound for all CTL properties under this kind of circular dependency of the additive and non-critical overriding composition. For the critical overriding composition, it may not. In such a circumstance, OIMC requires extra checking at removed transitions. Details are discussed in Section 4.2.1. $\square$

4.3 Properties Preservation at Extension States

Until this stage, we have used the assumption model checking in E to verify the conformance between B and E at exit states. During this model checking task, $\mathcal{V}_B(re, cl(p))$ at reentry states are seeded into the ending states of E. Unlike Section 4.1 dealing with the base states, this section is about the effect of conformance at exit states on the extension states.

Lemma 20 *Given a model B and a CTL property p, an extension E is attached to the model B via an exit state ex and a reentry state re. Suppose that the assumption function As is proper. If B and E are in conformance at ex, $\forall s \in S_E : \mathcal{V}_C(s, cl(p)) = \mathcal{V}_E(s, cl(p))$.*

Informally, this lemma claims that if base and extension conform, there is no change in truth values of extension states with respect to $cl(p)$ between model checking in entire C or in E only. Formally, $\forall \phi \in cl(p) : e_\phi = c_\phi \cap S_E$. The proof structure is as usual - proving the lemma to be valid for all CTL properties p.

Proof

i). $p \in AP$: the atomic label sets at all states in E are supplied by L_E and hence they are fixed. The lemma is valid for single properties with zero order, $|p| = 0$.

Suppose the lemma is valid for all CTL properties with order less than or equal to k. We prove it holds for any single property p of which $|p| = (k+1)$ via four operators $\mathbf{EX}\,f$, $\mathbf{EG}\,f$, $\mathbf{E}\,[f\,\mathbf{U}\,g]$ and $\mathbf{E}\,[f\,\mathbf{U}\,g]$. Since As is proper, certainly $\mathcal{V}_C(re, cl(p)) = \mathcal{V}_E(re, cl(p))$. Further due to the conformance, the same happens for ex. The proof is only concerned with the rest, i.e. pure extension states. Let $s \in S_E$ and $s \neq ex, re$.

ii). $p = \mathbf{EX}\, f$: All paths rooted at s are initially in E and then through re back to B. By the hypothesis, all states in these paths preserve their f labels. Moreover, As is proper at re. Comparing the respective computation trees rooted at s in C and in E, they are identical in terms of shape and labels with respect to $cl(f)$ at descendant states (in the initial portion from s to re). Hence, $\mathcal{V}_C(s,p) = \mathcal{V}_E(s,p)$.

The truth value with respect to p is preserved at all extension states: $\forall s \in S_E$: $\mathcal{V}_C(s,p) = \mathcal{V}_E(s,p)$. We have: $cl(p) = \{p\} \cup cl(f)$ and

- $\mathcal{V}_C(s,p) = \mathcal{V}_E(s,p)$ (due to above arguments).
- $\mathcal{V}_C(s, cl(f)) = \mathcal{V}_E(s, cl(f))$ (due to the hypothesis where $|f| = k$).

Overall, the lemma holds for $p = \mathbf{EX}\, f$ with the order $(k+1)$.

iii). $p = \mathbf{EG}\, f$: Similarly, we compare the computation trees rooted at s in both models C and E and see that they match the initial portion from the root s to state re. At all intermediate states, labels up to $cl(f)$ are matched in both trees. The branches from re are equivalently substituted by the seeded values $\mathcal{V}_B(re, cl(p))$ which is proper. So this substitution does not affect the truth value with respect to p at s.

Overall, the truth value with respect to p is preserved at s. By a similar argument in (ii) together with the inductive hypothesis on $cl(f)$, we can conclude that the truth values with respect to $cl(p)$ at s are preserved. The lemma holds for $p = \mathbf{EG}\, f$ with the order $(k+1)$.

iv). $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$: Identically, by comparing the computation trees rooted at s in both models C and E, we find out that they match the initial portion from the root s to state re. At all intermediate states, labels up to $cl(f)$ and $cl(g)$ are respectively matched in both trees. The branches from re can be substituted by the seeded values $\mathcal{V}_B(re, cl(p))$. This substitution does not affect the truth value with respect to p at s.

Overall, the truth value with respect to p is preserved at s. By a similar argument in (ii) together with the inductive hypothesis on $cl(f)$ and $cl(g)$, we can conclude that the truth values with respect to $cl(p)$ at s are preserved. The lemma holds for $p = \mathbf{E}\,[f\ \mathbf{U}\ g]$ with the order $(k+1)$

v). $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$: This case is similar to the case (iv) as two computation trees are matched respectively in terms of shape; and labels in $cl(f)$ and $cl(g)$ at each states. The theorem is also valid for $p = \mathbf{A}\,[f\ \mathbf{U}\ g]$ whose order is $(k+1)$.

vi). For the other normal properties, the proof is simple as they are derived from the above four normal properties. The proof is exactly as the counterpart in Theorem 15. The rest of six normal properties are proved.

Until this stage, part 1 of the proof is completed. We turn to part 2 involving $p = \neg f$ where f is normal; and $p = f \vee g$ where f and g are single CTL properties at which lemma holds. In the first case, the complementary set $e_f = S_E \setminus e_p$ of the set e_p does not change due to part 1. So does e_p. The lemma is valid for the negation operator. Similar, in the latter case, $e_p = e_f \vee e_g$ is not changed because both e_f and e_g are not changed.

Overall, the lemma on assumption model checking is valid for all CTL properties with respect to simple interface composition. □

Next we consider Lemma 20 under the generalized interface. At exit states, due to the conformance of B and E, $\mathcal{V}_C(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$ for all ex as long as the seeded truth values at their respective reentry states are proper. The same requirement is expected for dual states. From the perspective of a pure extension state, if all reachable

reentry states from that state are seeded with proper truth values, the arguments in the proof for the above lemma are still valid.

In summary, Lemma 20 can be extended to the theorem below for the generalized interface as long as the seeded truth values at all reentry states are proper. Indeed, this is the soundness issue of incremental verification mentioned in Section 4.2. According to the result in that section, the seeded values, i.e. the function As, at reentry states are surely proper in the additive and non-critical overriding composition. For critical overriding composition, OIMC may not be sound in some extreme cases of the critical overriding composition with circular dependency structure.

Unlike Theorem 17 addressing the base, the following theorem is another key of incremental verification but it focuses on the extension.

Theorem 21 *Given a base B and a property p, an extension E is attached to B at some interface states. Further, suppose that As is proper. If B and E are in conformance at all exit states, $\forall s \in S_E : \mathcal{V}_C(s, cl(p)) = \mathcal{V}_E(s, cl(p))$.*

Two theorems 17 and 21 serve as the foundation of incremental model checking.

4.4 The Scalability of Incremental Verification

This section addresses the scalability of incremental verification. The incremental verification method executes the assumption model checking for each exit state between B and E to check their conformance. Corresponding to each ex, the algorithm to verify a preservation constraint in the extension E is briefly described as in Section 4.1.4. If at all exit states, the truth values with respect to $cl(p)$ are matched respectively, B and E are composable. By a simple analysis, the complexity of this algorithm is shown below.

Let $\sharp S_E$, $\sharp R_E$ denote the numbers of states and transitions in E respectively. Similarly, $\sharp cl(p)$ is the number of sub-formulae in $cl(p)$. Note that $\sharp cl(p)$ is different from $|p|$, for instance, in composite CTL properties.

The above algorithm runs over E once and the conformance at all exit states can be verified. At each exit state, we need to verify that $\forall \phi \in cl(p)$, B and E agree with each other at the exit state with respect to ϕ. Therefore, the complexity of this algorithm is certainly proportional to $\sharp cl(p)$.

For any ϕ, the standard CTL model checking procedure is executed entirely within E. According to [6], the complexity of this model checking procedure for checking a property ϕ is $O(\sharp S_E + \sharp R_E)$. Therefore, given a property p adhered to B, the complexity of the incremental verification for any pair B and E is $O(\sharp cl(p) \times (\sharp S_E + \sharp R_E))$. This complexity is independent from the base B.

We consider the general case of n-th version (C_n) during software evolution as a structure of features B, E_1, E_2, ..., E_n where E_i is the extension to the $(i-1)$-th evolved version ($C_{(i-1)}$). The initial version is $C_0 = B$. We can prove that the complexity of verification does not change after adding feature E_n, i.e. the complexity of the incremental verification for confirming E_n not violating the property p in B is $O(\sharp cl(p) \times (\sharp S_{E_n} + \sharp R_{E_n}))$.

Lemma 22 *Given the extensions E_k where $k = \overline{1,n}$ are respectively in conformance with their bases, i.e. $C_{(k-1)}$*

- $\forall s \in S_B$: $\mathcal{V}_{C_n}(s, cl(p)) = ... = \mathcal{V}_{C_{(k+1)}}(s, cl(p)) = \mathcal{V}_{C_k}(s, cl(p)) = ... = \mathcal{V}_B(s, cl(p))$

- $\forall s \in S_{E_k}$: $\mathcal{V}_{C_n}(s, cl(p)) = ... = \mathcal{V}_{C_{(k+1)}}(s, cl(p)) = \mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{E_k}(s, cl(p))$

Proof

The proof is by induction. In the basic case, $i = 1$, the claim is correct as shown in case B and E_1 are composed, namely $C_1 = B + E_1$

- $\forall s \in S_B$: $\mathcal{V}_{C_1}(s, cl(p)) = \mathcal{V}_B(s, cl(p))$ (Theorem 17).
- $\forall s \in S_{E_1}$: $\mathcal{V}_{C_1}(s, cl(p)) = \mathcal{V}_{E_1}(s, cl(p))$ (Theorem 21).

Suppose the lemma is valid for any i-th evolved version, $i \leq (k-1)$. We prove its truth for C_k: $C_k = C_{(k-1)} + E_k$. Since E_k is in conformance with $C_{(k-1)}$, we have:

- By Theorem 17: $\forall s \in S_{C_{(k-1)}}$: $\mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{C_{(k-1)}}(s, cl(p))$. (1)

For $s \in S_{C_{(k-1)}} = S_B \cup S_{E_1} \cup ... \cup S_{E_{(k-1)}}$, there are two cases to consider.

First, if $s \in S_B$, by the above inductive hypothesis for $C_{(k-1)}$,

$$\mathcal{V}_{C_{(k-1)}}(s, cl(p)) = \mathcal{V}_{C_{(k-2)}}(s, cl(p)) = ... = \mathcal{V}_B(s, cl(p))$$

Also, from Equation (1) above:

$$\mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{C_{(k-1)}}(s, cl(p))$$

Therefore,

$$\forall s \in S_B\text{: } \mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{C_{(k-1)}}(s, cl(p)) = ... = \mathcal{V}_B(s, cl(p))$$

The claim is true for $s \in S_B$ in case of C_k.

Second, if $s \in S_{E_j}$ where $j = \overline{1, (k-1)}$, by the above hypothesis for $C_{(k-1)}$,

$$\mathcal{V}_{C_{(k-1)}}(s, cl(p)) = \mathcal{V}_{C_{(k-2)}}(s, cl(p)) = ... = \mathcal{V}_{C_j}(s, cl(p)) = \mathcal{V}_{E_j}(s, cl(p))$$

Besides, due to Equation (1),

$$\mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{C_{(k-1)}}(s, cl(p))$$

As a result, the claim is also true in case of C_k for $s \in S_{E_j}$ where $j = \overline{1, (k-1)}$.

For $s \in S_{E_k}$: $\mathcal{V}_{C_k}(s, cl(p)) = \mathcal{V}_{E_k}(s, cl(p))$ (from Theorem 21), the claim is also valid for E_k states in C_k version.

Overall, Lemma 22 is valid for C_k. By induction, the claim is also valid for n-th version. The proof is completed. □

We apply Lemma 22 to prove the scalability of incremental verification in the general case of n-th version. Let exs' be the set of new exit states formed by composing $C_{(n-1)}$ and E_n. The incremental verification method consists three activities:

1. Verifying property p of a base system, namely: $C_{(n-1)}$. (Activity 1)
2. Deriving $\mathcal{V}_{C_{(n-1)}}(ex', cl(p))$ as the set of preservation constraints for each exit state $ex' \in exs'$. (Activity 2)
3. Executing the above incremental verification algorithm in E_n to check its conformance with its base, i.e. whether $\mathcal{V}_{E_n}(ex', cl(p)) = \mathcal{V}_{C_{(n-1)}}(ex', cl(p))$. (Activity 3)

Activity 1 above can be ignored in this case because we have done it during evolving to $(n-1)$-th version. $E_{(n-1)}$ does not violate p in B. So in $C_{(n-1)}$, p continues to hold in B. The complexity of this activity is $O(1)$.

Activity 2 is provided as the secondary result at the end of Activity 1. The result can be retrieved directly from previous separate verifications of $C_{(n-1)}$ and E_n. There is no need to re-run model checking in $C_{(n-1)}$ to find $\mathcal{V}_{C_{(n-1)}}(ex', cl(p))$ at the exit state ex'. That claim can be justified in the following. Because $ex' \in S_{C_{(n-1)}} = S_B \cup S_{E_1} ... \cup S_{E_{(n-1)}}$, there are two cases to consider.

If $ex' \in S_B$, due to Lemma 22, $\mathcal{V}_{C_{(n-1)}}(ex', cl(p)) = \mathcal{V}_B(ex', cl(p))$. That is, the truth values of the base state stay the same if model checking in either C_n or B. The right-hand side is provided after verifying p in B initially. So this preservation constraints at ex' is already supplied.

In case $ex' \in S_{E_k}$ where $k = \overline{1,(n-1)}$, due to Lemma 22, $\mathcal{V}_{C_{(n-1)}}(ex', cl(p)) = \mathcal{V}_{E_k}(ex', cl(p))$. The right-hand side is also already given after the incremental model checking within $(n-1)$ previous versions during system evolution.

Therefore, the complexity of Activity 2 is $O(1)$. On the other hand, the complexity of Activity 3 is exactly that of the incremental verification algorithm on E_n, i.e. $O(\sharp cl(p) \times (\sharp S_{E_n} + \sharp R_{E_n}))$. The total complexity of ensuring E_n not violating p in B is the same as that of other E_j, $j = \overline{1,(n-1)}$. The incremental verification method maintains its scalability.

Theorem 23 *If all respective pairs of base ($C_{(i-1)}$) and refining (E_i) components conform, the complexity of OIMC to verify the consistency between E_n and B is independent from the n-th version C_n, i.e. it only executes within E_n.*

By the way, to enable OIMC, it is essential to allocate memory for each state s to record its preservation constraints $\mathcal{V}_{E_k}(s, cl(p))$. As any state could be an interface state in the future, we need to memorize the constraints at all states in the feature E_k. Hence, the memory cost for OIMC within a given E_k is $O(\sharp cl(p) \times \sharp S_{E_k})$.

4.5 An Example about Consistency among Single-Object Features

This section gives an example about a typical user account management in a simple library. Initially, the account management system provides basic feature of book reservation and updates user credit when he is overdue (B). Later, the system is extended to handle the case when he loses borrowed books (E - loss-handling feature). After the system evolves to this stage ($C_1 = B + E$), another change (E') in policy is made so that the system distinguishes students from faculties of the university. In this change, a faculty possesses higher priority in services of the library, $C_2 = C_1 + E'$. These changes to system are rather unanticipated. Basically, the system evolves in three different versions by changing feature set incrementally.

In this section, the following notations are used to denote states and events associated with $User$ object. They are:

- User states: *u.init*(*ial*), *u.wait*(*ing*), *u.borr*(*owing*), *u.chg* (charging fines), *u.crd* (updating the credit), *u.clr* (the account is clear), *u.ord*(*er*), *u.sdd* (deducting salary).

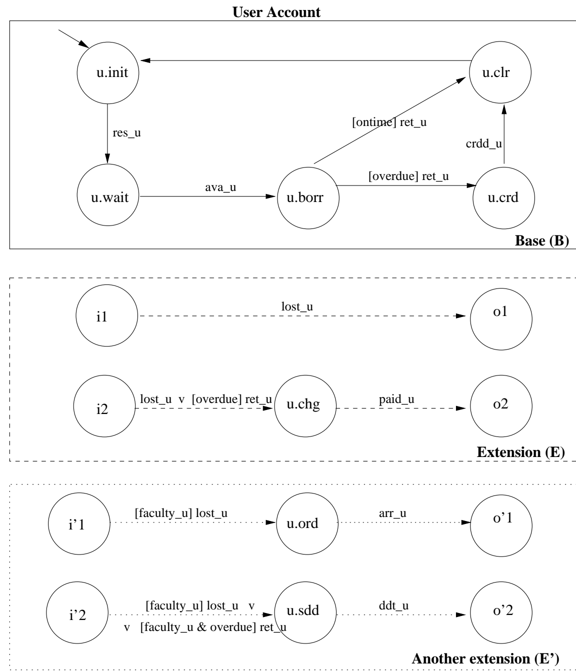

Figure 4.16: A base component and its refining components for a user account.

- Events: res_u (reservation request), ava_u (book available), $lost_u$ (loss), ret_u (book return), $paid_u$ (fine payment), $crdd_u$ (the credit update is done), arr_u (book replacement arrival), ddt_u (salary deduction).

- Guards: $faculty_u$ (faculty user), *ontime*, *overdue*.

The feature-based model of the account is shown in Figure 4.16 and Figure 4.17. The three layers are corresponding to three separate features: basic book reservation, loss-handling and faculty priority. The independent components are depicted in Figure 4.16. The figure is rather self-explained. In B, once the user reserves, he waits for the book if it is on loan to someone else. In E, if the user is waiting for some book currently on loan and that book is lost, he has to go back to the initial state because there is no book for his request anymore. If he actually loses the book, he has to pay money for the book. If the payment (for book loss or overdue) is on time, his credit is not affected. Otherwise, his credit will be deducted.

In E', some changes are made to both existing features: basic reservation and book-loss handling. First, when the faculty is waiting for the book on loan to someone else and that book is lost, he is not going back to the initial state like regular students. The library will order a replacement directly to the publisher. When the book arrives, the system notifies the faculty immediately. In case the faculty actually loses the book, the system will charge the fine directly to his salary and his credit is not affected at all.

The composite model of the account is shown in Figure 4.17. Within E, importantly, E overrides the transition ret_u with the guard $[overdue]$ of B, i.e. E changes courses of some scenarios in B. Considering the composition of E with B, obviously i_1 can be mapped with $u.wait$, $i_1 \leftrightarrow u.wait$, according to the plugging conditions among compatible interface states as both states specifies the state when the user is waiting for the book.

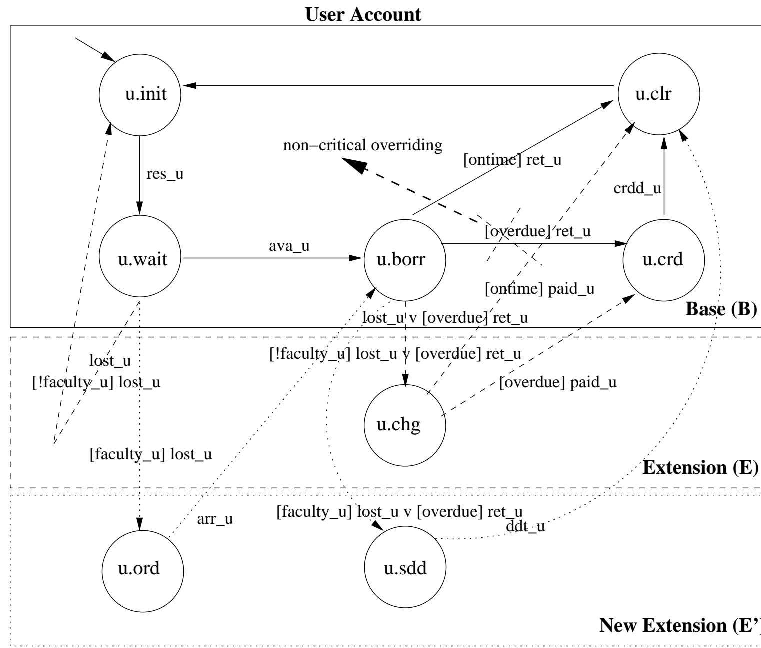

Figure 4.17: The state transition chart of a user account in the library management system

Similarly, $o_1 \leftrightarrow u.init$, $i_2 \leftrightarrow u.borr$, $o_2 \leftrightarrow u.clr$ and $o_2 \leftrightarrow u.crd$ [7].

For E', by a similar manner, we have the following mapping configuration between interface states: $i'_1 \leftrightarrow u.wait$, $o'_1 \leftrightarrow u.borr$, $i'_2 \leftrightarrow u.borr$ and $o'_2 \leftrightarrow u.clr$.

In this example, a property adhered to the base is that the user can always get to the final state, i.e. *u.clr* state. Informally, he eventually succeeds in borrowing a book under any circumstance. Of course, this informal meaning of the property is intended for the first feature in which books are never lost. For subsequent evolution steps, because some user may lose books, the property can be informally interpreted as the user always succeeds in borrowing a book as long as it is still in the library. In terms of CTL notation, the property can be expressed as: $p = \mathbf{AG}\,(\mathbf{EF}\,(state = u.clr))$. This property can be easily verified to hold on the base. We need to verify that property is not violated by the extension E with respect to B, then by E' with respect to C_1.

The following shows the application of the OIMC theory so far. The closure set of p is $cl(p) = \{p, f, a\}$, where:

- $f = \mathbf{EG}\,(state = u.clr)$
- $a = (state = u.clr)$

Within B, obviously,

- $\forall s \in S_B$: $B, s \models p$ and $B, s \models f$.
- On the contrary, $\forall s \neq u.clr : B, s \models \neg a$.

There are two exit states between B and E, *u.wait* and *u.borr*. Because they are unrelated in E, they can be considered separately. First, in case of *u.wait*, this is an

[7]Depending on the guard of $paid_u$, o_2 is mapped into two different states in B.

additive composition. Theorem 17 is applicable. In the extension, from exit state $u.wait$, there is only one reentry state $u.init$. Hence, by seeding $\mathcal{V}_B(u.init, cl(p))$ to the ending state $init$ for the assumption model checking within the extension E, certainly we have:

- $E, u.wait \models_{as} p$ and $E, u.wait \models_{as} f$.
- $E, u.wait \models_{as} \neg a$.

Therefore, E and B conform at $u.wait$ with respect to $cl(p)$.

Second, the exit state $u.borr$ is a little more complicated due to its overriding of the base transition $[overdue]\ ret_u$. In the extension, at $u.chg$, there are two paths. The one direct to $u.clr$ ensures that p and f hold at $u.chg$ directly. On the other hand, in the second path to $u.crd$, by seeding $\mathcal{V}_B(u.crd, cl(p))$ to the reentry state for the assumption model checking, we have:

- $E, u.chg \models_{as} p$ and $E, u.chg \models_{as} f$.
- $E, u.chg \models_{as} \neg a$.

From the derived truth values $\mathcal{V}_E(u.chg, cl(p))$, the truth values at $u.borr$ are: $\mathcal{V}_E(u.borr, cl(p)) = \{p, f, \neg a\} = \mathcal{V}_B(u.borr, cl(p))$. B and E are in conformance at $u.borr$.

In summary, Theorem 17 can be applied as B and E conform at all exit states. All states in the base B are not affected by E, namely p is preserved by the second feature E after evolving to C_1.

In the third feature E', there are two exit states: $u.wait$ (a B state at which E' partially overrides $lost_u$ transition into $lost_u \wedge \neg faculty_u$) and $u.chg$ (an E state). The assumption model checking to be executed within E' is similar to the previous counterpart in E. Identically, within E', it is easy to verify that:

- $\mathcal{V}_{E'}(u.wait, cl(p)) = \{p, f, \neg a\} = \mathcal{V}_B(u.wait, cl(p)) = \mathcal{V}_{C_1}(u.wait, cl(p))$. E' and C_1 conform at $u.wait$.
- E' and C_1 are in conformance at $u.chg$.

The conclusion is that p is preserved by both extensions E, E'. In this example, the scalability of incremental model checking is maintained as it only runs on E and E', independently from the base B and C_1 respectively.

Chapter 5

OIMC for Consistency among Multi-Object Features

5.1 The Fundamental Approach

The arguments in Sections 4.1 and 4.3 are based on the assumption that base and extension models are fully given in advance. That is true when each feature is encapsulated within a single object. The problem is more complicated when features crosscut several objects and those member objects may not synchronize in entering and returning from the extension. Constructing the extension model itself is hence a difficult task.

This part is the improvement from the static model [25] mentioned in Section 2.3.

5.1.1 A Formal Model of Features Crosscutting Multiple Objects

For simplicity, suppose that each feature in the system crosscuts the same number of objects: $o_1, ..., o_k$. Each o_i is corresponding to a pair of base (B_i) and extension (E_i). Unlike the model in Section 3.1, a multi-object feature contains many objects and they often communicate with each other via output events. Hence, the element Σ in Definition 2 should extend to cover output events. That is Σ is with the form *ie*/*oe* - where *ie* and *oe* are input and output events respectively.

Definition 24 *A* base *is a tuple* $\langle B_1, ..., B_k \rangle$ *of base object models and a tuple of interfaces, where* $B_i = \langle S_{Bi}, \Sigma_{Bi}, s_{o_{Bi}}, R_{Bi}, L_{Bi} \rangle$ *is defined in Definition 2. The interface is a tuple* $\langle I_1, ..., I_k \rangle$*, where* $I_i = \langle exit_i, reentry_i \rangle$.

Definition 25 *An* extension *is a tuple* $\langle E_0, ..., E_k \rangle$ *of extension object models. Each* $E_i = \langle S_{Ei}, \Sigma_{Ei}, \perp, R_{Ei}, L_{Ei} \rangle$ *is compatible with the respective* B_i *as of Definition 3.*

Definition 26 *Composing the base with the extension, through the interface* $\langle I_1, ..., I_k \rangle$ *produces a tuple* $\langle C_1, ..., C_k \rangle$ *of composed object models. Each* $C_i = \langle S_{Ci}, \Sigma_{Ci}, s_{0_{Ci}}, R_{Ci}, L_{Ci} \rangle$ *is defined from* $B_i = \langle S_{Bi}, \Sigma_{Bi}, s_{0_{Bi}}, R_{Bi}, L_{Bi} \rangle$ *and* $E_i = \langle S_{Ei}, \Sigma_{Ei}, \perp, R_{Ei}, L_{Ei} \rangle$ *as previously defined in Definition 5.*

Hereafter, the term local (or object) model is used for addressing individual objects, while global model is for the whole system with multiple objects. Further, assume that

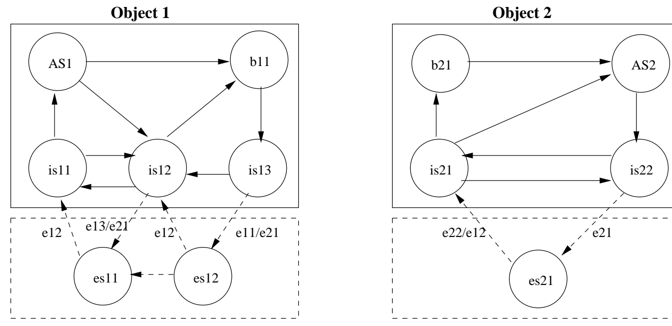

Figure 5.1: Composing a full extension model with a partial base feature.

message passing mechanism between objects are synchronous. Events in the model are categorized into *external* and *internal* classes. External events are visible from outside. The system and its environment communicate with each other via these events. On the contrary, internal events are passed among objects and hence invisible from outside. Invisible internal events are subsequently generated among objects during handling an external event. They occur during the system is in transition. By ignoring these internal events, the system is considered to transit from a stable state to another stable state due to an external event. Let Σ_{EE} denote the external event set. In Figure 5.1, the transition e_{11}/e_{21} of o_1 shows its handling of the external event e_{11} by signaling the internal event e_{21} to o_2.

5.1.2 Transforming Multi-Object Models into a Global Model

Because each feature is modeled separately, in order to apply the verification algorithm in Section 4.1.4, the models must be first transformed into a *global model*. At any point in time, the state of the whole system is represented by a global state which is essentially a cross-product of its object states.

Due to feature encapsulation, it is ideal to reveal only the interface to the other features. However, in order to give objects more freedom during their synchronization at the interfaces before entering their local extensions, further internal structure b_i of each object o_i is made visible from the corresponding extension E_i. Hence, if viewed from E_i, the state set of o_i is $S_{Bi} = exit_i \cup reentry_i \cup b_i \cup \{HS_i\}$, where HS_i represents the hidden part of B_i. In practice, objects usually synchronize well at the interface so the set of additional visible states b_i is rather small.

The global base model constructed from the local base models of k objects is expressed as $B = \langle S_{BG}, \Sigma_{BG}, s_{0_{BG}}, R_{BG}, L_{BG}\rangle$ where:

- $S_{BG} = \prod_{i=1}^{k} S_{Bi}$, $\forall s_{BG} \in S_{BG} : s_{BG} \overset{def.}{=} \langle s_1, ..., s_k\rangle$, where $s_i \in S_{Bi}$.
- $\Sigma_{BG} = \Sigma_{EE} \cap \bigcup_{i=1}^{k} \Sigma_{B_i}$.
- $s_{0_{BG}} = \prod_{i=1}^{k} s_{0_{Bi}} \equiv \langle s_{0_{B1}}, ..., s_{0_{Bk}}\rangle$.
- $R_{BG} \subseteq S_{BG} \times PL(\Sigma_{BG}) \to S_{BG}$.
- $L_{BG}(s) = \bigcup_{i=1}^{k} L_{Bi}(s_i), where\ s = \langle s_1, ..., s_k\rangle$.

The global composition model $C = \langle S_{CG}, \Sigma_{CG}, s_{0_{CG}}, R_{CG}, L_{CG} \rangle$ is similarly defined upon C_i's. That is, $C = \langle S_{CG}, \Sigma_{CG}, s_{0_{CG}}, R_{CG}, L_{CG} \rangle$ where:

- $S_{BG} = \prod_{i=1}^{k}(S_{Bi} \cup S_{Ei})$, $\forall s_{CG} \in S_{CG} : s_{CG} = \langle s_1, ..., s_k \rangle$, where $s_i \in S_{Bi} \cup S_{Ei}$.
- $\Sigma_{CG} = \Sigma_{EE} \cap \bigcup_{i=1}^{k} \Sigma_{Ci}$.
- $s_{0_{CG}} = s_{0_{BG}} \equiv \langle s_{0_{B1}}, ..., s_{0_{Bk}} \rangle$.
- $R_{CG} \subseteq S_{CG} \times PL(\Sigma_{CG}) \rightarrow S_{CG}$.
- $L_{CG}(s) = \bigcup_{i=1}^{k} L_{Ci}(s_i), where\ s = \langle s_1, ..., s_k \rangle$.

Here, any entity with subscript Ci is defined from the respective entity pairs with subscripts Bi and Ei as in Section 3.1 for single object model.

A system is in its global extension $E = \langle S_{EG}, \Sigma_{EG}, \perp, R_{EG}, L_{EG} \rangle$ if at least one of its objects are in the respective local extension. More specifically,

- $S_{EG} = S_{CG} \setminus S_{BG} = \prod_{i=1}^{k}(S_{Bi} \cup S_{Ei}) \setminus \prod_{i=1}^{k} S_{Bi}$.
- $\Sigma_{EG} = \Sigma_{CG}$.
- $R_{EG} \subseteq S_{EG} \times PL(\Sigma_{EG}) \rightarrow S_{EG}$.
- $L_{EG}(s) = L_{CG}(s)$.

This full representation of global base and extension models are too redundant in terms of states, especially interface states. We present some principles to compact E and its interface with B.

Based on local bases B_i and extensions E_i, $i = \overline{1, k}$, the candidates for global exit and reentry states are identified via four main factors: global exit/reentry events, invariants adhered to the base, use-case scenarios for interaction between features and the clients; and compatible interface states between components.

First, synchronization of message passing among local objects are utilized. Potential global exit (and reentry) events are identified. These exit events are external events causing the system to enter the extension. By an observation, this event typically connects an exit state $\langle s_1, ..., s_i, ..., s_k \rangle$ to an extension state $\langle s_1, ..., s'_i, ..., s_k \rangle$, where $s_j \in S_{Bj}$ and $s'_j \in S_{Ej}$, $j = \overline{1, k}$. Hence, this event is surely an exit event of o_i - the first object to start its crossing among k objects. Further, it must be an external event (Σ_{EE}).

In Figure 5.1, local exit events are: e_{11}, e_{21}, e_{13}. Given the transition e_{11}/e_{21} in Figure 5.1, $e_{21} \notin \Sigma_{EE}$ can not be a global exit event. Hence, only external events e_{11}, e_{13} are left as candidates for global exit events.

Similarly, in case of reentry events, a global reentry event must be an external reentry event. It is associated with the object which starts the handling of the event so that the system is back to the base at the end of the handling process. In Figure 5.1, since $e_{12} \notin \Sigma_{EE}$, the only candidate for global reentry event is e_{22}.

Associated with any global event are a key object o_i and the state of the object during which it receives the event. Depending on whether the event is exit or reentry, the state of o_i is a key local exit or reentry state contributing to the interface state in the global model. In Figure 5.1, is_{12}, is_{13} are key exit states, while is_{21} is a key reentry state.

Definition 27 *Suppose $ex[i]$ is a key local exit state of the object o_i, potential global exit states are those cross-products of local base states containing $ex[i]$, i.e. $ex_{BG} = \langle s_1, ..., ex[i], ..., s_k \rangle$. Potential global reentry states are similarly defined upon a key local reentry state $re[i]$ as $re_{BG} = \langle s_1, ..., re[i], ..., s_k \rangle$.*

In Figure 5.1, potential global exit state ex_{BG} is a cross-product of either is_{12} or is_{13} with any base state of o_2. On the other hand, re_{BG} is a cross-product of is_{21} with any base state of o_1.

Second, up to this stage, there are still possibly many potential global interface states. Certain combination of some object states, i.e. the coexistence of their respective atomic label sets, are in conflict. Formally, for any two objects o_i and o_j, $i, j = \overline{1, k}$ and $i \neq j$, if $\exists a \in AP : (L_{Ci}(s_i) \Rightarrow a) \wedge (L_{Cj}(s_j) \Rightarrow \neg a)$, then any state cross-product containing the both s_i, s_j never exists. Those cross-products are dropped and hence the restriction on global states is strengthened. With these invariants, besides simplifying the global state space, the set of potential interface states is also further compacted.

Third, usually the extension feature interacts with the base in some pre-defined sequences of external events. That is, the interacting scenarios are given in advance. By this order of external events, this scenario-based interaction between base and extension reduces the state space further. Unlike the first principle about deriving the order of internal events, this scenario-based principle deals with the order of external events.

Fourth, to enter and exit from the extension feature, certain conditions must be satisfied. The conditions L_{ex} and L_{re} are respective pre-condition and post-condition associated with a scenario of the extension feature. A potential exit state $ex_{BG} = \langle s_1, ..., s_k \rangle$ is *compatible* to be an exit state to the extension if the pre-condition L_{ex} to accept the extension events are satisfied. That is, $\bigcup_{i=1}^{k} L_{Bi}(s_i) \Rightarrow L_{ex}$. We can safely drop those potential exit states ex_{BG} at which the pre-condition is not satisfied from the set of potential exit states. Similarly, for reentry states, the associated post-conditions L_{re} must be satisfied after the control is returned to the base at state $re_{BG} = \langle s'_1, ..., s'_k \rangle$. That is, $L_{re} \Rightarrow \bigcup_{i=1}^{k} L_{Bi}(s'_i)$. This condition to match compatible states between features are used to eliminate redundant interface states.

Through the utilization of the above principles, the global state model is compacted and so are the interface states. The extension model is then constructed as any possible path running from an ex_{BG} and ending at a re_{BG}. Along the path are global extension states. The event connecting two adjacent global states is corresponding exactly the external event at the object initiating that transition.

5.1.3 Incremental Verification Within Global Model

The theoretical foundation of OIMC is constructed with respect to explicit state-based model. Hence, when dealing with multi-object features, model checking must be done in global state space. To do that, we need to transform tuple of individual object models into a single global state model before any model checking attempt is carried out. Like Section 4.1.4 on single-object features, the corresponding steps to verify feature consistency among multi-object features can be briefly presented below:

1. Constructing a explicit state-based model for the base feature B; and then verifying within this global model that a CTL property p holds.

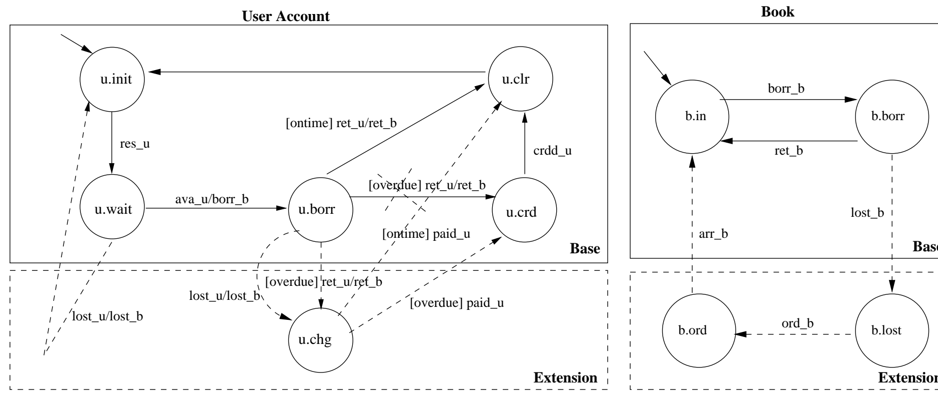

Figure 5.2: A simple library with two features: book reservation and loss-handling.

2. Within this model, for any state s, recording all truth values of $cl(p)$ at s. Those are preservation constraints to preserve p in the base if s is set to be an exit state for a future extension feature.

3. An extension feature E is attached with the base via some interface states. To verify the consistency between B and E, first we need to transform multiple object models and their respective extensions into a single global extension model. The OIMC procedure shown in Section 4.1.4 can be applied as usual because this model can be equally treated as a single-object model.

However, there is a subtle difference between single- and multi-object OIMC. After transforming multiple object models into a single global extension model, under any computation path from a exit state ex_{BG} to a reentry state re_{BG}, there are some extension states. If all extension states are not of the cross-product form containing some local state HS_i, the verification is straightforward as of Section 4.1.4. If p holds in the extension part as of this case, the extension and the base features crosscutting multiple objects are composable.

In the other case, there exists a state involving HS_i in the global extension model. OIMC does not work then. The only solution is to reveal further states in HS_i so that HS_i does not occur in the extension model any more, i.e. achieving the ease of OIMC at the cost of feature encapsulation.

5.2 An Example of Consistency among Multi-Object Features

This section presents a simple example about verifying multi-object feature consistency. It is a library with two participating objects: $User$ and $Book$ as shown in Figure 5.2. This library is more detailed than the counterpart in Figure 4.17 in Section 4.5. The two layers are corresponding to two separate features: basic book reservation and loss-handling. The figure is rather self-explained. Once the user reserves, he waits for the book if it is on loan to someone else. The book will be available for his use only after the previous user has finalized book-borrowing procedure, i.e. reaching $u.clr$ state. In the extension, if the user is waiting for some book currently on loan and that book is lost, he has to go back to the initial state because there is no book for his request anymore. If he actually loses the

book, he has to pay a fine. Importantly, the extension feature also charges the user if he is overdue, i.e. complete overriding of the base transition $[overdue]\ ret_u/ret_b$ happens here. If the payment (for book loss or overdue) is on time, his credit is not affected. Otherwise, his credit will be deducted.

In this section, the following notations are used to denote states and events associated with $User$ and $Book$ objects. They are:

- User states: $u.init(ial)$, $u.wait(ing)$, $u.borr(owing)$, $u.chg$ (charging fines), $u.crd$ (updating the credit), $u.clr$ (the account is clear).
- Book states: $b.in$ (in the library), $b.borr$ (on loan), $b.lost$ (loss), $b.ord$ (being ordered to the publisher).
- External user events: res_u (reservation request), ava_u (book available), $lost_u$ (loss), ret_u (book return), $paid_u$ (fine payment), $crdd_u$ (the credit update is done).
- External book events: arr_b (book arrival), ord_b (replacement order). Internal events (generated from user events): $borr_b$ (borrowed), ret_b (book return), $lost_b$ (loss).
- Guards: $ontime$, $overdue$.

In Figure 5.2, local exit events are: $\{lost_u, [overdue]\ ret_u, lost_b\}$. However, since $lost_u \hookrightarrow lost_b$, key exit events are $\{lost_u, [overdue]\, ret_u\}$. Similarly, key reentry events are $\{lost_u, [ontime]\, paid_u, [overdue]\, paid_u, arr_b\}$. They lead to key exit states $\{u.wait, u.borr\}$ and key reentry states $\{u.init, u.clr, u.crd, b.in\}$. From these sets, we can derive the candidates for global exit and reentry states as:

- Exit candidates: $\langle u.wait, b.in\rangle$, $\langle u.wait, b.borr\rangle$, $\langle u.borr, b.in\rangle$, $\langle u.borr, b.borr\rangle$.
- Reentry candidates: $\langle u.init, b.in\rangle$, $\langle u.wait, b.in\rangle$, $\langle u.borr, b.in\rangle$, $\langle u.crd, b.in\rangle$, $\langle u.clr, b.in\rangle$, $\langle u.init, b.borr\rangle$, $\langle u.clr, b.borr\rangle$, $\langle u.crd, b.borr\rangle$.

Because of the invariant adhered to the base, namely a book can not be on loan and in the library at the same time, we can remove unreachable states from the candidate sets such as: $\langle u.borr, b.in\rangle$,[1] $\langle u.clr, b.borr\rangle$, $\langle u.crd, b.borr\rangle$. Besides, $\langle u.wait, b.in\rangle$ is not a valid reentry state within the extension model because there is no corresponding exit state. The interface states are:

- Exit states: $\langle u.wait, b.in\rangle$, $\langle u.wait, b.borr\rangle$, $\langle u.borr, b.borr\rangle$.
- Reentry states: $\langle u.init, b.in\rangle$, $\langle u.crd, b.in\rangle$, $\langle u.clr, b.in\rangle$, $\langle u.init, b.borr\rangle$.

From these potential interface states, the global extension model is constructed in Figure 5.3.

A property adhered to the base model is that: Under any circumstance, the system can always reach the safe final state eventually, namely the user account is clear of the book, and the book (or its replacement after being extended to handle the book-loss feature) is in the library. In terms of CTL, the property can be expressed as: $p = \mathbf{AG}(\mathbf{EF}((u.state = u.clr) \wedge (b.state = b.in)))$. The closure set of p is $cl(p) = \{p, f, ua, ba\}$ where:

[1] $u.borr$ means $book.status = out$, whereas $b.in$ means $book.status = in$.

Figure 5.3: The global extension model of the library example.

- $f = \mathbf{EF}\,((u.state = u.clr) \wedge (b.state = b.in))$
- $ua = (u.state = u.clr)$ and $ba = (b.state = b.in)$ (user and book respective atomic propositions).

This property is confirmed within the base feature providing basic book-borrowing service. After a model checking task within the base B, we can derive the following in global base model:

- $\forall s \in S_B$: $B, s \models p$ and $B, s \models f$.
- $\forall s = \langle u.clr, \perp \rangle$: $B, s \models ua$; otherwise $B, s \models \neg ua$.
- $\forall s = \langle \perp, b.in \rangle$: $B, s \models ba$; otherwise $B, s \models \neg ba$.

The book-loss feature is verified not violating p. Applying the truth values in the base model to four reentry states as in Figure 5.3 as follows:

- Seeding values at $\langle u.init, b.in \rangle$, $\langle u.crd, b.in \rangle$ are $\{p, f, \neg ua, ba\}$.
- Seeding values at $\langle u.clr, b.in \rangle$ is $\{p, f, ua, ba\}$.
- Seeding values at $\langle u.init, b.borr \rangle$ is $\{p, f, \neg ua, \neg ba\}$.

Within the global extension model depicted in Figure 5.3, the assumption model checking is executed to show the truth values at exit states as follows:

- $\mathcal{V}_E(\langle u.borr, b.borr \rangle, cl(p)) = \{p, f, \neg ua, \neg ba\} = \mathcal{V}_B(\langle u.borr, b.borr \rangle, cl(p))$.
- $\mathcal{V}_E(\langle u.wait, b.borr \rangle, cl(p)) = \{p, f, \neg ua, \neg ba\} = \mathcal{V}_B(\langle u.wait, b.borr \rangle, cl(p))$.
- $\mathcal{V}_E(\langle u.wait, b.in \rangle, cl(p)) = \{p, f, \neg ua, ba\} = \mathcal{V}_B(\langle u.wait, b.in \rangle, cl(p))$.

In brief, we have the conformance at all exit states with respect to $cl(p)$. Hence, p is preserved by the extension feature E.

Chapter 6

OIMC Improvements

This chapter tries to improve some results in Chapter 4. In particular, we are concerned with the conformance condition between a base component and its refinement.

6.1 Relaxing the Conformance Condition

Like Chapter 4, we are concerned with property preservation in B after composing B with E. Specifically, exit states are of interest due to particulars of their associated computation trees. At each exit state ex, the computation tree rooted at ex within C is formed by combining the computation trees from B' [1] and E. Hence, the truth values at ex in C is derived from the counterparts in B' and E. In fact, we can prove that if the assumption function As is proper, $\forall\phi$: $\mathcal{V}_C(ex,\phi) = \mathcal{V}_{B'}(ex,\phi) \oplus \mathcal{V}_E(ex,\phi)$. Because B' and B are very similar, in most of the cases, $\mathcal{V}_{B'}(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$. Even when they differs, $\mathcal{V}_{B'}(ex, cl(p))$ can be derived rather quickly from $\mathcal{V}_B(ex, cl(p))$.

In addition, unlike the counterpart in Chapter 4, $\mathcal{V}_E(ex, cl(p))$ is derived by the assumption model checking within E in which the assumed truth values at a reentry state re are $\mathcal{V}_{B'}(re, cl(p))$. In the former, the seeded values are $\mathcal{V}_B(re, cl(p))$. The values from the model B' certainly reflect more accurately the computation tree at re in C.

Definition 28 *$\oplus$ is the composing operator defined over CTL properties of two sub-trees rooted at the same node.*

- *If ϕ is a single universal CTL property (ACTL), i.e. $\phi = \mathbf{A}f$*
  1. *$\phi \oplus \phi = \phi$.*
  2. *Otherwise, $\perp \oplus \neg\phi = \neg\phi \oplus \perp = \neg\phi$.*
- *If ϕ is a single existential CTL property (ECTL), i.e. $\phi = \mathbf{E}f$*
  1. *$\neg\phi \oplus \neg\phi = \neg\phi$.*
  2. *Otherwise, $\perp \oplus \phi = \phi \oplus \perp = \phi$.*
- *$\phi = \phi_1 \vee \phi_2$, $\phi \oplus \phi = (\phi_1 \oplus \phi_1) \vee (\phi_2 \oplus \phi_2)$.*
- *$\phi = \phi_1 \wedge \phi_2$, $\phi \oplus \phi = (\phi_1 \oplus \phi_1) \wedge (\phi_2 \oplus \phi_2)$.*

[1] B' is the remainder of B after removing all overridden transitions, if any.

However, by Definitions 7 and 11, $\forall \phi \in cl(p) : \mathcal{V}_M(ex, \phi) \in \{\phi, \neg\phi\}$ because all properties in $cl(p)$ are single. Within this paper, the operator $\oplus$ in essence operates on pairs of single CTL properties. The new conformance condition is proposed below. Basically, it is similar to the counterpart - Definition 14.

Definition 29 *B and E are in* conformance *at an exit state ex (with respect to $cl(p)$) if $\mathcal{V}_{B'}(ex, cl(p)) \oplus \mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$. They are in* strict conformance *if besides the regular conformance condition, with regards to any property in $cl(p)$ of the form $\mathbf{A}\,[f\,\mathbf{U}\,g]$ (or $\neg\mathbf{EG}\,f$) holding at ex, E does not make $\mathbf{A}\,[f\,\mathbf{U}\,g]$ false (or $\mathbf{EG}\,f$ true) at ex by patching a path of $(f \wedge \neg g)^*$ (or f^*) with another existing $(f \wedge \neg g)^*$ (or f^*) path in B' to make a complete cycle $(f \wedge \neg g)^*$ (or f^*) through ex in C.*

If the assumption function As is proper, the left-hand side of the equation can be proved to be exactly $\mathcal{V}_C(ex, cl(p))$. Therefore, the conformance between B and E ensures the preservation of truth values with respect to $cl(p)$ at ex. Compared with the condition proposed in Section 4.1, the above condition is more relaxed because the former proposes the condition $\mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$. The conformance condition is proved below to be weakened. That is, $\forall \phi \in cl(p)$, if $\mathcal{V}_E(ex, \phi) = \mathcal{V}_B(ex, \phi)$ then $\mathcal{V}_E(ex, \phi) \oplus \mathcal{V}_{B'}(ex, \phi) = \mathcal{V}_B(ex, \phi)$.

The proof follows. According to Definition 7, all member properties ϕ in $cl(p)$ are single. A single property ϕ belongs to either $ACTL$ or $ECTL$. In the former condition in Section 4.1, $\forall \phi \in cl(p) : \mathcal{V}_E(ex, \phi) = \mathcal{V}_B(ex, \phi)$. There are four cases to consider.

First, $\mathcal{V}_B(ex, \phi) = \phi \in ECTL$ implies $\mathcal{V}_E(ex, \phi) = \phi$. According to Definition 28 above, $\mathcal{V}_E(ex, \phi) \oplus \perp = \phi = \mathcal{V}_B(ex, \phi)$. That is, $\mathcal{V}_E(ex, \phi) \oplus \mathcal{V}_{B'}(ex, \phi) = \mathcal{V}_B(ex, \phi)$.

Second, $\mathcal{V}_B(ex, \phi) = \neg\phi$ and $\phi \in ECTL$. This means all paths in B rooted at ex do not satisfy ϕ. As B' is a sub-model of B, the set of paths from ex in B' is a subset of the counterpart in B. Certainly, $\mathcal{V}_{B'}(ex, \phi) = \neg\phi$. In addition, since $\mathcal{V}_E(ex, \phi) = \mathcal{V}_B(ex, \phi) = \neg\phi$. We have, $\mathcal{V}_E(ex, \phi) \oplus \mathcal{V}_{B'}(ex, \phi) = \neg\phi \oplus \neg\phi = \neg\phi = \mathcal{V}_B(ex, \phi)$ (according to Definition 28 and $\phi \in ECTL$).

Third, $\mathcal{V}_B(ex, \phi) = \phi \in ACTL$. Similar to the second case, as B' is a sub-model of B, $\mathcal{V}_{B'}(ex, \phi) = \phi$. Composing $\mathcal{V}_E(ex, \phi) = \mathcal{V}_B(ex, \phi) = \phi$ with $\mathcal{V}_{B'}(ex, \phi) = \phi$, the conformance condition in Definition 29 is derived.

Fourth, $\mathcal{V}_B(ex, \phi) = \neg\phi$ and $\phi \in ACTL$. Similar to the first case, we have: $\mathcal{V}_E(ex, \phi) \oplus \mathcal{V}_{B'}(ex, \phi) = \neg\phi \oplus \perp = \neg\phi = \mathcal{V}_B(ex, \phi)$.

After examining four cases, the proof is completed. □

In the similar manner to Chapter 4, the subsequent theorems and arguments built on this relaxed conformance definition are presented. Note that, since B' has already taken in account the removal of any overridden transitions, even critical, there is no need to care about the critical overriding composition as in Chapter 4 anymore. In this context, the composition types, such as additive, non-critical overriding and critical overriding, are equivalent in terms of OIMC procedure and verification soundness. The assumption function As is not of concern anymore.

Theorem 30 *Given a base B and a property p, an extension E is attached to B at some interface states. $\forall s \in S_B : \mathcal{V}_B(s, cl(p)) = \mathcal{V}_C(s, cl(p))$ if B and E conform with each other at all exit states.*

This theorem is equivalent to Theorem 17 in Chapter 4. Informally, this key theorem claims that provided the conformance between base and extension, the truth values of base

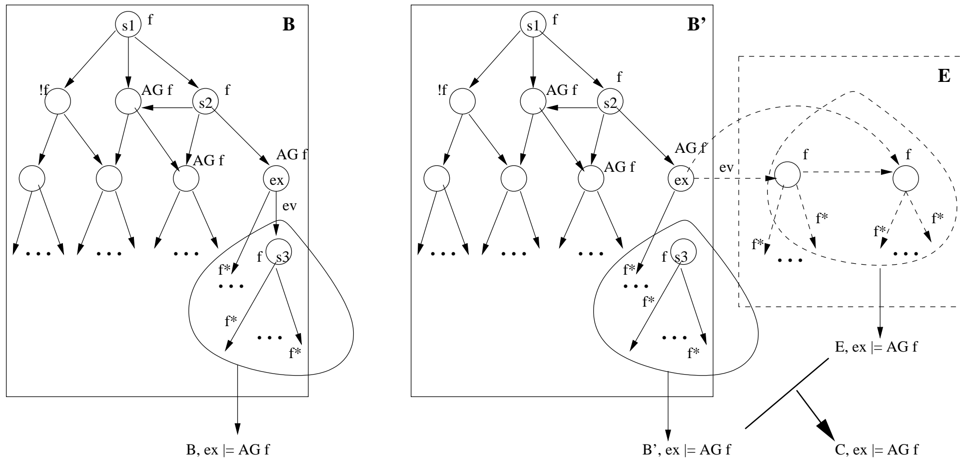

Figure 6.1: An illustration of B and E conformance in case of overriding composition. The truth value with respect to the property $p = \mathbf{AG}\ f$ is preserved at ex and all states in B.

states with respect to $cl(p)$ in B and in C are the same. The theorem holds regardless the composition type, either additive, critical or non-critical overriding. The detailed proof is quite similar with the counterpart in Chapter 4 and hence skipped. Note that comparing the conformance condition of Theorem 30 with the counterpart in Theorem 17, it is obvious that the condition presented in this section is more relaxed.

Figure 6.1 depicts the composition preserving the property $p = \mathbf{AG}\ f$ when B and E are in conformance. The composition is done via a single exit state ex. Further, E overrides the transition ex-s_3 in B. f^* denotes that f holds at all intermediate states along the computation path. In the figure, within B, $p = \mathbf{AG}\, f$ holds at s_2, ex and s_3 but not at s_1. As $\mathcal{V}_E(ex, p) = \mathcal{V}_{B'}(ex, p) = \mathcal{V}_B(ex, p) = \mathbf{AG}\ f \in ACTL$, B and E conform at ex. While the edge ex-s_3 is removed, the new paths in E together with the remaining computation tree in B' still preserve p at ex directly; and consequently s_2 indirectly. For s_1, its truth value $\mathcal{V}_C(s_1, p) = \neg p$ is preserved as well. On the other hand, s_3 is not affected by E. In this figure, we do not care about the descendant states in E. Thus, E is intentionally left open-end so that the reentry state re is not explicitly displayed. In this part, what E can deliver at ex is important regardless of ex's descendants. The arguments are still valid when the downstream of E converges to the reentry state re.

From Theorem 30, given a property p holds on B, it continues to hold in C if B and E conform with each other at all exit states. The following corollary is the answer to the fundamental issue prescribed in Section 3.2. It is equivalent to Corollary 18, except that this corollary is not concerned with the properness of assumption function As due to the weakened conformance condition in Definition 29.

Corollary 31 *Given a base B and a property p holding on B, an extension E is attached to B at some interface states. With respect to the relaxed conformance condition in Definition 29, E does not violate p adhered to B if B and E conform with each other at all exit states.*

With the change of model B into B', the incremental verification algorithm is changed a little bit as follows. Comparing with the counterpart in Section 4.1.4, we only change the seeding labels at reentry states from model B to B'. By this change, we are sure that the seeding is always proper, i.e. As is proper.

1. Seeding reentry states re with the corresponding $\mathcal{V}_{B'}(re, cl(p))$.

2. Executing the CTL assumption model checking procedure within E to check for ϕ, $\forall \phi \in cl(p)$.

3. At the end of the model checking task, checking if $\mathcal{V}_E(ex, cl(p)) \oplus \mathcal{V}_{B'}(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$.

4. If re is not an ascendant of ex in B', then simply skip to the next step. Otherwise, for the strict conformance, two extra checks for the non-existence of any $(f \wedge \neg g)^*$ or f^* loop to assure properties of the forms $\mathbf{A}\,[f\ \mathbf{U}\ g]$ and $\neg\mathbf{EG}\ f$ are required. Details of the checking are already shown in Section 4.1.4.

5. Repeating the procedure for other exit states.

6.2 The Soundness Issue

The relaxed condition presented in this chapter indeed solves some failures in cyclic dependency cases discussed previously. Reminding that in Section 4.2, there are some extreme cases in which the assumption function As is not proper at reentry states. Hence the composition of B and E fails to preserve the property in the base.

The reason for which the soundness issue is no longer of concern is because we seed each reentry state re with the labels $\mathcal{V}_{B'}(re, cl(p))$. It is certain that the labels are proper. That is, As is proper. As a consequence, the incremental model checking in E derives a reliable result. If $\mathcal{V}_{B'}(ex, cl(p)) \oplus \mathcal{V}_E(ex, cl(p)) = \mathcal{V}_B(ex, cl(p))$, the conformance is certain. The conclusion about base and extension consistency upon the relaxed conformance, i.e. Corollary 31, is not effected by the change of condition.

6.3 The Scalability Issue

This issue is not affected by the relaxed condition. As long as the bases and their respective extensions conform pair-wise, OIMC maintains the scalability as discussed in Section 4.4.

6.4 Parallelizing the OIMC Algorithm

Theorem 23 emphasizes on the fact that if the respective base (C_{i-1}) and extension (E_i) components pair-wise conform, although p is introduced by the inner most component B, any future extension E_n can still be independently verified to be consistent with B by OIMC. The input are seeded truth values at the out-states of E_n, whereas the output are truth values derived at in-states of E_n which in turn could be used as input for the OIMC of future extension components. This point serves as the key idea for parallelizing the above OIMC algorithm in component-based software.

Recently, due to the ever-growing complexity of software, there is an urgent need to improve performance of verification procedures. Besides modular verification as of OIMC, one approach is to execute verification process in parallel. Due to Theorem 23, a parallel version of the OIMC algorithm in Section 4.1.4 is possible and it is the focus of this section.

So far, we only consider the simple component-based software of a base and a refinement. In the general case, a component-based system S is formed through a sequence of components $C_1, C_2, ..., C_n$. Initially, $S = C_1$. Later, S is extended with C_2 and then C_3, ..., C_n in that order. Each component C_i is represented by the formal model in Definition 1. Associated with a component C_i is an interface $J_i = \langle in, out \rangle$, where in- and out-states are *terminal* states of the respective component (Definition 3 in Section 3.1).

The relation between components C's could be either refinement or COTS. A COTS component can be indeed regarded as a special case of refinement in which there is only a single exit state and no reentry state with the base. The computation tree of the COTS deviates from the base and never joins the base again. After composed with a COTS, instead of an assumption model checking within the COTS, because there is no reentry state, a standard model checking procedure can be executed entirely within the COTS to find the properties at the exit state. The conformance condition to ensure the consistency between the two components can be well applied as usual.

Regarding the general case of component-based software, the basic consistency issue among components can be described as the following. Initially, a property p is inherent to the first component C_1. The question is on how to ensure the consistency of other components with respect to p in the assembled system S. In this perspective, within S, C_1 is the base, all other components are considered as extensions.[2]

Consider a pair C_i and C_j in which C_j actually refines C_i. As OIMC requires the truth values at all reentry states (i.e. $\mathcal{V}_{C_i}(re, cl(p))$) as the input to its execution within the refinement, the truth values in the base must be determined before executing OIMC in the refinement. This ordering among components is used to schedule the component verification based on the OIMC algorithm. Formally, C_j depends on C_i, denoted as $C_j \xrightarrow{dep.} C_i$, if $\exists o \in J_j.out, re \in S_{C_i} : o \leftrightarrow re$.

If C_i and C_j are COTS, there is no order between C_i and C_j, namely the OIMC-based verification of C_i and C_j can be executed in parallel.

From the above dependency between components, among $C_1, ..., C_n$, there exists a dependency structure showing which components should be verified first, which one can be done later. This structure shows the degree of importance associated with a component with regards to the whole system. Since OIMC is based on assumption model checking which requires a sequential order of components during verification, the dependency structure should not contain any circular dependency among components, namely $\nexists\, i, j, .., k = \overline{1, n} : C_i \xrightarrow{dep.} C_j \xrightarrow{dep.} ... \xrightarrow{dep.} C_k \xrightarrow{dep.} C_i$. As components are integrated in the order $C_1, C_2, ..., C_n$, we assume that any component C_j only depends on the lower-index components, if any. That is, $\forall j, \nexists\, k > j : C_j \xrightarrow{dep.} C_k$. Surely, there is no cycle in the structure.

From the above arguments, any component in S is assigned with a number showing its verification order. The parallel version of the OIMC algorithm consists of two parts. The first part involves with assigning numbers to components. The second part is about actual parallel verification.

```
/* assigning order numbers to components */
```

[2]If p is not initially inherent to C_1 but introduced by C_2, then we consider p to be inherent to the base component $(C_1 + C_2)$. The issue is then translated into the consistency between $C_3, ..., C_n$ with respect to p in $(C_1 + C_2)$. This is identical to the basic case depicted above. If p is introduced subsequently by any other component C_k, the argument is similar and the problem also resembles the basic case.

```
int turn[n];
int maxturn = 0;
for k = 1 to n { turn[k] = 0; }

for i = 1 to n {
   for j = (i + 1) to n {
      if (C_j --dep.--> C_i) ∧ (turn[j] ≤ turn[i]) {
         turn[j] = turn[i] + 1;// C_j must be verified after C_i.
      } // end if
   } // end for
} // end for

for k = 1 to n { maxturn = max(maxturn, turn[k]); }

/* Parallel verification */
for k = 0 to maxturn {
   parallel for i = 1 to n { // executed in parallel
      if (turn[i] = k) {
         Seeding truth values to all out-states of C_i;
         Executing an assumption model checking in C_i to check φ, ∀φ ∈ cl(p);
         Storing the truth values for all in-states of C_i;
      } // end if
   } // end parallel for
} // end for

parallel for k = 1 to n { // read-only loop, hence executed in parallel
   parallel for j = (k + 1) to n {
      if i ∈ J_j.in, ∃ex ∈ S_{C_k} : i ↔ ex {
         if (V_{C_j}(i, cl(p)) ⊕ V_{C'_k}(ex, cl(p)) ≠ V_{C_k}(ex, cl(p))) {
         // C'_k - remainder of C_k after removing overridden transitions.
            Message(``C_j and C_k do not conform at the state i ↔ ex'');
            return(false);
         } // end if
      } // end if
   } // end parallel for
} // end parallel for
return(true);// p is preserved by C_i's.
```

Chapter 7

Model-Based Feature Implementation and Application

This chapter is mainly concerned with the application and realization of the feature-oriented software model. First, a typical application of the theory developed in previous chapters is about component specification and composition. This part proposes a way to facilitate the ideal paradigm of *plug-and-play* components in component-based software.

Next, the chapter is involved with the realization of formally specified features during AOSD implementation phase. A pseudo language is presented to describe formally features. Later, based on the specification of each feature, code transformation is briefed. Specifically, the corresponding Java codes implementing the feature are generated via meta rules. Subsequently, entities such as classes, data members, functions etc. between features are mapped together and then composed accordingly via Hyper/J or AspectJ to form the concrete composed system.

Finally, the relation between the OIMC foundation of previous chapters with an actual model checker (e.g. NuSMV2) is another topic. We examine the possibility of applying the OIMC idea to NuSMV - a well-known open-source model checker.

7.1 Component Specification and Consistency Verification

As an unanimity within the software engineering community, high quality software are structured from lowly coupled *components*. Within the component-based approach, composing components properly is very essential. Component-based software idealizes the *plug-and-play* concept. The current component technology generally supports component matching at the syntactic level. Components can be syntactically checked and hence *plugged*. However, they do not *play* as expected. A major issue of concern is the mismatches of the components in the context of an assembled system [11]. A main source of this phenomenon is because a component violates some property inherent to another. In our opinion, the problem is two-fold: underlying logic is not powerful enough to express component properties; and even if formally specified, it is difficult to verify the properties in an open way - future components are not known in advance. For instance, temporal inter-component constraints are difficult to formally specify, much harder to check among components with the current specification methods. This dissertation introduces the tem-

poral logic CTL into component semantic to facilitate component matching. Specifically, this thesis addresses two points in the issue: how to explicitly specify such a component semantic; and given that kind of information in the component interface, how to efficiently analyze components and to decide whether they are safe to be composed together.

Most current approaches for component interface definition deal with primarily syntactic issues among static interface elements such as *operations* and *attributes*, like those of the CORBA Interface Definition Language (IDL) [14]. Regarding a component's exact capability, essential semantic aspects of the component should also be described. This dissertation advocates the inclusion of two additional semantic aspects of component specification to facilitate proper component composition. Given a base component $B = \langle S_B, \Sigma_B, s_{o_B}, R_B, L_B \rangle$, the semantic aspects are: dynamic behavior (via state transition model in which only potential future interface states are visible to other components - Section 3.1) and their associated consistency constraints (via the truth values of $\mathcal{V}_B(s, cl(p))$ at such an interface state s, where p is a CTL property holding in the base component - Section 4.1).

7.1.1 Interface Signature

Component signatures are the fundamental aspect to the component interface. As commonly recognized, the traditional interface signature of a component contains *attributes* and *operations*. First, through attributes [1], the current state of a software component may be externally observable. The component's clients can observe and even change the values of those attributes. Second, the outside world interacts with the component through operations. The operations represent services or functions the component provides.

Unlike above two static aspects, the introduction of dynamic behavior of a component to the interface is recommended in this paper. Components in reality resemble classes in the object-oriented (OO) approach. This specification style hence follows the encapsulation principle of OO technology so that only essential information is exposed. Only the partial dynamic model of the component consisting of potential future interface states is visible to clients. The rest of the model can be hidden. Associated with a visible interface state s is the set of atomic propositions $L(s)$ (Definition 1). These propositions are often expressed via logic expressions among attributes above.

7.1.2 Interface Constraints

The interface signature only shows the individual elements of the component for interaction with clients in syntactic terms. In addition to the constraints imposed by their associated types, the attributes and operations of a component interface may be subject to a number of further semantic constraints regarding their use. In general, there are two types of such constraints: internal to individual components and inter-component relationships. The first type is simple and has been thoroughly mentioned in many component-related works [14, 40]. The notable examples are the operation semantics according to pre-/post-conditions of operations; and range constraints on attributes. For the second

[1]Attribute is termed as property in [14] which are essentially the entities expressing states of components. To distinguish them from temporal properties inherent to components in Section 7.1.2, those entities are named attributes.

type, current component technologies such as UML and OCL [40], OMG CORBA or Microsoft COM/DCOM etc are limited to a very weak logic in terms of expressiveness. For example, different attributes in components may be inter-related by their value settings; or an operation of a component can only be invoked when a specific attribute value of another falls in a given range etc [14]. The underlying logic only expresses the constraint at the moment an interface element is invoked, i.e. static view, regardless of execution history.

The dissertation introduces two inter-component semantic constraints. The first constraint is based on the *plugging* compatibility for a refining component to be plugged at special states of the base. This situation resembles the extension of use-case scenarios. The base gives the basic interacting scenarios of the component with clients. The extension component refines some of those scenarios further at a certain point from which the component deviates from the pre-defined course to enter new traces in the extension component. Such a point corresponds to an exit state in Definition 2.

On the other hand, the second semantic constraint emphasizes on how to make components *play* once they are plugged. Importantly, this constraint type is expressed in terms of CTL so its scope of expressiveness is enormous. In contrast to the logic above, CTL can describe whole execution paths of a component, i.e. dynamic view. Via OIMC in Section 4.1.4, a refining client E to a base component B can be efficiently verified on whether it preserves the property p of B.

Once composed, the new component $C = B + E$ exposes its new interface signatures and constraints. Static aspects like attributes and operations are simply the sum of those in B and E. The dynamic behavior of C is exposed according to the composition of corresponding visible parts of B and E. In terms of constraints, any potential interface state s is exposed with the set of propositions $L_C(s) = L_B(s)$ according to Definition 5. On the other hand, the consistency constraint at s is derived either from $\mathcal{V}_B(s, cl(p))$ (for any $s \in S_B$) or $\mathcal{V}_E(s, cl(p))$ which is resulted from the above execution of OIMC within E (if $s \in S_E$). Subsequent refinements to C follow the same manner as the case of E to B because of Theorem 23.

7.1.3 Component Specification and Composition

Component specification can be represented via interface signatures and constraints written in an illustrative specification language below. Indeed, a language similar to that of [2] for declaring and refining state machines in layering manner is used. Based on the exemplary specification, components are implemented as classes in typical object-oriented languages. Component composition is then done via class aggregation/merging. Component attributes and operations are declared in the object-oriented style like C++. The `virtual` keyword is used to only name an element without actual memory allocation. The element will be subsequently mapped to the actual declaration in another component. This mechanism resembles `mergeByName` in Hyper/J [36] in which component entities sharing the same label are merged into a single entity during component composition.

Figure 7.1 shows the dynamic model of a simple component, while below is the corresponding specification of the component. The interface signatures should declare: edges with name, start state, end state, transition guard and input event; as well as transition action. At the end are the semantic constraints of the component written in both types

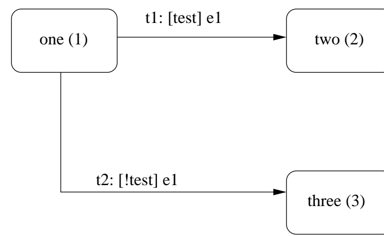

Figure 7.1: The dynamic behavior model of the "black" component.

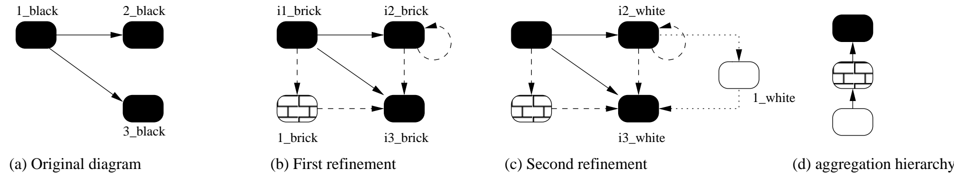

Figure 7.2: Component refinements and component composition via class aggregation.

shown in Section 7.1.2, namely plugging compatible conditions and inherent temporal properties at potential interface states. For illustration purpose, this producer-consumer example is very much simplified so that only some key transitions and states are shown. Because of this over-simplified model, the whole dynamic behavior of the component is visible to clients. In practice, regarding the encapsulation principle, only essential part of the model for future extension is visible. The rest of the model is hidden from clients. There are three components: "black" (the base B of Figure 7.2a - item-producing function); "brick" (the first refinement E of Figure 7.2b - variable-size buffer and item-consuming function); and "white" (the second refinement E' of Figure 7.2c - optimizing data buffer).

```
Component B {
Signature:
   states 1_black, 2_black, 3_black;

   /* edge declarations */
   edge t1:  1_black -> 2_black
   condition test // OK if adding k items to buffer
   input event e1 // producing k items
   do { produce(k)...  }; /* t1 action */

   edge t2:  1_black -> 3_black;
   ...  /* similarly defined */

   // operations and attributes declaration
   boolean test;
   int cons, prod;// consumed, produced items
   int buffer[];// a bag of data items
   ...
   init(){ state = 1_black; ...};
   produce(n){ prod = prod + n;...};

Constraint:
```

```
   /* compatible plugging conditions - CC */
   1_black_cc:  cons = prod;// empty buffer
   2_black_cc:  test = true, cons < prod;
   3_black_cc:  test = false, cons ≤ prod;

   /* Inherent properties - IP */
   1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
   2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
   3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

As components are composed with each other, they can be progressively refined or extended in layering manner. The process adds states, actions, edges to an existing component. The original component and each refinement are expressed as separate specifications that are encapsulated in distinct layers. Figure 7.2 shows this hierarchy: the root component is generated by the specification from Figure 7.1 or Figure 7.2a; its immediate refinements are in turn generated from component specifications according to the order in the Figures 7.2b and 7.2c.

```
Component E {/* for refining black */
Signature:
   states 1_brick, i1_brick, i2_brick, i3_brick;

   /* edges declaration */
   edge t3:  i2_brick -> i3_brick
   condition ...  // ready to consume
   input event ...  // consuming k items
   do { consume(k)...  }; /* t3 action */

   edge t4:  i1_brick -> 1_brick
   condition ...  // ready to change buffer size
   input event ...  // change the size
   do { changesize();...  }; /* t4 action */

   edge t5:  1_brick -> i3_brick;
   edge t6:  i2_brick -> i2_brick;
      // buffer inquiry only, consuming zero item
   ...  /* similarly defined */

   // operations and attributes declaration
   virtual int cons;// mapped with cons in B
   virtual int prod;// mapped with prod in B
   virtual int buffer[];// mapped with buffer in B
   consume(n){ cons = cons + n;...};
   changesize(){ buffer = malloc();...};

Constraint:
   1_brick_cc:  cons ≤ prod;
```

```
   i1_brick_cc:  cons ≤ prod;
   i2_brick_cc:  test = true, cons < prod;
   i3_brick_cc:  test = false, cons < prod;
}

Component E' {/* for refining black + brick */
Signature:
   states 1_white, i2_white, i3_white;

   /* edges declaration */
   edge t7:  i2_white -> 1_white
   condition ...  // ready to compact buffer
   input event ...// compact the data buffer
   do { resetbuffer();...  }; /* t7 action */

   edge t8:  1_white -> i3_white;
   ...  /* similarly defined */

   // operations and attributes declaration
   virtual int cons;// mapped with cons in B
   virtual int prod;// mapped with prod in B
   virtual int buffer[];// mapped with buffer in B
   resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
   1_white_cc:  cons ≤ prod, cons = 0;
   i2_white_cc:  test = true, cons ≤ prod;
   i3_white_cc:  test = false, cons ≤ prod;
}
```

Aggregation then plays a central role in this component implementation style. All the states and edges in Figure 7.2a are aggregated with the refinement of Figure 7.2b; and this figure is in turn united with the refinement of Figure 7.2c. The component to be executed is created by instantiating the bottom-most class of the refinement chain of Figure 7.2d.

The following explains the preservation of the constraint in B by all subsequent two component refinements E and E'. Informally, the property means that under any circumstance, the number of produced items by the component is always greater or equal to that of consumed items. In terms of CTL notation, $p = \mathbf{AG}\ (cons \leq prod)$. The closure set of p is hence $cl(p) = \{p, a\}$, where $a = (cons \leq prod)$.

Initially, B is composed with E. Interface plugging conditions are used to map compatible interface states among components. The base exposes three interface states 1_black, 2_black and 3_black. On the other hand, the refinement component exposes four interface states, namely 1_brick, $i1_brick$, $i2_brick$ and $i3_brick$. Based on the respective atomic proposition sets at those states, corresponding interface states are mapped accordingly. For instance, $L_B(1_black) = \{cons = prod\} \Rightarrow L_E(i1_brick) = \{cons \leq prod\}$. According to Definition 4, $i1_brick \leftrightarrow 1_black$. Also, because $L_E(i2_brick) = L_B(2_black)$,

$i2_brick \leftrightarrow 2_black$. Similarly, $L_E(i3_brick) \Rightarrow L_B(3_black)$, $i3_brick \leftrightarrow 3_black$. Here, $i1_brick$ and $i2_brick$ perform exit states of the base component, while $i2_brick$ and $i3_brick$ are reentry states.

The composite model of the two components $C_1 = B + E$ is shown in Figure 7.2b. After the designer decides on the mapping configuration between interface states, and properly resolves any mismatches at the syntactic level between B and E, the semantic constraint of consistency between the two due to p is in focus. The OIMC algorithm in Section 4.1.4 is applied as follows:

1. Copying $\mathcal{V}_B(s, cl(p))$ to the respectively mapped out-states $i2_brick$ and $i3_brick$ in E for any reentry state s such as 2_black and 3_black.

2. Executing assumption model checking within E to find $\mathcal{V}_E(i1_brick, cl(p))$ and $\mathcal{V}_E(i2_brick, cl(p))$.

3. Checking if $\mathcal{V}_E(i1_brick, cl(p)) = \mathcal{V}_B(1_black, cl(p))$ and $\mathcal{V}_E(i2_brick, cl(p)) = \mathcal{V}_B(2_black, cl(p))$. If so, B and E conform.

The model checking is very simple and hence its details are skipped. At the end, B and E components conform at all exit states. According to Theorem 17, p is preserved by the second component after evolving to $C_1 = B + E$.

C_1 is then extended with E'. Notably, the interface of the new component C_1 is derived from B and E as below:

```
Component C1 {
Signature:
   states 1_black, 2_black, 3_black, 1_brick;

   /* edge declarations */
   edge t1:  1_black -> 2_black;
   edge t2:  1_black -> 3_black;
   edge t3:  2_black -> 3_black;
   edge t4:  1_black -> 1_brick;
   edge t5:  1_brick -> 3_black;
   edge t6:  2_black -> 2_black;
   /* identical to each component's declaration */

   // operations and attributes declaration
   boolean test;
   int cons, prod;// consumed, produced items
   int buffer[];
   init(){ state = 1_black; ...}
   consume(n){ cons = cons + n;...};
   produce(n){ prod = prod + n;...};
   changesize(){ buffer = malloc();...};

Constraint:
   /* compatible plugging conditions - CC */
   1_black_cc:  cons = prod;
   2_black_cc:  test = true, cons < prod;
```

```
   3_black_cc:  test = false, cons ≤ prod;
   1_brick_cc:  cons ≤ prod;

   /* Inherent properties - IP */
   1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
   2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
   3_black_ip:  AG (cons ≤ prod), cons ≤ prod;
   1_brick_ip:  AG (cons ≤ prod), cons ≤ prod;
}
```

The approach in composing E' with C_1 is similar to the above, we have the following mapping configuration between interface states: *i2_white* $\leftrightarrow$ 2*_black*, *i3_white* $\leftrightarrow$ 3*_black*. The same result is achieved, p is preserved by E'. More importantly, the verification method is executed within E' only. After composing E', the component becomes $C_2 = C_1 + E'$ shown below:

```
Component C2 {
Signature:
   states 1_black, 2_black, 3_black, 1_brick, 1_white;

   /* edge declarations */
   edge t1:  1_black -> 2_black;
   edge t2:  1_black -> 3_black;
   edge t3:  2_black -> 3_black;
   edge t4:  1_black -> 1_brick;
   edge t5:  1_brick -> 3_black;
   edge t6:  2_black -> 2_black;
   edge t7:  2_black -> 1_white;
   edge t8:  1_white -> 3_black;
   /* identical to each component's declaration */

   // operations and attributes declaration
   boolean test;
   int cons, prod;// consumed, produced items
   int buffer[];
   init(){ state = 1_black; ...}
   consume(n){ cons = cons + n;...};
   produce(n){ prod = prod + n;...};
   changesize(){ buffer = malloc();...  };
   resetbuffer(){ prod = prod - cons; cons = 0;...};

Constraint:
   /* compatible plugging conditions - CC */
   1_black_cc:  cons = prod;
   2_black_cc:  test = true, cons < prod;
   3_black_cc:  test = false, cons ≤ prod;
   1_brick_cc:  cons ≤ prod;
   1_white_cc:  cons ≤ prod, cons = 0;
```

```
    /* Inherent properties - IP */
    1_black_ip:  AG (cons ≤ prod), cons ≤ prod;
    2_black_ip:  AG (cons ≤ prod), cons ≤ prod;
    3_black_ip:  AG (cons ≤ prod); cons ≤ prod;
    1_brick_ip:  AG (cons ≤ prod); cons ≤ prod;
    1_white_ip:  AG (cons ≤ prod); cons ≤ prod;
}
```

In brief, p is preserved by both extensions E and E'. In this example, the scalability of incremental model checking is maintained as it only runs on the refinements, independently from the bases B and C_1 respectively.

7.2 Layered Architecture for Feature-Oriented Software

The major goal of the illustrative specification language in the example of Section 7.1 is to minimize the "conceptual distance" between architectural abstractions and their implementation [2]. The specification language is similar to that of [2] for declaring and refining state machines in layering style. The architectural abstraction for component refinement is essentially layered.

The layered architecture is very effective in separating *concerns* [37]. A system usually consists of several concerns which are essentially high-level abstraction of some system requirements or goals. At the core of software engineering is the *"separation of concerns"* concept. Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts. It is difficult to manage and to evolve several concerns together, especially when they tangle each other. System complexity can be reduced significantly if each concern can be separately managed. In terms of system design, the layered architecture facilitates the concept by assigning each concern to a layer.

Fundamentally, given a system with several concerns, there are several associated *dimensions* of concerns such as: class, function, feature etc [37]. Thus, there could be several layered architectures for the system due to the system partition in different dimensions. The best layered architecture is the one in accordance to the dimension in which the tangling degree among layers is at minimum.

Regarding component refinement as of this dissertation, the layered architecture resembles the way components refine each other. The base component and each refinement are expressed as separate specifications that are encapsulated in distinct layers. As components are composed with each other, they can be progressively refined/extended in layering manner. The process adds states, transitions and actions to an existing component's behavior model. Figure 7.2d shows the layering hierarchy for the example in Section 7.1: the top layer corresponds to the specification from Figure 7.1 or Figure 7.2a; the below layers are in turn respectively associated with component specifications in Figures 7.2b and 7.2c.

With respect to the general feature-oriented software, the layered architecture also plays an important role in terms of both system development and evolution. First, for

system development, each component targets a particular system function/service. So, from the layered architecture's perspective, each layer corresponds to a component or a group of closely-related components. The layers are ordered from top to bottom according to the sequence of component compositions, i.e. base component on top, and then refinements sequentially. In the example of Section 7.1, there are three layers corresponding to the concerns: item-producing function; variable buffer size plus data-consuming function; and optimum memory usage. The layers are mapped with the base component B and refinements E, E' from top to bottom in accordance with the dependency among components. The development process then simply involves the composition of layers in a proper order. Because the separation of concerns is achieved, the total development cost is significantly reduced.

Second, with regards to system evolution, the layered architecture is especially resilient to system changes - the vital characteristic of open systems. The changes can arise in the form of either providing new functions/services to or removing some parts from the system. Even so, the system architecture still keeps its layering quality. If each service is encapsulated within a layer, any new service can be positioned into the proper position in the layering hierarchy. On the contrary, a service can be disabled from the system by removing the associated layer from the architecture. The key issue is then about whether system consistency among layers is maintained after some layers are inserted to or removed from the architecture. This issue is in essence about the consistency among components mentioned in Sections 4.1 and 7.1.

In brief, with inherent advantages such as the separation of concerns and the resilience to changes, the layered architecture is regarded as a candidate for open systems design in general and feature-oriented software design specifically, at least in terms of architectural abstraction.

7.3 Transforming Formal Feature Model to Codes

Software industry is witnessing the shift from traditional application-specific development approach to model-based software development. The most obvious evidence is the birth of OMG's MDA (Model-Driven Architecture). The MDA provides an open, vendor-neutral approach to the challenge of business and technology change. Based on OMG's established standards, the MDA separates business and application logic from underlying platform technology. No longer tied to each other, the business and technical aspects of an application or integrated system can each evolve at its own pace - business logic responding to business need, and technology taking advantage of new developments - as the business requires. Functional features mentioned in the thesis mainly belong to business and application logic. They can be specified in platform-independent models separate from the technology-specific implementation code. Depending on the specific target platform, from OMG MDA's perspective, the corresponding codes of features can be generated from the associated models.

Currently, in the software industry, there are several automatic tools supporting the transformation from text-based or graphical-based software entity description into a particular programming language. Notably, with the appearance of UML - a general purpose visual modeling language, there are many works on graphical-based code transformation. For instance, many tools support the transformation of UML diagrams such as class,

collaboration [9] and sequence [34, 39] diagrams into their respective programs. The key principle of graphical-based transformation is on mapping the source UML diagrams into UML meta model. The derived output is a meta model for the source UML diagram type [8, 30]. For example, when transforming sequence diagrams, through this mapping, a meta model for sequence diagram is derived. On this meta model, some pattern-based meta rules are then established according to some patterns within this model. Depending on the pattern, each meta rule is specified separately. Continuing the above code transformation of sequence diagrams, a possible simple rule is about constructing a method declaration within a class. Another rule is about conditional method invocation and branching. The common characteristic of these meta rules is context-free. Once the rules are completed, the semantic mapping from meta model and codes are ready. They can be well applied to transform diagrams into a specific programming language, e.g. Java [39].

Unlike graphic-based approach, in our opinion, text-based approach is simpler since the mapping between a text-based specification into a programming language does not require a meta model. Instead, it is simply a relation mapping between two languages. Hence, the transformation from a specification language like that in Section 7.1 and a programming language, e.g. Java, is rather direct. Essentially, the mapping is also context-free.

Because the formal specification model of feature-oriented software can be described in both equivalent forms: state-charts (graphical-based) or pseudo language (text-based) as in Section 7.1, the approach to transforming specification of features into codes can take either way.

7.4 Composing/Weaving Features via Hyper/J and AspectJ

Based on the proposed specification in Section 7.1, components are usually implemented as classes in typical object-oriented languages. Component composition is then done via class aggregation/merging. Aggregation plays a central role in this component implementation style. All members of the class implementing the base component in Figure 7.2a are aggregated with the refinement of Figure 7.2b; and the newly formed class is in turn united with the refinement of Figure 7.2c.

There are possibly many approaches to implement components from their specifications. For example, traditional object-oriented implementation techniques, *mixin layer* [35] or *aspect-oriented programming* (AOP) [17, 36] etc. This dissertation recommends the use of AOP. Aspect-oriented programming currently attracts a great deal of research from the community for its advantage in handling cross-cutting concerns. In fact, AOP outperforms object-oriented programming in capturing software concerns in modular way. As previously discussed, there could be several dimensions of concerns in any system. Object-oriented technology focuses on its *dominant* class dimension. From the layered architecture's perspective, every layer is associated to exactly a unique class. If concerns crosscut multiple objects as they usually do, the class dimension does not capture system variations well. As a result, corresponding codes for those concerns are scattered among objects (or layers in the class dimension). The objects are highly coupled so the evolution cost is certainly high, i.e. a change in any concern will trigger simultaneous updates at cross-cut objects. The OO approach is bad in such a case. Moreover, AOP is selected because of the transparency between layers in the layered design with *hyperslices* [36, 37]

so that the traceability among design and implementation stages is much improved.

The most notable AOP languages are AspectJ [38] and Hyper/J [36]. Their common approach is to capture multi-object crosscutting concerns of a system in separate modules. Each concern corresponds to a module. As their codes are centralized, the cost to handle changes to concerns is significantly improved. The job of the AOP languages is to weave the codes of such concerns into existing object-oriented classes of the system at appropriate places, e.g. *joint points* [38]. The overall result of the approach is the absence of code-tangling among objects. This approach is clearly different to the OO approach in which related codes of a concern are left scattering in objects.

After generating Java codes for each feature as briefed in Section 7.3, the corresponding entities among features such as classes, functions etc are mapped together accordingly. The mapping mechanisms are different between tools. In Hyper/J [36], this mapping is done via a *hypermodule* definition file. This definition file allows a feature entity to merge, to interleave and even to override the respective entity in another feature.

Unlike Hyper/J viewing features as layering on top of each other rather independently, i.e. features are on the peer-to-peer relation, AspectJ [38] views a *pointcut* as a subordinate feature to the system (main feature). Hence, pointcut features usually do not arrange in parallel with each other and with the main feature. Rather, they interleave or weave into it. Weaving features via AspectJ then requires a different mapping from that of Hyper/J.

7.5 NuSMV2

To our knowledge so far, all the current well-known model checkers such as SPIN [15], SMV [21, 22, 23] etc. do not support assumption model checking directly. The assumption aspect is the essential part of OIMC. To put OIMC into practice, it is necessary to adapt a current model checker with the assumption model checking mechanism. Among those, NuSMV [3, 4] is selected because several reasons.

1. NuSMV is a derived product of SMV. It inherits all the merits of SMV. Further, it re-engineers SMV for a much better design, documentation and comprehensibility.
2. It is an open-source and on-going project. So there is a great possibility to change the existing codes to deal with assumption verification.

The work on adapting NuSMV with assumption model checking is not yet finished. Further results will be provided in the future work.

Chapter 8

Related Work and Conclusion

A rigorous mathematical foundation of open incremental model checking (OIMC) is proposed. It starts by presenting a formal model for feature-based software together with the composition definition. Each feature is separately specified by a state transition model. Later, the fundamental issues of OIMC are suggested. Subsequent parts of the work devote mainly to the effort to answer the fundamental issues of handling software change: consistency, scalability and soundness.

This thesis is actually a detailed and improved combination of the earlier works [10, 26, 27]. Compared with those works, it contains several significant advantages. They are:

1. A precise and generalized formal model of feature-based software is proposed. In particular, the formal interface consisting of multi exit, reentry and dual states is considered. In addition, base behavior overriding is possible within the model and it can be permitted to a great extent.

2. Sound mathematical proofs for CTL properties verification are given. In fact, an uniform preservation constraint at exit states are required to guarantee *consistent* composition between B and E. That constraint is the conformance of B and E at all exit states, i.e. $\mathcal{V}_B(ex, cl(p)) = \mathcal{V}_E(ex, cl(p))$.

3. An algorithm is presented to carry out the incremental model checking process in the extension in which some extra loop-checking care is required to ensure the *strict* conformance condition.

4. The soundness issue concerning with the reliability of OIMC is also investigated. We prove that under additive and non-critical overriding composition, results delivered by the incremental verification is sound, whereas it may not be in some critical-overriding cases with circular dependency structure.

5. The scalability of Theorem 17 is also discussed. The incremental model checking procedure maintains its scalability as it only depends on the size of the extension for any subsequent modules to be added to the newly found model C, as long as the extensions and their respective bases pair-wise conform.

6. A systematic procedure to transform a multi-object feature models into a global model suitable for the OIMC algorithm. Some principles such as external/internal exit and reentry events, base invariants, interaction scenarios and compatible-interface

conditions are introduced to deal with the problem of state explosion due to taking cross-product of object states.

7. Some attempts to improve the existing theoretical foundation of OIMC. Specifically, a relaxed conformance condition between features actually helps to overcome the soundness failures in extreme cases of circular dependency structure mentioned above. By seeding the truth values in model B' instead of those in B, the assumption function As during OIMC is certainly proper. As a consequence, the result delivered by OIMC is sound. Another attempt is to parallelize the OIMC algorithm in the context of general component-based software.

8. Some important applications based on the foundation are suggested. Notably, the results can be applied to deal the most challenging issue in component-based software, namely component composition. The work advocates the inclusion of *dynamic behavior* and *inter-component consistency* as the semantic constraints to each component interface. The proposed specification style is particularly useful for component-based software in general, especially feature-oriented software during system evolution.

OIMC is indeed based on two fundamental theorems 17 and 21 under the conformance between base and extension. The former is about properties preservation at base states, while the latter is for the counterpart at extension states.

Modular model checking is rooted at assume-guarantee model checking [18, 31]. However, unlike the counterpart in hardware verification [13, 18], software modular verification [20] is restricted by its sequential execution nature. Therefore, properties at the interface states are required to be stricter. Incremental model checking inspires verification techniques and the theoretical foundation further. "Separation of concerns" aims to an ideal software paradigm that is open to other software modules, one module per concern, to be plugged in or removed at will without the expense of modules' mutual consistency and validity. Under such an open software paradigm, the interfaces between modules are critical for modules to work properly. The formal interface model between the base and the extension features as well as the preservation constraints at those interface states proposed in this thesis are a part of the effort towards that goal. In a reference to the existing programming tools in AOSD paradigm, a foremost advantage of our model in facilitating downstream programming tools like Hyper/J [36] or AspectJ [38] is the ability to verify consistency and to check for behavioral correctness among concerns within this formal model. Ensuring consistency among concerns is one of the most challenging questions for AOSD nowadays. Bridging the formal specification in this thesis and those programming tools is an important topic within AOSD area.

Regarding the assumption aspect in component verification, [12] presents a framework for generating assumption on environments in which the component satisfies its required property. This work differs OIMC in some key points. First, the constraints in OIMC are explicitly fixed at $\mathcal{V}_B(ex, cl(p))$ for any exit state ex, whereas based on a fully specified component model including error states, [12] generates assumption about operation calls by which the environment does not lead the component to any error state. Second, the approach in [12] is viewed from a static perspective, i.e. the component and the external environment do not evolve. If the component changes after adapting some refinements, the assumption-generating approach is re-run on the whole component, i.e. the compo-

nent model has to be re-constructed; and the assumption about the environment is then generated from that model.

Comparing to the modular verification work such as [13, 18, 31], there is a fundamental difference in characteristic between those and the work of both [10] and ours. Modular verification in those work are rather closed. Even though it is based on component-based modular model checking, it is not prepared for component addition. If a component is added, the whole system of many existing components and the new component is re-checked altogether. On the contrary, the approach in [10] and this work is incrementally modular and hence more open. We only check the new system partially in terms of new component and its interface with the rest of the system. Certainly, this merit comes at the cost of "fixed" conditions at exit states. This "fixed" constraint can deliver a false negative of conformance to legal extensions. One of the future work is to reuse the assumption model checking within the extension E to check the base B, even if the two do not conform at all exit states.

It is essential to consider the effectiveness and complexity of modular model checking of the extension only with respect to the complete model checking of both the base and extension. There is a relationship between these two factors. If the extension and the base are very lowly coupled, i.e. the features offered by these two collaborations are quite orthogonal, then the interface will be very small and the collaboration-based model checking is significantly more effective. On the contrary, if the extension and the base communicate to each other via a large interface, the modular verification is quite complicated then. Under such a case, it might be better to check the complete composition of the two instead of each collaboration separately.

At this stage, well-known existing model checkers such as SPIN [15] or SMV [21, 22, 23] do not support this incrementally modular verification technique. Constructing a front-end preprocessor transforming a partial model into a suitable form for SPIN or SMV is very essential. Assumption model checking is quite new to the field of model checking. Thus, current model checkers does not support this checking style yet. To enable OIMC, the capability to provide assumption model checking is critical. So adapting such a checker, for instance NuSMV [3, 4], to handle this model checking style is the next step in the research direction.

Finally, like the proposal about encapsulating dynamic behavior model into component interface, i.e. state-full interface, two closely related works [5, 7] also advocate the use of *light-weight formalism* to capture temporal aspects of software component interfaces. More specifically, this paper simply relies on state transition model in the most general sense, while the approach in [5, 7] presents a finer realization of state-full model in which states are represented by control points in operations of components; and edges are actually operation calls. That approach focuses on the order of operation calls in a component [1]. By formalizing a component through a set of input, output and internal operations, the compatibility between component interfaces with regards to the structure of component operations is defined and checked. In addition, the two approaches target different aspects of consistency. This paper is concerned with component consistency in terms of CTL properties, whereas the approach in [5, 7] is involved with the correctness and completeness of operation declarations within components. Instead of substituting each other, the two approaches are hence more about complement to each other.

[1] In [5], operations are named as methods.

Bibliography

[1] F. Armour and G. Miller. *Advanced Use Case Modeling: Software Systems*. Addison-Wesley, 2001.

[2] D. Batory, C. Johnson, B. MacDonald, and D. V. Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proc. International Conference on Software Reuse*, July 2000.

[3] R. Cavada, A. Cimatti, G. Keighren, et al. *NuSMV 2.2 Tutorial*. CMU and ITC-irst, nusmv@irst.itc.it, 2004.

[4] R. Cavada, A. Cimatti, E. Olivetti, et al. *NuSMV 2.2 User Manual*. CMU and ITC-irst, nusmv@irst.itc.it, 2004.

[5] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, M. Jurdzinski, and F. Y. C. Mang. Interface compatibility checking for software modules. In *Proceedings of the Computer-Aided Verification - CAV*. LNCS Springer-Verlag, 2002.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[7] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Symposium on Foundations of Software Engineering*. ACM Press, 2001.

[8] G. Engels, R. Heckel, and J. M. Kuster. Rule-based specification of behavioral consistency based on the UML meta-model. In *International Conference on the Unified Modeling Language, UML'01*, pages 272–286, 2001.

[9] G. Engels, R. Hucking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In *International Conference on the Unified Modeling Language, UML'99*, pages 437–488, 1999.

[10] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proc. Symposium on the Foundations of Software Engineering*, September 2001.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, 1995.

[12] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings of the International Conference on Automated Software Engineering*, 2002.

[13] O. Grumberg and D. E. Long. Model checking and modular verification. In *International Conference on Concurrency Theory*, volume 527 of *Lecture Notes of Computer Science*. Springer-Verlag, 1991.

[14] J. Han. An approach to software component specification. In *Proceedings of International Workshop on Component Based Software Engineering*, 1999.

[15] G. J. Holzmann. *Basic Spin Manual*. AT&T Bell Laboratories, http://spinroot.com/ spin/ Man/ Manual.html, 2004.

[16] T. Katayama. Evolutionary domains: A basis for sound software evolution. In *Proc. IWPSE*, 2001.

[17] G. Kiczales, J. Lamping, et al. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming - ECOOP'97*, pages 220–242. Springer, 1997.

[18] O. Kupferman and M. Y. Vardi. Modular model checking. In *Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[19] L. Lamport. *LaTeX- A Document Preparation System*. Addison-Wesley Publishing Co., 1986.

[20] K. Laster and O. Grumberg. Modular model checking of software. In *Conference on Tools and Algorithms for the Constructions and Analysis of Systems*, 1998.

[21] K. L. McMillan. *The SMV System*. CMU, http://www.cs.cmu.edu/ modelcheck/ smv/ smvmanual.r2.2.ps, 1992.

[22] K. L. McMillan. *Getting Started with SMV*. Cadence Berkeley Labs, Cadence Design Systems, 1999.

[23] K. L. McMillan. *The SMV Language*. Cadence Berkeley Labs, Cadence Design Systems, 1999.

[24] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1997.

[25] T. T. Nguyen. Formalization and evolution of collaboration-based object-oriented methodology. Master's thesis, Japan Advanced Institute of Science and Technology - JAIST, August 2002.

[26] T. T. Nguyen and T. Katayama. Dynamic behavior and protocol models for incremental changes among a set of collaborative objects. In *Proc. IWPSE*, pages 45–50, 2003.

[27] T. T. Nguyen and T. Katayama. Towards a sound modular model checking of collaboration-based software designs. In *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 88–97, 2003.

[28] T. T. Nguyen and T. Katayama. Handling consistency of software evolution in an efficient way. In *Proc. IWPSE*, pages 121–130, 2004.

[29] T. T. Nguyen and T. Katayama. Open incremental model checking. In *Proc. SAVCBS - Specification and Verification of Component-Based Systems*, pages 134–137, 2004.

[30] D. H. Park and S. D. Kim. XML rule-based source code generator for UML case tool. In *Asia-Pacific Software Engineering Conference, APSEC'01*, pages 53–60, 2001.

[31] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes of Computer Science*. Springer-Verlag, 1999.

[32] D. Rosenberg and K. Scott. *Use Case Driven Object Modeling With UML: A Practical Approach*. Addison-Wesley, 1999.

[33] J. Rumbaughm, I. Jacobson, and G. Booch. *The Unified Modeling Language - Reference Manual*. Addision-Wesley, 1999.

[34] N. Sangal, E. Ferrel, K. Lieberherr, and D. Lorenz. Interaction schemata: Compiling interactions to code. In *Technology of Object-Oriented Language and Systems,TOOLS USA'99*, pages 268–277, 1999.

[35] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP*, 1998.

[36] P. Tarr and H. Ossher. *Hyper/J(TM) User and Installation Manual*. IBM Research, IBM Corp., 2000.

[37] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N-degrees of separation: Multidimensional separation of concerns. In *Proc. ICSE*, pages 109 – 117, 1999.

[38] The AspectJ Team. *The AspectJ(TM) Programming Guide*. Xerox Corporation., 2001.

[39] M. Thongmak and P. Muenchaisri. Design of rules for transforming UML sequence diagrams into Java code. In *Asia-Pacific Software Engineering Conference, APSEC'02*, pages 485–494, 2002.

[40] J. Warmer and A. Kleppe. *The Objects Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

Publications

[1] T.T. Nguyen and T. Katayama, "Collaboration-Based Evolvable Software Implementations: Java and Hyper/J vs. C++ Templates Composition", ACM - Proceedings IWPSE (International Workshop on Principle of Software Evolution), pp.29-34, 2002.

[2] T.T. Nguyen and T. Katayama, "A Framework for Unanticipated Software Changes", Proceedings USE (International Workshop on Unanticipated Software Evolution), pp.114-129, 2003.

[3] T.T. Nguyen and T. Katayama, "Dynamic Behavior and Protocol Models for Incremental Changes among a Set of Collaborative Objects", IEEE - Proceedings IWPSE (International Workshop on Principle of Software Evolution), pp.45-50, 2003.

[4] T.T. Nguyen and T. Katayama, "Towards a Sound Modular Model Checking of Collaboration-Based Software Designs", IEEE - Proceedings APSEC (Asia-Pacific Software Engineering Conference), pp.88-97, 2003.

[5] T.T. Nguyen and T. Katayama, "Handling Consistency of Software Evolution in an Efficient Way", IEEE - Proceedings IWPSE (International Workshop on Principle of Software Evolution), pp.121-130, 2004.

[6] T.T. Nguyen and T. Katayama, "Open Incremental Model Checking", Microsoft Research - Proceedings SAVCBS (Specification and Verification of Component-Based Systems), pp.134-137, 2004.

[7] T.T. Nguyen and T. Katayama, "A Formal Approach Facilitating the Evolution of Component-Based Software", IEEE - Proceedings IWPSE (International Workshop on Principle of Software Evolution), pp.49-52, 2005.

[8] T.T. Nguyen and T. Katayama, "Constructing Open Systems via Consistent Components", International Colloquium on Theoretical Aspects of Computing (ICTAC), Springer-Verlag LNCS 2005.

[9] T.T. Nguyen and T. Katayama, "Specification and Verification of Inter-Component Constraints in CTL", Microsoft Research - Proceedings SAVCBS (Specification and Verification of Component-Based Systems), pp.15-22, 2005.