SPECIFYING and VERIFYING COLLABORATIVE BEHAVIOR IN COMPONENT-BASED SYSTEMS

Levent Yilmaz

Dissertation submitted to the Graduate School of Virginia Polytechnic Institute and State University in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy
in
Computer Science

Approved By

Dr. Stephen Edwards (Chair) _____

Dr. James D. Arthur _____

Dr. J.A.N. Lee _____

Dr. Richard Nance _____

Dr. Binoy Ravindran _____

March 13, 2002
Blacksburg, VA

Keywords: verification, validation, testing, component-based, collaborative behavior

Abstract

In a parameterized collaboration design, one views software as a collection of components that play specific roles in interacting, giving rise to collaborative behavior. From this perspective, collaboration designs revolve around reusing collaborations that typify certain design patterns. Unfortunately, verifying that active, concurrently executing components obey the synchronization and communication requirements needed for the collaboration to work is a serious problem. At least two major complications arise in concurrent settings: (1) it may not be possible to analytically identify components that violate the synchronization constraints required by a collaboration, and (2) evolving participants in a collaboration independently often gives rise to unanticipated synchronization conflicts. This work presents a solution technique that addresses both of these problems. Local (that is, role-to-role) synchronization consistency conditions are formalized and associated decidable inference mechanisms are developed to determine mutual compatibility and safe refinement of synchronization behavior. More specifically, given generic parameterized collaborations and components with specific roles, mutual compatibility analysis verifies that the provided and required synchronization models are consistent and integrate correctly. Safe refinement, on the other hand, guarantees that the local synchronization behavior is maintained consistently as the roles and the collaboration are refined during development. This form of local consistency is necessary, but insufficient to guarantee a consistent collaboration overall. As a result, a new notion of global consistency (that is, among multiple components playing multiple roles) is introduced: causal process constraint analysis. A method for capturing, constraining, and analyzing global causal processes, which arise due to causal interference and interaction of components, is presented. Principally, the method allows one to: (1) represent the intended causal processes in terms of interactions depicted in UML collaboration graphs; (2) formulate constraints on such interactions and their evolution; and (3) check that the causal process constraints are satisfied by the observed behavior of the component(s) at run-time.

Research Grants

- Principal investigator (PI) for "Force Level Automated Certification of Downward Compatible Baseline Software," Naval Surface Warfare Center, Small Business Innovation Research (SBIR) Phase I, II research and development grants.
- PI for "Upward Compatible Baseline Support Framework for Effective Force Level System Regression Testing and Certification," Naval Surface Warfare Center, SBIR Phase I, II research and development grants.
- PI for "Processor Technology Insertion Recertification," Naval Air Warfare Center, SBIR Phase I research grant.
- PI for "Synthetic Natural Environment Interoperability," Army Simulation Training and Instrumentation Command (STRICOM), SBIR Phase I research grant.

I am grateful to Naval Surface and Air Warfare Centers as well as U.S. Army Simulation Training and Instrumentation Command (STRICOM) for funding my V&V research so far under the grants N00178-00-C-3029, N00178-00-C-3039, N68335-01-C-0045, and N61339-02-C-0021.

Dedicated to the memory of my mother, Nazmiye Yilmaz.

Acknowledgements

I would like to thank my advisor, Dr. Stephen Edwards, for not only giving me the courage to pursue various reincarnations of the topic presented in this work but also listening and suggesting directions that enabled me to formulate and refine the ideas culminated in this dissertation. I am grateful to my committee members Dr. James D. Arthur and Dr. Richard Nance for their support, enthusiasm, and advise. I will always be indebted to them for being role models as computer scientists with common sense and wisdom. I would like to thank Dr. JAN Lee for letting me always see the big picture and for extending his office hours so often to discuss both my progress and philosophical aspects of verification and validation. Thanks to Dr. Binoy Ravindran for discussing and showing me the proliferation of issues of verification and validation in the context of real-time systems domain. Special thanks to Nick Karangelen and Leland Tankersley of Trident Systems Incorporated for providing me a flexible research environment and spending their valuable time discussing potential research ideas as well as business development opportunities. I would also like to thank my parents for their unwavering love, support, and encouragement for my academic pursuits. Finally, special thanks to my wife, Funda, for her understanding and patience. She has asked for so little but has given me so much.

Table of Contents

CHAPTER 3: THE PRINCIPLES OF FORMALIZATION FOR CONSISTENCY CONDITION IDENTIFICATION and SPECIFICATION

CHAPTER 4: SAFE REFINEMENT AND MUTUAL ROLE COMPATIBILITY ANALYSIS OF INTERACTION POLICIES FOR LOCALCONSISTENCY

List of Figures

CHAPTER 1

INTRODUCTION

In the past decade, the object-oriented paradigm has been considered and promoted as a solution to most software development headaches. Many software engineers viewed object-orientation as the answer to numerous problems such as enhancing software quality, reusability, and enabling seamless development methodology. In spite of its wide adoption, the major justification of the object-oriented paradigm —systematic and dependable reuse based on plug-compatible and reusable objects— has yet to be achieved. In fact, there has been a gradual realization that object-oriented programming languages and programming alone do not have enough impact to transform the software industry from custom-boutique development to "component-based" development.

As a result, the development of applications or components from pre-certified, plug-compatible software components has become a trend growing in importance. In this paradigm, new components are built as compositions of pre-packaged software components (i.e., classes) assembled to form large composite components or applications that consist of *cooperating and communicating* objects, each with well-defined responsibilities. Component-oriented development is described as the natural extension of object-oriented programming where components are defined to be static abstractions with contractually specified interfaces that enable them to be composable [Nierstrasz, 1995].

1.1 COMPONENT CERTIFICATION WITH REUSE

Although a considerable amount of research has been devoted to component and composition-based concepts and techniques, the impact of this technology on component certification has yet to be fully explored. The clients of reusable components will expect the parts they purchase and use to work correctly with respect to their *catalog*

descriptions. Therefore, a concrete component must be trustworthy; that is, it must correctly implement the abstract service specification on which its clients will depend.

Since a reusable component is developed with specific contextual assumptions and is unlikely to comply completely and without conflicts with the domain of interest, the challenge of certification with reusable components is far greater than that of conventional software. For the correctness of applications developed from reusable components, local certification must establish their properties outside of the context in which they are embedded [Weide & Hollingsworth, 1992]. However, components cannot be built in isolation; instead, they must observe compatibility constraints that are often phrased as architectural rules [Lea, 1999; Dellarocas, 1997a]. Therefore, development with reuse requires knowing whether the certified properties of a component will facilitate its playing certain *roles* to establish a specific purpose in the context in which it is embedded.

For that reason, the verification as well as reuse of concurrent, persistent, and/or secure components faces challenges not seen, for example, in most UI toolkits and data structure libraries [Lea 1999]. Factors such as specification and design policies (i.e., synchronous or asynchronous, local or remote, callback, forwarding, or delegation), reflective, multilevel design (composite components), security, safety, liveness, fault-tolerance, auditability, recovery policies make specification, certification, and reasoning of reusable components difficult.

Furthermore, with regard to verification, the nature of component-based development methodology introduces new sources of problems, particularly from the point of view of collaboration assumptions. Such problems are explained in detail in this chapter. Other issues introduced by the object-oriented paradigm can be found in object-oriented software testing literature [Barbey and Strohmeier 1994; Binder 1994; Smith and Robson 1992].

Component certification involves the application of static analysis, inspections, formal verification and testing to demonstrate the correctness of the reusable components with respect to their published properties. It refers to the task of establishing both the properties and invariants of a component. At first glance, in the context of reuse the two

most obvious approaches for demonstrating the fitness and suitability of components involve formal verification and testing.

Formal verification refers to the formal process of symbol manipulation according to well-defined proof rules and reasoning mechanisms associated with the formal languages used to write abstract behavioral specifications and concrete implementations Although the term has broader definitions in other software engineering communities, here, verification is referred to as formal demonstration of the compliance of an artifact with respect to its specification [Wallace and Fujii 1989]. On the other hand, testing involves running operations on test data and inspecting the results to determine whether the component actually does what is expected.

Multiple facets of a reusable component require certification. This dissertation introduces a synchronization-centered and reuse-oriented component certification framework for composite object-oriented components that have explicit architectural interaction constraints that must be fulfilled to function correctly. The framework would be useful particularly in real-time, embedded, concurrent systems, and *multi-agent system* domains. Note that in domains where component-to-component negotiation and interaction protocols are explicit, protocol-centered reuse is the common denominator [Pintado, 1995].

1.2 THE PROBLEM STATEMENT

Software designs consist of *components, roles*, and, *collaborations*. Actors that represent the *components* play *roles* collaboratively to implement *collaborations*. Collaborations are descriptions of how a group of components jointly behave when configured together in specific ways. For the design and code implementing such components and collaborations, software designs must provide coherent organization. Conventional software design methodologies arrange designs around components: each module denotes a component and a collection of component modules form the complete design. However, recent trends in software design lead to design organizations around collaborations (i.e., protocol-centered reuse) rather than components to facilitate more reusable designs and implementations. In such a design each module reflects a collaboration feature and contains components synchronizing through interaction to play roles relevant to

implementing that *collaboration feature*. Based on the observations of this emerging trend, the main thrust of this research is motivated by the following question: Given that the collaboration-based software designs based are successful and reuse effective at implementing software product lines, can this trend and viewpoint suggest and excite a new emerging prospect in certifying such designs?

Reusing collaborations under a collaborative behavior design introduces two major certification complications: (1) components that do not conform to the synchronization requirements of the collaboration context (i.e. role) cannot usually be identified analytically early in the development and (2) evolving collaboration designs and code often produce conflicts and inconsistencies with the rest of the (re)used components. Hence, the activities of disparate reused components cannot usually be coordinated as intended, since they are not originally designed around the contextual collaborative behavior imposed by the domain into which they are embedded. These complications lead to the formulation of the following questions that will be addressed in this dissertation:

Given generic parameterized collaborations and components with specific roles, how do we verify that the provided and required synchronization models are consistent and integrate correctly?

How do we guarantee that the synchronization behavior is maintained and kept consistent with the collaboration constraints as the roles and collaboration are refined during development?

How do we verify if the emergent cross-cutting synchronous causal behavior, which arises due to the causal interference, interaction, and coordination of components, is as intended? Furthermore, how do we determine whether the coordination of roles played by each used component is evolved as expected during the evolution of the collaboration as the system is maintained?

Why is this Problem Important and Difficult? Software technology is undergoing a transition from monolithic systems or components, constructed according to a single unified overall design derived from scratch into collections of semiautonomous, heterogeneous, and independently designed components, developed and managed by different organizational units with little knowledge of each other. Controlling the activities of the potentially active disparate components and ensuring that they are coordinated and evolved as intended around a given scenario and pattern are among the serious problems inherent in such collections. In collaboration-based designs, components —particularly active objects— exhibit state-dependent behavior and may require that particular synchronization constraints be obeyed. Hence, an object's synchronization contract with its clients should specify its behavior for all permissible ways of interaction.

Existent widely-used and accepted contracts (i.e., behavioral pre and postconditions) do not directly capture both this interactive notion of behavior and descriptions of the intended use of the components. Existing modeling languages and behavioral contracts based on model-based specifications that provide pre and postconditions for each individual function of a component are widely used for describing the services that components provide.

Such contracts specify the structural characteristics and functional assumptions of a component; however, they do not address how a component expects to interact with its clients (i.e., interaction assumptions and guarantees). Furthermore, they do not address how the overall configuration coordinates to achieve its behavior. However, the process through which the component accomplishes its behavior is an intrinsic characteristic that establishes a certain functional or non-functional goal.

Hence, as part of a collaboration design, we associate each role with a type specification, called interaction policy. Interaction policies denote the synchronization contracts or, in other words, interaction protocols of components with respect to a single role. They provide a language independent specification of external interface that constitutes the visible synchronization behavior in terms of an interaction-oriented abstract model.

What is Required? There are several questions that this research seeks to answer: Which representation scheme and model should be used to represent synchronization contracts (i.e., interaction policies) of a component at the abstract level? What are the necessary and sufficient conditions and corresponding decision procedures that facilitate tracking and maintaining semantic dependencies between components based on the pertinent synchronization model? That is, how do we verify that the roles integrate correctly, and how do we ensure that the incremental enhancements to collaboration role models are consistent with the original commitments made at previous levels of abstractions? Furthermore, given that the used components fit into the roles expected from them, is the actual emergent collaborative behavior, which arises from the cascading of local interactions, achieved or evolved the way intended by the designer? Specifically, the following three subproblems facilitate systematically addressing the main problem:

- In the context of collaborative behavior design, in order to facilitate the realization and maintenance of precise behavioral dependencies among role abstractions, what kind of local synchronization consistency conditions are relevant? Which modeling perspective and semantic domain are appropriate to formalize these consistency conditions? Under this semantic domain, what are the necessary and sufficient conditions for their correct realization?
- With regard to the semantic domain, what are the proper analytical techniques to facilitate determining if the identified local consistency conditions are established correctly?
- Given that roles interact and causally interfere each other, and that the behavior of a particular role might depend on others with which it has causal connections, how do we define and determine if the *emergent* global causal processes are as intended and whether the components are playing their roles within the limits and constraints of the expected patterns?

Each of these problems is briefly discussed in the following subsections.

1.2.1 Local Role Consistency in Collaboration-Based Software Design

In the establishment of a market for software components, the separation of specification and implementation is a well-known prerequisite. Clients choosing the components based on their catalog descriptions (i.e., contract specifications) expect them to behave correctly with respect to their published contracts. In general, as shown in figure 1.1, contracts can be divided into four main levels with increasingly negotiable properties [Beugnard et al. 1999].

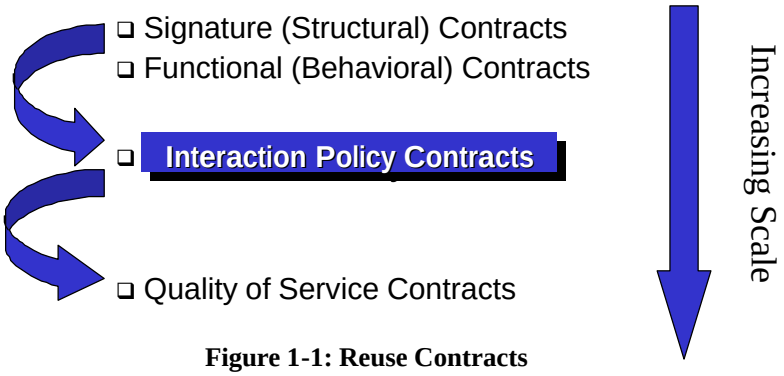


Figure 1-1: Reuse Contracts

The first level, that of basic contracts, is required to simply define the signature and types. The second level, based on behavioral contracts, improves the level of client confidence, particularly in a sequential context. The third level, emphasizing the synchronization and interaction pattern contracts, improves confidence particularly in concurrent and distributed settings. The fourth level, quality of service contracts, quantifies the quality of service.

Although consistency notions exist for functional and structural contracts, still missing with regard to collaboration-based software with collaborative behavior reuse is a general consistency notion that emphasizes **synchronization contracts**. Hence, determining methodologically whether the synchronization constraints of used components' roles fit into their local context (i.e., horizontal consistency) or are safely extended (i.e., vertical consistency) is still an elusive goal. In particular, the major issues that need to be addressed involve the types of consistency relationships and the means for modeling and expressing synchronization constraints to facilitate the defining of conditions for consistency relations.

1.2.2 Decision Procedures for the Analysis of Local Consistency Conditions

Once functional and signature compatibility is established, synchronization play a key role in assessing the fitness of components in a collaboration. The preliminary analysis for synchronization fitness (consistency) of a component is determined by its interaction policy compatibility with respect to its context and depends on the interaction semantics under consideration. For object interaction, different interaction and request scheduling mechanisms are devised. Conventional environments such as C++, Smalltalk, Objective-C are based on unconditional acceptance where the temporal traits are implicit and all interface services are uniformly available even though a particular message ordering may be expected.

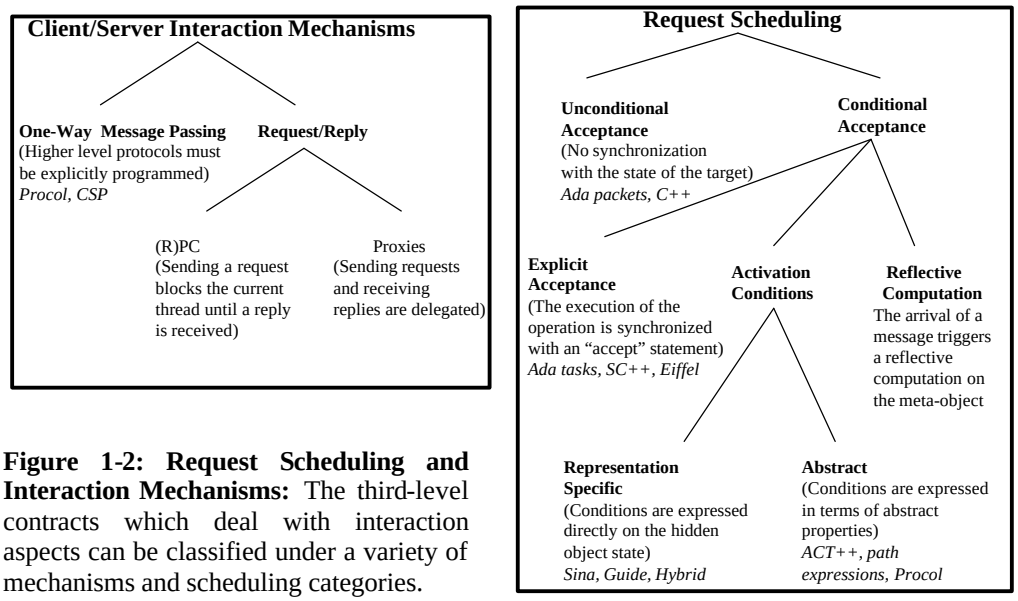


Figure 1-2: Request Scheduling and Interaction Mechanisms: The third-level contracts which deal with interaction aspects can be classified under a variety of mechanisms and scheduling categories.

On the other hand, as depicted in Figure 1.2, there exist development environments based on conditional acceptance, where message acceptance and activation conditions are explicitly represented and are non-uniformly available (i.e., Ada tasks, Synchronous C++, Synchronous Java, Procol, Java/CSP, Sina, Hybrid, POOL-T). With *explicit acceptance*, the requests are scheduled by means of explicit statements. Acceptance may be based on message contents as well as the object's current state. With *activation conditions,* the requests are accepted on the basis of predicates over object states.

Given the semantic domain used to express the synchronization consistency conditions based on a certain interaction and request scheduling mechanism, for correct realization, the next step involves developing decision procedures to analyze the conditions mapped to the semantic domain. This involves procedures determining (1) role compatibility (horizontal consistency), (2) safe refinement (vertical consistency) of role and collaboration evolutions, and (3) correct realization and implementation of the expected local role interaction policies.

1.2.3 Causal Process Constraint Analysis for Achieving Global Consistency

For achieving global consistency, analysis of local synchronization constraints for horizontal and vertical consistency is necessary but insufficient. This is due to the fact that the synchronization behavior of a role often depends on other roles that, because of shared data state space and control and data dependent causal connections, interfere with its interaction decisions. Furthermore, for many component engineering tasks, including reasoning, verification, and reuse, such causal connections depict why an artifact is configured as it is. This is especially important when non-functional aspects are derived from a particular design rationale based on certain causal processes embedded in the component.

Hence, to achieve global consistency, there is a need to provide methods for capturing, constraining, and analyzing expected causal processes. Due to cascaded local synchronous interactions, the *emergent* collaborative behavior designates how the component's behavior depends upon the composition of its constituent components and how the component behavior is accomplished. Here, the conjecture asserts that in order to verify that a component achieves its goal as intended by the designer, one must show not only that the final expected condition is satisfied but also that the components participated in the expected causal process that resulted in the expected function. Hence, the causal processes underlying the intended design patterns of the component designate a certain aspect of the designer's rationale and purpose to establish certain non-functional as well as functional goals. In order to realize this rationale and purpose it is essential to provide a means of (1) representing the intended causal processes, (2) formulating constraints on causal processes and their evolution, and (3) verifying whether the causal

process constraints are satisfied by the observed behavior of the component. In this way, it would be possible decide if the components (re)used in the intended process work to achieve particular functional or nonfunctional goals.

1.3 THE LIMITATIONS OF CONVENTIONAL APPROACHES

Previous work on verification and testing of reusable object-oriented components has mostly focused on ad hoc application or extension of conventional techniques based on functional or abstract data type theory (model-based and algebraic approaches) focusing on functional assumptions of the component. While benefiting from a large body of existing literature, in reuse-oriented, component-based development with regards to the domain of real-time, embedded, concurrent systems where protocol-centered reuse and component-to-component negotiation and interaction patterns are explicit (i.e., multi-agent systems), these approaches possess several disadvantages:

1. Common interface-based specification and verification techniques are limited in their ability to deal with interaction constraints and nonuniform service availability. An interface alone defines syntactic conformance. Implementations must respond to the listed operations. An interface enriched with descriptions of the nature of operation arguments and results defines per-operational functional conformance. But even this does not always provide sufficient information for designers and implementers to ensure that components behave as desired. For example, a new interface-compatible (functionally compatible) component may not actually interoperate with an existing one if it fails to synchronize by sending the same kinds of messages and the implicit callbacks under the same temporal and causal constraints to other components in the course of providing services.

2. Existing approaches do not explicitly distinguish between basic and composite components. Not included in current certification models are the derivation of the satisfaction of component constraints by the specification of reused and provided components and determination of whether the constituent components actually play their intended roles (i.e., rationale and knowledge of purpose) to achieve overall functionality as well as non-functional goals. Rationale and knowledge of purpose are particularly important in the context of reuse and component engineering where results of such implicit design decisions help determine where components can appropriately and advantageously be used.

3. Component level reasoning in the context of reuse requires precise and clear understanding of interface level semantic dependencies such as the meaning of conformance between refinements of abstract components, conformance between abstract and concrete components, and fitness and suitability of the component under certification in its architectural context (Pintado, 1995).

1.4 BRINGING THE METHODS TOGETHER FOR COLLABORATIVE BEHAVIOR VERIFICATION USING UML-RT

In this section the proposed solution is illustrated in the context of a collaborative behavior design. The utility of the methods is discussed in the context of UML-Real Time (UML-RT), which is an extension of UML. Note, however, the methods are not specific to UML-RT. Section 1.4.1 briefly discusses UML-RT role modeling aspects. The approach is briefly discussed in the context of UML-RT in section 1.4.2.

1.4.1 Concepts of UML-RT

UML-RT places strong emphasis on UML collaboration concept, along with notions of *capsules*, *ports*, *connectors*, *protocols*, and *protocol roles*. Capsules are complex and potentially concurrent active architectural components. They interact with their surroundings through one or more signal-based boundary objects called ports. Collaboration diagrams describe the structural decomposition of a capsule class.

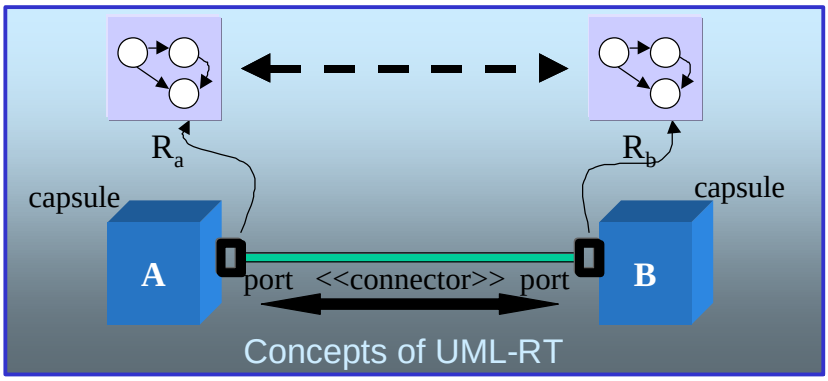


Figure 1-3: Components of UML-RT Role Models – The Local Perspective

Figure 1.3 illustrates the components of UML-RT role models. A *port* is a physical part of the implementation of a capsule that mediates the interaction of the capsule with the outside world—it is an object that implements a specific interface. Ports realize

protocols, which define the valid flow of information (signals) between connected ports of capsules. In a sense, a protocol captures the contractual obligations that exist between capsules. Because a protocol defines an abstract interface that is realized by a port, a protocol is highly reusable. *Connectors* capture the key *communication relationships* between capsules. Since they identify which capsules can affect each other through direct communication, these relationships have architectural significance. Collaboration diagrams, which capture architectural design patterns, are used to describe the structural decomposition of a capsule class. They use the primary modeling constructs of capsules, ports, and connectors to specify the structure of software components. Capsules use ports to export their interfaces to other capsules. The functionality of simple capsules is realized directly by finite state machines, whose transitions are triggered by the arrival of messages on the capsule's ports. Capsules themselves can be decomposed into internal networks of communicating sub-capsules. The state machine and network of hierarchically decomposed sub-capsules allow the structural and behavioral modeling of arbitrarily complex systems

1.4.2 Collaborative Behavior Verification with UML-RT

In this section discussions with regard to how the approach would facilitate verification of collaborative behavior under a parameterized collaboration pattern is provided in the context of UML-RT design representation constructs.

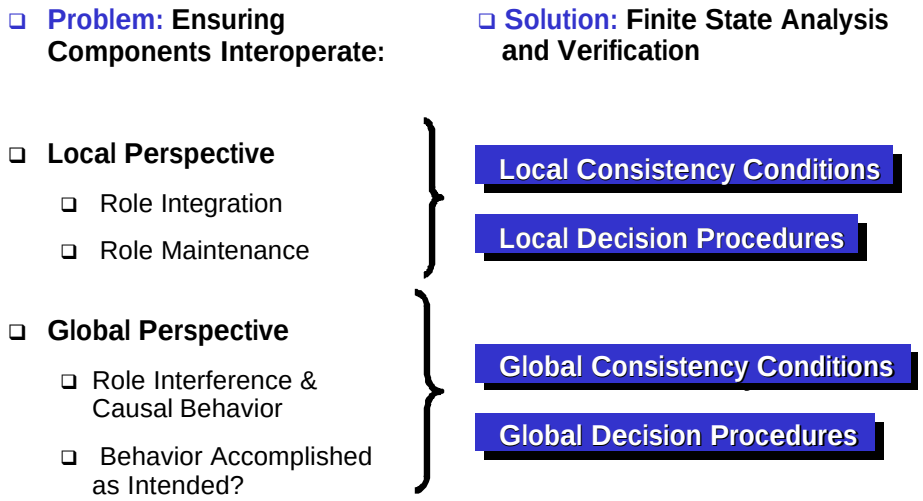


Figure 1-4: Problem –Solution Domain Mapping

Figure 1.4 depicts mapping the solution domain to the original problem domain discussed in section 1.2. More detailed outline of the consistency conditions along with the associated decision procedures is given below in figure 1.5.

Figure 1-5: List of Consistency Conditions and Inference Procedures

The safe role refinement, role compatibility, and protocol consistency analysis methods, which will be discussed next, represent the horizontal and vertical consistency analysis decision procedures for the local consistency analysis problem discussed in section 1.2.2. The substitutability principles and synchronization invariants utilize the notion of vertical consistency, whereas connector protocol consistency is derived using the horizontal consistency notion.

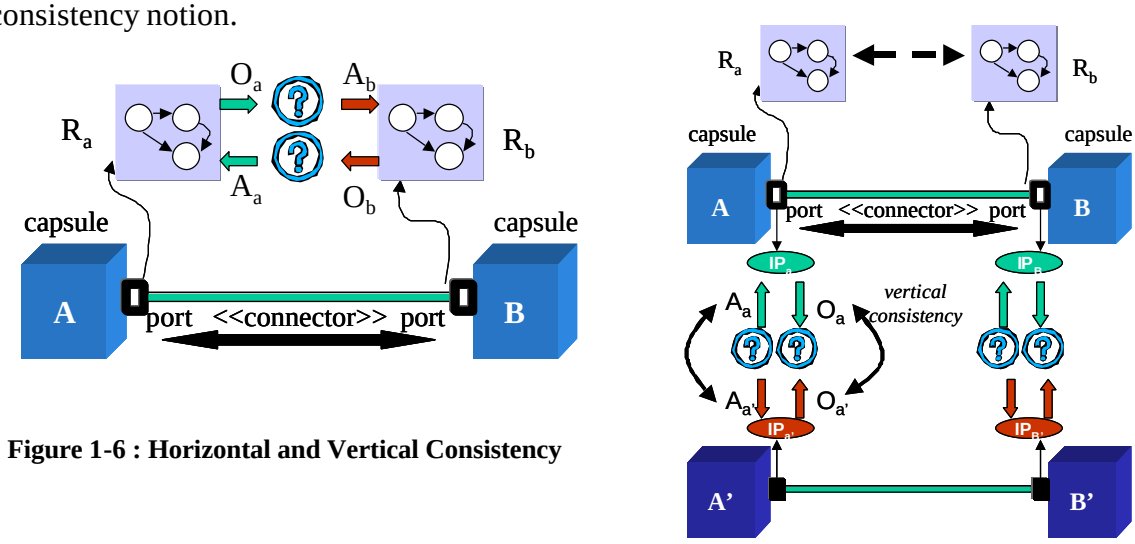


Figure 1-6 : Horizontal and Vertical Consistency

Figure 1.6 characterizes the horizontal and vertical consistency notions. In each design configuration A_i and O_i denote the assumptions and obligations of the capsule ports.

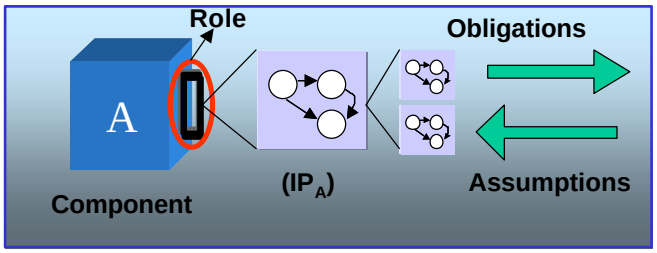


Figure 1-7 : Assumptions and Obligations of a Role

As depicted in figure 1.7, the assumptions of a port characterize what the environment is supposed to expect and assume regarding the proper usage of the component, while obligations denote the guarantees that the component is supposed to deliver as long as the clients use the component in conformance with its assumptions. The interaction policy of a component, denoted as IP_A in figure 1.7, facilitates the derivation of the assumption and obligations. Given the formal models of assumptions and obligations, the horizontal consistency refers to the compatibility of assumptions and obligations of a port with respect to the obligations and assumptions of its peer capsule port, respectively, as shown in figure 1.6.

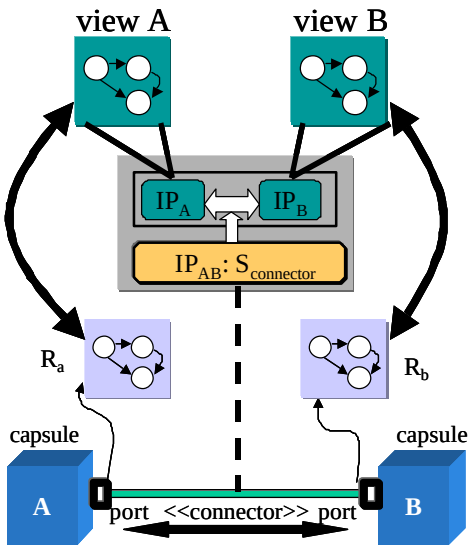


Figure 1-8: Local Connector Consistency

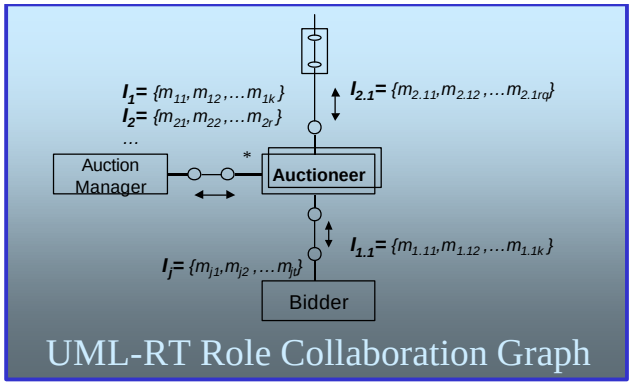


Figure 1-9: Achieving Global Consistency - *Global consistency* is checked using causal process constraints to determine if the collective behavior due to interaction and interference of ports are achieved/evolved as intended. By means of the introduced ***causal process constraint analysis*** method, the occurrence of expected collaborative behavior and constraints on them is verified.

Safe refinement on the other hand captures the consistency of the original assumption and obligations to the refined assumption and obligation, respectively, to assure that the refined component includes all the behavior of the original role.

It is not always practical to expect reused components to be consistent with the context they are embedded. In such cases designer develops a connector component that facilitates the mediation of interactions between incompatible ports. Figure 1.8 denotes a connector projected onto the viewpoints of the ports that it associates. Connector consistency entails (1) the projection of the connector protocol onto its assumptions and obligations with respect to each port and (2) horizontal consistency assessment between the projected protocol and each connected port. Figure 1.9, on the other hand, denotes the global perspective captured by the collaboration graph. Causal process constraint analysis utilize UML collaboration graphs to derive expected overall causal behavior based on the partial ordering of interactions and checks if the constraints on them are satisfied by the observed behavior.

1.5 CONTRIBUTIONS

The main contribution of this dissertation is the development of a framework and associated suite of methods developed under a reference prototype that enable component engineers of reusable components to reason about interaction assumptions of components under a collaboration-based development supported by design languages and methodologies such as UML-RT, Catalysis. In particular, the framework provides mechanisms for determining the fitness of the components with respect to the published architectural/interaction design dependencies and causal integration dependencies reflecting the internal component causal process constraints.

The model provides a certification framework based on the following principles: (1) interaction policies and coordination protocols need to be enforced, (2) the enforcement needs to be decentralized, (3) coordination protocols, causal processes, and associated constraints need to be formulated explicitly, rather than being implicit in the code of the components involved, and (4) policies and protocols should be refined and deployed incrementally, in an evolutionary manner. Utilizing augmented *parameterized UML collaboration graphs*, the certification framework is founded on a role-interaction

abstraction refinement model. It aims to facilitate (1) reasoning about the rules of engagement based on abstract interaction assumptions/obligations of component role specifications, (2) validation of interaction policies by detecting interaction violations, and (3) formulation of constraints and a means of testing for deviations from intended causal processes. To achieve that objective we

- developed reasoning methods founded on a reuse-oriented model of certification to facilitate prediction and analysis of the interaction behavior of component interfaces at different levels of abstraction, given the interaction models of its composites and architectural context,
- developed representation and reasoning methods and operators for capturing the internal collaboration constraints and knowledge of purpose (i.e., component internal constraints),
- embodied the techniques developed during the research in an evolving reference prototype "Verification Engineer's Assistant Tool" (VEAT). VEAT helps designers and verification engineers reason about refinement and semantical correctness of interconnections at an abstract interface level, and it provides methods for interface violation detection of the architectural assumptions at the concrete component level.

Being founded on a clear conceptual view of component engineering, the approach provides component engineers systematic guidance for tracking and maintaining semantic dependencies among components. The verification model is founded on precisely defined semantic design dependencies—such as *implements, extends, uses* relationships—among the interaction interfaces on which the component under certification depends. The interface model is represented in terms of *UML Statechart* and denote the externally visible synchronization behavior. The patterns of interactions are restricted to the languages that can be generated by finite state machines. The underlying component interaction and request scheduling semantics is synchronous. The necessary and sufficient conditions for preventing particular interaction policy related interface-level faults are identified: safe interface refinement, valid interface connections, interface violation detection. For the collaborative behavior under certification, the developed decision procedures detect the safe extension, composition, and violations of the synchronization contract assumptions and obligations. Verifying through these valid composition, extension, and interface violation detection procedures will not guarantee

that the future compositions will work; rather, it will guarantee that the aspects modeled in the point of view represented by that interaction-oriented interface model will be honored.

The certification strategy incorporates a method for detecting deviations from the expected internal causal interaction process. Such causal processes provide an account of how the desired goal of the component under certification is achieved; hence, they reflect the internal rationale of the component. Although most rationale research is devoted to recording the exploratory activities of the component design process, the research attempts to understand the causal dependencies among the roles of used components. The approach complements other approaches that deal particularly with the functional assumptions of the component. Components with per-functional conformance (functionally compatible) may not be protocol-compatible since they are not designed to interact under an expected synchronization requirement. Therefore, in order to achieve effective reuse, both expected functional properties and synchronization behavior should be certified.

1.6 ORGANIZATION

The dissertation is organized as follows: In chapter 2 current, state of the art methods of formal verification, testing methods for components, formalisms used in protocol-centered reuse, and component interaction dynamics are reviewed. Chapter 3 introduces the formalisms and principles underlying the certification model, as well as the identification of and specifications for consistency conditions. Chapter 3 also introduces a general component evolution model, which provides the context for the certification process. Chapter 4 describes the decision procedures used to determine the correct realization of local consistency conditions identified in chapter 3. Chapter 5 provides the mechanisms for interaction policy validation, along with methods to check the suitability of the causal event descriptions with respect to the derived causal behavioral model. Chapter 6 outlines the reference prototype and the design issues of the individual subsystems constituting the reference prototype. In chapter 6, three case studies are used to demonstrate the utility of the developed approach. Finally, chapter 7 concludes by summarizing the presented work and discussing contributions and future research.

CHAPTER 2

COMPONENT CERTIFICATION WITH REUSE

As witnessed by the vendor implementations of OMG Corba and other composition technologies such as OLE, ActiveX, Parts, VisualAge, and JavaBeans, component-based software is gaining momentum. Logically, therefore, the development of components or applications from pre-certified and plug-compatible components presents a key emerging market. However, this marketing makes sense only if developers can certify components to ensure customers that they will behave as expected, that they will perform as promised, and that they can be interconnected safely to existing components. The key issues and current trends in reusable component certification can be classified under four main dimensions:

(1) Investigation not merely of the **properties** of reusable components requiring certification (i.e., functional assumptions, configuration and structure, timing, space, performance, and security) but also of a **means for deciding** if the components possess the required properties.

(2) Investigation of certification **issues unique to reuse** and component-oriented development (i.e., COTS certification, domain analysis and validation, behavioral component relationships, safe extensibility, valid composability, component qualification, adaptation, structuring, assembly, and inferring system properties using component properties).

(3) Investigation of **economic considerations** such as span of application, criticality, marketability, and lifetime of components and their impact on certification (i.e., criticality and risk assessment methodologies for COTS software).

(4) Investigation of quality assessment methodologies and schemes for reusable component libraries to label a component to **characterize trustworthiness** and the degree of which certification has been performed (i.e., component dependability, leveled, ranked, and enumerative approaches).

The main thrust of the presented research fits into the first and second dimensions of the above characterization. As a complementary view to functional aspects, this research focuses on architectural assumptions or interaction constraints of the components. The intend is to develop a certification strategy founded on a conceptual view of component engineering in a development context both protocol-centered and reuse-oriented. Therefore, the two key facets to the approach are the reuse-oriented behavioral component relationships and the component protocols reflecting the interaction assumptions.

A central issue in the construction, maintenance, and reasoning of software components is the precise and explicit understanding of the interface-level relationships among components (i.e., specifications, templates, concrete components) involved in modern component-based systems [Gibson 1997], in which extension and composition concepts are fundamental and essential. Therefore, an effective reuse-oriented certification framework should be founded on a clear conceptual view of component engineering rather than ad hoc application or extension of conventional techniques. Recently, the research community has focused on investigating behavioral component relationships, and as a result of this work, general frameworks for understanding component systems have been proposed [Gibson 1997; Gibson and Weide 1997].

Devising interface definitions and attaching meanings to components are prerequisites for component-based development and for reasoning in general about component relationships. Interface declarations provide a basis for specifying capabilities in component-based systems at varying levels of precision and formality. An interface describes only those services that other components may depend on, in terms of a set of constraints on a family of components. Common interface-based specification techniques are limited in their ability to deal with dynamics, architecture, and interaction constraints. An interface alone defines syntactic conformance. Implementations must respond to the listed operations. An interface enriched with descriptions of the nature of operation arguments and results defines per-operational functional conformance. But even this knowledge does not always provide sufficient information to assure designers and implementers that components will behave as desired.

For example, a new interface-compatible or functional compatible component may not actually inter-operate with an existing one if in the course of providing services it fails to send to other components the same kinds of messages under the same temporal sequencing and causal constraints. There exist recent proposals that would alleviate this problem by augmenting interfaces with such protocol descriptions. These methods are mentioned in detail in Section 2.2.1. Efforts also exist to develop compiler and tool support for debugging object protocols [Butkevich *et al.* 2000].

The certification framework is founded on a reuse-oriented model and emphasizes temporal architectural interaction assumptions rather than functional assumptions. First, in Section 2.1, the existent, widely applied formal verification and testing methods for component-based and object-oriented software are reviewed. Then the current state of the practice and art of object interaction dynamics, as well as protocol representation and verification methods are examined. This examination is followed by the review of the efforts of the software community to alleviate the problems attendant upon designing and maintaining reusable components. In Section 2.2.1 and Section 2.2.2, the current state of component protocols and the behavioral component relationships on which the framework will be founded is examined.

2.1 FORMAL VERIFICATION AND TESTING APPROACHES FOR OBJECT-ORIENTED COMPONENTS

Formal verification and testing has been investigated for many years, a fact that has given rise to a large body of literature. Myers [1979], Wallace and Fujii [1989], and Beizer [1990] have produced some of the most influential publications on the topic of conventional software testing. Recently, the software testing community has tackled new sources of problems introduced by the OO paradigm without giving any consideration to the kinds of problems unique to reusability [Binder, 1994; Binder, 1995; Cheatham & Mellinger, 1990; Smith & Robson, 1992]. On the other hand, Hegazy [1989] has investigated some interesting testing issues related to reusable components such as controllability, observability, and genericity.

Due to increasing interest in this problem from both the research and commercial sectors, advances have been relatively rapid. A previous working group at WISR5 has

addressed certification issues largely from a theoretical perspective. A more recent working group at WISR8 focused on practical issues related to the mechanics of component certification. However, there remain a number of unresolved problems.

In the area of component certification, several groups are working concurrently, as evidenced by teams at each WISR since 1992 employed specifically or tangentially with research on the topic [Dunn & Knight, 1992; L. D. Systems-East, 1996]. The major reuse libraries have established policies and procedures for performing component certification or evaluation [C. IBM, 1992; I. Software Productivity Solutions 1995]. The purpose of such research and data collection is not merely to provide assurance that each component meets some standard of quality but also to inform potential users of the results of intensive evaluation performed using the specification for the component or using constraints identified by the evaluator.

Although formal verification and testing are essential elements in the success of component-based software engineering, advances in these areas have not been widely applied or adapted for unique problems of reusable component verification. The Reusable Software Research Group at Ohio State University is one of the few research groups which has done extensive work in the field of local certification and modular reasoning about the behavior of component software, as indicated by their web site (http://www.cis.ohio-state.edu/rsrg/bib-certify.html). In the next two subsections the advances in two areas related to object-oriented component-based software verification and testing are examined and reviewed: formal verification and object-oriented software testing.

2.1.1 Formal Verification of Component Software

Since component software involves a higher risk of poor verification than does conventional software, it is essential that a reusable, concrete component is proven completely dependable and accurate with respect to its published properties. Since testing itself is not capable of proving the absence of errors, it is necessary to support it with formal verification. Furthermore, as Weide and Hollingsworth [1992] state, local certification of components is a prerequisite for tractable reasoning and verification of large software systems built using these components. Formal reasoning is based on

comparing two formal objects, i.e., demonstrating the correctness of a program with respect to a specification, proving properties about a given system description. There are two general approaches for showing the correspondence of two formal objects: theorem proving and model checking. A generalization of model checking is also known as *model-based verification* and is discussed in Section 2.1.2.

Model checking, which works on finite transition systems, has proven to be a successful technology for verifying hardware [Clarke, Grumberg, & Long, 1994; Wing & Farahani, 1995]. It exploits the finiteness property of the hardware by performing an exhaustive analysis of the transition system's states. The model checking approach is also based on the assumption of finiteness of the state space of the software. Applying model checking to software may seem inappropriate at first glance due to the possibly infinite state space of software. However, based on abstractions of the state space of software, a model of the system rather than the system itself can be used, thus raising the level of abstraction at which the reasoning is performed. Therefore, while one can still fall short of exact reasoning about the original system, it is possible to perform approximate reasoning more quickly and with less effort.

The roots of formal verification based on theorem proving occur in the initial work of Floyd [1968], who first introduced the concept; Hoare [1969], and Dijkstra [1976]. Verification is based on demonstrating that given the truth of a predicate before a program node, when the node is executed, then the post-condition predicate would be true. Hoare [1969] extended Floyd's model by placing it within the formal context of predicate logic. He based this approach on a series of axioms describing the behavior of each language construct and proving, using mathematical logic, that the program has the desired pre- and post-conditions. Dijkstra [1976] developed a model similar to Hoare's axiomatic model, called *predicate transforms*, which computes the weakest preconditions and then demonstrates that they imply the given precondition of the specification. These approaches constitute the foundation of axiomatic approaches.

The use of data types, object-orientation, and modularization have led to another model called *algebraic* specifications. Specification and verification based on algebraic methods focus on the axioms defining the interactions of operators of the signature with

one another. Additional works based on the concept of an ADT can be found in Alphard [Shaw 1981], Modula-2 [Ernst *et al.* 1982], Ada/ANNA [Luckham *et al.* 1987], and Larch/Ada [Guaspari et al 1990]. Along with composition rules and a context-induction "prover" that can demonstrate the relationship between component specifications at different levels of abstraction, a pure algebraic approach to the specification of reusable components is shown to be useful in retrieving and identifying appropriate implementations for reuse [Cramer, Doberkat, & Goedicke, 1994]. Functional correctness and verification based on operational semantics have also received attention from the formal methods community. In functional correctness a program is considered as a function from some input domain to some output domain. Assuming that it is also possible to represent the specification as a function, verification process have to show that they are equivalent functions. The operational model is based on using some more abstract interpreter and showing that the program and the abstracted program have equivalent properties. Vienna Definition Language (VDL) and denotational semantics are two examples based on the concept of operational semantics. Krone [1988] provides verification rules using the proof system of RESOLVE [Hegazy 1989].

The object-based concurrent systems community reveals the development of a trend toward providing useful perspectives for object system verification. Most of the approaches are based on some form of process calculi (i.e., CCS calculus, pi calculus). Other approaches rely on trace semantics and actor systems. The verification work involving specific OO languages such as POOL [America, 1989] and FOOPS [Goguen & Meseguer, 1986] are based on Hoare-style proof techniques and algebraic techniques, respectively. Another study of object-oriented systems verification is that of Fiadeiro and Maribaum [1994], which uses temporal logic to prove that the properties to be verified are logical consequences of the axioms of the specification.

2.1.2 Model-Based Verification

Formal methods in component certification focus on specification and verification of a components using formal languages and associated proof systems. In contrast, recent efforts have attempted to create a model-based verification process—illustrated in Figure

2-1—founded on selective and pragmatic application of formal methods, using simplified models that focus on error detection rather than formalized proofs [Gluch & Weinstock , 1998].

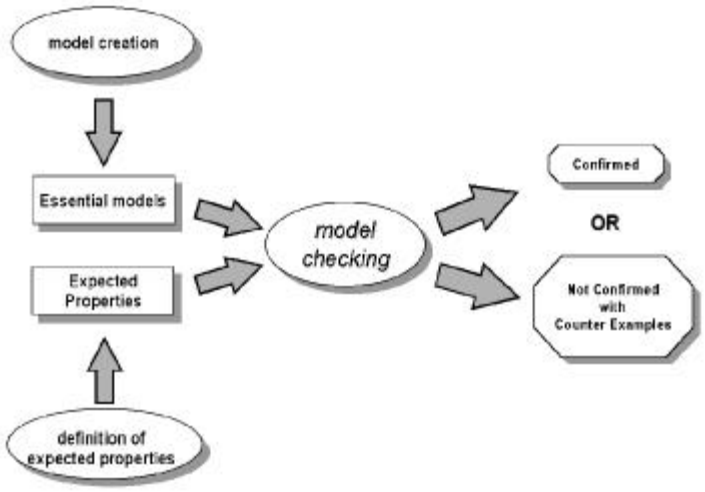


Figure 2-1: Model-Based Verification

By deriving test suites and new analysis methods based on partial models that capture the essence of the component under certification, model-based verification typifies a lightweight approach to the usage of formal and semi-formal methods. The models evolve during the component development, ultimately leading to the final software artifact. The method used for describing the models and for formalizing the evolutionary steps provides a basis for verifying the transformational accuracy of the overall development process.

In model-based verification, a well-founded formalism of software engineering models is chosen to capture the essential properties and behavior of the system. The formalism should have a well-defined language and methodology that includes mathematical and logical concepts permitting deduction and reasoning. The formal or semiformal models are then used to reason about the behavior of the component at higher levels of abstraction and to provide a basis for test suite generation.

2.1.3 Object-Oriented Component Testing

In this section the object-oriented software testing literature and those techniques, which are applicable to testing of components, are reviewed. There are several books written for

object-oriented analysis, design, and software development in general. The perception about object-oriented testing can be classified as follows:

- Testing is not discussed [Martin and Odell 1992; Shlaer and Mellor 1992].
- There is no need for a new approach; testing is not affected by the object-oriented paradigm [Booch 1991].
- Object-orientation makes testing easier due to the new principles and concepts which the paradigm introduces [Rumbaugh *et al.*1991; Wirfs-Brock *et al.*1990].
- The object-oriented concepts such as encapsulation, inheritance, and polymorphism make object-oriented testing different and imply new ways of testing [Berard 1990].
- Object-oriented development strategies are significantly different than traditional strategies; thus there is a need for new strategies for testing. Moreover, due to the incremental and iterative nature of object-oriented development, unit and integration strategies must be redefined and testing must be a continuous activity with particular emphasis on regression testing [Booch 1994; Jacobson *et al.* 1992].

Three different approaches to object-oriented testing are observed. these approaches are categorized by the degree and manner they incorporate other paradigms.

The conventional approach: In this approach existing well-known techniques are used in testing without any adaptation or extension for object-oriented paradigm. In particular, these techniques are applicable at the method level testing. That is, the methods of objects are actually functions and procedures which use imperative language constructs. Thus, conventional techniques are applicable for method level testing.

The adaptive approach: There is vast amount of research conducted in procedural software testing. The useful techniques borrowed from earlier research can be used in a new way by adapting or extending to include object-orientation.

The specific approach: In this approach new techniques, which are based on object-oriented formalisms, are used to develop verification and validation strategies.

The differences between object-oriented and procedural software are discussed in [Binder 1994]. The differences between object-oriented and procedural software are discussed in Binder [1994]. There, techniques are discussed that investigate the feasibility of adapting conventional testing techniques to alleviate the differences between these two types of software so they can be used within object-oriented testing. One of the first works on considering an object as the unit of testing and proposing adaptations of conventional techniques is produced by Fiedler [1989].

Perry and Kaiser [1990] discuss the theoretical view of object-oriented testing by describing the applicability of Elaine Weyuker's test adequacy criteria. Although the criteria were originally designed for procedural software, they are useful enough to determine an adequate level of testing for object-oriented software. The authors list each of the criteria and discuss the applicability to object-oriented paradigm.

Another interesting approach to testing is the adaptation of aliasing analysis techniques to uncover errors. D'Souza and LeBlanc [1994] examine the use of pointer aliasing as a testing technique. They summarize and evaluate the existing techniques and then show that using pointer aliasing information can be as powerful in its ability to detect errors as other techniques. The basic idea involves traversing the run-time structure obtained by the test case execution, which is stored in a table of pathnames. Each entry in the table is composed of a pathname, an object identifier, and the dynamic type of the object associated with that pathname. Then the table is sorted with object identifiers. Finally, the table is scanned for duplicate object identifiers, which due to sorting appear adjacent to each other. The duplicates indicate that the corresponding pathnames alias the same object, which can be an anomaly.

One of the well known testing techniques in the context of procedural paradigm is data flow testing. Although conventional data flow testing has been successfully applied to testing of programs developed using the procedural paradigm, it is not powerful enough to be used for object-oriented paradigm [Parrish *et al.* 1993]. However, there are

new adapted data flow techniques that are appropriate for object-oriented paradigm. Parrish *et al.* [1993] argue against the use of only specification-based techniques. They propose a theory for the adaptation of conventional flow graph-based strategies to testing object-oriented programs. They develop a formal framework to model classes with flow graphs. Then analogous definitions of node coverage, branch coverage, path coverage, definition coverage, use coverage, and du-path coverage are defined on the class flow graph. Each node in the graph refers to a method of the class. The edges between nodes represent the feasible sequence of method invocations which can be derived from the model-based specification of the class.

Hierarchical data flow testing technique, which is an extension of the existing data flow techniques, is proposed by Subramanian, Tsai, and Kirani [1994]. The authors decompose an object-oriented program into three different hierarchies: classes, objects, and attributes. They describe class flows, object flows, attribute flows and then define the interrelationships among them. Definitions such as “defined,” “referenced,” and “killed,” which are used in traditional data flow testing, are extended for class, object and attribute flows. The analysis is based on an algorithm with two phases. In the first phase class flow, object flow, and attribute flow are performed; in the second, any ambiguities raised during the class and object flow stages are resolved. As do the authors of previous approaches, Subramanian, et al., show the usefulness of the technique for method sequence anomalies. One of the most important contributions of this approach is its shifting of focus from variables to objects and classes; thus anomalies at this level of abstraction can also be revealed.

In Chen and Low [1995] an extension of traditional data flow analysis is proposed. The technique promoted by the authors uses the memory location of variables and class members rather than the improper sequence of actions on a variable. This procedure allows for the detection of anomalies based on pointer variables and class operators.

Another technique for adapting data flow testing to object-oriented paradigm is discussed in Harrold and Rothermel [1994]. They argue that most of the existing approaches are based on specification-based testing techniques and that there is a necessity for augmenting these approaches with code-based or white box techniques.

Their technique defines three levels involved in performing data flow class testing: (1) *Intra-method testing*, which performs data flow testing for individual modules; (2)*Inter-method testing*, which tests the interaction of methods inside a class; and (3) *Intra-class testing*, which tests the sequence of calls that can be initiated by the user of the class..

Most of the approaches suggested for object-oriented testing do not mention the domain structure and rules for which the model is developed. However, the structure of the domain and the knowledge structure represented by the relationships among objects also require verification. For instance, Addy [1997] discusses the importance of verification and validation in the domain engineering phase of reuse-based software engineering. Kandelin and O'Leary [1995] extend their previous research on rule-based system verification to develop verification approaches for object-oriented AI models. Their approach uses meta-knowledge from the domain to examine the completeness, consistency, correctness, and redundancy of the object model.

By identifying faults specific to the object-oriented paradigm, one can develop testing strategies either by adapting conventional techniques or by developing pure object-oriented approaches. Hayes [1994] concentrates on the necessity of developing fault taxonomy to guide testing. She examines several fault types and associated verification techniques that are mentioned in [Firesmith 1993] and [Purchase and Winder 1991]. She argues that in order to adapt conventional techniques it is necessary to understand the specific components of an object-oriented system. She identifies five components: (1) objects, (2) associated messages and methods, (3) class hierarchies (inheritance, aggregation, association relationships), (4) external interfaces, which are connections to databases, users, communication channels etc., and (5) tools and utilities. The tools and utilities are regarded as highly usable components. She argues that these components require a certification procedure since they will be reused in a variety of systems without customization. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a model.

Firesmith [1993] analyzes the problem of testing and recommends a hierarchical testing strategy based on the existing techniques. He identifies both the differences

between the object-oriented and procedural paradigm and their implications for testing. Based on the levels of testing (i.e., object, class, subassembly, message, exception, scenario) a taxonomy of errors and their priority for testing are presented. Then he suggests a verification technique for each error type listed in the taxonomy. The techniques proposed are conventional black-box testing, white-box testing and inspections to detect these specific errors. However, Firesmith does not explain how to apply the techniques to uncover errors.

There are other approaches that focus on test case generation and testing using different views of the system: object model, dynamic model, and functional model. Usually these techniques are influenced by the Object Modeling Technique (OMT) discussed in [Rumbaugh *et al.* 1991] for the design and modeling of object-oriented systems. Thus the test cases are prepared based on three different models: (1) Object Model, (2) Functional Model, and (3) Dynamic Model.

Poston [1994] discusses a project in which automated testing tools are integrated with the OMT life cycle. The object model identifies the classes, attributes and their relationships (i.e., inheritance, association, aggregation). Although the object model incorporates name, type, and initial values of attributes, it does not include domain descriptions. The testers add textual annotations to the object model to make it test-ready. The dynamic model introduces graphical notations for the specification of events, states, transitions, and hierarchies of states. However, creating a test-ready dynamic model is hard and time consuming. After creating test-ready models, a test case generator must be used to develop cases based on such design techniques such as boundary value analysis, equivalence class testing, cause-effect graphing, and event-based testing.

Kung, Hsia, Toyoshima, Chen, Kim, and Song [1995b] developed a methodology that uses a test model composed of three views of the system. Those views are represented by three types of diagrams: 1) the object relation diagram (ORD), 2) the block branch diagram (BBD), and 3) the object state diagram (OSD). Those diagrams are extracted from the source code using a reverse engineering technique (Kung, Gao, Hsia, Toyoshima, and Chen, 1993). The ORD diagram represents aggregation, association, and inheritance relationships. The BBD deals with those aspects of the system concerned with

the control structure, transformation of values, and functional dependencies. The OSD considers the dynamic model of the component [Kung, Suchak, Gao, Hsia, Toyoshima, and Chen, 1994]. The usefulness of the test model is summarized in Kung, *et al.* [1995b] as follows:

- The tester and maintainer can understand the structure and relationships among the components of an object-oriented program.
- The model allows the tester and maintainer to find strategies based on the test model to reduce testing effort.
- The model facilitates the definition and analysis of test criteria..

Another approach advocated in Software Architects' Synthesis model (SASY) by McGregor and Korson [1994] is part of a comprehensive testing process utilizing the iterative OO development approach. The authors represent SASY as a tool for structuring the development process and for allowing the integration of the testing process during each iteration stage. Three main categories of testing are identified: testing the development process, testing analysis and design models, and testing implementation. McGregor and Korson focus on and suggest strategies for the last two categories, namely, testing the implementation and models developed during the analysis and design phases. The system model testing process is based on the completeness, consistency, and correctness criteria. Use-cases are identified to develop interaction models, which are checked for errors, ambiguities and omissions. Class-Responsibility-Collaboration (CRC) cards are prepared for modeling individual classes during the specification phase. Then several checks are performed on these CRC cards to demonstrate their correctness and consistency with respect to the object model.

The testing proceeds by using the models of OMT; that is, object model, dynamic model, and functional model. During this phase not only the accuracy of the transformation from the analysis model to design model must be verified but also the consistency of these different views must be considered. For the system model testing

process the authors propose guidelines for testing the three different views of the system: object model, functional model, and dynamic model.

The testing process usually involves the generation of cases which have input and expected output pairs, such that the program processes the input and then compares the actual output to the expected output (i.e., oracle) in order to determine whether the test case is successful or not. Object-orientation represents a significant departure from conventional process-oriented models in which there is a clear division between process and data. In the process-oriented model the data is static, and the domain-dependent semantics are represented by complex processes reflected by large procedures. By contrast, object-oriented modeling encapsulates both the data and its domain-dependent semantics. Smith and Robson [1990] also argue this modeling difference in order to demonstrate that the testing process defined above is not suitable for object-oriented paradigm.

Algebraic-Specification-Based Testing approaches use algebraic specifications to object-oriented testing. One of the earliest systems to address the question of testing data abstractions is DAISTS (Data Abstraction Implementation Specification and Test System) [Gannon *et al.* 1981]. In DAISTS the axioms of an algebraic specification are used as an oracle for the testing implementations of the Abstract Data Type (ADT). A test case consists of a set of arguments for the left-hand side of the axiom. The test case is given as input to the left-hand side and right-hand side of an axiom and then the results are checked by a user-supplied equality (EQN) function.

Doong and Frankl [1994] describe an approach, called A Set of Tools for Object-Oriented Testing (ASTOOT) that is based on the algebraic specification of ADTs. ASTOOT consists of three components: driver generator, compiler, and simplifier. Under this system, the user utilizes an interactive test generation tool (based on the compiler and simplifier) to create a sequence of operations from algebraic specifications. The simplifier then uses rewrite rules provided by the compiler to produce two equivalent sequences of operations that form a test case. Then diver generator executes the test cases.

A more recent approach proposed by Hughes and Stotts [1996], called DAISTISH, involves a perl script which uses the formal specification of an ADT; it combines this specification with the actual code and then creates a test driver. DAISTISH is similar to DAISTS; however, DAISTISH is more effective in creating test drivers for languages that use side effects to implement ADT. This approach is also similar to ASTOOT; however, the authors concentrate on the semantic-specific issues of correctly duplicating the objects, rather than on automating the test case selection problem. The differences between these tools are summarized in Hughes and Stotts [1996].

Also worth mentioning are the retesting and class firewall concepts developed by Kung *et al.* [1995a, b]. These programs undergo continual changes. By the use of functional decomposition principles, the procedural paradigm provides a well-defined tractable structure. Hence the effects of a change can be traced easily. However, in the case of object-oriented software, the effects must be propagated to all the objects with relationships to the object under change. The Object Relation Diagram (ORD) and Block Branch Diagram (BBD) are used for identifying the regions of the code in need of retesting. The details of constructing the class firewall, its use in retesting, and an algorithm for creating an optimal test order for the firewall are described in Kung, Gao, Hsia, Lin, and Toyoshima [1995a].

Reuse is one of the key issues in object-oriented development. Classes, components, and highly reusable model libraries can be reused for different problem domains without wasting the effort for redeveloping the same functionality. However, since these components will be reused in a variety of systems without customization, they require a certification procedure. The methodology developed by Wohlin and Runeson [1994] provides a useful technique, based on usage modeling, for certifying components of a system. The objective of their method is to obtain reliability measure for the components. During development for reuse, they suggest that components can be certified with usage testing by developing usage profiles and then applying a certification model that provides the level of confidence in them. They generate test cases by using a hierarchical usage model by a state hierarchy model which uses Markov chains to model the state transitions. They attach probabilities to usage and behavior transitions and then

use these measures to develop test sequences. The test sequences facilitate the collection of failure data and provide a basis for the certification measure.

Harrold, McGregor, and Fitzpatrick [1992] discuss an incremental approach for the testing of class hierarchies. They suggest an order for testing the inheritance hierarchy that starts from the top. That is, testing starts with the base classes and continues down the hierarchy. The testing process progresses by testing features in isolation and then testing the interactions among features. The authors' algorithm uses test histories from the parent classes, which are incrementally updated with information about the derived class's differences from the parent class. Each new test history allows the tester to differentiate the reusable test cases from the parent class, along with any attributes for which new test cases must be generated. The authors' approach supports three levels of attribute visibility: hidden, accessible to derived classes, and accessible to all classes. They model the inheritance concept in terms of a class and a modifier, resulting a derived class..

Chung and Lee [1992] offer another study related to inheritance. The authors show that when repeated inheritance occurs errors may arise as a result of name-confliction. The authors build a directed graph of the inheritance structure where nodes denote the classes and edges denote the inheritance. The root classes are traversed using breadth-first traversal, and then parent classes are added to the ancestor sets of the child classes. The ancestor sets are examined to find the repeated inheritance structures, which are then classified hierarchically based on their euler region numbers (r). Then these hierarchies are used in testing as follows: URI(1): every class in a repeated inheritance must be tested at least once; URI(2): every repeated inheritance with r=2 must be exercised; and URI(3): every closed region with r=3 must be exercised. The authors propose that after such hierarchies are discovered, McCabe's cyclomatic testing strategy should then be applied to them.

The complexity of the object-oriented paradigm arises from the dynamic and diverse patterns of interactions and collaborations among the objects. Testing individual methods and classes is not enough; integration of class methods (i.e., intra class integration testing) and of methods of different classes via association relationships (i.e.,

inter-class integration testing) are also necessary. Unlike that of procedural and functional models, object-oriented models do not possess an easily tractable system decomposition structure. Thus, top-down and bottom-up integration strategies are not very useful. Inheritance and aggregation relationships and dynamic binding thus pose challenging integration-level problems for the object-oriented paradigm.

Jorgensen and Erickson [1994] discuss this problem of structure and behavior in the case of object-oriented software. They argue that the event-driven nature of object-oriented software forces a declarative spirit on testing. The two constructs proposed for the integration level of testing are as follows:

Method/Message Path (MM-Path): a sequence of member function invocations which starts with a method and ends when a method which does not issue any messages is reached.

Atomic System Function (ASF): an input event followed by a MM-Path, which ends with an output port event.

These constructs provide a framework for integration testing. However, as the systems become larger and more complex it becomes necessary to formally specify and capture the object interactions through MM-Paths.

Overbeck [1994] provides a detailed integration level testing strategy for object-oriented systems. He focuses on developing a testing infrastructure based on the relationships between classes with different inheritance schemes. In this approach each class being tested passes the self test (class unit test) if for all methods in the class message input/output is correct, all intra-class method sequences conform to the expected sequence, and every server class of the one under test is used appropriately. Once the class passes the self test, the strategy focuses on the client/server relationships (collaborations and interactions) among the classes (i.e., contract-test). To clarify, consider two classes—A and B—where A uses a service provided by a method of B, the class A passes the test if it passes the self test (class unit test), it meets the preconditions of B, and A uses B's output correctly.

McGregor and Korson [1994] also provide an integration approach in their testing process model. They identify two levels of interactions: those among the methods of a

class and those among the methods of different classes due to associational relationships. They suggest investigating a pair of methods that directly or indirectly affect each other to produce incorrect results. The strategy proceeds by constructing test cases that identify attributes modified or accessed by two or more methods. Typically, each pair of such methods should be executed. However, in the cases of polymorphism and dynamic binding, the definite inter-class relations cannot be identified. The authors discuss applying orthogonal array-based techniques to utilize possible interactions and create corresponding test cases. However, as the order of interactions increases, the management of the technique becomes difficult.

Graham, Drakeford, and Turner [1993] suggest using black-box and white-box testing for individual methods. They recommend an incremental strategy similar to the approach discussed in Harrold, *et al.* [1992] for reducing the number of test cases that need to be re-run during the testing of inheritance structure. For object integration testing, an object instantiation tree that shows which objects can be instantiated from which others allows the tester to provide a top-down testing order. This strategy creates a plan for the integration of program units.

One of the problems of object-oriented testing is the state dependent behavior of objects [Binder 1995]. A state-based approach is given in [McGregor and Dyer 1993]. The test strategy relies on the development of a state model of a class from OOA/D specification or by examination of the implementation of the class under test. Each class keeps an invariant and each test case checks whether the current state complies with the invariant. They argue that due to polymorphism and dynamic binding, possible number of client objects and polymorphic servers can be large and they propose the use of orthogonal arrays to facilitate selection. Turner and Robson [1993b] propose a new technique, called state-based testing, for the validation of object-oriented systems. They mention that due to the interactions between the features and the state of the object, new techniques are necessary. Their approach defines the state of an object as the combined values from all of its data members. Once the states are identified, the methods and the input and output states are examined. That is, the input states for which the method is applicable and the output states which result by the application of the method are

identified. Then test cases are generated to create the initial condition (starting state for a method) along with the code which validates the starting state. After this, the method is applied and the resulting state is compared to the expected final state.

2.2 PROTOCOL-ORIENTED PERSPECTIVE FOR ACTIVE OBJECT ORIENTED COMPONENT SPECIFICATION and CERTIFICATION

Thus far, the reviewed literature in the context of formal methods, model-based verification, and testing ideals mainly with the functional assumptions of components. In this section the focus is on existing work involving a particular type of non-functional component characteristic: the interaction dynamics perspective. Section 2.2.1 reviews existent component protocol specification mechanisms, along with a variety of object interaction semantics utilized by these methods. Section 2.2.2 reviews the object interaction semantics in more detail from a synchronous interaction point of view.

2.2.1 Component Protocol Specifications

Adding protocols not only provides additional constraints on interface descriptions but also helps in the task of composing software, provides a basis for component engineers to use tools to test component compliance with constraints, and assists users of the components by formally documenting the usage assumptions.

There have been several proposals for augmenting interface descriptions with protocols that represent the sequencing constraints. Campbell and Habermann's path expressions [1974] is one of the earliest methods to explore sequencing constraints on method invocations. Nierstrasz [1995] also incorporates protocols into object descriptions. Both approaches express server protocols (receives); neither includes client protocols (import requirements or sends). Yellin and Strom [1997] extend protocol descriptions by explicitly representing both the message receives and sends, which are essential in reuse-oriented development. In the software architecture community Garlan (1995) and Luckham, Kenney, Augustin, Bryan, and Mann [1995] have worked on adding protocol constraints to component connections and components, respectively. The work on *Gluons* by Pintado [1995] on *protocol-centered reuse* and by Arapis [1995] on temporal specification object models focus on using protocols as glue to hold the

messages of an interface. The message sequence specification and testing approach used by Kirani and Tsai [1994] augments component interfaces with server protocols by regular grammars; the authors argue that this specification can be used at runtime for interface violation detections. However, they do not provide any means for achieving such a goal.

There exist many research and commercial prototype environments, as well as theoretical conceptualization efforts, that offer mechanisms for synchronous or asynchronous interaction styles. Those include:

- Ada Rendezvous: The Ada task select and accept statements provide representation specific explicit acceptance mechanisms for synchronous interaction.
- CSP (Concurrent Sequential Processes): CSP is a modeling language that has been proposed by Hoare. It is at the origin of Ada select statement, but after the definition of Ada, the select statement is revised to a symmetrical nature so that it is possible to both accept and send calls.
- Promela: Promela is a language for modeling of programs and verification of their temporal properties (i.e., absence of deadlock, correct sequence of actions). The language is supported by a tool, called SPIN, that check these properties. Promela offers statements similar to parallel wait (i.e., select statement); however, instead of method calls, it provides channels.
- sC++ (Synchronous C++) and sJava (Synchronous Java): sC++ adds to C++ the concept of active object that contains a separate thread of control. An active object communicates with its environment via synchronous messaging. Synchronous Java extends the java language with parallel wait and synchronization mechanisms to support synchronous interaction.
- Regis: The language is very similar to sC++ and is accompanied by an architecture editor utilizing an architectural description language (Darwin)
- SDL: SDL is a language adapted to protocols and is proposed by standards organizations ISO and ITU. Similar to Promela it uses channels for

communication. Several transitions from an automaton state can be executed depending on the events. This provides a construct similar to parallel wait.

- LOTOS: This language is also proposed to be an international standard. LOTOS defines multi-rendezvous.
- PROCOL: Procol (Protocol-constrained Concurrent Object Language) is a concurrent object-oriented language. PROCOL defines a protocol within each object definition. The protocol constrains sequences of message communication and through guards controls access to methods of the object.
- Software Adaptor: This work [Strom and Yellin 1997] concerns with how well the software components work together if connected. If two components match perfectly, they can simply be connected without any glue mechanism. Otherwise, a *software adaptor* is used as a bridge between components.

2.2.2 Object Interaction Dynamics

Based on the roles played by the participants of the interaction, object interaction patterns can be classified according to two broad categories: (1) *Producer-consumer*, in which interaction involves uni-directional communication; and (2) *Client-server*, which involves bi-directional, synchronized communication. Examples of producer-consumer interaction include selection of a menu option, the sending of a purchase order to a supplier, and a report generator sending a report to an output device. When client-server interaction is involved, the client requests a service and blocks until a reply is returned from the server. As opposed to the asynchronous messaging emphasized by the producer-consumer interaction style, the focus in this dissertation is on synchronous messaging among reusable components.

Figure 2.2 depicts the structure of an interactive transition, where the *guard* refers to the condition firing the transition, *action* refers to the terminating activity that can affect local attributes. The *events* are due to the results of the actions and are optional. The *reply event* in client-server interaction falls into this category.

In conjunction with the interaction mechanisms and request scheduling means illustrated in Figure 2.2, interaction aspects are usually classified as either *(1) Readiness* or *(2) Communication* mechanisms. Next, we describe these properties in detail and point to those properties which are investigated in detail.

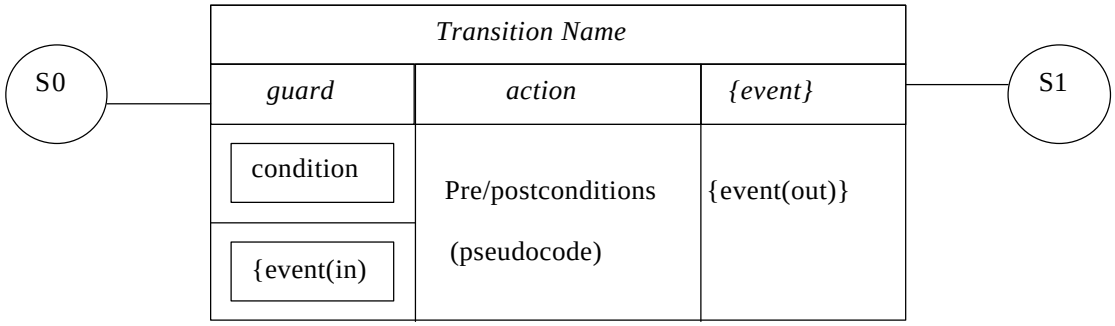


Figure 2-2 Object Interaction Dynamics

Readiness: A transition fires up when an event specified in its guard occurs along with the satisfied conditions. This raises the issue of what happens when an event is generated due to the transition and the recipient is not available to receive the interaction request. There are three main alternatives: (1) *Inaction:* the event is intentionally ignored by the recipient, (2) *Queuing:* the event is queued by the recipient and processed as soon as the object moves into a state in which the event can be processed. (3) *Exception:* the event causes an error condition.

Communication: There exists a variety of communication styles such as *asynchronous, synchronous,* and *preemption*. Asynchronous communication involves unblocked one way messaging which can be either unbuffered or buffered. Synchronous communication, on the other hand, requires the sender to block until the recipient finishes the transition.

The following categorization reflects the widely known and used primitive interaction protocols.

Synchronous Method Call: This occurs as a request/reply or client/server interaction mechanism between two objects. The reply is an implicit event representing the completion of the required service.

Callbacks: Instead of issuing acknowledgments, event consumers may generate events or callbacks that can be dispatched by the senders.

Forwarding: Where this is concerned, one object may incorporate a protocol as an event mediator accepting and sending requests on behalf of collaborators.

Multicast/Broadcast: In multicast/broadcast, an event producer may generate an event received by each member of a set of consumers interested in that particular event.

High-level interaction protocols can be defined in terms of primitive interactions and interaction protocols. Software components interact in a variety of ways as intended and expected by their designers. Such interaction requirements are fundamental aspects of a component that are supported by many design modeling languages and environments. Using state diagrams, man existing component design representations capture the interaction point of view of components via *dynamic models*. The functional and object models—which reflect the dataflow and data, respectively—are strongly supported in conventional programming systems. However, the dynamic model and interaction aspects require significant programmer involvement and are only supported by a small number of languages. Several concepts have been proposed to extend OO languages to concurrency and explicit interaction and request scheduling mechanisms. Examples include delayed evaluations (a concept in Actor-based languages), Ada tasks, synchronous channels and exceptions of Eiffel, and active objects of Synchronous C++ and Synchronous Java.

2.2.3 Finite State Verification

Finite state verification and testing are usually viewed as competing approaches for validating object interaction dynamics. In finite state verification, a finite model of the

system is first constructed, usually by abstracting irrelevant details, then the finite state verification methods explore the state space to determine whether a given property holds.

Conceptually, finite state verification views systems as a collection of states in which the system can be in at any time during the execution. The basic approaches used by finite state verification involve the generation of computational reachability graphs and the searching of those graphs to determine property compliance. A reachability graph represents all reachable states of the system to the extent that this system is modeled by the finite state verification approach of choice. Paths through the reachability graph denote the potential executions of the system. A path in the reachability graph is executable if it corresponds to a real execution of the system. All other states are considered to be spurious. If there is a path from the start state of the reachability graph to some state, *s*, such that the property is violated on this path, *s* is called a violation state. Many finite state verification approaches are capable of checking two general kinds of properties: safety and liveness.

2.3 COLLABORATION-BASED SOFTWARE VERIFICATION AND TESTING

A variety of research groups have recently investigated the compositional verification of collaboration designs. To prove certain behavioral properties about individual and composition of collaborations, some of these group—such as Fisler and Krishnamurthi [2001]—employ finite state verification. Others, such as Engels, Kuster, and Groenewegen [2001], use CSP-type process-oriented formalisms to analyze object-oriented behavioral models for certain consistency properties. The interface automata concept introduced by Alfaro and Henzinger [2001] investigates light-weight formalisms that capture the temporal aspects of software with an approach the authors call *optimistic view.* Additionally, Butkevich, et al. [2000] describe an extension of the Java programming language that supports static interaction conformance checking and dynamic debugging of object protocols. Their approach considers only the incoming requests and omits the output obligations of components, as well as global causal processes. Also, the UML model integration testing approach discussed by Hartman, Imoberdorf, and Meisinger [2001] illustrates the issues involved in collaboration testing.

CHAPTER 3

THE PRINCIPLES OF FORMALIZATION FOR CONSISTENCY CONDITION IDENTIFICATION and SPECIFICATION

The motivation for the proposed certification framework and the underlying foundation of this chapter derives from the experiences of programmers, designers, and architects—including me—of software components and systems. As outlined in the problem statement, software designers generally encounter two major problems when they attempt to reuse components built by other people: (1) the pieces do not fit with each other and (2) changing design and code often produce conflicts and inconsistencies with the rest of the pieces.

The first problem arises primarily due to the lack of formality and incomplete representation of the semantics of the interfaces, along with their syntax. While formalisms, such as model-based and algebraic methods, are widely advocated by academia, practitioners rarely use them due to the complexity of their underlying mathematical basis. Lack of clarity and certainty with regard to related decision problems also plays a role in their general rejection. The second problem emerges due to the inherent complexity of the software. Specifically, the cited problems are effects of the compound causes of *composition, evolution*, and *complexity.*

The key approach to the problem of composition involves interface models with enhanced syntactic interfaces, where semantic information expressing the intent of the interface provider facilitates correct and effective reuse. The problem of evolution is approached in this dissertation through constructive use of interfaces to determine and maintain *semantic dependencies* by utilizing *behavioral component relationships* as opposed to ad-hoc structural relationships, which fail to provide meaningful context-independent dependency information. Given that both interfaces and implementations

evolve during development, keeping track of how interfaces are used facilitates consistency and conflict mitigation and analysis. Enriching syntactic interfaces with semantic information and capturing semantic dependencies via behavioral component relationships are not new concepts. However, the lack of previous success in systematic and methodological reasoning about reused components begs the question*: how can they be made to work?* An important clue to that question stems from the issue of managing complexity. The approach used in this dissertation tends to view complexity from a different perspective, from within the context of interaction and role-based consideration of components.

The shift from algorithms and a data structures point of view to a components and interactions viewpoint suggests an entirely different perspective on complexity. The type of complexity that prevails in current research considers the problem of scale and the sheer wealth of small, individual component details, rather than the inherent complexity of individual details. The framework presented in this dissertation facilitates management of the mass of small-scale details: it uses interaction and collaboration points of view to focus on solving composition and evolution (both abstraction and implementation) problems by tracking semantic dependencies and the organizational rationale underlying the component under review.

3.1 CERTIFICATION UNDER A DEVELOPMENT WITH PROTOCOL-CENTERED REUSE PROCESS

Rather than being written from scratch, composite components are increasingly constructed from reusable components and coordination. This recent approach shifts development to a new life cycle model. In Section 3.1.1, the development-with-reuse lifecycle, as well as the different aspects involved in certification are briefly discussed. Section 3.1.2 focuses on protocol and collaboration perspectives and presents a reuse-oriented certification process utilizing UML collaboration graphs.

3.1.1 Development Process with Protocol-Centered Reuse

The first phase of this lifecycle involves *component qualification/acquisition*, where components that satisfy the composite component requirements are retrieved from

libraries and external vendors. Next, as depicted in figure 3.1, the *component assembly* stage facilitates the integration of these components into a cohesive new component. Continuous *component analysis* determines the compatibility of the components and determines if reused components fulfill their expected behavior and thus satisfy intended behavioral specifications. Modifications are made until the component is eventually judged acceptable. As the design and development proceeds, the *component adaptation* phase facilitates replacing existing components with new ones or wrapping original components in order to achieve the desired requirements.

During component development designers focus on a variety of views based on a set of aspects that addresses an architectural focus. Although the structural and behavioral views form the core of analysis and validation concerns, *protocol* and *quality-of-service* aspects are usually not at the center of these processes. This is particularly the case in component development with reuse, because components are not originally developed to participate in collaborations required from the context in which they are embedded.

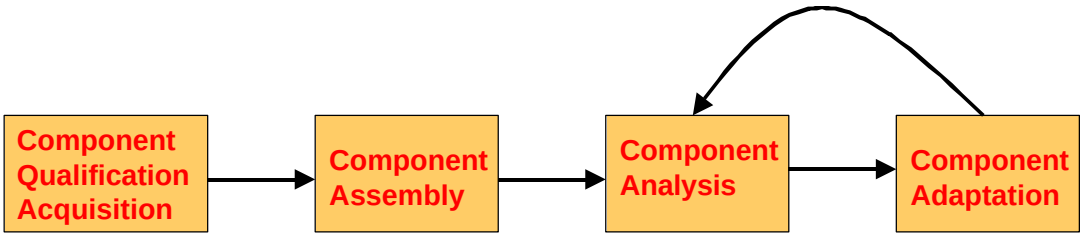


Figure 3.1: Development Phases (Partial) The component design stage refers to design and development of individual composite components that constitute the overall large scale system. The developed framework focuses on the certification process for synchronous parameterized collaboration-based design of these components that are required to conform to reused design patterns.

The entire process is recursive, focused on the collaborations between components or objects. The concepts of type models, collaborations, refinement, frameworks and patterns can be applied at all levels and across the entire process. Using external actors, a software system itself plays a role in business collaborations. It can be characterized separately by each collaboration in which it participates, and it is specified as a large type (i.e., role interaction policies). Its internal context is designed as a collaboration of finer-grained components.

3.1.2 Protocol-Centered Certification Perspective with Parameterized Collaboration Graphs

A variety of perspectives and aspects are involved in the development of a component. Protocols and collaboration patterns constitute two of the key perspectives. Likewise, *UML collaboration graphs* are widely used in software engineering practice to capture (1) the structural descriptions and associations of participants and (2) the description of their communication patterns.

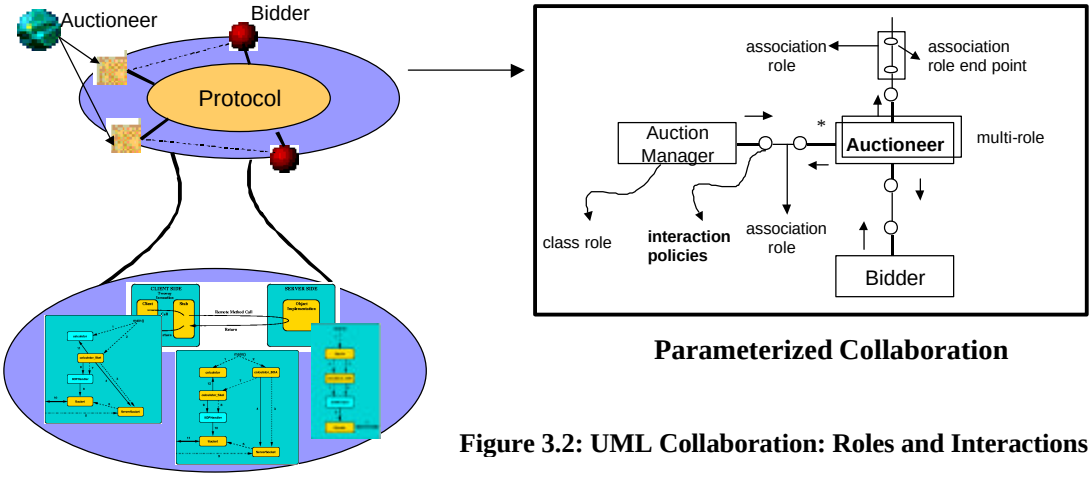


Figure 3.2: UML Collaboration: Roles and Interactions

Figure 3.2 depicts a collaboration that includes sets of both classifier and association roles that define the participants needed in the context of the component under development. In the same figure, the causal processes denoting the UML collaboration graph interaction sequences involved in the collaboration are depicted as the realization of the protocol. The OMG-adapted UML collaboration graph semantics is presented in Appendix A. However, in this section, the main elements of a collaboration are briefly overviewed:

(1) Classifier Role: A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required by the collaboration. A classifier role may be connected to a set of *Association Roles* via *Association End Roles*. In Figure 3.2, *auction manager*, *auctioneer*, and *bidder* act as participants in the collaboration.

(2) AssociationRole: An association role defines a restricted view of an association that mediates and links classifier roles. Association roles are compositions of a set of *AssociationEndRoles,* corresponding to association ends of the base association.

(3) AssociationEndRole: An association end role is part of an association role and specifies the connection of an association role to a classifier role.

(4) Collaboration: A collaboration defines a set of roles to be played by instances and links, as well as a set of interactions that define the communication between the instances when they play the roles.

(5) Interaction: An interaction specifies the communication between classifier and association roles. Interaction involves a set of messages defining the communication between a set of instances conforming to specific roles.

(6) Message: A message defines a particular communication between roles that potentially represent a set of instances. A message defines a particular message in an interaction. A communication can call an operation, receive a method call, or raise a signal, among other functions.

A **parameterized collaboration**, such as shown in Figure 3.2, represents a design construct that can be used repeatedly in different component designs. The participants of the collaboration, including classifier roles and association roles, form the parameters of the generic collaboration. The parameters are bound to particular model elements in each instance of the generic collaboration. Such a parameterized collaboration can capture the structure of a design pattern. Collaborations may be expressed at different levels of granularity, and generalizations can be formed about them. Hence, they can be refined and specialized as the development progresses.

Given the definitions of the constructs of parameterized collaboration graphs and the component-based development life cycle phases, the following algorithm depicts how the proposed certification methods interleave in a design situation involving parameterized collaboration graphs.

Protocol-Centered Certification with Parameterized Collaboration Graphs: Here, a certification process is outlined to focus on interaction policies of components with a reuse-oriented development perspective based on parameterized collaboration graphs.

Certification Process with Parameterized Collaboration Graphs

Input: PCG($p_1, p_2, \ldots, p_n$): Parameterized Collaboration Graph

(1) Let the parameterized collaboration graph be denoted by ***PCG($p_1, p_2, \ldots, p_n$)*** where $p_1, p_2, \ldots, p_n$ denote the participants of the collaboration.

(2) For each participant p_i, $1 \leq i \leq n$

   a. Find a component $C_{candidate} = \{R_1, R_2, \ldots, R_k\}$ where $R_1, R_2, \ldots, R_k$ are the roles that the component $C_{candidate}$ plays.

   b. Check if any R_j, $1 \leq j \leq k$ can substitute p_i.

   c. If R_j, $1 \leq j \leq k$ cannot substitute p_i and no other alternative exists

      i. augment R_j or develop a new role to obtain R_{new}.

      ii. check if R_{new} is compatible with the context of p_i (i.e., associated roles)

      iii. if R_{new} is not compatible with the context of p_i then

         1. develop an association role (adaptor) $AR(R_{new}, p_i)$.

         2. make sure association end roles are compatible with the participant roles.

(3) **Repeat**

   a. Refine the graph by extending classifier and association roles and make sure *refinements are safe*.

   b. Define or refine causal interaction processes and their constraints.

   Until the ***PCG($p_1, p_2, \ldots, p_n$)*** meet the contextual requirements.

(4) Implement/instantiate the collaboration.

(5) Validate that local role interaction policies are implemented correctly and that the observed causal processes do not deviate from the intended interaction process and comply with the constraints imposed upon the processes.

(6) Evolve/adapt the implementation to adjust to the contextual requirements. Make sure that the causal processes are evolved as intended conforming to the rationale of the component designer.

3.2 ROLE-BASED ANALYSIS WITH REUSE: AN INTERACTION-ORIENTED BASIS FOR THE CERTIFICATION MODEL

Traditionally, object-oriented modeling has structured component and system descriptions around programming language constructs such as classes and packages, but has not provided architectural interaction-based semantic constructs for components and their interconnections.

3.2.1 Role-Based Modeling for Compositional Reasoning

Role-based modeling facilitates the explicit representation of collaboration responsibilities. A *role* describes a collaboration participant fulfilled by a specific component when instantiated with a particular connector. Hence, a component can play one or more roles. For instance, an *event notification* protocol might comprise roles like *event publisher, event channel,* and *event push consumer*. *Role interfaces* model role behavior in terms of provided and required services.

Multiple role interfaces for a single component and directed usage of other roles' interfaces can be specified. *Interaction protocols* describe sequences of such requests among multiple roles. Interaction protocols refer to services of role interfaces, and they describe role interface usage for various interactions of the collaboration. For instance, in the *event notification protocol,* the interaction *connecting as Push Consumer* may require that an event consumer register itself to the component playing the role of *event channel* through the usage of *event channel* and *consumer admin* interfaces to obtain a *proxy push supplier* object reference. Other *event notification* interactions include *pushing event data* and *pulling event data* interaction sequences that may be expressed by informal UML collaboration graphs. Note that although such informal diagramming techniques help communicate design information, they lack the sophisticated formal basis for facilitating sound reasoning about the interaction behavior of components and their role interfaces.

The introduced framework aims to bridge this gap. Furthermore, reuse of a component in a new context with new roles and interaction protocols requires new assessment and certification viewpoints to assure that the component under certification delivers the required services as intended, without role interference and inconsistency.

3.2.2 Safe Evolution: Interaction Abstraction Refinement

Construction of composite components from reusable artifacts and components require precise and clear understanding of the design and integration dependencies among components. The role of behavioral component relationships is to describe the dependencies explicitly and provide information regarding how components should be used in conjunction with other components in a systematic way. Such design and integration dependencies evolve as component design progresses and the abstract representations are refined. However, the refinement process requires carefully designed systematic mechanisms assuring that original assumptions and obligations regarding integration and design dependencies are not violated.

In the context of reuse, the abstraction order and the correctness assessment of forward transformations require careful analysis of the dependencies between concrete and abstract components. Gibson [1997] and Gibson and Wade [1997] discuss such a general framework to describe the formal characterization of behavioral relationships between artifacts involved in modern component-oriented development. Their framework is general and does not constrain the behavioral space of the components. Similarly, Cramer, et al. [1994] mentions how a formally-defined implements relation can verify components using the algebraic specification approach.

An *Abstraction Refinement Model* similar to the model discussed in [Nance *et al.* 1989] is adapted to capture (1) the evolution aspect of the abstract integration and design dependency models and (2) each refinement step in facilitating systematic reasoning for the interaction assumptions of the component under certification. The design dependencies refer to constraints involved when the component under certification is defined in terms of other components, whereas in the context of the component under certification, the integration and coupling of the used components constitute the integration dependencies. The certification model involves three main elements:

- Component Role Descriptions: These descriptions range from abstract models to concrete realizations (i.e., implementations) of the component under certification and the associated dependent components.

- Abstraction Order: A pre-order that relates the component descriptions by a relative-correctness order. It facilitates the expression of refinement relationships. For example, if *p* and *q* are component descriptions and if $p \acute{I} q$, then *p* can be substituted for *q* and honors all the constraints and obligations imposed by *q*. Interpretation of the safe refinement, as described here, varies depending on the formalisms used, as well as the conditions necessary and sufficient for relative correctness.
- Forward Transformations: Transformations of the system descriptions constitute the heart of mapping abstract representations to reach concrete realizations. The correctness of these transformations is evaluated using certification methods associated with the formalisms and abstractions used for the component descriptions.

The structure of the certification model not only characterizes evolutionary development but also provides a context for the process of systematic certification. The sequence of transformations from the most abstract components to the concrete realizations constitutes a chain. The certification model ensures that each link in the chain respects the contractual design and integration dependency assumptions while adding more details, as well as that the actual concrete realization implements the abstract contracts and constraints expected from the component. Three such relationships, illustrated in Figure 3.3, facilitate tracking and maintaining semantic dependencies among the collaboration participants in the context of the certification process. The key terms for those relationships are.

- *Implements:* A concrete component *C* implements abstract component *A* if *C* exhibits the behavior specified by *A.*
- *Uses:* A component *X* uses a component *Y* if and only if the behavior of *X* depends on the behavior of *Y.*
- *Extends:* A component *X* extends component *Y* if and only if all of the interface and behavior of Y is included in the interface and behavior of *X*.

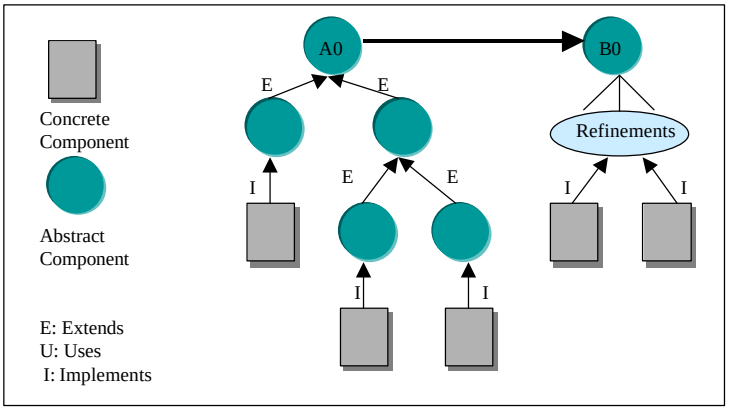


Figure 3-3 Interaction Abstraction Refinement

The *implements* and *extends* relationships and their correct realization refer to vertical transformation of component descriptions and models, whereas the *uses* relationship is helpful in assessing the compatibility of dependent components based on their mutual expectations. The correct realization of these relationships requires assessment methods developed and tailored specifically for the formalisms used in component descriptions (i.e., abstract and concrete components). The certification model founded on behavioral component relationships requires three stages. It must assess the interface compatibility and reusability between dependent components at every level of abstraction (i.e., reusability relation). It must also use the certification behavioral model to determine the safety of the component extensions, as well as the safe evolution of the reused components and the environmental context. Finally, it must determine the correctness of the abstract component in terms of the concrete component and demonstrate the mutual compatibility between the two..

The *extends* relation is usually defined in terms of a checklist of formal properties that must hold between the specifications of the two abstractions at different levels. Such relations are available for abstract data types based on their functional specifications. This definition of *extension* guarantees that certain properties of the more abstract specification are preserved by the less abstract, refined version. In the context of certification with reuse, ensuring a safe refinement relationship has a variety of implications. First, a safely refined component should continue to be compatible with the rest of the entities upon which it depends. Second, a component that implements an

abstract component (i.e., specification) that safely refines the original component's abstraction should be capable of being used safely in the context of the component under certification. The above formulation for safe refinement in the context of ADTs is provided as a general model of *extension* and should be modified and interpreted according to the formalism used for modeling the component's abstract behavioral space.

Determining the reusability of a component in the context of component under certification is an essential element of the certification process at every level of abstraction. In this section a general specification matching method is formally described to provide a basis for interaction-oriented interface compatibility.

Similar to Chen and Cheng [2000], the reusability of components is formalized in terms of correctness with respect to specifications. To keep the scheme as general as possible, we represent component specifications in terms of pre/post conditions but do not constrain the behavioral space or choose any specific axiomatization method. This method facilitates application of the reusability concept for the interaction interfaces introduced in the next subsection.

A component C is **reusable** if it implements a query specification $Q: (Q_{\text{pre}}, Q_{post})$ such that $Q_{\text{pre}}\ \{C\}\ Q_{post}$ holds. That is, the observable behavior of the component must conform to the required behavior imposed by the context in which it is embedded. A reusable component should be partially reusable and able to participate in the expected causal process underlying the component under certification. A partially reusable component is only a candidate for reuse.

The query specification might vary from first-order predicate calculus to process-centric and state-based formalisms representing the input/output behavior of the component. Therefore, the problem of reusability assessment is to demonstrate that the actual input/output behavior of the reused component is in conformance with the required behavior.

Formally, this matching process can be defined as a boolean function

$$\textbf{\textit{Match: Spec}}\ \mathbf{X}\ \textbf{\textit{Spec}}\ \mathbf{\rightarrow}\ \mathbf{\{T, F\}}$$

Consider two specifications *R* and *Q* where *Match(R,Q)* holds. That means, logically, the following should hold:

$$Q_{pre} \Rightarrow R_{pre}$$

$$R_{post} \Rightarrow Q_{post}$$

That is, the reusable component should at least have a weaker precondition on the input state space and a stronger post condition on the output state space.

Implements relationship, on the other hand, refers to the correct realization of the abstract component and is a conformance relationship between the abstract state space and the actual concrete implementation. The relation addressed in this work is not a total correctness, but rather conformance to a given specification. Establishing the properties of a component in isolation is considered to demonstrate that as long as the client components reuse the component under certification in compliance with the certified properties, the component will deliver its specified output behavior.

3.3 A SEMI-FORMAL ROLE-BASED COMPONENT MODEL: THE INTERACTION DYNAMICS PERSPECTIVE

This section introduces an abstract model that includes component and interface descriptions, and models of structural and behavioral dependencies, as well as constraint representations. This model is used throughout the certification process and guides the application of the certification methods. The interaction model and the organizational rationale represented by the causal process underlying the component under certification play key roles in the certification process. In this section, the key constructs underlying the abstract model are outlined. Finally, these constructs are mapped to the representation language—augmented UML collaboration graphs—used throughout the framework.

3.3.1 Components, Interaction Channels, and Interaction Behavior

This section introduces the abstract model that characterizes the interaction interfaces, component dependencies, and compositional and behavioral constraints shaping the certification process.

The abstraction consists of the following elements upon which a component's interactions aspects are defined. First, the components are the primitive or composite elements from which the component under certification is built. Each component is defined in terms of its roles. Hence, a component provides an interaction interface constituting a set of roles, *R*, each defined by a role interface. Each role interface specification declares a set of channels through which the component interacts (i.e., method calls, firing of events).

The development proceeds through refinement and extension of role interfaces. An *adaptor* component with a *composition* protocol is provided for incompatible but connected role interfaces. The refinement of component role interfaces and adaptor protocols concludes with the final component implementation. This stage implements the required local interaction policies and the expected causal interactions that emerge as a result of integration dependencies among the constituent components. As discussed above, role interfaces play a key role in the abstraction and are defined in more detail below.

Role Interfaces: Components are the basic building blocks of a component-based system, and several branches of software engineering emphasize the explicit modeling of interactions as a major step during analysis and design. This is a typical component in the development of concurrent, reactive embedded systems. Message sequence charts have found their way into most common object-oriented analyses and design methodologies. Such modeling aids are semi-formal and do not facilitate reasoning about safe refinement, reusability analysis based on specification matching, or consistency.

Here, a semi-formal means is presented for representing such interaction policies and protocols to enable capturing of behavioral component relationships from the interaction dynamics perspective. In this abstraction model, each component possesses a set of role interfaces, and each interface consists of a set of interaction channels. An interface is defined between interacting peers.

An interaction interface is defined in terms of its set of *roles*, composed of interaction channels, as shown in Figure 3-4. Each role description consists of a *syntactic* and a *semantic* part. The syntactic part specifies the names, types, and direction of

message channels (i.e., incoming/outgoing method calls, events) that connect the two components.

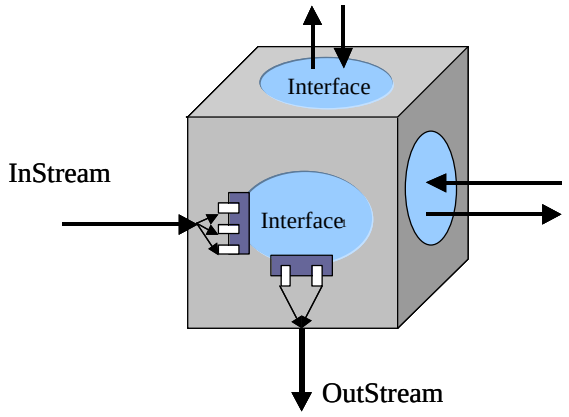


Figure 3-4: Component Role Interfaces

Let *R* denote the set of role interfaces constituting the interaction interface. The set of channels in each role interface are further decomposed into set of *InStream* and *OutStream* channels according to their direction. The semantics of each role interface is defined in terms of *streams*, finite or infinite sequence of messages, to represent the communication and interaction histories. A stream with discrete time, called *timedstream*, from the set *Messages* is an element of the set

$$(Messages^{*})^{\mu}$$

that stands for

$$N \backslash \{0\} \rightarrow Messages^{*}$$

Given a set of roles *R*, constituting the interaction interface of the component under certification, *[R]* is defined as the set of its *interaction mappings:*

$$[R]: R \rightarrow (Messages^{*})^{\mu}.$$

The elements of the interaction mappings are the interaction histories. Each role $r \hat{I} R$ consists of a set, C_r, of channels. A channel is an association between two

interconnected roles; specifically, they are the incoming and outgoing method calls and events constituting the interface of the role. Given these basic definitions of role interfaces and interaction channels, the interaction interface of a component is formally characterized as a predicate, called *interaction predicate* (*IP*), on the set of interaction mappings that may occur during the interaction between the component under certification and its mate role interfaces. That is, *IP* denotes the assumptions and obligations of a component with respect to a set of roles. Formally, *IP* corresponds to the mapping:

$$\textbf{\textit{IP: [R]}} \rightarrow \textbf{B}, \text{ where } \textbf{B} \text{ stands for boolean, or equivalently}$$

$$\textbf{\textit{IP: (R} } \rightarrow \textbf{\textit{ (Messages}}^{*}\textbf{\textit{)}}^{\mu} \textbf{\textit{)}} \rightarrow \textbf{B,} \text{ also}$$

$$IP.m \equiv \mathrm{P}(m.r_1, m.r_{2,\ldots,} m,r_k) \circ P_1(m.r_1) \cup P_2(m.r_2) \cup \ldots \cup P_k(m.r_k),$$

where P_i denote the logical expressions denoting the possible states to which the set of message sequences, $m.r_i$, puts the interaction state of role r_i. The projection of the *IP* on a specific role is defined as

$$\textbf{\textit{IP}}_{ri}\textbf{\textit{: (R} } \rightarrow \textbf{\textit{ (Messages}}^{*}\textbf{\textit{)}}^{\mu} \textbf{\textit{)}} \rightarrow \textbf{B,} \text{ where } \textbf{\textit{R}}=\{r_i\}; \textbf{\textit{ hence IP}}_{ri}=P_i\ (m.r_i)$$

Role Decomposition: The role interface interaction constraints of a complex component might potentially interweave a set of role interaction policies into a single composite policy. Such compound and composite policies exist when a component assumes a variety of roles with respect to several components with which it is associated. The decomposition is similar to the decomposition of a protocol used in the context of *protocol consistency*, which will be discussed in chapter 4.

Component Interaction Behavior: The interaction interface concept is refined to derive interaction behavior for the component under certification. A relational component interaction behavior (*IB*), is given in terms of the mapping from the *I/OStreams* of the collection of role interfaces to boolean. The set of role interface channels are partitioned

into input channels, *InC*, and output channels, *OutC*. That is, *InC* and *OutC* refer to the input and output channels, respectively, of the role interfaces provided and required by the component, as well as by its constituents. The input/output behavior of the component under certification is defined as

$$C_{IB}: (InStream \rightarrow OutStream) \rightarrow B,$$

where

every $m_{in} \in InStream$ is an element of $(InC_1 \cup InC_2 \cup \ldots InC_n)$ and

every $m_{out} \in OutStream$ is an element of $(OutC_1 \cup OutC_2 \cup \ldots OutC_n)$.

The relevant role interfaces involved in the definition of the component interaction behavior are only those reused or provided by it. Hence, the above mapping represents the feasible input/output streams of the component under certification, considering all of its role interfaces collectively interacting with the component's environment as well as with its constituent components.

Note that in order to generate particular output streams as expected by context, each role interaction interface assumes particular constraints in its input. Hence, the input/output behavior constraints of the component under certification are decomposed into a set of (*assumption, obligation)* pairs for each role interface that it includes. This decomposition leads to compositional reasoning based on interface connections. These notions of constraints on the input and output behavior are analogous to pre/post condition concepts in the *design by contract* paradigm.

As discussed above, each role *R* has a set of input and output channels, *C*. For object-oriented components, the defined methods are entry points of the input channels, whereas the message send or method calls denote output channels. The *assumption predicate* for each role interface $r \in R$ is defined as

$$A_r : [I] \rightarrow \mathbf{B} \text{ where } [I] = (R \rightarrow (Messages^*)^\mu) \text{ and } R=\{r\}, Messages \in (InC_r).$$

The *obligation predicate* for each role interface $r \in R$ is defined as

$$G_r : [O] \rightarrow \mathbf{B} \text{ where } [O]=(R \rightarrow (Messages^*)^\mu) \text{ and } R=\{r\}, Messages \in (OutC_r)$$

A function, called ***Match***, is introduced to denote (1) the compliance and conformance relationships between the association role protocols of the component and the role interfaces (both the provided and required interfaces) with which it is associated and (2) the conformance between the directly-connected role interfaces.

Let i_r *and* o_r denote the input and output stream valuations as defined in the component interaction behavior definition. I_r and O_r are defined as the set of mappings of i_r*s and* o_r*s* based on the projections of the input and output streams onto the role *r*. Also, o_r denotes the output streams valuation generated by the role under consideration in response to the set of input stream valuation depicted by i_r. R_M denotes the roles associated with the adaptor *M*. The i'_r *and* o'_r denote the outgoing and incoming messages from the mediator with respect to role *r*. The i'_r, for instance, denotes the valuations (mappings) involving the messages depicted by i_r with the reverse direction.

$$Let\ R_M = \{r_1, r_2, \ldots, r_n\}$$

$$\textbf{Match(M, } \mathbf{R_M}) \equiv ((\forall\ r \in R_M \text{ and } \forall\ i_r \in I_r\ \exists\ o_r \in O_r\ \mathbf{IP_r}(i'_r) \Rightarrow \mathbf{A_r}(i_r) \wedge \mathbf{G_r}(o_r) \Rightarrow \mathbf{IP_r}(o'_r))]$$

where r_i denotes a particular role that participates in computation through *Mediator* (i.e., an association role). $\mathbf{IP_r}(i'_r)$ denotes the interaction predicate of *Mediator* projected onto its obligations with respect to role *r*. Similarly, $\mathbf{IP_r}(o'_r)$ denotes the interaction predicate of *Mediator* projected onto its assumptions with respect to role *r*. $\mathbf{A_r}(i_r)$ and $\mathbf{G_r}(o_r)$ denote the assumptions and obligations of the role *r* against the mediator. Combining the above specifications, along with the previous definitions, the component interaction behavior constraints are specified as follows:

$$\mathbf{C_{IB}}(I, O) \equiv [\forall\ r, cr \text{ that are directly connected } \mathbf{compatible(r, cr)} \wedge (\forall\ r \in \mathbf{R}\ \forall\ i_r \in I_r\ \exists\ o_r \in O_r\ \mathbf{A_r}(i_r) \Rightarrow \mathbf{G_r}(o_r)) \wedge (\forall\ M\ \forall\ r \in R_M \text{ and } \forall\ i_r \in I_r\ \exists\ o_r \in O_r\ \mathbf{IP_r}(o) \Rightarrow \mathbf{A_r}(i_r) \wedge \mathbf{G_r}(o_r) \Rightarrow \mathbf{IP_r}(i))$$ where $i = o_r$ and $o = i_r$ with messages in the opposite direction.]

In the above formulation *r* and *cr* denote the directly connected roles of the set of all roles **R**, whereas *M* denotes the set of adaptors (association roles) developed/reused by the

component engineer. It must be assumed that the assumption and obligation decompositions of a particular role used in reasoning are implied by the projected interaction predicate for that particular role. It must also be assumed that for each role interface the satisfying of an assumption predicate on a given input message stream implies the satisfying of the obligation predicate on the resultant output stream. If these two conditions are met, one can finally assume that the correct interaction behavior of a component will depend on the compliance relationship on the abstract state space of the connected role interfaces and the consistency of the adaptors with respect to individual role interaction policies.

3.3.2 Role Interaction Policies and Adaptor Composition Protocols

The interaction-oriented model outlined in Section 3.3.1 facilitates capturing the interaction behavior of the component under certification in terms of role interfaces, mutual role agreement, and the compatibility of the component's internal protocol with the constituent roles interaction policies.

The *interaction policy* of a role interface is defined first in terms of the definition of channels' syntax. The definition also takes into account the patterns of interactions that depict both constraints on the message sequencing and coordination with respect to a particular role interface. Role interfaces, associated with interaction policy specifications, enable meaningful, controlled assembly of the components according to their roles. Roles can be undertaken by separate threads of control, and their purpose is manifold: to define the achievable objectives, to indicate the ensuing organizational relationships between participants, to set the channels through which interactions take place, and to dictate the patterns of interaction deemed feasible. The organization's rules accompany the role interaction policies, and they define the regulation of interactions among the components' incompatible role interfaces in order to achieve the desired scenarios and required causal processes (see Section 3.3.3). These rules are called the *composition protocol.* Implemented by mediator components, composition protocols coordinate, regulate, and resolve interferences involving interactions that take place during the computation.

3.3.3 Causal Behavior Model: Introducing Organizational Rationale into Verification using Causal Process Modeling

This section considers how a component's behavior depends upon the composition and configuration of its own components. The behavior of the organization depends upon the ways in which the roles are enacted, as well as the degree to which organizational rules are followed. In this context, the term "design rationale" encompasses both the decisions that underlie the conception of a component and the reasoning that lies behind those decisions. Such decisions help determine where and which reused components can be advantageously and appropriately used to achieve desired functional as well as non-functional goals.

The rationale is viewed here as the relation of the component's parts to other elements of the design; that is, the way in which a component interacts with other components. Such a view of rationale captures the role that entities play in the overall functioning of the component and designates the use of a part or modification to resolve undesirable effects (introduce or prevent effects).

Causality plays an essential role in understanding the purpose and rationale of a design. In order to say that a component has achieved its goal, which may be expressed as a condition on the state of the component, one must not only show that the condition is satisfied but also prove that the components used participated in creating the result. The goal involves determining whether the used components actually play their intended roles and purpose in achieving the overall functionality. This requires representing the causal process among the interacting components. Causal relations can be defined in a variety of ways. For example, a causal relationship between events in two threads might be implied either by two threads synchronizing by means of locks or by writing and reading the same object. Since such causal relations are directly implied by the semantics of the operations in the computation generating the events, they are called *computational cause*.

With regard to the issue of the component under certification, the study projects on using interaction-based causal relationships for determining the correctness of or deviations from the causal process patterns of constituent components. The causal model consists of a set of events *E,* along with two ordering relationships on *E*. The *temporal*

relationship orders the events and interactions according to a clock. The *causal relationship* defines an ordering with respect to the cause of an event. Chapter 5 formally defines and demonstrates how the causal event model facilitates reasoning about the causal process underlying the component under certification.

Causal relationships between events are represented in the proposed model by the dependency partial order. An event *B* depends on an event *A*, written ($A \circledR B$) if and only if:

- *A* and *B* are generated by the same thread, and *A* is generated before *B.* (threads are sequential; all events generated by a thread have a total dependency ordering), or
- A thread is triggered by *A* and then generates *B*, or
- $A \rightarrow B$ and $B \rightarrow C$; then $A \rightarrow C$ (transitivity).

3.3.4 Interaction Policy and Causal Process Modeling of Synchronous Components using UML Collaboration Graphs

As discussed in section 3.1, the evolutionary development process of a component can be viewed in the context of the *Interaction Abstraction Refinement Model.* One of the key aspects of the model is the representation of component descriptions from the most abstract to the concrete implementation. In the certification model, an *abstract component* description is composed of two main sections: (1) the role interfaces, association role composition protocol constraints, and the configuration of roles, (2) the causal processes among the deployed components and associated constraints upon these processes.

Existing modeling languages such as UML provide the modeling constructs required by the abstraction. UML is a visual modeling language that permits efficient specifying, visualizing, understanding, and constructing of solutions, particularly for software design problems. It provides modeling views to address *structural, behavioral, user, implementation,* and *deployment* issues. The user-model view encompasses a problem and solution from the perspective of those individuals whose problem the solution addresses. This view presents the goals and objectives of the problem owners, as well as the solutions they require. The structural view considers the static or structural aspects of a problem and solution. The behavioral view addresses the dynamic or

behavioral aspects, interactions, and collaborations among problem and solution elements. The implementation model view delineates the structural and behavioral aspects of the solution's realization. Finally, the environment model view encompasses the structural and behavioral aspects of the domain in which a solution must be realized.

The interaction aspects and the causal process modeling requirements imposed by the introduced framework require the deployment of a collaboration viewpoint among roles. The *UML collaboration graphs* describe interactions among classes and associations modeled as message exchanges among roles participating in the collaborations. Collaboration graphs may also exist in generic form to describe a set of components and associations involved in message-exchange sequences; that is, the collaboration among class roles and association roles, in addition to their interactions. These graphs define a context that constitutes a set of related class roles and associated roles used for a specific purpose. A detailed, more formal overview of collaboration graphs is given in Appendix A. This overview is adapted from OMG standard.

Next, we discuss how the role interface, adaptor protocol, and causal processes are mapped to collaboration diagram constructs. Collaboration diagrams contain the following elements, represented visually in Figure 3.5:

(1) *Class roles* represent roles that objects may play within the interaction policies. They are the classes that define roles or specific parts played by class participants in interactions or collaborations. They are used to model roles that entities play within interactions or collaborations with respect to specific roles. They also represent a restricted view of components by defining what is required of a component for its interaction and collaboration with respect to a particular role of a component in a particular context and configuration.

(2) *Association roles* represent roles that links—transport and adaptation mechanisms—may play within interaction policies. They designate roles that bind actual links when collaborations are used, and they have association role ends that connect association role paths to two or more other class roles. Multi-roles are

association roles representing a set of objects or links that participate in interactions. When connected to class roles representing sets of objects, they must have multiplicity of expression.

(3) *Message flows* represent messages sent between objects via links. They involve the sending and receiving of messages among class roles via association roles or direct connection where role compatibility is not an issue. These messages include method calls as well as signals sent and received between active components

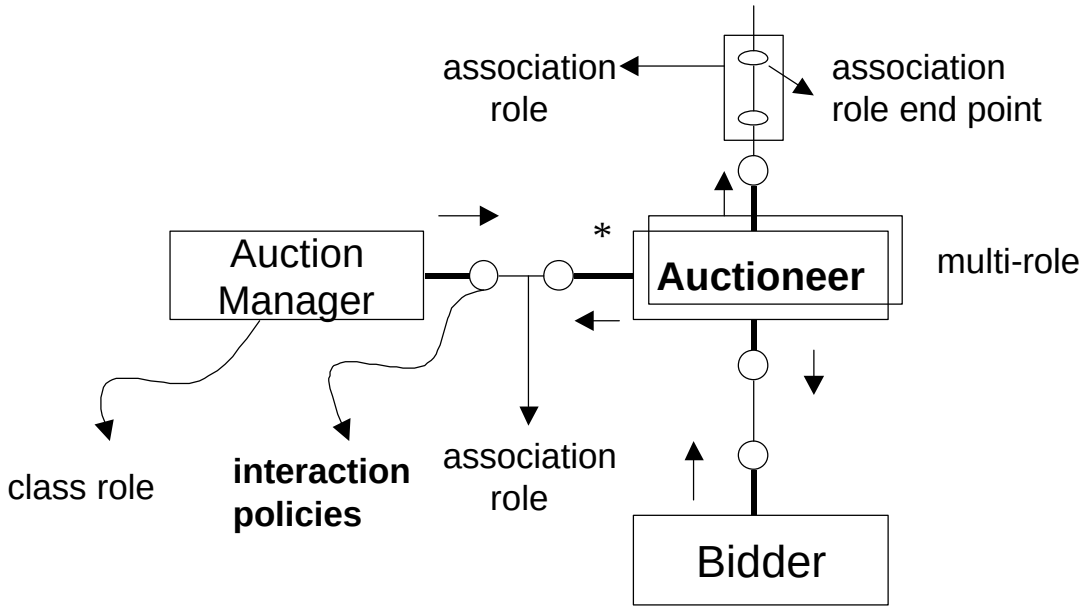


Figure 3.5: Auction Process Collaboration Diagram

Using the above-defined elements, it is possible to map this semi-formal, interaction-oriented model based on the roles that define interaction policies, adaptor protocols, and causal processes. Basically, the role interaction policies augment the class roles, with interaction semantics denoting the local interaction polices of a particular role with respect to an association end role or a particular connected role.

Each association role maps to an adaptor and constitutes a protocol that—unless the roles are determined to be compatible—mediates the interactions among the participant class roles. The message flows outline the causal process involving one or more class and association roles. The sequence numbers of the message flows indicate the order of messages within the next higher level of nesting or passing of control. Each

collaboration diagram starts with message 1. Messages that differ in one integer form a sequence at the same level of nesting. For instance, at the same level of nesting, message 1.1 precedes message 1.2, message 1.2 precedes message 1.3, and so forth. Among any individual nesting level, a message is considered to be concurrent when it originates from different class roles. For example, if message 1 and message 2 originate from different class roles, they are deemed concurrent.

Once more, consider the collaboration graph depicted in Figure 3.5. An auction single bidder. The interaction policies of incompatible class roles are linked through association roles. Each association end role denotes the assumptions and obligations that the adaptor (i.e., association role) imposes on the connected class role.

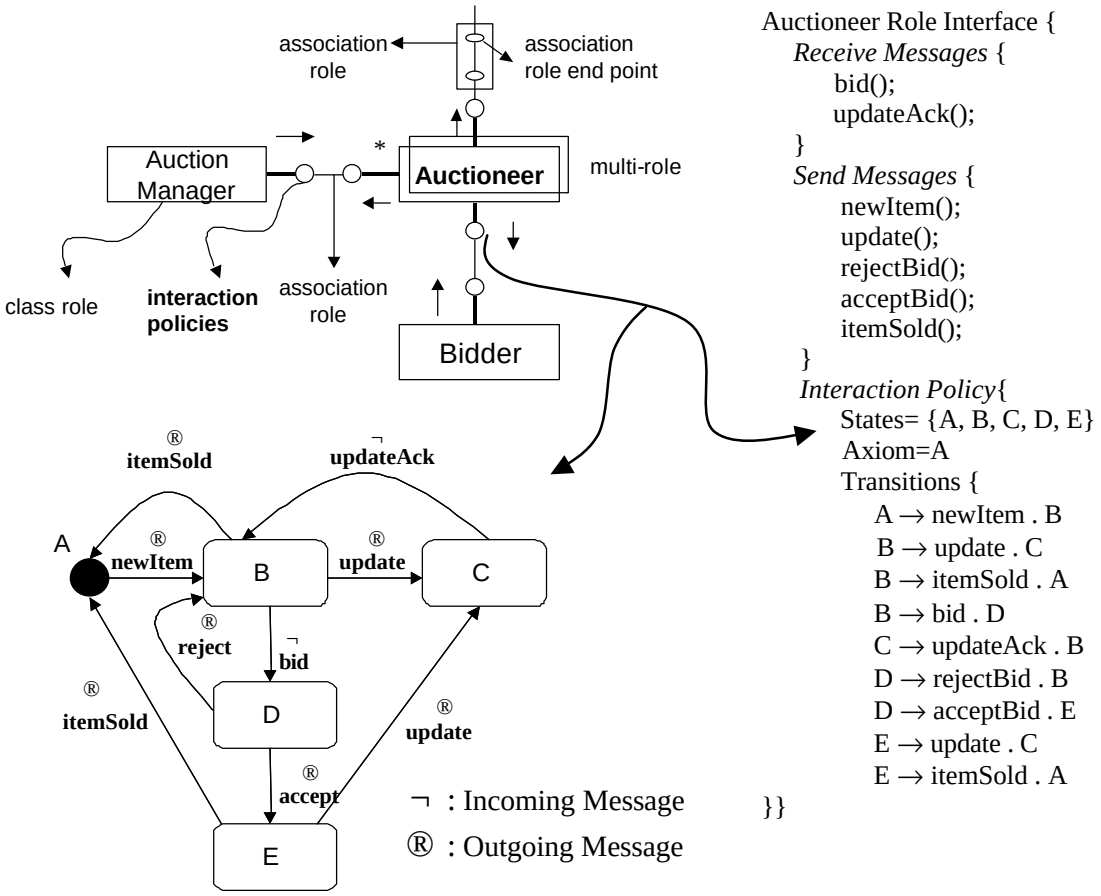


Figure 3.6: Role Interaction Policies

The collaboration graph is augmented to represent the interaction polices of individual roles. The association end roles and the semantically-enhanced service access points

denoting the interaction policies of class roles are represented in terms of UML state charts.UML state charts represent state machines, graphs of states and transitions that describe the responses of the component to outside stimulus. In this formulation and usage, as illustrated in Figure 3.6, states define the interaction condition and service availability of components. Transitions define relations between these conditions of service availability. The arrows associated with the interactions denote the direction of the messages. An arrow that points to the left denotes an incoming message, whereas an arrow that points to the right denotes an outgoing message.

Chapter 5 discusses in detail the representation of causal processes in terms of the message flows in collaboration graphs, as well as the linguistic (syntactic) abstraction and associated semantics of the constraints imposed on them.

3.3.5 Limitations and Prospects of Synchronous Role Interaction Policy and Causal Process Formalisms: Decidability vs. Expressiveness

The utilized role interaction policy representation formalism is based on finite automata. Hence, the expressions and coordination patterns that can be described by this formalism is constrained with the regular expressions. This restriction brings with it weaknesses in terms of expressing certain coordination mechanisms required by the domain, while it enables decidable analytical computation of consistency conditions for safe refinement and mutual role compatibility.

One can distinguish two levels of interactive expressiveness: *sequential* vs. *multi-instance*. In a sequential interaction interface (i.e., role), there is a single input/output message stream, representing a non-interleaved interaction with respect to a single associated role. An unbounded multi-instance collaboration needs to process and coordinate interaction streams, each one representing interaction with a separate autonomous component. The components interact simultaneously and independently with the unbounded multi-instance role without necessarily any awareness of the presence of other components. Moreover, the number of components playing the collaboration role imposed by the domain can change dynamically: components can start and terminate their interaction with the multi-instance role without the awareness of other components.

The need for active management for interaction, and dynamic reconfiguration of interactive entities makes multi-role interaction difficult to formalize and to model than sequential interaction. On the other hand, it allows the set of behaviors for multi-role collaborations to be strictly richer than for sequential interaction. For instance, even in the case of the preliminary semaphore case study of chapter 6, the logic of ordering of the clients while waiting the semaphore to be unlocked is beyond the representation capability of the regular expressions. Similarly, in the case of the auctioneer-bidder case study of chapter 6, the relationship between auction manager and auctioneers constitute the multi-instance interactions defined above.

Depending on the complexity of the coordination mechanism several solutions exist to capture and formalize the collaborations irreducible to regular expressions, finite automata, and directed acyclic graphs (causal processes). Behavioral contracts defined in terms of the first-order predicate calculus constraining the internal state space of the component is widely used to capture algorithmic notions of coordination of multiple roles. For instance, in the case of the semaphore case study, queuing and retrieving of clients based on certain rules or regulations using proper data structures facilitate capturing richer collaboration scenarios involving unbounded number of component instances. Note, however, many decision procedures required for the consistency conditions introduced in this research are undecidable under first-order predicate calculus. For instance, the safe refinement decision procedure is simply based on the implication relation. In particular, in terms of the predicate calculus, the satisfiability of

$$p'(a_1, a_2, \ldots, a_n) \Rightarrow p(b_1, b_2, \ldots, b_m),$$

where **p** and **p'** are predicates and a_i, b_j $1 \leq i \leq n$, $1 \leq j \leq m$ are symbols, is undecidable.

Since analytical methods fail to determine establishment of such consistency conditions, run-time testing and validation mechanisms such as interface violation detection [Edwards *et al.* 1998] and state-based testing techniques overviewed in chapter 2 facilitate establishment of the correctness of such coordination mechanisms. The interaction dynamics perspective for coordination formalization provides a variety of expressive languages in a hierarchical manner (i.e., Chomsky hierarchy). Even though the

expressiveness of such languages are weaker compared to algorithmic constructs that can be utilized for such unbounded multi-instance coordination problems, they enable capturing practically significant interaction and coordination scenarios.

The four major classes of languages are regular expressions, context-free languages, context sensitive languages, and recursively enumerable sets in the order of increasing level of expressiveness. The properties of each language are discussed in detail by Hopcroft and Ullman [1978]. The main concern here is the justification of the prospects of using regular expressions with respect to other expressive languages.

Context-free languages are produced by context-free grammars and are computationally represented by pushdown automata, which extends finite automata with a memory in the form of a stack. Hence, while it is possible to recognize and represent coordination streams of the form $a^n b^n$, $n \geq 1$ using context-free languages, regular expressions cannot recognize such interaction histories (i.e., streams of messages). Hence, no finite automata can capture such a policy. Regular expressions are generated by constrained versions of context-free grammars, which are called regular grammars. A context-free grammar is a quadruple (*N, T, P, S*) where *N* is a finite set of variables, *T* is a finite set of terminal symbols, *P* is the set of rules, *S* is a special variable of *N* called the start symbol. The sets *N* and *T* are disjoint. The rules of a grammar consist of elements of the set. The rule [*R,w*] is also written as $R \rightarrow w$. A rule of this form is also called an *R* rule. As a convention, the terminal symbols are denoted by lowercase letters or strings. The variable symbols (non-terminal symbols) are denoted by capital letters or strings. In particular, if all productions of a grammar are of the form $A \rightarrow wB$, $A \rightarrow wB$, or $A \rightarrow w$, where *A* and *B* are variables and *w* is a (possibly empty) string of terminals, then the grammar is called a regular grammar.

The largest family of grammars in the Chomsky hierarchy permits of the productions of the form $\alpha \rightarrow \beta$, where α and β are arbitrary streams of grammar symbols, with $\alpha \neq \varepsilon$. Such unrestricted grammars characterize recursively enumerable languages. The stream a^i, where *i* is a positive power of 2 is an example string that can be generated by such unrestricted grammars. The context sensitive languages that cover context-free

languages are restricted forms of recursively enumerable languages. The restriction is that for each rule of the grammar $a \rightarrow b$, the length of b is at least as long as the length of a.

The hierarchy theorem [Hopcroft and Ullman 1978] denote that regular sets are properly contained in context-free languages; context-free languages that do not contain empty string are properly contained in context sensitive languages; finally context-sensitive languages are properly contained in recursively-enumerable languages. Furthermore, any computational problem involving a certain language that is not decidable is also undecidable in the language that properly contains it.

It is important for formal languages to have decidable algorithms for answering various questions concerning emptiness, finiteness, equivalency to facilitate automated analysis [Hopcroft and Ullman 1978]. The consistency conditions introduced in chapters 3 and 4 require certain relationships between role interaction policies to be computationally decidable. For instance, as depicted in chapter 4, the safe refinement decision procedure requires $\boldsymbol{L(R_{old}) \subseteq L(R_{new})}$ and $\boldsymbol{L(R_{new}) = L(R_{old})}$ to be computable, where $L(R_{new})$ denote the language of streams derivable by the interaction policies of refined role R_{new}, and $L(R_{old})$ denote the language of streams derivable by the interaction policies of role R_{old}. Similarly, mutual role compatibility consistency conditions and the associated decision procedure require similar equivalency checks to be performed on the abstract representation of policies. While the equivalency problem (i.e., if two finite automata accept the same language) is decidable [Hopcroft and Ullman 1978] for regular expressions, the same problem is undecidable for context-free languages. Hence, the same problem is undecidable for context sensitive and recursively enumerable sets as well. More specifically, given G_1 and G_2 as arbitrary context-free grammars and R as an arbitrary regular set, the following problems are undecidable [Hopcroft and Ullman 1978]:

$$\boldsymbol{L(G_1) = L(G_2),\ L(G_1) \subseteq L(G_2), L(G_1) = R, R \subseteq L(G_1).}$$

Furthermore, directly extending the request scheduling and interaction mechanism to asynchronous messaging under this framework is not feasible. The mutual role compatibility analysis consistency conditions require collaborating components to be deadlock-free. While this problem is computationally decidable for synchronous

interactions, it is proven to be undecidable for asynchronous components [Strom and Yellin 1997]. Hence, the role interaction policies are represented in terms of regular expressions and the causal processes are represented as directed acyclic graphs to facilitate decidable analytic computation of the consistency conditions.

3.4 REASONING FOR CONSISTENCY AND ROLE COMPATIBILITY DURING ABSTRACTION REFINEMENT

This section discusses semantic dependency relationships and their significance in predictable and systematic composite component certification. Here the steps involved in the certification model, along with the principle formalization of the methods are provided.

The abstract component specification based on an augmented collaboration diagram incorporates the roles to be played by the component under certification and its composites. It also designates the explicit and enforced rules of engagement between the component under certification and its constituent components. Development with reuse entails inclusion of specific, separately-developed components into the context of the component under certification. The component abstraction continuously evolves to facilitate the adaptation and assembly of any component into its new context. This process of refining and detailing abstractions demands constant detection and management of inconsistencies.

In this context, the focus is on three main problems. The first involves the concept that the refined interaction policies and composition protocols need to conform to the commitments set forth at the previous level of abstraction. The second problem concerns the idea that associated role interfaces need to be mutually agreeable and consistent. Finally, the third demands that the governing rules of engagement (organizational rules) enforced by the adaptor composition protocols (association roles' interaction policies) need to comply with the constraints imposed by the interaction policies of the individual role interfaces associated with the adaptor.

3.4.1 Interaction Policy/Association Role Protocol Refinement Notions

The refinement notions of role interaction policies and composition protocols are analyzed separately. The role interaction policies depict the UML state chart-based rules of service enactment constraints. The composition protocols are the composite policies embedded with the association roles that bind disparate class roles.

Component Role Refinement: The refinement of individual role interfaces is based simply on the implication relation. Safe refinement of all roles of a particular component is essential for that of the component's abstraction. Conditions for the refinement are devised such that the collaborating components would still be in agreement with the refined interaction policies of the new role interface. We consider two types of refinement, independent and simultaneous.

Independent refinement occurs when a single role interface is modified. Before mutual compatibility of the refined role and its connected role interface, provided by the connected component, can be achieved, one must demonstrate that the refined role interface interaction policies can at least engage in the expected interaction patterns, as dictated by level of abstraction that occurs prior to refinement. Formally, the constraints on the input patterns should be weaker, while the output constraints should be stronger. Hence, the refined role interface would be able to accept the expected input streams from its connection, as well as additional patterns required by the refinement, while the outputs of the refined interface should be acceptable by the input pattern assumptions of the connected role interface.

Furthermore, due to internal non-determinism of the finite state abstraction of the protocol and interaction policies, those sets of message traces or streams that fail to be accommodated by the refined protocol must also be considered failures in the original protocol. That is, it is essential to incorporate *failure semantics*. Chapter 4 discusses the concept of failure in more detail. The *failure* predicate is defined similar to the concept of traces, as follows:

failure: (*In/OutStream* ® $2^{FailureSet}$) ® *B* where

Instream and *OutStream* denote the input and output channel streams of role interfaces, respectively. If the protocol is unable to react to or generate a particular message while in a state that the i/o stream brings it into, a message is considered a failure, hence a member of *FailureSet*, **F= *F(iÅ o)***, where ***F(iÅ o)*** denotes the set of such messages. Using the role interface formalism illustrated in Section 3.3, the role interface interaction policy assumption and obligations are related as follows: Let O_r and I_r be the mappings denoting the set of feasible output and input stream valuations for role *r*.

$$O_r = O_r'$$

$$I_r \subseteq I_r'$$

$$\forall i \in I_r \ \forall f \in F(i): (A_r(i) \Rightarrow \exists i' \in I' \text{ s.t. } i=i' \wedge A_r^{refined}(i') \wedge (failure_{refined}(i',f) \Rightarrow failure(i,f))$$

$$\forall o' \in O_r' \ \forall f \in F(o'): G_r^{refined}(o') \Leftrightarrow \exists o \in O \text{ s.t. } o=o' \, G_r(o) \wedge failure_{refined}(o',f) \Leftrightarrow failure(o,f)$$

The above formulation requires the refined interface to handle all the traces permissible in the more abstract policy specification. Furthermore, all input/output stream failures of the refined specification are also input/output failures of the abstract specification. It is required that the output stream of the refined policy to be equivalent to the output stream of the original interaction policy. This condition is sufficient to ensure that the output streams of the refined policy will be accepted by the receiving role interface. In *simultaneous refinement*, the constraint on the set of output streams of the refined policy is relaxed to incorporate additional output streams. This relaxation must ensure that the receiving role interfaces and the composition protocol can consistently handle new sets of output streams. This notion is depicted in the following formulation:

$$O_r \subseteq O_r'$$

$$I_r \subseteq I_r'$$

$$\forall i \in I_r \ \forall f \in F(i): A_r(i) \Rightarrow \exists i' \in I' \text{ s.t. } i=i' \wedge A_r^{refined}(i') \wedge (failure_{refined}(i',f) \Rightarrow failure(i,f))$$

$$\forall o \in O \ \forall f \in F(o): G_r(o) \Rightarrow \exists o' \in O' \text{ s.t. } o'=o \, G_r^{refined}(o') \wedge failure_{refined}(o',f) \Rightarrow failure(o,f)$$

compatible*(r, cr) where connected(c,cr)=true*

Association Role Protocol Refinement: The protocol refinement is simply an implication relation. Let the interaction behavior of the original and refined protocols be depicted as

$$\boldsymbol{P_0[I,O]: (I \rightarrow O) \rightarrow B} \text{ and}$$

$$\boldsymbol{P_1[I',O']: (I' \rightarrow O') \rightarrow B}.$$

Two types of refinement are considered, *independent refinement* and *simultaneous refinement*. *Independent refinement* occurs when new mediation and state transition rules refine the adaptor protocol, but individual role interaction policies remain the same. The implication relation defined on the common input/output channels refers only to those channels related to those of associated role interface interactions. The input channels of the refined protocol should at least contain all those of the original protocol, and for input assumptions the protocol interaction behavior predicate of the original protocol should imply the interaction predicate of the refined protocol. The refined set of input channels forms the superset of the original input channels.

Furthermore, the set of permissible message traces on the output channels associated with the role interfaces should be equivalent in both protocols. Note that since the protocol is the only artifact being refined, it is necessary to constrain its output channel behavior.

That is, existing role interfaces might not accommodate new input streams that are mediated and forwarded by the protocol, since their interaction policies may be violated. These notions of refinement are depicted in the following formulation:

$$\text{Let } R_M = \{r_1, r_2, \ldots, r_n\}$$

$$\boldsymbol{IP: (R_M \rightarrow (Messages^*)^\mu) \rightarrow B,}\ Messages^* \in (messages(I) \cup messages(O))$$

$$\boldsymbol{IP': (R_M \rightarrow (Messages^*)^\mu) \rightarrow B,}\ Messages^* \in (messages(I') \cup messages(O'))$$

$$[(\forall r \in R_M \text{ and } \forall i_r \in I_r \ \forall o_r \in O_r\ \boldsymbol{IP_r(i_r) \Rightarrow IP'_r(i_r) \wedge IP'_r(o_r) \vee IP_r(o_r))} \wedge$$

$$\boldsymbol{IP'_r failure(i_r,f) \Rightarrow IP_r failure(i_r,f))} \text{ where } f \in \boldsymbol{F}(messages(i_r)) \wedge$$

$$\boldsymbol{IP'_r failure(o_r,f) \vee IP_r failure(o_r,f))}] \text{ where } f \in \boldsymbol{F}(messages(o_r))]$$

With the exception of relaxation on the obligation constraints, s*imultaneous refinement* is similar to independent refinement. That is, the output constraints are weaker with respect to a set of specific roles—each of which is denoted by *r*—as long as that role is refined to be compatible with the relaxed obligation constraints of the output message stream:

$$\textit{Let } R_M = \{r_1, r_2, \ldots, r_n\} \text{ and } R^{refined}{}_M = \{f_1, f_2, \ldots, f_m\}$$

$$\textbf{IP: } (R_M \ \textbf{®} \ (\textbf{Messages}^*)^{\mu}) \textbf{®} \ \textbf{B}, \ \textit{Messages}^* \ \hat{I} \ (\textit{messages } (\textbf{I}) \ \grave{E} \ \textit{messages } (\textbf{O}))$$

$$\textbf{IP': } (R_M \ \textbf{®} \ (\textbf{Messages}^*)^{\mu}) \textbf{®} \ \textbf{B}, \ \textit{Messages}^* \ \hat{I} \ (\textit{messages } (\textbf{I'}) \ \grave{E} \ \textit{messages } (\textbf{O'}))$$

$$[(\forall r \ \hat{I} \ R_M - R^{refined}{}_M \text{ and } \forall i_r \ \hat{I} \ I_r \ \forall o_r \ \hat{I} \ O_r \ \textbf{IP}_r(i_r) \ \text{Þ} \ \textbf{IP'}_r(i_r) \wedge \textbf{IP}_r(o_r) \ \hat{U} \ \textbf{IP'}_r(o_r)) \wedge$$

$$\textbf{IP'}_r \textbf{ failure } (i_r, f) \ \text{Þ} \ \textbf{IP}_r \textbf{ failure } (i_r, f) \text{ where } f \ \hat{I} \ F(\textit{messages}(i_r)) \wedge$$

$$\textbf{IP'}_r \textbf{ failure } (o_r, f) \ \hat{U} \ \textbf{IP}_r \textbf{ failure } (o_r, f)] \text{ where } f \ \hat{I} \ F(\textit{messages}(o_r))] \wedge$$

$$[(\forall r \ \hat{I} \ R^{refined}{}_M, \forall i_r \ \hat{I} \ I'_r, \exists o_r \ \hat{I} \ O'_r \text{ s.t. } \exists o = i_r \text{ and } i = o_r \ \mathbf{G}_r(o) \ \text{Þ} \ \textbf{IP'}_r(i_r) \wedge \textbf{IP'}_r(o_r) \ \text{Þ} \ \mathbf{A}_r(i))]$$

$$\text{with } \textit{destination } (o) \ \text{¹} \ \textit{destination } (i_r) \text{ and } \textit{destination } (i) \ \text{¹} \ \textit{destination } (o_r)$$

The above formulation ensures not only that the input channel behavior of the refined adaptor is weaker with respect to a particular role, but also that the behavior of the output channel is relaxed as long as its constraints imply those of the associated role. The valuations depicted by i and o_r denote the same messages with different directions. For instance, while $i_r \ \hat{I} \ I_r$ defines the messages incoming to the adaptor from role r, $o = i_r$ includes the outgoing messages of role r with respect to the adaptor's projected protocol onto role r.

3.4.2 Class Role Interaction Policy Compatibility

The role compatibility method uses semantic analysis to ensure the congruity of connected role interface interaction policies. That is, interaction policies of connected interfaces are checked to ensure that they are consistent with each other. More specifically, collaborating components receive and send messages in a synchronized manner; that is, when a collaboration mate sends a message, the receiving party is shown to be in a state to receive and act upon the message.

Each role interface declares the set of input and output channels in terms of the provided (required) services and the input (output) events. The services refer to bi-directional method calls, whereas the events are unidirectional.

Let R_1 and R_2 be two associated roles in the abstract component representation. Let $PS_{R!}$, RS_{R1}, $A_{r1}(i)$, and $G_{r1}(o)$ be the provided services, required services, and assumptions on the input and output channel behavior valuations of the role interface R_1, respectively. The channels, services, and channel behavior of R_2 is defined similarly. Next, a notion of interaction policy and protocol consistency is formulated.

In order for two role definitions to be mutually compatible, the required services of one role need to be provided by the other, and the assumption constraints of the interaction policy of a role should be satisfied by the associated role. Formally, the compatibility relation is defined as follows: In the formulation, *i* and *o* denote the elements of channel valuation mappings for the role under consideration onto the input and output messages, respectively. The mapping is projected onto messages by using the function *messages*:

$$RS_{R!} \subseteq PS_{R2} \text{ and } RS_{R2} \subseteq PS_{R1}$$

$$\forall o \in O_{R1}\ G_{R1}(o) \Rightarrow \exists i \in I_{R2}\ A_{R2}(i) \text{ with } messages(o)=messages(i)$$

$$\forall o \in O_{R2}\ G_{R2}(o) \Rightarrow \exists i \in I_{R1}\ A_{R1}(i) \text{ with } messages(o)=messages(i)$$

The above formulation requires successful matching of (1) the provided and required services to (2) the assumption and obligation constraints of each role. That is, the required services of a particular role need to be a subset of the provided services of its mate. Furthermore, the obligations of a role with respect to its mate should be compatible with the assumptions of the mate component. More specifically, the output stream constraints, called obligations, of a particular role with respect to its mate need to match the constraints on or assumptions about the input streams of the mate component. Conversely, the input stream constraints of the same role must comply with the output stream obligation constraints of its mate. Note that the output messages of a role interface are the input messages of its mate with opposite directions.

3.4.3 Connector (Association Role) Composition Protocol Consistency

Recent studies in component development and verification, as well as validation processes, indicate that inconsistency is common throughout the development and evolution stages. Inconsistencies occur because the different reusable components developed for different contexts have conflicting assumptions with respect to each other. The introduced framework facilitates global consistency management through pair-wise consistency analysis between the viewpoints of associated role interaction policies. Bridging the inconsistencies is beyond the scope of the presented work; however, introducing mediating protocols that act as facilitators among disparate viewpoints is a common solution. Therefore, practitioners usually deploy protocols that receive messages, mediate, and then forward them to other components that have viewpoints different from the originating component. These protocols are embedded as part of the component logic to determine, regulate, and adapt the component's behavior against the environment and composites. Thus, they can provide the desired services while bridging the inconsistent perspectives of the associated components.

Basically, in the framework, *protocol consistency* refers to the compatibility of the composition protocol with each of the role interaction policies. The need for such compatibility and consistency validation arises due to the sheer wealth of small conflicting details introduced by the reused components' interaction policies. Our position is that this interpretation of complexity—as opposed to the intricate complexity of individual pieces—currently dominates component and system development in the context of reuse. By keeping track of whether assumptions and obligations are satisfied in the construction and evolution of the protocol and the interaction policies, the component designer can better understand the impact of the forward transformations induced as the abstraction is refined.

The composition protocol incorporates all knowledge and rules regarding the means of coordinating and enacting role interface interactions among the environment, component under certification, and the constituent components. In addition to identifying internal rules used solely to regulate and adapt mismatched roles, it also stipulates how and in which order the role enactment occurs. Note that the composition protocol does

not cover the causal processes enacted among the constituent components. Instead, the micro-architecture section of the abstraction defines the set of rules and constraints relevant to these casual processes.

For each interaction policy being tested by the mutual compatibility diagnosis, the composition protocol consistency involves two stages. The first stage projects the protocol onto the interaction policy services of the component and the role interface services of the associated component or the environment. The second stage consists of the mutual compatibility check, which covers the projected interaction constraints imposed by the protocol and the interaction policy constraints of the connected role.

Formally, the consistency check is defined as follows: let the composition protocol interaction constraints be defined as $C_{Protocol}$. The projection of the protocol constraints onto the required and provided services with respect to role r is defined as $\tilde{O}_r^{Protocol}$. The projection consists of the required and provided services and events as well as the derived interaction policies with respect to r.

Given the projection, $\tilde{O}_r^{Protocol}$, let the *derived* assumption and obligation constraints of the projection with respect to the role r be $A_{\tilde{o}}(i)$ and $G_{\tilde{o}}(o)$, where i and o are elements of the set of mapping valuations for the input and output channel streams, respectively. Here o and i denote the elements of the set of mappings for the outgoing channel streams for the projected role, $O_{\tilde{o}}$, and incoming channel streams for the role under consideration, I_r, respectively. The mappings o' and i' are defined conversely:

$$\boldsymbol{RS_{\tilde{O}\,\mathbf{rProtocol}} \acute{I}\ PS_R \ and\ RS_R \acute{I}\ PS_{\tilde{O}\,\mathbf{rProtocol}}}$$

$$\boldsymbol{\forall\ o \hat{I}\ O_{\tilde{o}},\ G_{\tilde{o}}\ (o) \Rightarrow\ \exists\ i \hat{I}\ I_r\ A_r(i)}$$

$$\boldsymbol{\forall\ o \hat{I}\ O_r,\ G_r(o) \Rightarrow\ \exists\ i \hat{I}\ I_{\tilde{o}}\ A_{\tilde{o}}\ (i)}$$

with ***messages(o)=messages(i)*** and **direction (*messages(o)*) ¹ *direction (messages(i))***

The rules indicate three requirements. First, the collaboration mate must provide every service required by a role. Secondly, the output channel obligation constraints of the projected role must be compatible with the input assumptions constraints of the role interface under consideration.

Finally, the output channel obligation constraints of the role under consideration must be compatible with the input assumption constraints of the projected role interface derived from the protocol.

3.5 TESTING COMPONENTS FOR CONFORMANCE TO SYNCHRONOUS ROLE INTERACTION POLICY AND CAUSAL PROCESS COMPLIANCE

The previous section discusses the reasoning method for ensuring an abstract component's conflict and consistency with its semantic dependencies. The *safe refinement* (i.e., extends relation) and the *uses* relationships form the dependencies for which the necessary and sufficient conditions are identified for continuous certification of a component in the context of reuse. The last relationship discussed in the context of the behavioral component relationships viewpoint is the *implements* relationship.

With regard to this relationship, the goal is to demonstrate that the actual concrete component **implements** the abstraction correctly and as *intended* by the designer. Basically, the *implements* relation is based on testing mechanisms to determine the conformance of the concrete component to the abstract interaction policy and composition protocol, as well as to causal process and evolution constraints.

The implements relation assessment entails four related activities. The first one involves generating operational interaction scenarios as test suites from the models of interaction policies. The second one involves creating the infrastructure to operate and monitor the component execution in order to collect information useful in determining interaction policy violation and causal process deviation. The third activity uses model-based analysis and checking of the causal process conformance to determine if the component under certification works and evolves in the ways intended by the designer. Finally, the fourth one includes model-based analysis and checking how the implementation evolves as behaviors are introduced, prevented, or guaranteed, while adapting the component under certification and its constitutes.

Given the operational interaction models of the environment, the generated test scenarios have to ensure that the interaction policies are exercised and covered to a certain degree. Along with the decision procedures for processing compliance, the rules

for interaction integrity and interaction policy constitute the basic components of the *implements* relation. Such rules emphasize the integrity of the role interfaces.

The current state of the practice focuses on results and conformance to desired functional and non-functional requirements without determining whether the desired services function as intended by the designer. Therefore, the organizational rationale depicted by the expected causal process underlying the component requires causal process representation and new mechanisms for analysis.

To this end, interaction test suite generation, interface violation detection, causal behavioral model generation, and constraint analysis constitute the key components of the *implements* relation.

3.5.1 Test Suite Generation and Execution

Given component C and its associated roles $R_1, R_2,\ldots,R_m$, test suite generation utilizes the role R_{env}, which is one of the roles defined in a role set, to derive a sequence of messages that will drive the component's implementation. With respect to the context in which it is assumed to situated, R_{env} denotes the model of assumptions and obligations of the component under certification.

The role interface consists of the signatures of the incoming and outgoing messages, as well as sequencing constraints on them. The set of sequenced messages is generated from the dual of R_{env}, R^{dual}_{env},, which is either derived from the original role or provided by the component designer. If we let the constraints be represented as the set, $Constraints(R^{dual}_{env})$, which contains all feasible message sequences that can be generated by the interaction policy labeled transition system, then the message sequences, $TestSequences = \{S_1, S_2, \ldots, S_k\}$, derived by the test generator, are acceptable if and only if $TestSequences \hat{I} Constraints(R^{dual}_{env})$.

The test derivation algorithm generates only those test sequences compatible with the policy constraints. Every test sequence in the set, $TestSequences = \{S_1, S_2, \ldots, S_k\}$, is called a *scenario*. The execution of each scenario constitutes a separate test session. Each sequence set is generated under coverage criteria, and the complete execution of all scenarios in the set achieves the coverage criteria.

3.5.2 Interaction Violation Detection

Each test scenario includes a sequence of interactions between the test driver simulating the environment and the component under development. A correct component is expected to demonstrate role compatibility with respect to every other collaboration mate. Here, the existence of a specific component situated between every role of a component and its associated mate is considered.

This component, called *violation detector*, connects to collaborators and mediates the interactions between every provided and required role connection defined in the interaction interface, as well as all connected constituent roles. The *violation detector* is based on the concept of a composite interaction policy computed from the individual roles involved in the connection. Let *violation detector* component be represented as VD_{ab}, and the involved roles as R_a and R_b. Role R_a is affiliated with component C_a and role R_b is affiliated with C_b. Assume that C_a is associated with role set $[C_a]=\{R_a, R_{a1},\ldots R_{aj}\}$, whereas C_b is associated with the role set $[C_b]=\{R_b, R_{b1},\ldots R_{bj}\}$. Here, the following simplified assumptions are made: (1) roles are orthogonal and pair-wise independent; that is the methods and events associated with the roles of a role set are distinct, and (2) the roles are deployed or implemented on separate threads of control. These assumptions are made to simplify the recording of which incoming and outgoing message belongs to which role interconnection. The violation detector incorporates a composite interaction policy that mediates interactions between R_a and R_b, as shown in Figure 3.7. This composite is derived automatically from the individual policies. By providing two different interfaces, the implementation of the violation detector acts as a C_b to C_a and as C_a to C_b :

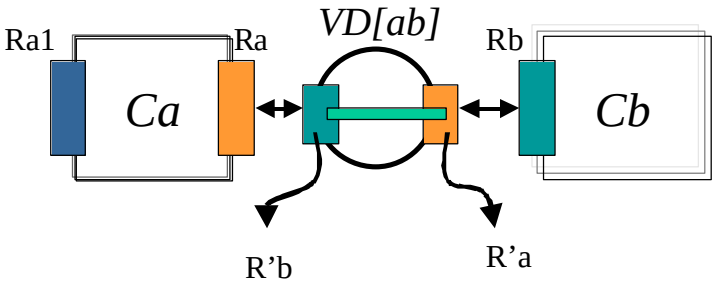


Figure 3.7: Interaction Violation Detection

The violation detector detects and identifies four different types of violations. The R_a obligation violation involves deviations in the outgoing message sequence patterns from R_a; the R_a assumption violation, deviations in the incoming message from C_b, directed to C_a; the R_b obligation violation, deviations in the incoming message sequence patterns from R_b; and the R_b assumption violation, deviations in the incoming message from C_a, that is directed to C_b.

3.5.3 Causal Behavioral Model Generation and Constraint Analysis

The generation of a test suite creates a set of message sequences, each of which is called a *scenario.* A single test session includes execution of a single scenario. The local interactions among each connected role are examined at run-time through attached violation detectors. Note, however, that the internal causal process initiated by the scenario execution often spans at least several local interaction policies.

This process depicts how the component is intended or designed to deliver the required service. That is, the causal process indicates, implicitly, how the configuration produces the behavior. Developing a formal representation and analysis facility to enable understanding and querying of how the component works requires capturing underlying processes and identifying, at a variety of levels of abstraction, the causal links among them. Figure 3.8 illustrates how the cascading of several locally-connected components can, through their local interactions with neighbors, enable the emergence of global causal threads. The behavior emerging from the conglomerates of local interactions depends on how the components are configured and composed. The derivation of a causal behavioral model depicting the computational paths from the set of local interactions is very expensive due to the state space explosion problem. Hence, a new technique, which can be categorized as *hybrid simulation verification*, is developed.

This technique combines the best of both worlds by synthesizing an intermediate analysis method. This method uses the execution of a test scenario to simulate the component with a test driver that emulates the context of the component under certification. The technique uses test scenario execution to limit the generation of causal computation graphs for computations consistent with the simulation. This limited computation graph, called the *causal behavior model*, can include orders of magnitude

smaller than the potential graph that could be derived from analyzing the local interaction policies of roles.

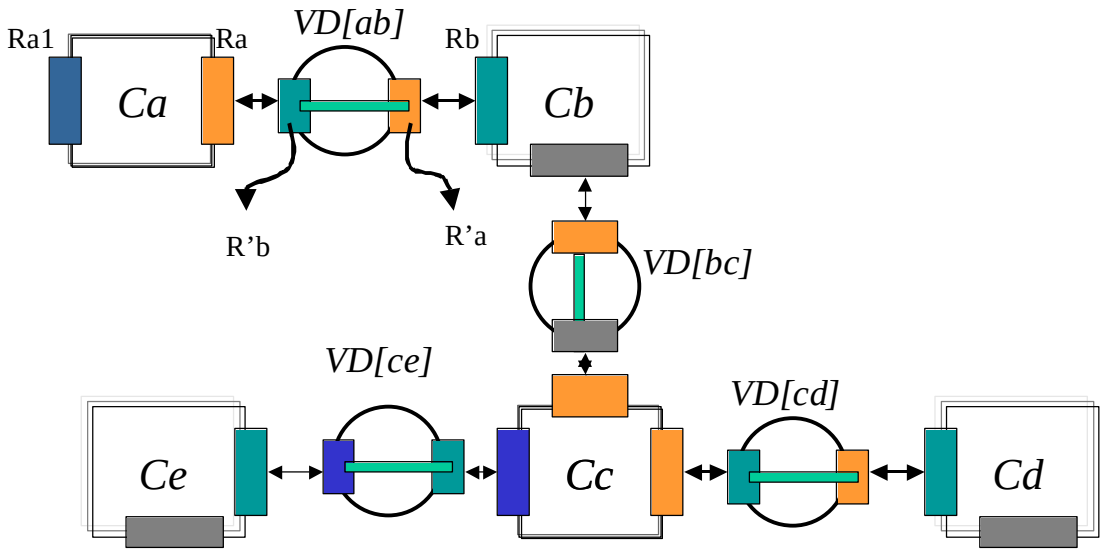


Figure 3.8: Causal Process Observation

The constraint analysis utilizes the derived behavior model to determine whether causal process constraints are satisfied. Formally, the *causal process description* is a directed acyclic graph with a distinguished *start* node. Each node in the graph represents an event, a message sent, or a message received by a component implementing one of the roles depicted in the component abstraction. The links between the nodes denote a causal dependency between two nodes. The causal event model introduced in Chapter 5 outlines the rules for causal dependency. *Causal process constraints* denote temporal, logical, and evolution restrictions on the causal processes that exist in the derived behavior model. The behavior model is derived by transforming and mapping the interaction logs, which are generated by the logging framework utilized by the violation detector components between each role, onto the matched causal processes. *Causal process constraint analysis* utilizes the behavior model and evaluates the constraints by mapping them onto a constraint tree representation. Each constraint is mapped onto a constraint tree to facilitate the application of a tree-based, bottom-up evaluation to the causal behavioral model derived from the interaction logs.

CHAPTER 4

SAFE REFINEMENT AND MUTUAL ROLE COMPATIBILITY ANALYSIS OF INTERACTION POLICIES FOR LOCAL CONSISTENCY

Chapter 3 discusses abstract role interaction and causal process models that capture essential interaction aspects of an evolving component abstraction. The abstraction refinement and compatibility analysis components of the certification framework involve the safe refinement, interaction interface compatibility, and protocol consistency/invariant satisfiability methods, each defined in terms of semantic dependencies among abstract components. This chapter formalizes the interaction roles and protocols vital for delineating the necessary and sufficient conditions for correct realization of the semantic dependencies; namely, *extends* and *uses*.

The certification introduced in this dissertation is based partly on the notion of augmenting an abstraction refinement framework with concepts such as reusability and interface compatibility. Formally representing the semantic dependencies among abstract components and reasoning for their correct realization form the basis of the certification model. Starting from the most abstract component, the certification process for abstract components consists of two actions. The first action determines the interaction policy compatibility of interaction roles and the composition protocol consistency if a protocol is declared, while the second ascertains whether the refined abstraction is safe. This process depends upon whether the role interfaces and the composition protocols (association roles) are refined such that the interaction policy invariants and the coordination policy commitments made at the previous level of abstraction are not violated.

This chapter is organized as follows. In Section 4.1, the notion of interaction policy and composition protocol modeling is formalized. Section 4.2 discusses the safe refinement for interaction assumption and obligation constraints, as well as composition protocols. Section 4.3 focuses on the necessary and sufficient conditions for role compatibility, while Section 4.4 extends these concepts to protocol consistency (i.e., association end role consistency) and the satisfying of interaction policy invariants.

4.1 FINITE STATE MODELING OF ROLE INTERACTION POLICIES AND COMPOSITION PROTOCOLS

To demonstrate the development of finite state models of interaction policies and composition protocols, syntactic grammars of the abstract model representations are used, as defined in Chapter 3. Each role interface in the abstract model is composed of provided/required services and events, plus constraints on their enactment. Formally, each role interaction policy is declared as a UML state chart whose transition rules can be depicted as a regular grammar $G=(V,T,P,S)$. In the transition table, every transition rule, **ruleTerm** '=>' **ruleTerm** **'.' ruleTerm**, is translated into a connection within the finite state model $M=(Q, S, d, q_0, F)$. Here Q refers to the set of states, S denotes the set of transitions, and d, q_0, F refers to the transition relation, initial state, and acceptor states, respectively. Note that the finite state models of the interaction policies are essentially partial and non-deterministic. In this context, the acceptor state information is irrelevant for the purpose of modeling the non-terminating nature of event and service policies.

Generating a finite state model from an interaction policy grammar is straightforward. The set of non-terminal symbols, V, is interpreted as the set of states, Q, while the set of non-terminal symbols, T, is interpreted as S, the set of the transitions in the finite state model. For each rule a '=>' b . c, where a and c are elements of V in the transition table, we define $d(a,b)=c$, which indicates that after receiving an event or service request, b, the interaction finite state model advances to state c from state a.

As discussed above, the finite state model that depicts each role interaction policy is defined formally as a four tuple $M=(Q, S, d, q_0)$ such that

- Q is the finite set of states,
- S is the finite alphabet denoting the provided and required services and events,
- $d \subseteq Q \times S \times Q$ is the transition relation,
- q_0 is the initial state.

Note that interaction policies are usually fine grain and, in comparison to the composition protocols that mediate and coordinate constituent roles, possess a lower complexity in terms of the number of states, events, and transitions. The composition protocols are often hierarchical and encapsulate a great deal of complexity due to the sheer wealth of details in gluing, mediating, and adapting the large number of roles provided by the component under certification, as well as the abstractions of the reused constituent components. Hence, in order to facilitate application of the analysis techniques introduced in this chapter, in this section the existence of a mechanism for mapping hierarchical finite state models to flat state models is assumed.

4.2 SAFE REFINEMENT NOTIONS FOR FINITE STATE MODELS OF INTERACTION POLICY AND COMPOSITION PROTOCOLS

This section introduces the necessary and sufficient conditions, along with the decision procedures, for determining if the refinement of interaction policies is safe. It bears noting that the interaction semantics is synchronous and that the utilized model is a finite automata. Both the interaction policies and the composition protocols use the same representation to facilitate the application of the concepts to protocol and policy refinement. The development of the safe refinement ensues by using independent policy refinement. Those types of refinement defined in Chapter 3 are developed using independent refinement as a basis. Remember that independent policy refinement was defined in Chapter 3 as

$$\boldsymbol{O_r = O_r'}$$

$$\boldsymbol{I_r \subseteq I_r'}$$

$$\boldsymbol{\forall\, i \in I_r\ \forall\, f \in F(i): (A_r(i) \Rightarrow \exists\, i' \in I' \text{ s.t. } i=i' \wedge A_r^{refined}(i') \wedge (failure_{refined}(i',f) \Rightarrow failure(i,f))}$$

$$\boldsymbol{\forall o' \in O_r'\ \forall\, f \in F(o'): G_r^{refined}(o') \vee \exists\, o \in O \text{ s.t. } o=o' \, G_r(o) \wedge failure_{refined}(o',f) \vee failure(o,f)}$$

Basically, the refined policy constraints on the input channels (i.e., assumptions) of role interfaces should be weaker, while the constraints on the output channels (i.e., obligations) should be stronger. Considering the interaction aspect, this viewpoint translates to the fact that the refined role should at least be able to accept all input sequences and patterns of the high-level abstract role specification. Furthermore, the failures of the refined role specification should also be failures of the original abstract interaction policy. Since the role under consideration is the only role that is being refined, the output behavior needs to be as strong as in the previous policy. That is, it is essential to ensure that the output refinement does not violate the input assumptions of the connected roles to which the output channels of the role under consideration are directed.

It is sufficient to assure that the possible output trace patterns that can be produced in compliance with the obligation constraints of the refined role are equivalent to the trace patterns that can be produced by the original role interface. It is also necessary to assure that the failures that might occur in the refined policy should also be failures in the original policy, so that the clients of the role do not experience unexpected message patterns that violate their state-dependent behavior. Failures on the input state and output state spaces might occur due to non-determinism introduced by the abstraction.

To facilitate the development of decision procedures that would enable us to determine if the above conditions hold, finite state models of interaction and composition protocols are used to introduce the following definitions on which the identified properties are formalized.

Given an interaction policy model $M=(Q, S, d, q_o,$ let $s \hat{I} S^*$ and $s' \hat{I} S^*$ be sequences of observable actions and let $a \hat{I} S$.

(a) $L(p) = \{ s \hat{I} S^* \mid \$ p \text{ s.t. } p \Rightarrow^s p' \}$ is the language of $p \hat{I} Q$.

(b) $S(p) = \{a \hat{I} S \mid p \ ® \ a \ . \ p'$ is in the interaction policy rules; that is $d(p,a)=p' \}$ are the set of immediate actions (i.e., events, services) permissible at state p.

(c) $D(p,a) = \{p' \hat{I} Q \mid p \ ® \ a \ . \ p'$ is in the interaction policy rules; that is $d(p,a)=p' \}$ are the immediate states following action a.

(d) $A(p,s) = \{S(p') \mid \$\ p\ s.t.\ p \Rightarrow^{s} p'\}$ is the acceptance set of *p* after *s*. The acceptance set *A(p,s)* represents the set of action capabilities of *p* after *s*. A set is in *A(p,s)* if it is possible for a state to be reached from *p* via *s* such that the only possible actions are those in *S(p')*. The fact that *A(p,s)* may contain more than one such set indicates that non-deterministic choices may occur during the execution of *s*.

Given the interaction policy definitions for the original and refined role interfaces in terms of their finite state models as $M=(Q, S, d, q)$ and $M_{refined} = (Q_{refined}, S_{refined}, d_{refined}, q_{refined})$, the following properties are defined to facilitate the development of the safe refinement notion. It is assumed that the set of services (i.e., S, $S_{refined}$) can be decomposed into input and output services. That is, for S, we define $S_{[in]}$ and $S_{[out]}$ for input and output service and event channels exclusively. $S_{refined}$ is decomposed similarly using the interaction policy descriptions. Similarly, $L_{[in]}(p)$ and $L_{[out]}(p)$ are defined to denote the input language and output languages of interaction state *p*. Later, Section 4.4, demonstrates how to decompose finite state models to facilitate the computation of input and output stream patterns from interaction policies that incorporate both input and output services.

4.2.1 Independent Role Interface Assumption Refinement

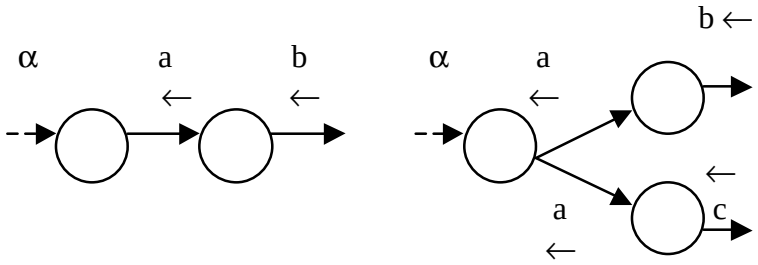
Given the models $M=(Q, S, d, q)$ and $M_{refined} = (Q_{refined}, S_{refined}, d_{refined}, q_{refined})$, and considering the above definitions, the laws defining the argued necessary and sufficient conditions of *independent* safe refinement for the assumption constraints are delineated as follows:

Law 1: $S_{[in]} \acute{I} S_{refined[in]}$

Law2: (a) $L_{[in]}(q) \acute{I} L_{[in]}(q_{refined})$.

That is, let q and $q_{refined}$ be the axioms or the initial states of the original and refined interaction policies, respectively. These laws are necessary to ensure that the interaction policy of the refined role interface is capable of undertaking at least all of the input interaction patterns assumed by the original role interface.

However, due to the non-determinism introduced by the abstraction, the above law is not sufficient to assure the safeness of the refinement. Consider the following scenario depicted below:



(Original) (Refined)

The original policy finite state model indicates that after receiving a sequence of observable actions, ***a***, the interaction policy enables service *a* followed by service request *b.* On the other hand, the refined policy might move into a state from which—after receiving the sequence *a* followed by service *a*—service request *b* becomes unavailable. Furthermore, if action *c* is an output event or service, then the connected role might experience an unexpected reception unless it is prepared to act in response to that request at its current state of interaction. As defined above, the acceptance set *A(p,s)* provides a basis for determining whether every unexpected service pattern in the refined policy can be described in terms of an unexpected pattern in the original pattern.

To facilitate the description of the *law of failures* (law 3), the following inclusion relationship between the state spaces of the original and refined protocols are defined:

Definition: Let $X, Y \in 2^S$, then **X << Y** if and only if $\forall S \in Y, \exists S' \in X$ *s.t.* $S' \in S$

Law 3 (Law of Failures): $\forall s \in S^*, A(q,s) << A(q_{refined},s)$. That is, for every state, *p,* in the refined policy, there exists a state, *p'*, in the original interaction policy such that the set of permissible actions from *p'* is a proper subset of the permissible actions from state *p*.

Note that due to non-determinism, the set of reachable states on certain actions would contain several states as opposed to a single state. In the above formulations of permissible actions, reachable states need to be extended for multi-state variants.

Definition: Let P be a set of states, called a *multi-state*. $S(P)=\{a \in S \mid \exists p \in P, p'$ *s.t.* $p \xrightarrow{} a . p'$ is in the interaction policy rules; that is $d(p,a)=p'\}$ and $P \xrightarrow{} a . P'$ iff $P'=\{p' \in Q \mid \exists p \in P, p \xrightarrow{} a . p'$ is in the interaction policy rules; that is $d(p,a)=p'\}$. That is, $P'=\bigcup_{p \in P} D(p,a)$.

Next, the relations between the original and the refined role interaction policies that need to be maintained are introduced in order to ensure that the above laws are established. Based upon the above discussions, constraints on the input channels are considered, followed by those on the output channels. For the safe refinement of assumptions regarding the original role interface, the above laws hold if and only if the following relations on the finite state models also hold:

Property 1: $S_{[in]} \subseteq S_{refined[in]}$

Property 2: $S(q) \subseteq S(q_{refined})$ where q and $q_{refined}$ are the axioms or initial states of the original and refined policies, respectively.

Property 3: $\forall p'_{refined} \in P_{refined}, \exists p \in P$ *s.t.* $S(p) \subseteq S(p'_{refined})$

Property 4: $\forall a \in S(P)$, *if* $P \xrightarrow{} a . P'$, $P_{refined} \xrightarrow{} a . P'_{refined}$ then substitute P by P' and $P_{refined}$ by $P'_{refined}$ in property 3 and also check to see multi-state equivalent of property 2 holds.

Let $M=(Q, S, d, P)$ and $M_{refined}=(Q_{refined}, S_{refined}, d_{refined}, P_{refined})$ be the projections of the models onto input channels, and P and $P_{refined}$ denote the initial multi-states of the original and refined role interface interactions.

The following algorithm represents a decision procedure that will determine whether the refinement of an interaction policy is safe. It utilizes the above laws and associated properties among the finite state models of the interaction policies. Basically, the algorithm traverses the reachability graphs of the interaction policies' state spaces. This process determines if the above properties hold.

First, the algorithm ascertains whether the set of services provided by the refined role interface is a superset of the original role interface's set of provided service channels. The next step determines both if the permissible set of the refined role interface's input

patterns is a superset of the original role interface input sequences and whether the failures of the refined role interface can be described in terms of those pertinent to the original role interface. The search for these properties are accomplished by initiating a graph search starting from the initial states of both the original and refined role interface interaction policies. The transitions of the original interaction policy are used to traverse the state space of the refined policy.

Note that due to non-determinism the reachable states are multi-states, and in the worse case scenario, the algorithm has a time complexity of 2^{cm+dn}, where m and n are the number of states in the original and refined policies, respectively. The time complexity for processing each state in the original and refined policies is constant, represented by c and d, respectively. The number of subsets of a set of size n is 2^n; hence, the upper bound on the number of multi-states in which the role interface interaction state can exist is 2^n. For each multi-state traversed at the original and refined policy state spaces, the above properties are sought to determine if the assumptions of the role interfaces are safely refined.

Safe Refinement Decision Procedure for Role Interface Assumption Constraints:

Input: $M=(Q, S, d, q)$ and $M_{refined} = (Q_{refined}, S_{refined}, d_{refined}, q_{refined})$ are the projections of the original and refined role interaction policy finite state models onto input channels.

Output: returns TRUE if the refinement is safe; otherwise returns FALSE.

Algorithm:

R=∅

if not($S_{[in]} \acute{I} S_{refined[in]}$) then return false

if not($S(q) \acute{I} S(q_{refined})$) then return false

$R = R \grave{E} [\{q\}, \{q_{refined}\}, S(q)]$

repeat

$u = [C, C_{refined}, A]$ where $A \neq \text{Æ}$ and choose an action $a \hat{I} A$

$R = R - u$

$R = R \grave{E} [C, C_{refined}, A\text{-}\{a\}]$

Compute C' and $C'_{refined}$ where $C_{refined} \to a . C'_{refined}$ and $C \to a . C'$

if [*C'*, $C'_{refined}$, *A'*] $\notin$ *R* for some *A'* then

if not(*S(C')* $\acute{I}$ $S(C'_{refined})$) then return false

if not($\forall$ p $\hat{I}$ $C'_{refined}$, $\exists$ p' $\hat{I}$ *C's.t. S(p')* $\acute{I}$ *S(p)*) then return false

R = R $\grave{E}$ [*C'*, $C'_{refined}$, *S(C')*]

until $\forall$ [*X*, *Y*,*ActionSet*] $\hat{I}$ R, *ActionSet* = $\varnothing$

The above algorithm determines whether two interface role interaction policies are related in such a way that the refined role interaction policy can be substituted for the original policy such that the original assumptions—that is, constraints on the input channels—are still satisfied. A simplified example is used to demonstrate how the algorithm works. The following illustration incorporates two role interaction policies projected onto input channels, as well as constraints on the permissible patterns of input streams reflecting the assumptions of the role interface.

Example: Consider the following two role interfaces. The *refinedRole* is an extended version of the original interaction policy, with new message and transition rule augmentations.

interface type Role is interactionpolicy import: export: φ(), γ(); events: in: out: states: X_0, X_1; constraints: ($X_0 \rightarrow$ φ . X_1; $X_1 \rightarrow$ γ . X_0) end interface	interface type refinedRole is interactionpolicy import: export: φ(), γ(), κ(); events: in: out: states: Y_0, Y_1; constraints: ($Y_0 \rightarrow$ φ . Y_1; $Y_1 \rightarrow$ γ . Y_0; Y1$\rightarrow$ (φ | γ | κ) . Y_1) end interface

The application of the safe refinement algorithm, as depicted below, generates and traverses the reeachability graphs of both the refined and original interaction policies to determine if all traces of the original policy are feasible in the refined policy. Furthermore, failure compatibility is also checked for unexpected receptions.

Step by step application of the algorithm:

$R = \{[\{X_0\}, \{Y_0\}, \{\varphi\}]\}$

$R = \{[\{X_0\}, \{Y_0\}, \varnothing], [\{X_1\}, \{Y_1\}, \{\gamma\}]\}$

$R = \{[\{X_0\}, \{Y_0\}, \varnothing], [\{X_1\}, \{Y_1\}, \varnothing], [\{X_0\}, \{Y_0, Y_1\}, \{\varphi\}]\}$

$R = \{[\{X_0\}, \{Y_0\}, \varnothing], [\{X_1\}, \{Y_1\}, \varnothing], [\{X_0\}, \{Y_0, Y_1\}, \varnothing]\}$

=> **Algorithm returns TRUE.**

According to the above laws and associated properties, the decision-procedure checking for these properties indicates that the refined policy may accept additional and unexpected input sequences of requests that the client will not use. It may also eliminate some non-determinism in the original policy by providing fewer possible transitions among states. On the other hand, the original policy can have more states and transitions, as long as they can be explained from the viewpoint of the refined policy. Note that the safe abstraction refinement relationship has to be transitive to assure that the all refinements in the path—from the most abstract interaction policy specifications to the most concrete—comply with the assumption and obligation constraints of the original contract.

Theorem 1: *Safe refinement relationship is a preorder relation*.

Proof: To prove that the above relation is a preorder, we demonstrate that the relation is *reflexive* and *transitive* through proof by construction.

Reflexivity: Given two interaction policies with the same finite state models that have the same states, transitions, and transition relationships, one can immediately confirm the *reflexivity* since (1) $L(q) \acute{I} L(q)$ and (2) $\forall s \hat{I} S^*, A(q,s) << A(q,s)$ where q is the initial state of both of the finite state models.

Transitivity: Assume p, q, and r are the initial states of three finite state models of interaction policies P, Q, and R where Q refines P and R refines Q.

$$L(p) \acute{I} L(q) \text{ and } L(q) \acute{I} L(r) \text{ then } L(p) \acute{I} L(r).$$

For the failures, one can argue that given

$$\forall s \hat{I} L(p), A(p,s) << A(q,s) \text{ and } \forall s' \hat{I} L(q), A(q,s') << A(r,s').$$

Using the definition, **X << Y**, if $\forall S \in Y, \exists S' \in X$ *s.t.* $S' \subset S$, it is possible to claim that the transitivity relationship holds due to the transitivity of subset relation.

4.2.2 Independent Role Interface Obligation Refinement

Section 4.2.1 is devoted to the safe refinement of the assumption constraints of interaction policies. In this section the laws and the decision procedures are tailored to assure that during refinement the contractual output obligation constraints of each role interface are safely extended. Originally, safe interaction policy refinement is defined in Section 4.2 as

$$\textbf{(1) } O_r = O_r'$$

$$\textbf{(2) } I_r \subset I_r'$$

$$\textbf{(3) } \forall i \in I_r \; \forall f \in F(i) : (A_r(i) \Rightarrow \exists i' \in I' \textbf{ s.t. } i=i' \wedge A_r^{refined}(i') \wedge (failure_{refined}(i',f) \Rightarrow failure(i,f))$$

$$\textbf{(4) } \forall o' \in O_r' \; \forall f \in F(o') : G_r^{refined}(o') \Leftrightarrow \exists o \in O \textbf{ s.t. } o=o' \; G_r(o) \wedge failure_{refined}(o',f) \Leftrightarrow failure(o,f)$$

The necessary and sufficient conditions for the safe refinement of *input channel assumption* contracts make use of the rules *(2)* and *(3)* above.

Given the models $M=(Q, S_{[out]}, d_{out}, q)$ and $M_{refined}=(Q_{refined}, S_{refined[out]}, d_{refined[out]}, q_{refined})$ that are projections of the general interaction policies, as well as the above definitions, the laws that delineate the necessary and sufficient conditions of *independent* safe refinement for the obligation constraints are defined. In order to do so, the above rules *(1)* and *(4)* are used as follow:

Law 4: $S_{[out]} = S_{refined[out]}$.

Law 5: (a) $L_{[out]}(q) = L_{[out]}(q_{refined})$.

Law 6 (Law of Failures): $\forall s \in S_{[out]}^{*}, A(q,s) << A(q_{refined},s)$. That is, for every state, *p*, in the refined policy interaction model, there exists a state, *p'*, in the original interaction model, such that the set of permissible output actions from *p'* is a proper subset of the permissible actions from state *p*.

Property 5: $S_{[out]} = S_{refined[out]}$

Property 6: $S(q) = S(q_{refined})$where q and $q_{refined}$ are the axioms or initial states of the original and refined policies, respectively.

Property 7: $\forall\ p'_{refined} \in P_{refined}, \exists\ p \in P$ *s.t.* $S(p) = S(p'_{refined})$

Property 8: $\forall\ a \in S(P)$, *if* $P \rightarrow a\ .\ P'$, $P_{refined} \rightarrow a\ .\ P'_{refined}$ then substitute P by P' and $P_{refined}$ by $P'_{refined}$ in property 3 and also check to see multi-state equivalent of property 2 holds.

Safe Refinement Decision Procedure for Role Interface Obligation Constraints:

Input: $M=(Q, S_{[out]}, d_{out}, q)$ and $M_{refined} =(Q_{refined}, S_{refined[out]}, d_{refined[out]}, q_{refined})$ are the projections of the original and refined role interaction policy finite state models onto output channels.

Output: returns TRUE if the refinement is safe; otherwise returns FALSE.

Algorithm:

R=∅

if $S_{[out]} \neq S_{refined[out]}$ then return false

if $S(q) \neq S(q_{refined})$ then return false

$R = R \cup [\{q\}, \{q_{refined}\}, S(q)]$.

Repeat

$u = [C, C_{refined}, A]$ where $A \neq \varnothing$ and choose an action $a \in A$

$R = R - u$

$R = R \cup [C, C_{refined}, A\text{-}\{a\}]$

Compute C' and $C'_{refined}$ where $C_{refined} \rightarrow a\ .\ C'_{refined}$ *and* $C \rightarrow a\ .\ C'$

if $[C', C'_{refined}, A'] \notin R$ for some A' then

if $S(C') \neq S(C'_{refined})$) then return false

if not($\forall$ p $\in C'_{refined}, \exists$ p' $\in C'$*s.t.* $S(p') \neq S(p)$) then return false

$R = R \cup [C', C'_{refined}, S(C')]$

until $\forall [X, Y, ActionSet] \in$ R, $ActionSet = \varnothing$

The algorithm for verifying safe obligation refinement is a procedure involving simultaneous state reachability analysis. It is similar to assumption refinement, with the exception that the output constraints are stronger than input constraints. That is, it is ensured that the assumptions of associated roles connected to the role under consideration are not violated. Due to the fact that the role under consideration is the only policy changed, it is essential to make sure that the output patterns are compatible with the input channel constraints of the connected role. It is possible that with respect to the original obligation policy, the obligation refinement is not safe; however, the new policy might be accommodated by the refined assumption policies of the connected roles. Hence, the failure of safe role refinement by itself does not preclude the enhancements. In such circumstances, though, it is essential to check role compatibility with respect to the connected role interaction policy.

4.2.3 Association Role (Adaptor) Composition Protocol Refinement

As dictated by its role interfaces, the component under certification could play several roles with respect to its constituent components and its environment. Coordination and mediation of collaborations among the roles and the environment are achieved by the composition protocol.

The composition protocol is based on the same formalism that defines individual role interaction policies. Therefore, the same procedures are used to assure that the refinement is safe and does not violate the established assumptions and obligations of individual role interfaces. The composition protocol refinement rules are defined in Chapter 3:

$$\text{Let } R_M = \{r_1, r_2, \ldots, r_n\}$$

$$\textbf{IP: } (R_M \rightarrow (\text{Messages}^*)^\mu) \rightarrow \textbf{B}, \ \text{Messages}^* \in (\text{messages}(\textbf{I}) \cup \text{messages}(\textbf{O}))$$

$$\textbf{IP': } (R_M \rightarrow (\text{Messages}^*)^\mu) \rightarrow \textbf{B}, \ \text{Messages}^* \in (\text{messages}(\textbf{I'}) \cup \text{messages}(\textbf{O'}))$$

$$[(\forall r \in R_M \text{ and } \forall i_r \in I_r \ \forall o_r \in O_r \ \textbf{IP}_r(i_r) \Rightarrow \textbf{IP'}_r(i_r) \wedge \textbf{IP'}_r(o_r) \wedge \textbf{IP}_r(o_r)) \wedge$$

$$\textbf{IP'}_r \textbf{ failure } (i_r, f) \Rightarrow \textbf{IP}_r \textbf{ failure } (i_r, f) \text{ where } f \in \textbf{F}(\text{messages}(i_r)) \wedge$$

$$\text{IP'}_r \text{ failure } (o_r, f) \Leftrightarrow \text{IP}_r \text{ failure } (o_r, f)] \text{ where } f \in \text{F}(\text{messages}(o_r))]$$

The above rules consider only those transitions relevant and projected onto the input and output channels of the provided and required roles. Projection of a protocol onto a selected subset of input output channels is discussed in Section 4.4. The above rules also indicate that with respect to each associated class role, the message traces of the original protocol should form a subset of the traces for the same role under the refined protocol. The refined protocol may introduce new message traces with respect to the role interfaces as long as the individual policies can tolerate them.

To determine if the protocol refinement is safe, the properties and the laws defined for safe refinement of interaction policy assumptions and obligations are combined. Founded on the same formalism as interaction policies, the composition protocol incorporates not only observable input/output actions but also internal actions and states used in mediating and resolving role interferences. Section 4.4 discusses the method for eliminating these internal actions to obtain a projected finite state model. In this section, the existence of projected protocol state models are assumed, both for the assumptions and obligations of the association role, against each role in R_M. The refinement check for each role $r \hat{I} R_M$ is performed in two steps:

(1) *Projected Assumption Constraints Refinement*: The properties of interaction policy safe refinement are tailored to facilitate the checking of this rule. That is, given the projected models for the assumption constraints $A_r = (Q, S, d, P)$, and $A_r^{refined} = (Q_{refined}, S_{refined}, d_{refined}, P_{refined})$, the following properties are utilized:

1: $S \acute{I} S_{refined}$

2: $S(q) \acute{I} S(q_{refined})$where q and $q_{refined}$ are the axioms or initial states of the original and refined protocols, respectively.

3: $\forall p'_{refined} \hat{I} P_{refined}, \exists p \hat{I} P$ s.t. $S(p) \acute{I} S(p'_{refined})$

4: $\forall a \hat{I} S(P)$, if $P \xrightarrow{} a . P'$, $P_{refined} \xrightarrow{} a . P'_{refined}$ then substitute P by P' and $P_{refined}$ by $P'_{refined}$ in property 3 and also check to see multi-state equivalent of property 2 holds.

(2) *Projected Obligation Constraints Refinement*: The next step involves projection of the adaptor composite policy (i.e., protocol) onto the obligations of the component roles against which the refinement is being performed. Using the obligation refinement rules defined in Section 4.1.2, the projected original policy $G_r=(Q, S, d, P)$, and refined projected policy $G_r^{refined}=(Q_{refined}, S_{refined}, d_{refined}, P_{refined})$ are checked to ensure safe output obligation constraint refinement.

4.2.4 Simultaneous Refinement of Interaction Policies and Association Role (Adaptor) Composition Protocols

Previous notions of refinement focused on independent refinement, which assumes that the role interface or the association role (adaptor) protocol are the only entities extended in a refinement step. In this section, this constraint is relaxed and the implications of simultaneously refining a set of policies are considered.

Analysis of simultaneously refining interaction policies derives from analysis of the refinement of (1) the *assumptions* and (2) the *obligations.* The point of view regarding simultaneous refinement of assumptions is exactly the same as that of independent refinement of role interfaces, since extending the assumptions of an individual role does not impose restrictions on other roles; hence, it is isolated.

As discussed in Section 4.2.2., the point of view regarding refinement of obligations prohibits the addition of new output channels and new output traces. This occurs because under independent refinement, the new traces and channels may not necessarily be acceptable by the receiving connected role interface.

Under simultaneous refinement of role interface *obligations*, the previous constraint is relaxed. However, a new constraint is added to assure that the refined role is consistent with its refined connected role, *cr*. Basically, a two-phase algorithm is used that applies the independent role assumption refinement algorithm, followed by a role compatibility check, discussed in Section 4.3.

$$O_r \subseteq O_r'$$

$$\forall o \in O \; \forall f \in F(o): G_r(o) \Rightarrow \exists o' \in O' \text{ s.t. } o'=o \; G_r^{refined}(o') \wedge failure_{refined}(o',f) \Rightarrow failure(o,f)$$

$$compatible(r, cr)$$ where *c,cr* are directly connected roles.

As discussed in Chapter 3, the simultaneous refinement of the association role requires that the following rules hold:

$$\text{Let } R_M = \{r_1, r_2, \ldots, r_n\} \text{ and } R^{refined}{}_M = \{f_1, f_2, \ldots, f_m\}$$

$$\mathbf{IP: (}R_M \mathbf{\ ®\ (Messages}^*\mathbf{)}^\mu \mathbf{)® B,}\ \ Messages^* \hat{I}\ (messages\ (\mathbf{I}) \grave{E}\ messages\ (\mathbf{O}))$$

$$\mathbf{IP': (}R_M \mathbf{\ ®\ (Messages}^*\mathbf{)}^\mu \mathbf{)® B,}\ \ Messages^* \hat{I}\ (messages\ (\mathbf{I'}) \grave{E}\ messages\ (\mathbf{O'}))$$

$$[(\text{"}\ r\ \hat{I}\ R_M - R^{refined}{}_M \text{ and } \text{"}\ i_r\ \hat{I}\ I_r\ \text{"}\ o_r\ \hat{I}\ O_r\ \mathbf{IP_r(i_r)}\ \text{Þ}\ \mathbf{IP'_r(i_r)}\ \wedge \mathbf{IP_r(o_r)}\ \hat{U}\ \mathbf{IP'_r(o_r)}) \wedge$$

$$\mathbf{IP'_r\ failure\ (i_r,f)}\ \text{Þ}\ \mathbf{IP_r\ failure\ (i_r,f)} \text{ where } f\ \hat{I}\ \mathbf{F}(messages(\mathbf{i_r})) \wedge$$

$$\mathbf{IP'_r\ failure\ (o_r,f)}\ \hat{U}\ \mathbf{IP_r\ failure\ (o_r,f)}] \text{ where } f\ \hat{I}\ \mathbf{F}(messages(\mathbf{o_r}))] \wedge$$

$$[(\text{"}\ r\ \hat{I}\ R^{refined}{}_M, \text{"}\ \ i_r\ \hat{I}\ I'_r, \$\ o_r\ \hat{I}\ O'_r \text{ s.t. } o = i_r \text{ and } i = o_r\ \mathbf{G_r(o)}\ \text{Þ}\ \mathbf{IP'_r(i_r)} \wedge \mathbf{IP'_r(o_r)}\ \text{Þ}\ \mathbf{A_r(i)})]$$

$$\text{with } destination\ (o)\ \text{¹}\ destination\ (i_r) \text{ and } destination\ (i)\ \text{¹}\ destination\ (o_r)$$

Basically, the rule requires that while obligations remain unchanged, the projected and derived policies for unchanged class roles must have weaker assumptions. On the other hand, the assumptions and obligation constraints of derived policies need to be compatible with the refined class roles. The method includes two phases. In the first, for unchanged class roles, role assumption and obligation refinement algorithms are applied to the projected role and the mate class role. Phase 2 demands that for refined class roles, assumption and obligation compatibility must be verified. The role compatibility algorithm is introduced in Section 4.3.

4.3 ROLE INTERFACE COMPATIBILITY FOR FINITE STATE MODELS OF INTERACTION POLICIES

Role interface compatibility is the capability of role interfaces to exchange messages consistent with their interaction policies without unexpected receptions and deadlocks. The challenge is to decide whether a particular role interface is a valid mate of another by checking for signatures as well as interaction policy assumptions and obligations. As component role interfaces are integrated into the abstract collaboration-based design being refined during the development process, it becomes essential to determine whether they fit into the overall framework.

Simply looking at the interface signatures that constitute names and types of the parameters and returned results is inadequate in a context where dynamic interaction policies, non-uniform service availability, and state-dependent behavior are key aspects. Often there are rules constraining the order in which messages are sent and received. For instance, a *file open* call has to precede a *file read* operation. Such rules and interaction patterns can be represented easily by the role interface interaction policies presented in this dissertation.

The role interface compatibility notions introduced in this section focus on sequencing constraints realized as finite state models depicting the interaction policies of role interfaces. Here, the trivial *interface signature* compatibility that constitutes simple syntactic verification of required and provided services is skipped. Hence, it is assumed that such syntactic checks are performed and that the roles under consideration are either interface-compatible or can adapted. Adaptation of an interface means that the component it represents has been wrapped and now possesses the required role interface. Given the interaction policies and constraints of the two connected roles in terms of finite state models $R=(Q, S, d, q)$ and $R_c=(Q_c, S_c, d_c, q_c)$, interaction policy compatibility is defined as follows:

The states of the provided role R and connected role R_c are given as Q and Q_c, respectively. Services and actions defined in the role interfaces are given as S and S_c. The actions and services under S and S_c are decomposed into S^{in}, S^{in}_c to denote input channels, and S^{out}, S^{out}_c to denote output channels. An *interaction state* for the two role interfaces is defined as a pair, $q_{IS} = (s, t)$, where $s \hat{I} Q$ and $t \hat{I} Q_c$. The *interaction sequence* among these two role interfaces is represented as an infinite sequence of interaction states of the form $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_{n-1} \rightarrow \alpha_n \rightarrow \ldots$,where each a_i is an interaction state,

- $\alpha_1 = (q, q_c)$
- $\alpha_{i+1} = (s_{i+1}, t_{i+1})$ where $\alpha_i = (s_i, t_i)$, $(s_i \rightarrow m \,.\, s_{i+1}) \hat{I} d$ (i.e., $d(s_i, m) = s_{i+1}$)

$(t_i \rightarrow m\ .\ t_{i+1})\ \hat{I}\ d_c$ and $[(m\ \hat{I}\ S^{in}$ and $m\ \hat{I}\ S^{out}{}_c\ )$ or $(m\ \hat{I}\ S^{out}$ and $m\ \hat{I}\ S^{in}{}_c)]$. That is, if m is *send message* for role interface R then it should be *receive message* for R_c or vice versa.

If we let $Interactions(R, R_c) = \{\alpha \mid \alpha$ is an interaction sequence$\}$ then the interacting role interfaces are free of unexpected receptions; hence, they are role compatible if in $\forall$ a $\hat{I}$ $Interactions(R, R_c)$ where a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_{n-1} \rightarrow \alpha_n$ and $a_n = (s_n, t_n)$, the following holds:

Property 1: if $s = s_n$, $(s \rightarrow m\ .\ s')\ \hat{I}\ d$ and $m\ \hat{I}\ S^{out}$, then there exists an interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (s', t')$ for some $t'\ \hat{I}\ Q_c$.

Property 2: if $t = t_n$, $(t \rightarrow m\ .\ t')\ \hat{I}\ d_c$ and $m\ \hat{I}\ S_c{}^{out}$, then there exists an interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (s', t')$ for some $s'\ \hat{I}\ Q$.

According to the above properties of the interaction state and the sequence of the connected role interfaces, the two roles can be compatible only in specific situations. For example, they are compatible when the interaction sequence reaches a state where R is in a state to receive message m, and the connected role R_c will be in a state where it can receive that message. This means that there is an interaction history, and at that point the message can be exchanged.

The goal in formulating these properties is to lay out the foundations of an algorithm that will decide whether two role interfaces are compatible. The following argument demonstrates the existence of an algorithm to compute this decision.

Let the set of composite interaction states that denotes the compatible states of the interacting roles be a subset of the Cartesian product of the set of states of individual role interaction policy finite state models. Hence, $COLLAB(R, R_c)\ \acute{I}\ Q$ X Q_c, is defined as the set of interaction states that are role compatible for R and R_c as follows:

- ***(q, q_c) Î COLLAB(R, R_c)***
- **if *(s,t) Î COLLAB(R, R_c),* (*s ® m . s') Î d* , (*t ® m . t'·) Î d_c*, [(*m Î S^{in}* and *m Î $S^{out}{}_c$*) or (*m Î S^{out}* and *m Î $S^{in}{}_c$*)] then *(s,t) Î COLLAB(R, R_c).***

The above steps implicitly provide an algorithm that will compute the permissible role interaction states to facilitate determining whether any given interaction state is in the computed set. A finite state space search algorithm is utilized to construct *COLLAB(R, R_c)*. The approach tailors the core-joint state traversal algorithm used in sections 4.2.1 and 4.2.2 to traverse the state spaces of interaction policies; it also determines by implicitly building the states of *COLLAB(R, R_c)* whether the above properties prove satisfactory.

Role Compatibility Decision Algorithm

Input: *R=(Q, S, d, q)* and *R_c=(Q_c, S_c, d_c, q_c)*

Output: TRUE if the roles are compatible; **FALSE** otherwise.

R=∅

if [(S_{in} ¹ $S^{out}{}_c$) or (S_{out} ¹ $S^{in}{}_c$)] then return false

if not [($S_{in}(q)$ Ê$S_{out}(q_c)$) and ($S_{in}(q_c)$ Ê $S_{out}(q)$] then return false

R = R È [*{q}, {q_c}, S(q)*]

Repeat

u = [*T*, T_c, *A*] where *A* ¹ Æ and choose an action *a Î A*

R = R – u

R = R È [*T*, T_c, *A-{a}*]

Compute *T'* and *T'$_c$* where T_c ® *a . T'$_c$ and T® a . T'*

if [*T'*, *T'$_c$*, *A'*] ∉ *R* for some *A'* then

if not [($S_{in}(T')$ Ê $S_{out}(T'_c)$) and ($S_{out}(T')$ Í $S_{in}(T_c)$)] then return false

if not[(∀ p Î *T'$_c$*, ∃ p' Î *T' s.t.* $S_{in}(p)$ Ê$S_{out}(p')$) and

(∀ p' Î *T'*, ∃ p Î T_c' *s.t.* $S_{in}(p')$ Ê $S_{out}(p')$)] then return false

R = R È [*T'*, *T'$_c$*, *S(T')*]

until ∀ [*X, Y,ActionSet*] Î *R, ActionSet* = ∅

The above algorithm ensures that the input and output channels of role interfaces are compatible. That is, it determines whether role interfaces define the input and output channels expected or generated by the other mate, decides whether initial interaction states are compatible, and assures that every state reachable from a compatible state is interaction and failure compatible.

Role compatibility ensures that the associated roles are interaction consistent. However, unless the refinement preserves compatibility as component roles are refined, it would be essential to ensure that the associated roles remain interaction compatible. The following theorem states that as interaction roles are refined, it would be unnecessary to check for role compatibility, since that state would be ensured by safe role refinement.

Theorem 2: *Role refinement preserves role compatibility.*

Proof: Role refinement is defined in terms of assumption and obligation refinement. Suppose that finite state models $M=(Q, S, d, q)$, $M_{refined}=(Q_{refined}, S_{refined}, d_{refined}, q_{refined})$, and $M_c=(Q_c, S_c, d_c, q_c)$ represent the original interaction policy, refined policy, and the connected role interaction policy, respectively. It is further assumed that the original interaction model, role M, is role compatible with the connected role M_c. The following demonstrates that safe obligation and assumption refinement implies role compatibility. That is, assuming that $M_{refined}$ is a safe refinement of the original model M, it can be shown that M_c and $M_{refined}$ are role compatible. Assumption refinement requires the following laws to hold:

Laws 1,4: $S_{[in]} \acute{I} S_{refined[in]}$ and $S_{[out]} = S_{refined[out]}$.

Laws 2,5: (a) $L_{[in]}(q) \acute{I} L_{[in]}(q_{refined})$ and $L_{[out]}(q) = L_{[out]}(q_{refined}).$.

Laws 3,6 (Law of Failures): $\forall s \hat{I} S_{[in]}^{*}, A(q,s) << A(q_{refined},s)$ and $\forall s \hat{I} S_{[out]}^{*}, A(q,s) << A(q_{refined},s)$. That is, for every state, p, in the refined policy interaction model, there exist a state, p', in the original interaction model such that the set of permissible actions from p' is a proper subset of the permissible actions from state p.

Role compatibility between M and M_c requires the following properties to hold between the state spaces of the interaction policies: If we let $Interactions(M, M_c) = \{\alpha \mid \alpha$ is an

interaction sequence*}* then the interacting role interfaces are free of unexpected receptions and hence are role compatible if $\forall$ a $\hat{I}$ *Interactions*(M, M_c) where a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_{n-1} \rightarrow \alpha_n$ and $a_n = (s_n, t_n)$, the following holds:

Property 1: if $s = s_n$, $(s \rightarrow m \,.\, s') \hat{I} d$ and $m \hat{I} S^{out}$, then there exist an interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (s', t')$ for some $t' \hat{I} Q_c$.

Property 2: if $t = t_n$, $(t \rightarrow m \,.\, t') \hat{I} d_c$ and $m \hat{I} S_c^{out}$, then there exist an interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (s', t')$ for some $s' \hat{I} Q$.

Since M and $M_{refined}$ are trace and failure compatible, for each $s = s_n$, $(s \rightarrow m \,.\, s') \hat{I} d$ and $m \hat{I} S^{out}$, there exists a sequence of actions and services that brings the refined policy to a state v such that $(v \rightarrow m \,.\, v') \hat{I} d_{refined}$ holds. Otherwise, the policies would not be trace equivalent. Since the connected policy is not modified, property 1 holds. Similarly, according to property 2, $t = t_n$, $(t \rightarrow m \,.\, t') \hat{I} d_c$ and $m \hat{I} S_c^{out}$, there exists an interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (s', t')$ for some $s' \hat{I} Q$. According to the trace and failure equivalency law of the safe refinement, there must exist a state w in the state space of the refined policy such that $w \rightarrow m \,.w'$, after processing the same sequence that the original policy processes. And any failure of the new refined policy is already known to be a failure of the original policy as well. Hence, the refined policy is capable of participating in the interaction sequence a = $\alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_3 \rightarrow \alpha_4 \rightarrow \ldots. \alpha_n \rightarrow \alpha_{n+1}$ such that $a_{n+1} = (w', t')$ for some $w' \hat{I} Q_{refined}$..

4.4 ASSOCIATION ROLE CONSISTENCY AND INTERACTION INVARIANT SATISFIABILITY

Protocol consistency refers to the compatibility of the composition protocol with respect to individual interaction policies of the role interfaces constituting the component under certification (i.e., provided/required role interfaces). Protocol consistency analysis is a key component of assessing whether protocol refinement is still congruent with the

expectations of the individual role interfaces. In the case of simultaneous role interface refinement, it also facilitates determining if any of the role interfaces violate protocol expectations. Note that independent role refinement is automatically consistent with the protocol due to Theorem 2.

Protocol consistency is very similar in concept to role interface compatibility. With respect to a specific role interface, the analysis includes an additional step for projecting protocol constraints onto a finite state model containing interactions. That is, the analysis extracts from the general composition protocol the obligation and assumption constraints imposed by the protocol with respect to the chosen role interface. The projected finite state model is checked against the role interface interaction policy to determine if the two are role compatible.

Formally, in Chapter 3 the consistency check is defined as follows: Let the composition protocol interaction rule structure, as defined in the abstract collaboration, be $C_{Protocol}$. Let also the projection of the protocol onto the required and provided services with respect to role R be defined as $\prod_R^{Protocol}$. The projection consists of the required and provided services and events, as well as derived interaction policies with respect to R. Given the projection, let the *derived* assumption and obligation constraints of the projection with respect to the role R be $A_{\tilde{o}}(i)$ and $G_{\tilde{o}}(o)$, where i and o are elements of the domain of input and output channel stream valuations, respectively.

$$RS_{\tilde{O}\,RProtocol} \subseteq PS_R \text{ and } RS_R \subseteq PS_{\tilde{O}\,RProtocol}$$

$$\forall o \in O_{\tilde{O}},\ G_{\tilde{O}}(o) \Rightarrow \exists i \in I_r\ A_r(i)$$

$$\forall o \in O_r,\ G_r(o') \Rightarrow \exists i \in I_{\tilde{O}}\ A_{\tilde{O}}(i)$$

with $messages(o)=messages(i)$ **and direction (**$messages(o)) \neq direction (messages(i))$

The procedure for checking protocol consistency involves three steps. The first step identifies services and actions in the protocol that either use or expect the services and actions provided by the role under consideration. Step 2 projects the composition protocol onto a finite state model that incorporates only those actions and services selected in the previous step. Finally, Step 3 tests role interface compatibility between the two finite state models. The first step is trivial given the role interface model and its

input/output channels. The third step is also discussed in the previous section. Therefore, in this section, we focus on Step 2 and introduce the procedure that facilitates the mapping of the protocol (i.e., association role) onto a reduced order finite state model (i.e., association end role) that entails the interaction policies with respect to the specific role interface. The following sections establish the steps of the procedure.

4.4.1 Î-instantiation of the Association Role Composition Protocol

Î-instantiation refers to the process of replacing every transition on the finite state model, other than the identified relevant actions and services, with an $\hat{I}$ transition to create a new, non-deterministic, finite state model.

Let the original finite state model of the protocol be $M_{protocol}=(Q_p, S_p, d_p, q_p)$ and the specific role interface finite state model under consideration be $M_{role}=(Q_r, S_r, d_r, q_r)$. Those actions and services of the protocol that utilize and expect services originating from and going into the role interface are identified as S_P , depicting the projected channels. The rest of the protocol transitions defined under the protocol, $S_p - S_P$, are substituted with $\hat{I}$ transitions. The resulting *Î-instantiated* finite state model is represented by

$$\mathbf{M'_{protocol} = (Q_p, S_P \grave{E} \{\hat{I}\}, d'_p, q_p)\ where}$$

$$\mathbf{\forall\ s \hat{I} Q_p\ \forall\ a \hat{I} S_P,\ d'_p(s,a) = d_p(s,a)\ and\ \forall\ a' \hat{I} S_p - S_P\ s.t.\ d_p(s,a')=s'\ define}$$

$$\mathbf{d'_p(s,\hat{I})=s'}$$

4.4.2 Reduced-Order Finite State Model of the Composition Protocol

The *Î-instantiated* finite state model, $M'_{protocol} = (Q_p, S_P \grave{E} \{\hat{I}\}, d'_p, q_p)$, includes all transitions but those irrelevant to the role interface under consideration. Formally, the *Î-instantiated* finite state model defines a *non-deterministic finite automaton with Î-moves* with the transition relation that maps Q_p x $S_P \grave{E} \{\hat{I}\}$ to 2^{Qp} .

To compute trace and failure compatibility, the key components of role compatibility, it is crucial to obtain a non-deterministic *reduced order* finite state model, $M_{\tilde{O}}=(Q_{\tilde{o}}, S_P, d_{\tilde{o}}, q_{p,\hat{}})$, of the projected protocol without $\hat{I}$ moves. This is essential, since the finite state space traversal is performed based on the joint actions and services

of the role interfaces. The intention is to extend the transition function d'_p to $d_{\tilde{O}}$ that maps Q_p x S^*_P to $2^{Q\tilde{O}}$. Therefore, $d_{\tilde{O}}(q,w)$ will contain all states p such that one can reach from q to p along a path labeled w. In constructing $d_{\tilde{O}}$, it is important to compute the set of states reachable from a given state q using only $\hat{I}$ transitions. This question is equivalent to the question of what vertices can be reached from a given source vertex in a directed graph. In the finite state model, the source vertex is the vertex in state q, and the directed graph in question consists of all and only the arcs labeled $\hat{I}$. One can use $\hat{I}$-*TransitiveClosure(q)* to denote the set of all vertices p such that there is a path from q to p labeled $\hat{I}$.

Considering the non-deterministic and the multi-state nature of the core state space traversal algorithm, the multi-state equivalent of $\hat{I}$-*TransitiveClosure* is defined as $\hat{I}$-*TransitiveClosure(P)*, where P is a set of states, be $\dot{E}_{q \text{ in } P}\, \hat{I}$-*TransitiveClosure(q)*.

Now, one can define $d_{\tilde{O}}$ for the reduced-order finite state model $M_{\tilde{O}}=(Q_{\tilde{O}}, S_P, d_{\tilde{O}}, q_p)$ as follows:

1. $d_{\tilde{O}}(q, \hat{I}) = \hat{I}$-**TransitiveClosure(q).**
2. **For** w **in** S^*_P **and** a $\hat{I}$ S_P **,** $d_{\tilde{O}}(q,wa) = \hat{I}$-**TransitiveClosure(P), where** $P=\{p$ **| for some** r **in** $d_{\tilde{O}}(q,w)$, p **is in** $d'_p(r,a)\}$.

For the reasons depicted above, it is convenient to extend the new transition relation. Hence, $d_{\tilde{O}}(R,w) = \dot{E}_{q \text{ in } R} d_{\tilde{O}}(q,w)$ is defined. Note that $d_{\tilde{O}}(q,a)$ is not necessarily equal to $d'_p(q,a)$, since $d_{\tilde{O}}(q,a)$ includes all states reachable from q by paths labeled a (including paths reachable through arcs labeled by $\hat{I}$), while $d'_p(q,a)$includes only those states reachable from q by arcs labeled by a. Similarly, $d_{\tilde{O}}(q,\hat{I})$ is not necessarily equal to $d'_p(q,\hat{I})$.

Once the reduced-order finite state model, $M_{\tilde{O}}=(Q_{\tilde{O}}, S_P, d_{\tilde{O}}, q_p)$, is created, the next step is to determine role interface compatibility to assure that the projected protocol is consistent with the role under consideration.

4.4.3 Protocol Consistency and Invariant Satisfiability

As discussed above, protocol consistency analysis is equivalent to testing for role interface compatibility. The reduced-order finite state model, $M_{Õ}=(Q_{Õ,}, S_P, d_{Õ,}, q_p)$ and the interaction policy model of the role under consideration are checked to determine whether their interaction assumptions and obligations are mutually compatible. That is, using the algorithm introduced in section 4.3, the following properties are checked for conformance:

$$RS_{Õ\,RProtocol} \acute{I} PS_R \text{ and } RS_R \acute{I} PS_{Õ\,RProtocol}$$

$$" o \hat{I} O_{Õ,}\ G_{Õ}(o) Þ \$ i \hat{I} I_r\ A_r(i)$$

$$" o \hat{I} O_{r,}\ G_r(o') Þ \$ i \hat{I} I_{Õ}\ A_{Õ}(i')$$

with $messages(o)=messages(i)$ **and direction (**$messages(o)) \neq direction (messages(i))$

Here, $G_{Õ}(o)$ and $A_{Õ}(i)$ denote the obligations on the output channels and assumptions for the input channels of the projected model, respectively. $A_r(i)$ and $G_r(o)$ denote the obligations and assumptions of the role interface under consideration. The compatibility is determined on the basis of whether the obligations of the projected protocol satisfy the assumptions of the role interface, and vice versa.

The above strategy can be applied as well to a variety of user-defined properties and invariants defined as finite state models. Here the focus is on invariants, but the same concept applies to user-defined properties, as long as these properties are defined in terms equivalent to the formalism defining invariants.

An *invariant* is considered as a design property that must be satisfied by every composition protocol throughout all refinement steps, from the most abstract level to the lowest level abstraction. Invariants and their usage in the design-by-contract paradigm are quite popular; they are also widely advocated under object and component-based development paradigms. Traditionally, invariants define the conditions on the values of the instance variables; that is, the state of the objects, that need to be true throughout the lifetime of an object. The invoked methods must conserve the state of the object such that the invariant condition is satisfied at all times.

One can consider an invariant as a role collaboration pattern that must be supported by the composition protocol. Note that the protocol consistency considers the mutual compatibility both of the composition protocol and of a particular policy regarding role interface interaction. On the other hand, invariants can characterize the expected pattern in which individual role interfaces are enacted. That is, invariants consider horizontal integration aspects of role composition, as opposed to vertical integration of the provided/required role interaction policies and the composition protocol. However, they do not cover the integration dependencies among the role interfaces of the constituent components. Such constituent role integration dependencies are discussed in Chapter 5 under the concept of causal process constraints.

Each invariant is assumed to be specified as a finite state model, $M_{INV}=(Q_{INV}, S_{INV}, d_{INV}, q_{INV})$. It is also fair to assume that it might include cross-cutting aspects of interaction among several role interfaces to depict the rules that govern how individual role interfaces are enacted. The steps for *invariant satisfiability* number three. Step 1 identifies services and actions in the protocol that use or expects specific services and actions provided by the specific roles under consideration. Step 2 projects the composition protocol onto a finite state model that incorporates only those actions and services selected in the previous step. Finally, Step 3 performs a safe refinement analysis to demonstrate that the derived reduced-order finite state model is a safe refinement of the user-provided invariant model. Steps 1 and 2 have been discussed previously in the context of protocol consistency. Step 3 of the invariant satisfiability problem is equivalent to protocol refinement , and it is tailored to invariant satisfiability as follows.

Two predicates P_{INV} and $P_{\tilde{O}}$, for finite state models $M_{INV}=(Q_{INV,}, S_{INV}, d_{INV,}, P_{INV})$ and $M_{\tilde{O}}=(Q_{\tilde{O},}, S_P, d_{\tilde{O},}, P_{\tilde{O}})$, are defined to denote the mappings from the input/output channel streams to Boolean. The invariant model and the reduced-order finite state model derived from the composition protocol do not distinguish between input and output channels. This is because both models cross-cut across the role interfaces and hence input and output channel distinction is not relevant. Using the predicates, P_{INV} and $P_{\tilde{O}}$, the *invariant satisfiability* is formalized as

$$P_{INV}: (I \rightarrow O) \rightarrow B, P_{\tilde{o}}: (I \rightarrow O) \rightarrow B$$

$$\forall i \in I, \forall o \in O \forall f \in F_{\tilde{o}}(i/o), P_{INV}[i,o] \triangleright P_{\tilde{o}}[i,o] \wedge failure_{\tilde{o}}[i/o,f] \triangleright failure_{INV}[i/o,f].$$

The properties checked by the decision procedure to determine if the above formulation holds are equivalent to the properties depicted in Section 4.2.3 are as follow:

1: $S_{INV} \sim S_{\tilde{o}}$

2: $S(q_{INV}) \sim S(q_{\tilde{o}})$where q_{INV} and $q_{\tilde{o}}$ are the axioms or initial states of the invariant and derived protocols, respectively.

3: $\forall p'_{\tilde{o}} \in P_{\tilde{o}}, \exists p \in P_{INV}$ s.t. $S(p) \sim S(p'_{\tilde{o}})$

4: $\forall a \in S(P_{INV})$, if $P_{INV} \rightarrow a \,.\, P'_{INV}$, $P_{\tilde{o}} \rightarrow a \,.\, P'_{\tilde{o}}$ then substitute P_{INV} by P'_{INV} and $P_{\tilde{o}}$ by $P'_{\tilde{o}}$ in property 3 and also check to see multi-state equivalent of property 2 holds.

The decision procedure that determines whether the above properties are satisfied by the reachable state spaces of the involved finite state models is introduced in Section 4.2.3 and is not repeated here.

CHAPTER 5

HOW COMPONENTS ARE INTENDED TO WORK and EVOLVE: LOCAL INTERACTION POLICY TESTING and CAUSAL PROCESS ANALYSIS FOR GLOBAL CONSISTENCY

When constructing systems from reusable components, an engineer is faced with the problem of dealing with potential inconsistencies and architectural mismatches due to conflicting dynamic interaction patterns. Chapters 3 and 4 laid out the principles of formalization and demonstrated how to reason about local interaction aspects early and at a high level of abstraction. In this chapter, the focus is on the *implements* relationship between a concrete component and its abstraction. That is, proper run-time methods are introduced to demonstrate that the rules and constraints on the individual role interaction policies, as well as the emergent causal processes, are honored.

How components are configured and composed determines the organizational behavior that will emerge from the conglomerates of local interactions. Due to the state space explosion problem, deriving from the set of local interactions a causal behavioral model depicting the computational paths is very expensive. Hence, a new technique, which can be categorized as *hybrid simulation verification*, is developed. By synthesizing an intermediate analysis method, this technique combines the best of both worlds of simulation and verification. It uses test scenario execution to simulate the component with a test driver that will emulate the context of the component under certification. Test scenario execution also limits the generation of causal computation graphs to the computations consistent with the simulation. This limited computation graph, called the *causal behavior model*, can include orders of magnitude smaller than the potential graph that could be derived from analyzing the local interaction policies of roles.

To meet all specified requirements, development with reuse involves a cycle of component acquisition, adaptation, assembly and evolution. Taking this into account, as the constituent components are continuously adapted, the *implements* relationship constitutes *safe evolution* operators to assure that the component evolves as intended. New constraint operators are introduced to represent the intention of the evolution. These operators, as well as the causal process constraints briefly introduced in Chapter 3, are discussed in detail in this chapter.

In this work, two main types of constraints are the key foundations: (1) role interaction policies and (2) causal process constraints. Correct implementation of the abstract component requires that the concrete component under certification, its environment, and its constituent components interact in compliance with their individual role interface interaction policies. Furthermore, the causal process underlying the component—that is, the interactions among the constituent components and the way the functionality is delivered—becomes a key concern in the context of reuse. Each reused component has an organizational role in the overall framework, and the assigned roles are implicitly captured by the way constituent components interact within their context. Capturing the causal process and reasoning about the patterns in it become useful to designers who constantly use functionality to determine if reused components conform to required scenarios. In development without reuse, the component interfaces and their behaviors are designed and implemented from scratch around the use-cases, which ensures that they will fit more effectively into given interaction scenarios. This is unfortunately not the case in development with reuse. To determine if the concrete component is a faithful implementation of the abstraction and that constituent components actually play their intended roles, an *interaction testing* approach along with a *hybrid simulation-verification* method are introduced to reason about compliance of the causal process. To this end, in this chapter, the means for specifying the expected causal processes, along with the associated constraint language is described in more detail.

This chapter is organized as follows. Section 5.1 introduces potential general implementation strategies for interaction protocols and protocols. Section 5.2 discusses the test suite generation algorithm, as well as the generation of composite interaction state

automata for interface violation detection. Sections 5.3 and 5.4 focus on causal process capture, behavioral model generation, and constraint analysis.

5.1 IMPLEMENTATION OF COMPONENT PROTOCOL AND INTERACTION POLICIES

Protocols and interaction policies serve as useful abstractions of how components interact. Several mechanisms exist that can map such abstraction onto programmatic APIs used by programmers; hence, this abstraction can be implemented in several ways.

The component sending or receiving a message may view the message as (a) an asynchronous message, (b) a blocking call until delivery, or (c) a synchronous call with return. In asynchronous messaging, after sending the message, the sender immediately returns to the computation. Actor-based systems, Java Messaging System libraries (JMS), CORBA, and one-way IDL supporting languages are examples of systems that enable asynchronous messaging. In synchronous call with return, the sender sends the message (i.e., call a procedure or method) and blocks until the receiver finishes the computation and returns. Part (b) defines an intermediate semantics where the sender blocks until the recipient receives the message but before the result is returned. Such interaction mechanisms utilize proxy objects that block on behalf of the sender and also enable the sender to query whether the result is available. The messages received by a component might be (a) queued until requested, (b) immediately invoke method handlers, or (c) serially invoke handlers as dispatched by the receiving end. To implement queued messaging, the environment needs to make queues explicitly addressable. Many RPC and distributed object systems support (b) and (c). For instance some implementations of CORBA provide single or multi-threaded object skeletons (i.e., the causal process case study). In the former case the server queues up incoming calls and dispatches them serially; in the latter, the server skeleton starts a new thread to handle each incoming message.

Components implementing the protocols and interaction polices can be either active or passive. Most passive component implementations do not explicitly represent the synchronization aspects unless they provide special generic synchronized components (i.e., java synchronized components). On the other hand, active components—in

particular, task sequencing and rendezvous-based objects—have autonomous threads of control. Protocols that contain states wherein both parties can simultaneously send and synchronize require active components.

In this research, due to the decidability issues discussed in chapter 3, synchronous interaction semantics is utilized. Although the developed methods and techniques target synchronous message passing, they are also applicable to the intermediate proxy-based interaction semantics. To this end, the utility and the applicability of the developed approach are demonstrated by using the Synchronous Java platform, which provides an active object concept with its own thread of control. An active object acts as a task with non-uniform service availability to make interaction policies and constraints explicit. The test bed is further discussed in detail in Chapter 6.

Roles can be implemented in a variety of ways. They can be implicitly embedded in the global state model of the component under certification, implemented with a separate thread of control, or interwoven with the component, as in *aspect-oriented* programming. As discussed in Kendall [2000], the role-object pattern serves as a potential micro-architecture for assembling roles provided and required by a component. The pattern omits details of implementing the roles. Roles can be implemented via passive components or active ones with state-dependent behavior and non-uniform service availability. To demonstrate the methods developed by the research presented here, active components that have explicit interaction and rendezvous constraints are utilized. The composition protocol can similarly be implemented by the main component as a separate, finite state control loop or an active thread of control inside the components that implement the association roles of the generic collaboration patterns.

5.2 ROLE INTERACTION TESTING: INTERACTION TEST SEQUENCE GENERATION AND INTERACTION POLICY VIOLATION DETECTION

The abstract component constitutes two main sections, focusing on the interaction policies and causal processes that represent the organizational rationale. The interaction policies depict constraints on the interaction patterns of the individual components delivering and using, respectively, the provided and required roles. The environment role interaction policy and provided role interface of the component are combined to derive

test scenarios that facilitate the development of a test driver. Section 5.2.1 introduces the *interaction policy test* infrastructure components, while Section 5.2.2 introduces the interaction test sequence generation mechanism.

5.2.1 Local Interaction Policy Validation Infrastructure

A software component or system is run through different operational scenarios to test for deviations from expected results. The viewpoint or aspect of a system's characteristics under scrutiny is the key influential factor in deciding the form and basis of the test strategy and tactics.

As opposed to data input/output or state change inspection via message sequences, the interaction perspective requires an infrastructure that focuses on evaluating the correctness of the interaction patterns. The state dependent behavior is defined in terms of an interaction state. On the contrary, most existing techniques, even the state-based testing techniques reviewed in Chapter 2, deal with internal data state change analysis. To drive components according to their interaction policies, the introduced approach takes into account both input/output channels and policy violations.

The infrastructure is composed of (1) the environment model, (2) the component under certification, (3) test sequences, (3) components, and (4) violation detectors. The environment model is a component that generates and receives messages according to the required role interaction policy. The violation detector connects the component that models the environment to the component under development.

A violation detector incorporates the implementation of a *composite interaction finite state model*, discussed later in this chapter. It mediates and records incoming and outgoing messages and checks for policy violation. The interaction violation detector provides two interfaces: the environment and the component under certification. It acts as the environment against the component, and vice versa. The component under certification receives messages and regulates the enactment of component services that implement the provided and required roles.

As shown in Figure 5.1, the component under certification implements protocols that mediate interactions among the constituent roles. A violation detector exists between each component to implement the provided role and its connection. Composition protocol

implementation refers to the rules that invoke the proper methods in the expected order to govern enactment of the roles. Such rules entail the order, sequencing, and synchronization patterns involved in invoking the components.

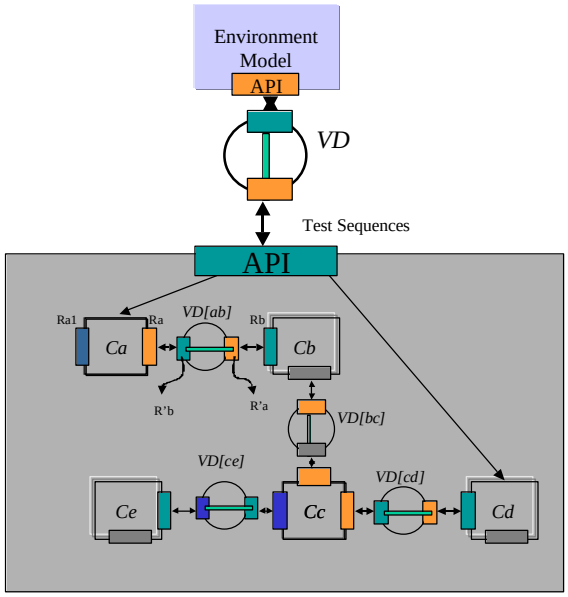


Figure 5.1: Component Under Certification

Next, the issues of test sequence derivation and composite interaction policy derivation are introduced. The composite interaction policy constitutes the interaction control loop of the violation detector between the roles. The test sequence derivation is used to generate message send and receive sequences in compliance with the component's interaction policy

5.2.2 Interaction Test Sequence and Violation Detector Generation

To facilitate test driver implementation, the test sequence generation is automated. The main utility of the test driver is to activate the actual component using the test sequences, along with mechanisms that detect violations as the constituent components interact with each other. Given the expected interaction policy of the environment of the component, $M_{ENV}=(Q_{ENV}, S_{ENV}, d_{ENV}, q_{ENV})$, to facilitate test sequence generation using graph traversal concepts, the model is transformed to a graph, $G=(V,E)$, where V denotes the set of nodes and E denotes the set of edges. The transformation is performed with the following steps:

Set V = Q_{ENV} and

*"**q, a, q' s.t.** d_{ENV}**(q,a) = q' let (q,q')** Î **E*** with the label ***a.***

Once the state connection information constructs the adjacency matrix of the graph, a depth-first search is performed to generate the depth-first tree representing reachability information on the state space. Next, the depth-first tree is traversed to generate test cases. Each root-to-leaf path is reported as a test case. The algorithm shown in Figure 5.2 traverses the graph to generate a depth-first tree that keeps track of the reached nodes and the path that leads to their traversal. The complexity is linear on the size of the graph; that is, its time complexity is *O(V+E).* Given the depth-first tree, the following recursive algorithm generates test sequences by simply identifying the root-to-leaf paths and storing each path as a sequence of nodes, denoting the message sequences for each test case. Note that different search strategies yield different test sequences. The approach utilized in this research covers all specified interaction policy states. Graph theory and the variety of search strategies could facilitate generation of sequences that cover all transitions, all terminating paths, and so on. However, such approaches are beyond the scope of the main trust of the research, which requires identification by some means of the test sequences involving the message send and receive operations.

```
Algorithm: DFSG(G)
Input: G(V,E): Input graph

Output: T: Depth First Tree of the state space
for each u Î V do
  visited[u]=false
for each u Î V do
   if visited[u]=false then DFS(u)
end.

DFS(u)
  visited [u] =true, I=0;
  for each v Î Adj[u] do
      if visited[v]=false then
          child[u,I]=v, trans[u,I]=x where (u,v) Î E with label x
          I=I+1
          DFS(v)
end.
```

Figure 5-2: Depth-First Search of the Interaction State Space

Algorithm TSG(T)

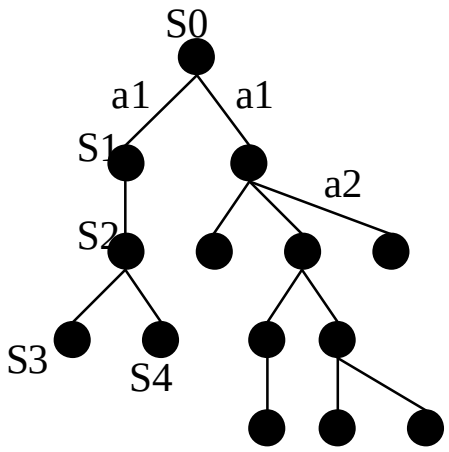
Input: Depth First Tree: T
Output: Test sequences

TS = NULL
x=root(T);
TSGen(*TS,x,0*)
return *TS*
end.

Algorithm TSGen(TS, r,j)

if *r=NULL* then return *j*
else
if *childlist[r]*=NULL and *TS = {}* then
TS[j]=TS[j] È {init}
return *j*
else
for each child node *rc* of *r* do
if *rc* is the first child then
temp=TS[j]
TS[r]=TS[r] È {t}where d[r,t]=rc
next=TSGen(TS, rc, j)
else
j=next+1;
TS[j]= temp
*TS[j]=TS[j] È {t}*where*d[r,t]=rc*
next=TSGen(TS, rc, ,j)
end.

Figure 5-3: Test Sequence Generation



Test Derivation Tree

The *test derivation tree* is the reachability tree derived from the interaction policy rule base. Each node and edge in the tree represents a state and permissible interaction, respectively. Non-determinism in the policy representation due to incomplete information, such as lack of parameter values, would lead to branches with the same labels. The test engineer must identify semantic constraints on the parameter domain and state space to derive complete test sequences. Once the test sequences are identified, they are encoded into test objects. Each test object is an active synchronous java object that embeds the identified interaction sequence.

Figure 5.4: Test Derivation Tree

Section 5.2.1 introduced interaction violation detectors to ensure that the components that implement the required and provided role interfaces follow the imposed interaction

policies. The violation detector must check the compliance of the incoming and outgoing messages to the *assumption* and *obligation* constraints of the roles. This requires it to track the permissible composite interaction states for both of the roles being mediated. The basic approach for composing individual interaction state models is to generate the product of the state models. However, this method results in an unnecessary number of states, most of which are not feasible. Instead, the approach incorporates an incremental composition method and reduction algorithm built on reachability computations. That is, a composite model is generated stepwise. Starting with the initial state, all reachable states and transitions are computed simultaneously for both interaction models, and the composite model is generated during the state space traversal.

The following algorithm traverses the state space and generates a new composite interaction model where the state set is at least the reachable subset of ***P X C***, where ***P*** is the set of baseline interaction policy states and ***C*** is the set of associated role interaction states.

Algorithm: Composite interaction policy state space generation
Input: P0, C0: initial states of the interaction policy specs.
Output: Composite interaction state model used during test suite execution.
Verify *init(C0) = init(P0)* else fail
else
{
L = [{P0}, {C0}, init(C0)]
Set <*P0,C0*> as the initial state of the composite interaction model
repeat
if *$ u Î L* where *u = [P,C,R]* and *R ¹ 0*
{
u = [P,C, R]
L=L-u
L=LÈ [P, C, R-{a}] for some *a Î R*
Compute *P'* where *P ® a . P'*
Compute *C'* where *C ® a . C'*
If [*P', C', R'*] *Ï L* for some *R'*
create a new state <*P',C'*> and update the adjacency matrix
by *d [<P,C>, a]= <P',C'>*
L=LÈ [P', C', S(P')]
}
until *" u Î L* where *u = [P,C,R], R = 0*
}

The output of the algorithm is a new composite state machine representing the interaction model of a single interaction policy for a given connection. Note that there can be more than one interaction policy between the baseline component and the connected component. If interference exists between the interaction policies, the test engineer is responsible for developing a composite interaction policy for the interfering interaction channels that will cover both the connection roles and the baseline interaction policies.

The generated composite interaction state model provides an abstract model of the mutual interaction commitments of the involved role interfaces. For the given role interaction policies, each violation detector embeds the finite state model generated by the above algorithm. Then, using the design pattern to be introduced in Chapter 6, the violation detector connects with the components implementing the interaction policies from which the model is generated. Once connected, the interaction violation detector generates the required messages in response to received messages and sends them to the target component in compliance with its interaction policies. The implementation aspects of the interaction adaptor are discussed in Chapter 6.

Note, however, that the interaction violation detector simply acts as an adaptor to provide a protocol for mediating interactions between the involved components. The adaptor's main goal is to perform violation detection. This can be performed in two ways. In the first method, the derived composite protocol constitutes only the permissible interaction patterns; hence, the implementation of the protocol used in mediating the interactions can in its current state reject any unexpected message. Under the second method, for each of the roles involved in the interaction, the adaptor can be augmented with additional roles for tracking compliance with assumptions and obligation constraints. That is, every incoming message from a specific component implementing the required role is checked against compliance with its *obligation constraints.* Similarly, any outgoing message from the interaction adaptor to that specific component is checked against the receiving component's *assumption constraints.* The same mechanism applies for the other component associated with the violation detector. Considering the non-deterministic nature of the interaction policy representation, both mechanisms need

proper pattern violation detection. The pattern detection can be on-line or off-line. In Chapter 6 demonstrates the run-time detection for deterministic protocols.

5.3 CAUSAL PROCESS AND CONSTRAINT SPECIFICATION

Local interaction policies consider those roles with pair-wise local connections that are associated with interaction constraints. They can be considered as local integration dependencies between constituent components implementing provided and required roles. In the context of the composite component under certification, the cascading of these local integration dependencies leads to the emergent global integration behavior of the constituent components. Such global integration constraints are considered under the horizontal integration aspect by utilizing mechanisms (i.e., UML collaboration graphs) to specify the causal processes underlying the computations. This section introduces a graph-oriented representation of causal event sequences based on the UML collaboration graphs. A simple constraint language is provided to enable the designer to constrain the expected causal process patterns and their evolution.

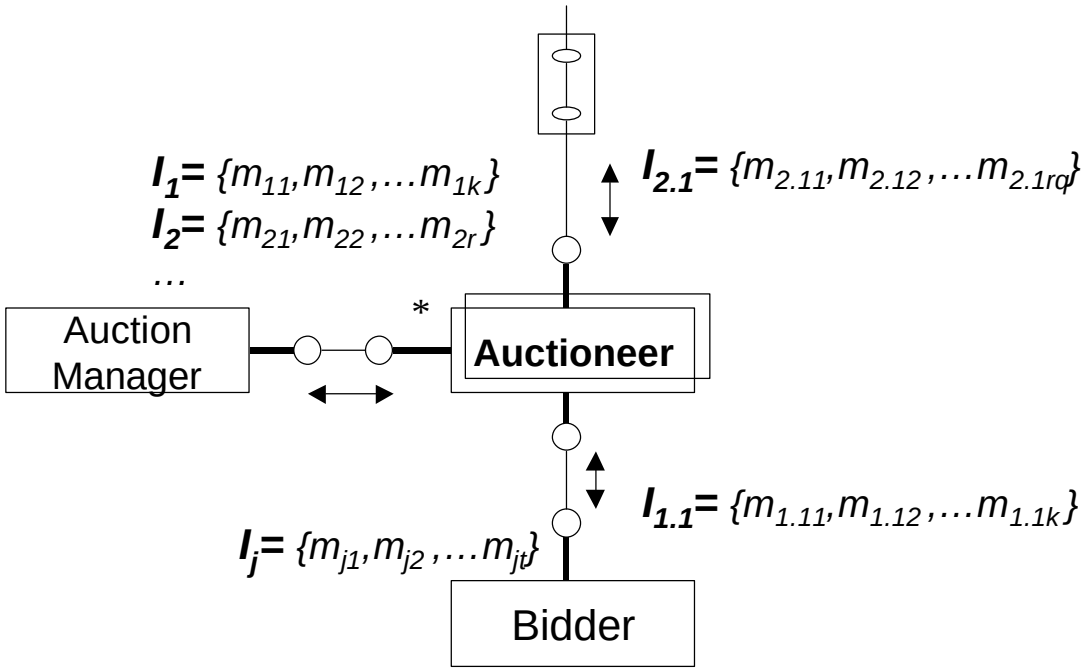


Figure 5-5: Causal Interaction Sequence Derivation from Collaboration Graphs

Figure 5.5 depicts the interaction aspect of a collaboration that is utilized to capture causal processes cross-cutting the used components. Each interaction constitutes a sequence of messages exchanged between a pair of connected roles. In figure 5.5, the overall interaction, $I = \{I_1, I_{1.1}, I_2, I_{2.1} \ldots I_j\}$, is decomposed into subinteractions such as I_1, $I_{1.1}$, I_2, $I_{2.1} \ldots I_j$, each of which constitutes message exchanges between peers. Note that UML collaboration interaction semantics provides a precedence relation among the interactions. That is, the interaction identifiers that are distinguished in one integer form (i.e., 1, 1.1, 1.1.1, 1.1.2, 1.1.3 and 1, 1.2, 1.2.1, 1.2.2, 1.2.3 are two such sequences) constitute a sequence of interactions.

5.3.1 Causal Interaction Graph Representation for Causal Process Capture

The interaction sequences discussed above can be viewed as directed acyclic graphs, with the nodes representing the events and the arcs representing the causal relationships among them. This representation makes it possible to distinguish between roles that interact and those that do not.

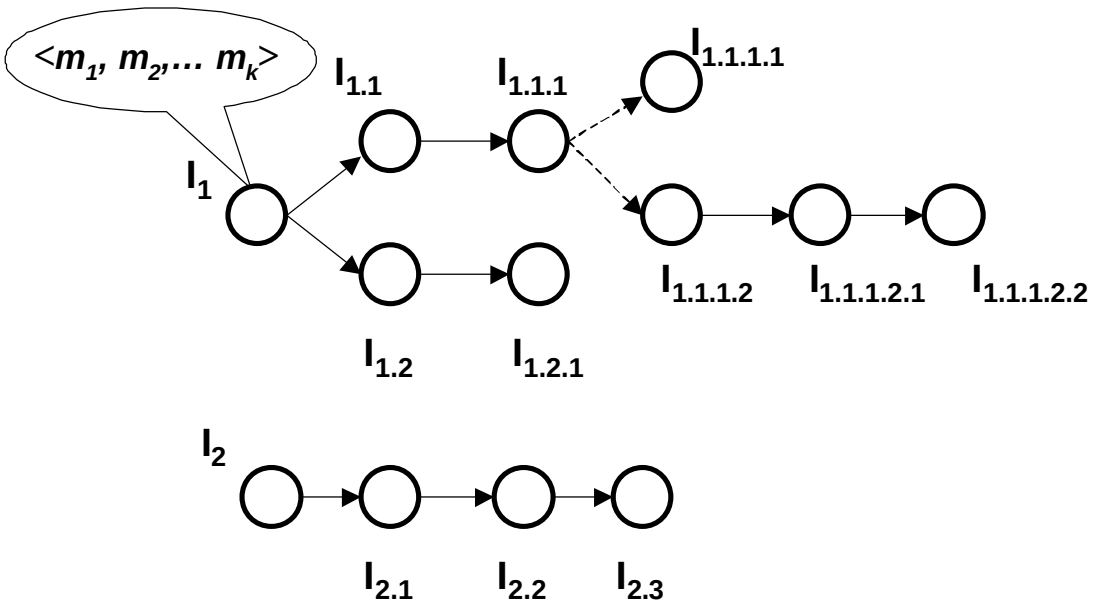


Figure 5-6: Causal Interaction Sequences

The interaction sequencing rules of UML collaboration graphs are represented in terms of directed acyclic graphs, as shown in Figure 5.6. These graphs are called *causal interaction graphs*. Each causal interaction graph defines a single *causal process*. Each root-to-leaf sequence of interactions is called a *causal interaction sequence*.

Each node in the causal interaction graph denotes an interaction containing a sequence of messages exchanged between two peers in conformance with their local interaction policies. Formally, a causal interaction graph is defined as $G = (V, E)$ where V denotes the set of interaction nodes and E denotes the causal relations; that is, $E \subseteq V \times V$ such that for every $(v, w) \in E$, v and w indicate a sequence of message send or receive actions. There are two types of edges in E: (1) *and* edges and (2) *or* edges. denote the fact that both causal paths will be taken consecutively, one after the other, whereas due to the guard conditions of messages, *or* edges denote alternative paths. Solid arrows represent *and* edges, while *or* edges are represented by dashed lines between the interaction nodes.

Each particular interaction may directly cause one or more interactions to occur, as depicted by *and* edges. For instance, a specific API invoked on a component implementing a particular role may utilize a set of API calls that initiate further role enactment. Therefore, a causal process defined in terms of causal interaction graphs constitutes a thread of event causation due to a particular original event. Each thread of interaction becomes a higher level of abstraction than an interaction itself, permitting a scalable view of what constitutes a computation.

5.3.2 Causal Process Constraint Definition Syntax and Operator Semantics

Causal processes are the expected computations that underlie services delivered by the component. The processes themselves constrain the overall computation space by imposing a specific causal pattern of interaction sequences. However, by themselves, causal processes cannot relate their occurrence to the occurrence of other causal processes and cannot represent the evolution of the computations. That is, operators and further constraints are needed to determine whether the occurrence of causal processes are temporally or logically related to each other, as well as whether the causal processes evolve as intended. Operators and constraints are also necessary to determine whether, as

causal process occurs in the overall computation under different test scenarios, the existential quantifiers hold (i.e., is a particular pattern an invariant for every possible computation?). Such properties can be extended to check for interesting properties in the overall causal event streams. Below the syntactic grammar of temporal, logical, and Boolean operators on causal processes are defined:

cpdConstraints ::= **constraints ':'** CPDSet
CPDSet ::= CPD CPDSet | *CPD*
CPD ::= (**always** |**sometimes**) Cpat | (**introduce | prevent | guarantee**) CPat
CPat ::= CedP | (**until | unless| since**) (CedP, CedP)
CedP ::= Ced **or** CedP | Ced
Ced ::= cedName **and** Ced | cedName Þ cedName | cedName

The constraints involve existential quantifiers (*always, sometimes*), evolution operators (*introduce, prevent, guarantee*), temporal operators (*until, unless, since*), and Boolean operators (*or, and, implies*). The set of constraints is selected to demonstrate the causal, temporal, and evolution aspects of causal processes that can be checked against those of the derived interaction behavior model. The details of the derivation of the interaction behavior model is left to Section 5.4. Here, the existence of such a model is considered to formalize the meaning of operators in terms of the observed causal processes.

The operands of the constraints are atomic causal processes defined in terms of interaction sequences, the choice of granularity of which depends on the selection by the component designer. As long as it does not contain *or* edges, any directed acyclic subcomponent of an original interaction event graph, including the original graph itself, can be chosen as the atomic unit of evaluation. *Or* edges denote alternative causal interaction sequences and need to be evaluated separately; however, alternative processes can be combined with *or* operators during constraint specification.

Any sub-graph of a causal interaction sequence can therefore be selected as an operand for constraint specification and as an atomic unit for evaluation. Encoded ordering among the interactions of such a sub-graph implicitly represents a linear

sequence of interactions, with each interaction denoting a sequence of messages among the peer roles of the collaboration graph.

Next, the semantics of each operator is introduced with respect to the behavioral model derived from event sequence logs observed during testing and run-time monitoring. Section 5.4 discusses how the behavioral model is derived from raw interaction logs using a causal event model. Here, its existence is assumed.

Let $\boldsymbol{M_{IB} = (CPD, E, P)}$ denote the graph representation of derived interaction behavior model for a single causal thread of interactions, where *CPD* denotes the set nodes representing the observed and matched causal processes, *E* denotes the causal relation between these processes, and *P* denotes the set of all non-circular causal process paths that can be generated from the graph model depicted by *CPD* and *E*. Each path, $\tilde{O}\ \hat{I}\ P$, refers to a sequence of causal process description nodes linked by the causal relation. Note that at least one behavioral model is derived as a result of a single test sequence execution.

The constraints on each behavioral model are defined in reference to the causal, temporal, and logical relationships between causal interaction sequences that are expected to exist in a single path of causal processes. Note, however, that due to concurrent interactions initiated from different components. Each with its own thread of control, there might exist simultaneous causal paths with independent causal processes. Constraints are defined such that causally dependent and related processes are evaluated together. The constraints that entail operators on such concurrent, composite, causal process expressions are evaluated separately. A causal process cpd_1 is caused by another causal process cpd_2, if there exists an interaction sequence in cpd_2 that has a direct causal link to an interaction in cpd_1. Given these basic definitions, one can now define the meanings of the constraint language operators.

Existential quantifiers consider the cases where a causal process is an *invariant* under every possible usage scenario of the component (i.e., always operator), or its occurrence is not essential for every scenario (i.e., sometimes). Each single scenario (i.e., test sequence) execution facilitates the generation of a set of models, where each model $\boldsymbol{M_{IB} = (CPD, E, P)}$ represents dependent emergent causal behaviors in reaction to

messages received by the component under certification. Existential quantifiers combine the results of the evaluations of individual M_{IB}s. The set of all M_{IB}s for all scenario executions (i.e., test sessions) of a component *C* is called *Scenarios(C).* The satisfiability of a causal process description for a given path, $\tilde{O}$ *of* M_{IB}, due to the execution of scenario (test sequence) *S*, is defined as

(*S*, M_{IB}, $\tilde{O}$ |= *cpd*), indicating that *cpd* Î $\tilde{O}$ or
simply **(M_{IB}, $\tilde{O}$ |= *cpd*),** when the scenario is implied.

Here, $\tilde{O}$ denotes the finite and linear sequence of filtered, matched, and mapped causal process description nodes. Using this basic notion of satisfiability, the existential quantifiers on causal processes are defined as follows:

Always cpd Û " ***S*** Î ***Scenarios(C), $ M_{IB} s.t. $*** $\tilde{O}$ Î ***P*, (S, M_{IB},** $\tilde{O}$ |= ***cpd)***
Sometimes cpd Û $ ***S*** Î ***Scenarios(C), $ M_{IB} s.t. $*** $\tilde{O}$ Î ***P, (S, M_{IB},*** $\tilde{O}$ |= ***cpd)***.

For every test scenario execution, the *always* operator requires that its parameter be satisfied by some causal process path. This notion of satisfiability under every test session is similar to invariant satisfiability, which requires the satisfying of causal process descriptions or composite expressions under every usage scenario of the component under certification. On the other hand, the *sometimes* operator determines if the causal process expression is satisfiable under a causal interaction in at least one scenario, but not necessarily in all of them. The temporal operators (i.e., until, unless, since) relate to the occurrence of relative ordering with respect to other causal processes.

Until* (*cpd*$_1$,*cpd*$_2$)** Û $ ***k ³ ***0, such that M_{IB},*** $\tilde{O}^{0,k}$ |= ***cpd*$_2$ and (M_{IB},** $\tilde{O}^{0,k-1}$ |= ***cpd*$_1$)**
and *M_{IB},* $\tilde{O}^{k+1,n}$ |¹ ***cpd*$_1$)**
Unless* (*cpd*$_1$,*cpd*$_2$)** Û ***M_{IB}, $\tilde{O}$ |= ***cpd*$_1$ iff (M_{IB},** $\tilde{O}$ |¹ ***cpd*$_2$)**
Since* (*cpd*$_1$,*cpd*$_2$)** Û $ ***k ³ ***0 s.t. M_{IB},*** $\tilde{O}^{0,k}$ |= ***cpd*$_2$ and (M_{IB},** $\tilde{O}^{0,k}$ |¹ ***cpd*$_1$) and**
(M_{IB}, $\tilde{O}^{k+1,n}$ |= ***cpd*$_1$)**.

The *until* operator combines the satisfying of two causal process expressions. If there is a node where the second expression is satisfied such that the first expression holds only on the prefix of the causal interaction path preceding the occurrence of the second expression, the *until* operator holds. The notation $\tilde{O}^{i,j}$ denotes the subsequence of the causal interaction path from the i^{th} to j^{th} node. On a given causal process path, the *unless* operator requires the first causal process expression to hold or be satisfied in case the second one does or is not. The *since* operator requires the first causal process expression to hold for a prefix of the interaction path that ends at the node, after which the second expression is satisfied. The first expression is required not to be satisfied after the second expression holds true.

Boolean operators define high-level composite causal process patterns in terms of the basic causal process descriptions that are exemplified by causal event graphs.

$$\textbf{or}\ (cpd_1, cpd_2, \ldots, cpd_q)\ \hat{U}\ M_{IB}, \tilde{O}\ \models cpd_1\ \textbf{or}\ M_{IB}, \tilde{O}\ \models cpd_2\ \textbf{or}\ \ldots\ \textbf{or}\ M_{IB}, \tilde{O}\ \models cpd_q$$

$$\textbf{and}\ (cpd_1, cpd_2, \ldots, cpd_q)\ \hat{U}\ M_{IB}, \tilde{O}\ \models cpd_1\ \textbf{and}\ M_{IB}, \tilde{O}\ \models cpd_2\ \textbf{and}\ \ldots\ \textbf{and}\ M_{IB}, \tilde{O}\ \models cpd_q$$

$$\textbf{implies}\ (cpd_1, cpd_2)\ \hat{U}\ M_{IB}, \tilde{O}\ \models^{1} cpd_1\ \textbf{or}\ M_{IB}, \tilde{O}\ \models cpd_2.$$

The development of a component with reuse is an evolutionary process in which the components are acquired, adapted, assembled, and fine-tuned in their new context to provide the desired service. Controlling and analyzing the evolution of a component in terms of the evolution of its causal processes would require mechanisms to ensure that intended behaviors are introduced, prevented, or guaranteed in accordance with any modifications. Hence the operators *introduce, prevent* and *guarantee* are created.

In the following formulation of these operators, $\tilde{O}'$ is a causal interaction path of *P'* that denotes the set of causal interaction paths derived from the modified component under a particular test scenario, S, whereas $\tilde{O}$ is an interaction path of *P* that denotes the paths for the original component implementation under the same test scenario. $M'_{IB}[S]$ and $M_{IB}[S]$denote the behavioral model derived as a result of the execution of scenario *S* for the modified and original components, respectively. ***P'*** and ***P*** denote the set of causal interaction paths modeled by $M'_{IB}[S]$ and $M_{IB}[S]$, respectively.

quarantee* (*cpd*)** $\hat{U}$ " ***S $\hat{I}$ ***Scenarios,*** $ ***M'$_{IB}$[S] s.t*** $ $\tilde{O}$' $\hat{I}$ ***P' (M'$_{IB}$,*** $\tilde{O}$' |= ***cpd)*** **and**

$ ***S*** $\hat{I}$ ***Scenarios ,*** $ ***M$_{IB}$[S] and*** $ $\tilde{O}$ $\hat{I}$ ***P s.t. (M$_{IB}$,*** $\tilde{O}$ |^1 ***cpd)***

prevent* (*cpd*)** $\hat{U}$ " ***S $\hat{I}$ ***Scenarios,*** " ***M'$_{IB}$[S] s.t.*** " $\tilde{O}$' $\hat{I}$ ***P' (M'$_{IB}$,*** $\tilde{O}$' |^1 ***cpd)*** **and**

$ ***S*** $\hat{I}$ ***Scenarios ,*** $ ***M$_{IB}$ [S] s.t.*** $ $\tilde{O}$ $\hat{I}$ ***P (M$_{IB}$,*** $\tilde{O}$ |= ***cpd)***

introduce* (*cpd*)** $\hat{U}$ $ ***S $\hat{I}$ ***Scenarios ,*** $ ***M'$_{IB}$[S] s.t.*** $ $\tilde{O}$' $\hat{I}$ ***P' (M$_{IB}$', ***$\tilde{O}$' |= ***cpd)*** **and**

" ***S*** $\hat{I}$ ***Scenarios*** " ***M$_{IB}$[S] s.t.*** " $\tilde{O}$ $\hat{I}$ ***P (M$_{IB}$,*** $\tilde{O}$ |^1 ***cpd)***

The parameter of a *guarantee* operator denotes a causal process description or a composite causal process expression. This operator requires its parameter to hold for the modified component on some causal interaction path in every test scenario, whereas the same expression holds on some of the interaction paths enacted by some test scenarios on the original component implementation. The *prevent* operator requires the causal process description or expression not to hold for any causal interaction path of the modified component under the given test scenario, while holding on some of the paths emerging from the execution of the original component under the some test scenarios. The *introduce* operator introduces the indicated causal process description or composite expression in at least one of the causal interaction paths that emerge from the execution of the modified component. Under execution of the same test scenarios, the same causal process does not exist in any of the causal interaction paths of the original component.

5.4 CAUSAL PROCESS BEHAVIORAL MODEL DERIVATION AND ANALYSIS

The proposed causal process constraints and the derived interaction behavior model facilitate checking whether, under the imposed constraints of the monitored computation, the intended causal processes exist. Causal process analysis requires a causal event model depicting a set of rules to formalize the conditions under which two events are causally related. The concepts underlying the model imply the implementation model; hence, different interaction models, systems, and models of computation would require proper models that fit into their framework. The presented model, which is congruent with the concepts and methods introduced in chapters 3 and 4, it is assumed that the composite component's provided and required roles are implemented by either passive components

or active components with their own thread of control, and that they communicate via synchronous messaging. The behavioral model derivation utilizes the causal event model, the connections among the roles of the collaboration graph, and the observed event log to generate the model. The constraint analysis constitutes the algorithms and methods that determine whether the derived model satisfies constraints.

5.4.1 A Synchronization-Based Causal Event Model

Causal processes are a convenient means of capturing the interaction behavior and scenarios of composite components. During component monitoring, two relations are associated with events of the observed interactions, *temporal* and *causal*. The temporal relation orders the events in each thread with respect to the system clock. If two events—*e*, *f*—are ordered by the temporal relation, this fact is denoted as $e \pounds f$. This relation is reflexive and transitive. Furthermore, it provides a total order with respect to the system clock. Conversely, the causal relation defines an order among the observed events according to the cause of an event. In general, the meaning of causal ordering is dependent on the causal model associated with the language or component model. The cause in the introduced model is based on *synchronization points* among the interacting, active components that constitute the composite component under certification. Since events cannot causally follow or precede themselves, the causal relation is non-reflexive. The relation is not symmetrical. That is, if two events *a* and *b* are causally related such that $a \pounds b$, then $b \pounds a$ is not true. Furthermore, the relation is transitive. That is, if $a \pounds b$ and $b \pounds c$, then $a \pounds c$ is true. Being non-reflexive, asymmetric, and transitive, causal relation defines a partial order that can be represented by directed acyclic graphs.

The introduced model has three main concepts: threads, connections, and synchronization points. In this model each role is depicted as a *thread*. The *connections* are defined in the abstract UML collaboration graph interactions as depicting the links between provided and required role services and actions. *Synchronization points* refer to the events of interest for observation and analysis. These events constitute the endpoints of the connections defined in the abstract component. Given these basic concepts constituting the causal model, the component execution exercised by the derived

interaction sequences generates a set of events *E.* The temporal relation *T,* and the set of connections *C,* along with the ordering rules presented in this section, facilitate construction of the causal relation. The theoretical basis for construction of the causal relation is a simplified model that makes certain assumptions. It is assumed that each thread refers to a single role, that connections are static and persistent throughout the lifetime of the component, and that causality is defined in terms of the synchronization points of threads, according to connections defined in the abstract component.

Threads are the producers and consumers of services and actions. They are the basic sequencing components of the composite component, and they provide a total order among the events that they produce and consume.

Events represent the activities and services of interest in the component being analyzed. Part of the process of abstract model specification involves defining whether an activity or service is of such interest. Role interfaces define the input and output channels that will be of interest to the analyst. Each event is associated with structural information that contains the event name, the originating/receiving thread, the time of the creation, or receipt of the event. An event is formally defined as *n-tuple* with at least the mentioned elements:

$$e = (e_{name}, e_{time}, e_{thread}.....)$$

Connections provide a way of depicting an ordered exchange of information, and in a sense, they represent how data flows among the synchronization points of the threads. Connections have two elements: a source name and destination name. Connections are formally defined as ordered pairs:

$$c = (c_s, c_d),$$

where c_s *and* c_d are the source and destination of the connection, respectively.

Execution of the component, along with its abstraction, produces an event set *E*, the temporal relation *T,* and the set of connections *C.* To determine their causality relation, two basic rules of causality are applied pair-wise to the elements of set *E*. The

first rule orders the events with respect to the threads, while the second one utilizes the connections to causally relate events.

(1) Role-Based Ordering: The sequential ordering of an individual thread's events against time provides a linear order of events causally related due to the control flow in the thread's execution pattern. Given a thread, the events generated by that thread are sequential in nature and are totally ordered. This relation is formally defined as follows:

$$\forall\ e, f \in E \text{ s.t. } e_{thread} = f_{thread} \text{ and } e_{time} < f_{time} \text{ then } e \rightarrow f$$

(2) Interaction-Based Ordering: The second ordering is implied by the connection descriptions in the abstract model. If two observed events are defined as the endpoints of a connection then a causal ordering is defined between them. Formally, this can be expressed as:

$$\forall\ e, f \in E, \text{ if } \exists\ c \in C \text{ s.t. } e_{name} = c_s \text{ and } f_{name} = c_d \text{ and } e_{time} < f_{time} \text{ then } e \rightarrow f$$

The above condition states that given two events, *e* and *f*, (i.e., a message send and receive events), if event *e* occurs before *f* and there is a connection with the source event *e* and destination event *f* , then there exists a causal relation between these two events. Note that, not all events *e* and *f* are causally related. It is an implementation detail to determine matching (*e,f)* pairs in the observed log of events.

Given the above rules and the components of the causal model, next the generation of a causal interaction behavior model is formally discussed to facilitate deriving the observed processes of the causal model.

5.4.2 Interaction Behavioral Model Derivation from Actual Interaction Behavior using the Causal Event Model

Model-based verification is an emerging discipline that aims to bridge the gap between the processes of testing and formal verification. Pragmatic, simplified models of systems

and components are utilized to facilitate the sort of formal reasoning vital for detecting inconsistencies, conflicts and errors on the models, as opposed to the formal process of symbol manipulation to prove the correctness of programs. The central foundation involves *abstracting* a particular *perspective*/*scope* of a component in terms of a *model* to analyze certain *properties* using *formalisms*. Hence, in model-based verification, well-formed, essential models are created using a particular formalism, and they analyzed relative to their expected behavior.

The current state of the practice involves generating essential models of the system before implementing the specifications and checking these models as early as possible against well-defined formal properties to detect errors. Note, however, that the approach presented here is based instead on deriving a causal behavioral model directly from the actual execution event logs of the concrete component. This model is then checked against the constraints to determine whether the intended causal processes occur and that the constraints imposed on them can be satisfied. Hence, the method pushes the model–based verification and checking further toward an implementation level.

Behavior model generation requires monitoring and recording of the sequence of relevant events enacted during the execution of the component and driven by the generated test sequences from the interaction policy of the environment model. The details of logging and monitoring are beyond the scope of the formalization of model derivation. Chapter 6 discusses at some length implementation details of the monitoring/logging environment used.

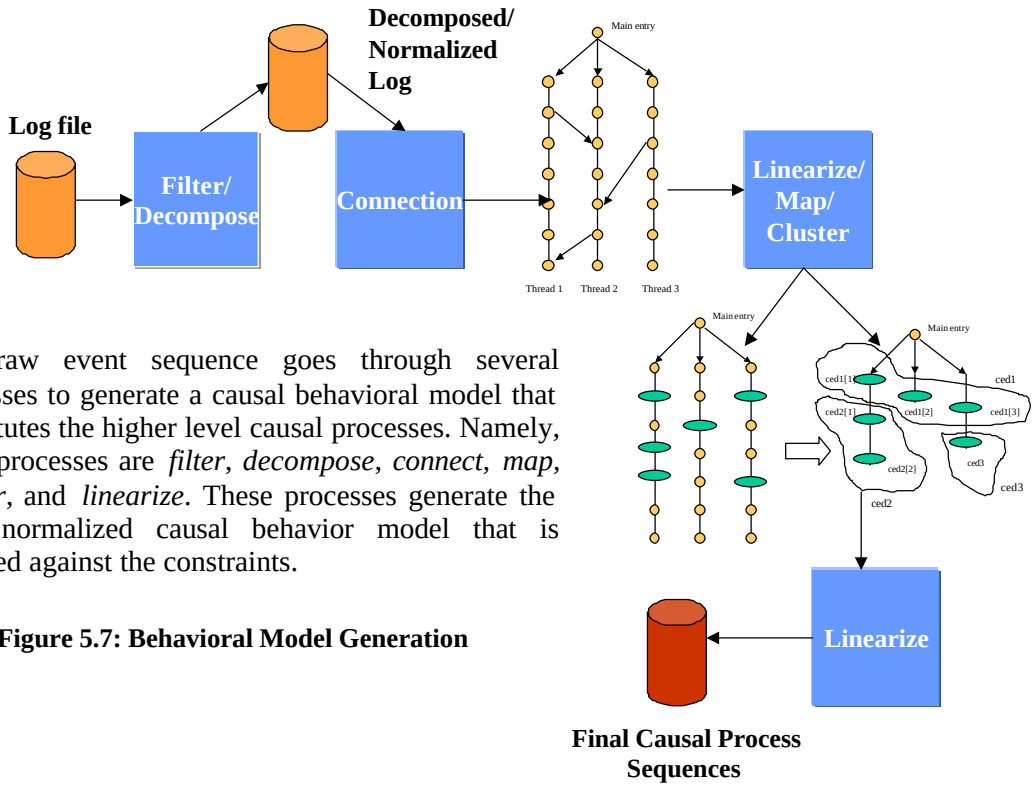
Execution of the component generates a sequence of logged events. These events include message send, receive, and event fire/consume operations performed by the component under certification, as well as by its constituents. This sequence is called $RawSequence[i] = \{e_o, e_1, e_2, \ldots, e_n\}$, indicating that the log belongs to the execution of the i^{th} scenario and each e_k denotes a service invocation or an action. Note that each sequence refers to the execution of a single scenario or test sequence.

For each test session there would be several scenarios or environment interaction sequences, each of which would generate a raw interaction sequence. Each sequence

contains the events that belong to the main component thread, as well as to the threads of the components, the roles of which are used in the collaboration graph.

The *filter* process selects those logged events that are of interest to causal process identification. Those events associated with interactions of the relevant causal processes are filtered to generate a reduced-order log. The *decompose* process partitions the log into *m* subsections, each of which keeps track of a particular thread. Hence, the *decompose* process partitions the log into separate event logs.

Each separate log section denotes the input/output (i.e., incoming and outgoing) interaction sequences of individual threads, each of which corresponds to an instance of a collaboration role. That is, the original sequence is decomposed into subsequences S_1, S_2, S_3,…, S_m, where S_i *Í S and* $S_i = \{e_{io}, e_{i1}, e_{i2}, \ldots, e_{ij}\}$ constitutes the events and services associated with thread T_i representing role R_i.



The raw event sequence goes through several processes to generate a causal behavioral model that constitutes the higher level causal processes. Namely, these processes are *filter*, *decompose*, *connect*, *map*, *cluster*, and *linearize*. These processes generate the final normalized causal behavior model that is checked against the constraints.

Figure 5.7: Behavioral Model Generation

The *connect* process links the causally related service and action points of threads associated through role connection. These service and action points are synchronization points, also defined in the connection section of the abstraction as causally related. The

outcome of the connect process is a directed graph that contains the service and actions for all threads, along with their connections, as shown in Figure 5.8.

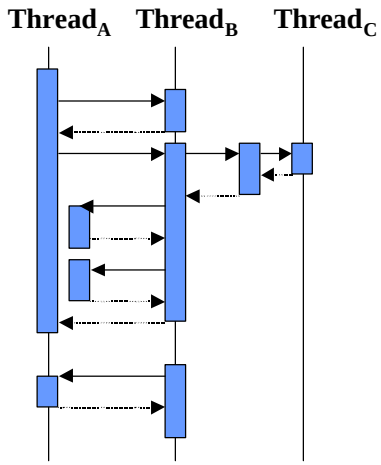


Figure 5-8: Causally Connected Threads

Note that the *decompose* process creates a thread-based ordering, whereas the *connect* process creates interaction (connection-based) ordering of events. Figure 5.8 characterizes the output of the *connect* process as a directed graph (DG). The main entry is the initialization event. Formally, the output is a graph, called *ThreadConnectionOrder= (V, E)* where *V* is the set of services and actions; that is *V = S*. The edges are derived based on thread and connection-based ordering as a result of decompose and filter operations. Formally, if $(e, f) \hat{I} E$ then either (1) $e_{thread} = f_{thread}$ and $e_{time} < f_{time}$, or (2) $\$ c \hat{I} C$ *s.t.* $e_{name} = c_s$ and $f_{name} = c_d$ and $e_{time} < f_{time}$

The *linearization* process simply generates causal sequences by traversing the graph taking the causal ordering of interactions into account. This leads to a set of sequences of causal interactions in conformance with the causal event model. The root node is the main entry node that characterizes the construction and initialization of the component. A leaf component of a DAG is a node that has a zero outdegree.

The output of the process is the causal sequence identified by the root-to-leaf traversal of the DAG. It is a set of sequences, each of which represents the causal sequences derived through thread-based and connection-based ordering of events. Formally, $S_{causal} = \{S_{c1}, S_{c2},\ldots, S_{ck}\}$ where each $S_{ci} = \{e_{ci,1}, e_{ci,2} , e_{ci,3}, \ldots, e_{ci,p}\}$ and for every consecutive pair of events *(e,f)* in the sequence, we have either (1) $e_{thread} = f_{thread}$ and $e_{time} < f_{time}$, or (2) $\$ c \hat{I} C$ *s.t.* $e_{name} = c_s$ and $f_{name} = c_d$ and $e_{time} < f_{time}$

The *map* process has two sub-processes. The first involves linearizing the causal process descriptions given as DAGs. This sub-process is similar to the *linearize* process defined above, with minor differences. That is, the linearizing of causal process descriptions leads to a single sequence of interaction by traversing the edges of the graph

such that explicitly encoded precedence relationships, such as shown in Figure 5.6, are taken into consideration. The output of the sub-process is a set of sequences, called *expected causal interaction sequences*, each of which is a linearized causal process. The second sub-process involves a *pattern matching* based on sequential pattern matching of the linearized causal process description sequences to serialized sequences of observed causal behaviors. Each linearized causal process sequence is a sequence of interactions,

$$cpd_i = <I_{i1}, I_{i2},\ldots, I_{in}>, \$\ j\ \ 1£\ j£\ n\ s.t.\ \ I_{ij}=<m_{ij1},m_{ij2},\ldots,m_{ijk}>,$$

where each $m_{ijt}\ 1£\ t\ £\ k$ is a single message.

The goal of this sub-process is to detect if the expected interaction sequences are observed under the exercised scenario. Note, however, that for a causal process to exist, all of the linearized interactions of that causal process description should be matched against the set of observed causal sequences. That is, observed causal sequences should contain the expected sequences derived from the linearization of the particular causal process description.

Pattern Matching: The following algorithm is applied in two phases to match expected linearized causal event sequences to linearized observed casual event histories:

```
PatternMatch(S,P)
[matches pattern P to sequence S]
  n = length[S]
  m = length[P]
  k=0
  for i= 0 to n-m do
      if P[1..m] = T[i+1..i+m]  then
         foundShift[k] =i
         k=k+1
end.
```

The first phase identifies each interaction, $I_{ij}=\{m_{ij1},m_{ij2},\ldots,m_{ijk}\}$, followed by the second pass that identifies the causal processes defined as sequences of interactions: $cpd_i = <I_{i1}, I_{i2},\ldots, I_{in}>$. The above pattern-matching algorithm returns every shift in the input sequence that occurs in the pattern described by the causal event sequence. In the new

model, every matched pattern is replaced by a single aggregate composite event representing the recognized interaction. The above algorithm assumes that the causally related events consequently occur in the observed causal interaction paths. A more relaxed version of the pattern matcher, known as longest common subsequence matching using dynamic programming, can be used in cases where causally related events that are separated in time are considered, as well as consecutive event patterns [Cormen *et al.*, 1990].

The new behavioral model generated is a directed graph representing the causal relationships between aggregate abstract events on which the component designer places constraints.

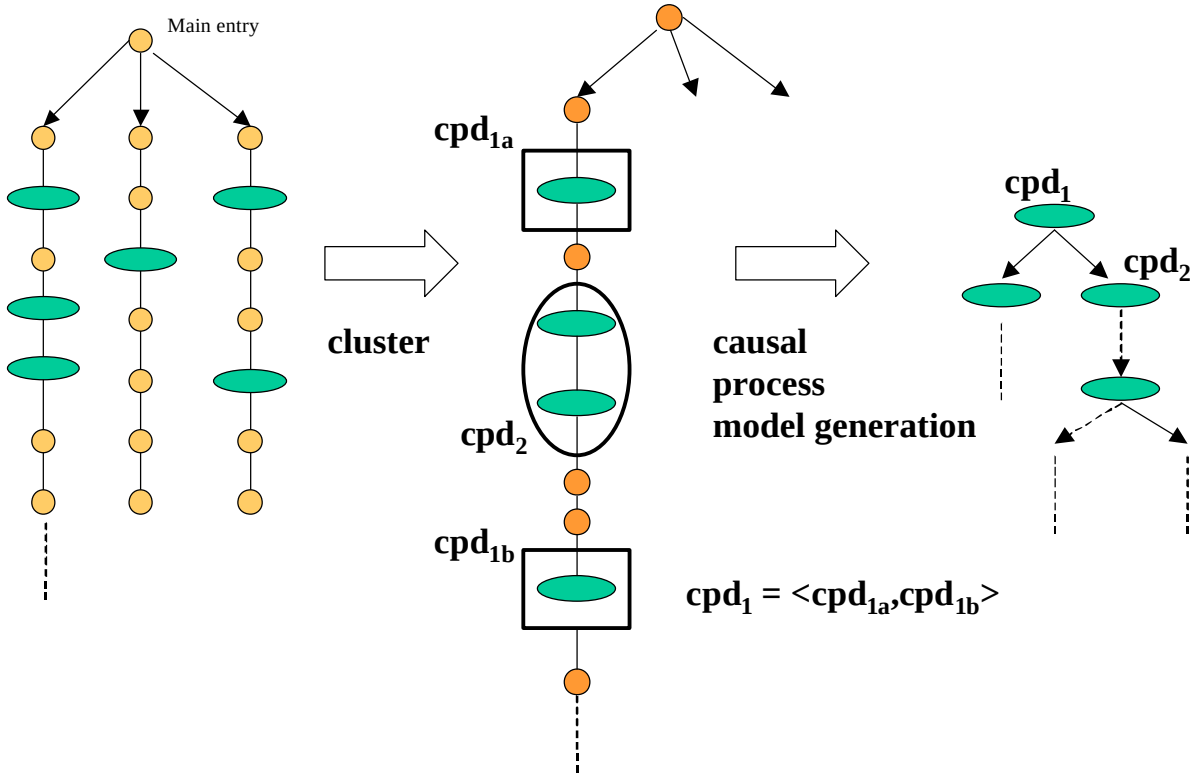


Figure 5.9: Interaction Clustering and Model Derivation

That is, the new model pulls the abstraction on-event sequences to interactions and process levels, the constituent components of which would make sense from the conceptual collaboration graph point of view.

Formally, by the pattern matching and filtering operations, a new set of sequences called, $S^R_{causal} = \{S^R_{c1}, S^R_{c2},\ldots, S^R_{ck}\}$ is generated to consider only the matched composite event patterns. Each of these event patterns constitutes a part of an *interaction* defined in the collaboration graph. In the new set of sequences, $S^R_{ci} = S_{ci} - M - U \; \grave{E} \; P$, where U is the set of unmatched events, M is the set as matched events, and P is the set of new patterns formed. Considering the fact that $S_{ci} = M \; \grave{E} \; U$, one can assert that $S^R_{ci} = P$; that is, the new sequence simply contains only the matched interaction patterns.

As shown in Figure 5.9, the *clustering* process is applied to the aggregated and *filtered* sequences to capture the causal process. This process itself contains several causal interaction patterns, as depicted earlier in Figure 5.6.

The causal sequences that constitute a particular causal process are clustered to derive a single node representing the recognized causal process. The procedure in clustering the sequences is as follows. For each *causal process description*, the contained *interactions* are identified. Note that the pattern matching sub-process serializes all the implied *interactions* of a particular causal process by *linearizing* the causal process description before applying the pattern-matching algorithm. Each causal process, CP_i, originally defined as a directed acyclic graph, is interpreted as sequence of interactions,

$$\boldsymbol{CP_i = <I_{i1}, I_{i2},\ldots, I_{in}>}.$$

For clarity and simplicity, it is assumed that $CP_i \; \text{Ç} \; CP_j = \text{Æ} \;\; i \neq j$, for any causal process description defined in the component abstraction. To further cluster $S^R_{causal} = \{S^R_{c1}, S^R_{c2},\ldots, S^R_{ck}\}$ such that it contains only causal processes and their execution order at a higher level of abstraction, the sequences are transformed into more abstract sequences

$$\boldsymbol{S^R_{ci} = <cp_1, cp_2, cp_3,\ldots, cp_m>}$$

that contain the causal processes. Analysis is performed with respect to the relevant model that constitutes the processes denoted by the constraints under consideration. Note that there would be several concurrent threads of causal processes originating from different components. Furthermore, it is possible to have more than one instance of

collaboration existing on separate threads of causal interaction sequences. The following algorithm is applied to every sequence to derive separate behavioral models:

Causal Behavioral Model Generation

Input: Linear sequence of clustered interactions: $S^R_{ci} = <cp_1, cp_2, cp_3, \ldots, cp_m>$,
Causal interaction graph: **CIG=(V, E)**, and
$CP_i = <I_{i1}, I_{i2}, \ldots>$ " $1 \pounds i \pounds m$ denoting interactions of each process to which partial process cp_i is part of.
Output: Causal Behavioral Model: **CBM,** denotes the causal dependency graph model

initialize ***CBM*** with the start node cp_1
set $j=2$ and *current* to the start node cp_1
repeat
select the next *cpd*, $cp_j \hat{I} S^R_{ci}$
if $\$ I_{jk} \hat{I} cp_j$ and $\$ I \hat{I} CP_{current}$ such that $(I, I_{jk}) \hat{I}$ **E**
add cp_j to the **CBM** along with the edge $(cp_{current}, cp_j)$
set *current* to the node cp_j inserted to **CBM**
else
if the interactions of $cp_j = < I_1, I_2 \ldots.>$ is a subset of the interactions of type, $CP_k = <I_{k1}, I_{k2}, \ldots>$, to which an ancestor partial causal process cp_k belongs then
set *current* to the ancestor node denoted by cp_k and $cp_k = cp_k + cp_j$
else
if the interactions of $cp_j = < I_1, I_2 \ldots.>$ has an interaction **I** such that there exist an ancestor node, $cp_k = <I_{k1}, I_{k2}, \ldots>$ and $(I_{kj}, I) \hat{I}$ **E** for some $I_{kj} \hat{I} cp_k$ then
add cp_j to the **CBM** along with the edge (cp_k, cp_j) and
set *current* to cp_j node.
else
if no node $cp_j = < I_1, I_2 \ldots.>$ that has a connection to a process of type CP_j exists then
create a new node cp_j under **CBM** and connect to the *current* node.
set *current* to cp_j
until all nodes in the sequence $S^R_{ci} = <cp_1, cp_2, cp_3, \ldots, cp_m>$ are traversed.

The above algorithm outlines how the causal behavioral model is generated. The clustered sequence that contains the detected processes is scanned, and the causal dependencies are constructed by consulting the *causal interaction graph*, which denotes expected interaction dependencies as defined by the original *collaboration graph*. A new behavioral model, represented as a directed acyclic graph (DAG), is constructed. As each node is processed, the algorithm checks to determine whether the node is causally related

to the previous node. If this is the case, then an edge is added from the previous node to the new node. Otherwise, the new node is determined to be part of a causal process, the preceding part of which has already been processed (Case 1), or to causally depend on an *interaction* of a causal process that is the ancestor node of the current node (Case 2). The new node can also be considered independent from any of the nodes (Case 3), and as such it cannot be described in terms of the causal relationships denoted by the causal interaction graph.

For Case 1, since the computation returns to the process from which it is originally enacted, the *current node* is set to be the ancestor node that has already been processed. The computation is expected to continue from there. For Case 2, the ancestor node, which enacts the current process, is found by searching toward the root node of the graph (ordered tree). Once that node is found, a new child node is added. The *current node* is set to be the current process connected to the found ancestor node. Under case 3, a new node is created and connected to the current node. The *current node* is set to be the new node inserted into the graph.

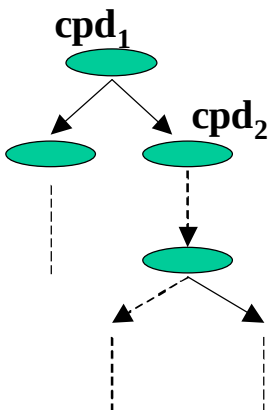


Figure 5.10: DerivedCausal Dependency Model

The observed causal dependency model, as shown in Figure 5.10, is represented as an ordered tree. Each node in the tree represents a particular causal process observed during the computation. An edge exists between two causal process nodes if and only if the source node has an interaction that itself causes an interaction of the target process. This interaction must be in accordance with causal interaction graphs based on the original collaboration graph. Each root to leaf path sequence denotes the observed causal dependencies in the observed computation. These sequences are behavioral in nature and are subject to constraint analysis.

The resultant behavioral model is an ordered tree denoting causal dependencies among the observed processes that emerge as a result of dependent interactions.

5.4.3 Constraint Analysis of the Derived Causal Interaction Behavioral Model

The derived behavior model, $G_M = (V_M, E_M)$, denotes the observed causal connections among higher-level causal processes aggregating one or more interactions. The set of process nodes and their causal dependencies are denoted by V and E, respectively. The causal behavioral model, $M_{IB} = (CPD, E, P)$, first introduced in section 5.3.2, can be obtained directly from G by letting $CPD = V_M$, and $E = E_M$. The set of root-to-leaf paths, P, including the sequences of recognized and clustered causal processes, are obtained using a path enumeration algorithm on directed graphs. Here the presentation of the path enumeration algorithm that generates paths of cardinality less than or equal to the number of nodes in the graph is omitted. Note, however, that the derivation of P from the G_M is presented in Chapter 6.

The constraints against which the derived causal behavioral model is checked impose temporal, evolution, and logical relationships on the observed causal processes as shown below. The execution of each single scenario results in a set of causal process sequences, stored in the set P. Constraint operators, such as *always* and *sometimes*, help the designer decide if the particular causal processes observed under all interaction scenarios enacted during the component certification process are invariant.

cpdConstraints ::= **constraints ':'** CPDSet
CPDSet ::= CPD CPDSet | *CPD*
CPD ::= (**always** |**sometimes**) Cpat | (**introduce | prevent | guarantee**) CPat
CPat ::= CedP | (**until | unless| since**) (CedP, CedP)
CedP ::= Ced **or** CedP | Ced
Ced ::= cedName **and** Ced | cedName Þ cedName | cedName

On the other hand, the evolution operators involve comparisons of successive versions under the test scenarios. The temporal operators, as well as the Boolean operators, apply a set of causal process patterns for each scenario. The results of these operators for each scenario are then combined to derive global results for quantifier and evolution operators.

The syntax grammar of the causal process constraints is delineated above. The logical operators derive from atomic process descriptions specifications for composite

and complex causal process patterns. Temporal operators such as *until, unless,* and *since* are proposed as a sample subset of feasible operators to facilitate analysis of the temporal relativity of the observed patterns. The evolution operators such as *introduce, prevent,* and *guarantee* facilitate comparative analysis of the component under certification as it evolves development. The quantifier operators are proposed to demonstrate that certain processes are satisfied across disparate test scenarios.

Once the behavioral models for each exercised interaction scenario are derived and enumerated into causal process sequences, one can view the complete observed behavior of the component as a behavior matrix. The rows and columns of this matrix represent the exercised scenarios and the individual sequences in *P* of each derived behavioral model for the corresponding scenario, respectively.

Formally, the behavior matrix is a two-dimensional *m* x *n* matrix, called *BM*. Here, *m* stands for the number of scenarios exercised, and *n* stands for the largest in cardinality of the set of causal process sequences derived as a result of each scenario. Each cell of the matrix is denoted as $P_{i,j}$, where *i* stands for the scenario *i* and *j* stands for the j^{th} sequence in the set, P_i, of causal process sequences associated with the scenario *i*. Hence, $P_{i,j}$ stands for the *jth* sequence of the causal process sequence set of the *ith* interaction scenario exercised during the run-time certification.

After defining a matrix-oriented representation of the behavioral model the next step is to test if the generated model satisfies constraints placed upon it. To determine if such constraints are satisfied, this approach requires a computational model to represent the constraints, as well as methods and reasoning mechanisms. To this end, The *constraint tree* formalism is introduced.

Definition: A **constraint tree**, *CT,* is a finite set of one or more nodes such that there is one designated node *R* called the root of *CT*. The remaining nodes in *CT –{R}* are also partitioned into $n \geq 0$ disjoint subsets $CT_1, CT_2, \ldots, CT_n$, each of which is a tree, and whose roots $R_1, R_2, \ldots, R_n$, respectively, are children of *R.* The leaf nodes that have out-degrees of zero are causal processes and the internal nodes that have out-degrees of one or more are the operators of the constraints.

sometimes (ced1 or ced2 and ced3)

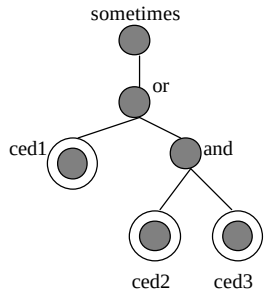


Figure 5-11: Constraint Tree

Constraint tree derivation: Consider the tree representation, shown in Figure 5.11, of the constraint *sometimes (ced1 or ced2 and ced3).* The constraint is parsed according to the constraint grammar to produce an abstract syntax tree for the in-order representation. Then, the tree is traversed to derive a pre-order representation of the constraint statement: *sometimes or ced1 and ced2 ced3*. Then the pre-order sequence is used to form the constraint tree in a top-down manner.

Causal Process Analysis: Causal process analysis constitutes the methods that check for the satisfying of constraints represented in constraint tree formalism against the behavior matrix $BM = (P_{i,j})$ for $i = 1,2,\ldots m$ and $j = 1,2,\ldots,k$ as discussed above. The satisfying of a constraint for the recognized causal process sequences of a scenario is represented as a Boolean vector. Hence, a two dimensional Boolean matrix, $SAT = (b_{i,j})$ for $i = 1,2,\ldots m$ and $j = 1,2,\ldots,k$ is used for all interaction scenarios used in the certification. Each Boolean vector for a given interaction scenario constitutes a row of the Boolean matrix.

We demonstrate the methods that check the satisfying of constraints against the derived causal process sequences of a particular scenario. The same method would be reapplied for each scenario to generate the complete Boolean matrix.

Let *BM* denote the *ith* row of the behavior matrix, which represents the set of causal process sequences captured during the *ith* interaction scenario. Similarly, *SAT* denotes the Boolean vector that stores the computed Boolean value denoting the satisfying of each constraint by the sequences of the *ith* scenario. *SAT* is computed gradually, using the constraint tree and the set of causal process sequences associated with the *BM.*

Using the constraint tree, the satisfiability vector is constructed from the bottom up. Initially, each leaf of the constraint tree is associated with a Boolean vector. The

vectors are propagated upward in the tree, and the rules associated with the operators at the internal nodes are applied to the vectors of its children to generate a new Boolean vector. The derived Boolean vector is further propagated up toward the root. The resulting Boolean vector at the root of the constraint tree represents the satisfying of constraints for the set of causal processes in each interaction scenario.

Algorithm: Constraint Evaluation
Input: BM_i: Behavior vector for ith scenario,
CT: Constraint Tree
Output: SAT_i: Satisfiability vector for the ith scenario against user-defined constraint

for each $u \hat{I} leaves[CT]$ set $visited[u] = true$, set $u_b = <b_1,\ldots,b_n>$
where $b_j = 1$ *if* $u \hat{I} BM_i[j]$ *else* $b_j = 0$
$unprocessed = internal[CT] \grave{E} leaves[CT]$
while $unprocessed \neq \{Root\}$

- choose a set of siblings $S = \{s^1, s^2,\ldots, s^k\} \acute{I} unprocessed$ such that $p = parent(s^i)$ for an arbitrary $1 \pounds i \pounds k$ and for each $s^r \in S$ $1 \pounds r \pounds k$ $visited[s^r] = true$
- Compute the boolean vector as $p_b = s_b^1 \grave{O} s_b^2 \grave{O} \ldots \grave{O} s_b^k$ where $\grave{O}$ denote the operator application imposed by the internal node p, set $visited[p]=true$
- set $unprocessed = unprocessed - S$

endwhile
return $Root_b$
end.

Formally, the operator application is defined as follows: Let *operator* stand for the operator (i.e., *and, or, implies*) associated with the node under consideration and let $bv_1=[bv_{1,0},bv_{1,1},\ldots,bv_{1,n}]$, $bv_2=[bv_{2,0},bv_{2,1},\ldots,bv_{2,n}],\ldots,bv_n=[bv_{1,0},bv_{1,1},\ldots,bv_{1,n}]$ denote the Boolean vectors associated with the child nodes of the operator node. The operator application is defined as **bv** = $bv_1 \circledR bv_2 \circledR \ldots \circledR bv_n$. Depending on the type of the operator, the operation can simply be applied to individual vector points (i.e., or, and) directly. For operators such as *unless, until,* etc., additional precedence information is necessary.

The above algorithm that has complexity $O(CT)$, where $|CT|$ is the number of nodes, considers all nodes of the tree in a bottom up manner to compute the Boolean vector which represents the satisfying of constraints for each sequence in the interaction scenario. The Boolean vector, $p_b = s_b^1 \grave{O} s_b^2 \grave{O} \ldots \grave{O} s_b^k$, is computed according to the

rules associated with the internal node. For instance, if *p* is an *or* node than the vector is computed as $p_b = s_b^1 \text{ È } s_b^2 \text{ È} \ldots s_b^k$.

For each operator, the operator application is defined as

(1) resultVector = *operator(vector*$_1$*,… vector*$_k$*)* for internal nodes or

(2) resultVector =*evaluate (causalprocessid, BM*$_i$*)* for leaf nodes.

Each leaf node in the constraint tree denotes a particular causal process and is evaluated against the set of causal process sequences, BM_i, caused by the *ith* interaction scenario. The internal nodes, which denote the constraint operators, evaluate the input Boolean vectors according to the rules of the operator to generate new such vectors. The rules for the operators are defined as follows:

- ***evaluate*** *(cpd, BM*$_i$*):* The evaluation of a particular causal process, *cpd*, against the set of sequences depicted by BM_i, results in the vector $<b_1, b_2, \ldots, b_k>$ where for $1£ j £ k$ $b_j = 1$ if $cpd \text{ Î } BM_i[j]$; otherwise $b_j = 0$.
- ***or*** *(v*$_1$*, v*$_2$ *…, v*$_q$*) :* The evaluation of the *or* operator on the results of the evaluation of its *q* sub-expressions (i.e., operands) is a Boolean vector $<b_1, b_2, \ldots, b_k>$ where for $1£ j £ k$ $b_j = v_1[j] \text{ È } v_2[j] \text{ È } \ldots \text{È } v_q[j]$.
- ***and*** *(v*$_1$*, v*$_2$ *…, v*$_q$*) :* The evaluation of the *and* operator on the results of the evaluation of its *q* sub-expressions is a Boolean vector $<b_1, b_2, \ldots, b_k>$ where for $1£ j £ k$ $b_j = v_1[j] \text{Ç } v_2[j] \text{ Ç } \ldots \text{ Ç } v_q[j]$.
- ***implies*** *(v*$_1$*, v*$_2$*) :* The evaluation of the *implies* operator results in boolean vector $<b_1, b_2, \ldots, b_k>$ where for $1£ j £ k$ $b_j = not(v_1[j]) \text{ È } v_2[j]$.
- ***until*** *(v*$_1$*, v*$_2$*) :* The evaluation of the *until* operator requires additional information regarding the relative timing of when causal processes occur in the derived sequences. The formal representation, *Until* (cpd_1, cpd_2) $\text{Û } \$ k ³ 0$ *such that* $M_{IB}, \tilde{O}^{0,k} \models cpd_2$ and $(M_{IB}, \tilde{O}^{0,k-1} \models cpd_1)$ and $(M_{IB}, \tilde{O}^{k+1,n} \neq cpd_1)$, can be translated into the concrete rules as follows: (1) A pair of indices, (s_i, e_i), identifying the originating and conclusion points of the causal process cpd_i is associated with each Boolean element of an input vector that has *true* value.

Basically, each interaction of the processes identifies a point in time. (2) For each sequence derived from the model, it is checked if the following rule holds:

$$s_1 < s_2 \text{ and } e_1 < e_2.$$

If the rule holds then a Boolean value of true is returned for the corresponding expression. Hence, the result is a Boolean vector $<b_1, b_2, \ldots, b_k>$ denoting the satisfying of the operator for each sequence.

- ***unless*** *(v_1, v_2) :* The evaluation of the *unless* operator requires additional information regarding the relative occurrence of causal processes in the derived sequences. The formal representation, *Unless* (cpd_1,cpd_2) $\hat{U}$ $M_{IB}, \tilde{O} \models cpd_1$ iff ($M_{IB}, \tilde{O} \not\models cpd_2$), is translated into concrete rules as follows: The evaluation of the *unless* operator results in a Boolean vector $<b_1, b_2, \ldots, b_k>$ where for $1 \pounds j \pounds k$ $b_j = 1$ *iff* $[(v_1[j]=1$ *and* $v_2[j]=0)$ *or* $(v_1[j]=0$ *and* $v_2[j]=1)]$. More succinctly, b_j = [(not ($v_1[j]$) and $v_2[j]$) or ($v_1[j]$) and not ($v_2[j]$))].
- ***since*** *(v_1, v_2) :* The evaluation of the *since* operator requires additional information regarding the relative occurrence of causal processes in the derived sequences. The formal representation, *Since* (cpd_1,cpd_2) $\hat{U}$ $\$ k \geq 0$ *s.t.* $M_{IB}, \tilde{O}^{0,k} \models cpd_2$ and ($M_{IB}, \tilde{O}^{0,k} \not\models cpd_1$) and ($M_{IB}, \tilde{O}^{k+1,n} \models cpd_1$), is translated into concrete rules similar to the *until* operator. Each Boolean element that has a true value in each vector is associated with the indices (s_i, e_i), of the occurrence and the conclusion point in time of the expression cpd_i. The following rule is used to derive the satisfiability vector for the operator: $s_1 > e_2$

Satisfiability: For constraints that have one of the above operators as the root node, the satisfying of the constraint is defined based on the derived Boolean vector. That is, if the final evaluation vector for the causal process expression (*cpe*) is $<b_1, b_2, \ldots, b_k>$, where each Boolean value represents evaluation results with respect to the particular causal sequence path of the model under interaction scenario, then the constraint is satisfied if there exists a Boolean value in the vector equal to true. Formally, $\boldsymbol{BM_i} \models$ ***cpd*** iff $\$ j, 1 \pounds j \pounds k, b_j = 1$.

- ***Always*** *(SAT_1, SAT_2,…, SAT_m)* : The evaluation of the *always* operator is analogous to checking for an invariant . That is, the Boolean vectors for each BM_i should all be satisfied according to the above definition of satisfiability. Formally, *" i, 1 £ i £ m* $\mathbf{BM_i}$*|=* ***cpd***.
- ***Sometimes*** *(SAT_1, SAT_2,…, SAT_m)* : The *sometimes* operator is an existential quantifier on the set of observed causal sequences and is satisfied if any member of the set of Boolean vectors, *SAT = {SAT_1, SAT_2,…, SAT_m}* is satisfied. This reflects the expectation that the intended causal process expression is not an invariant and does not necessarily occur under each interaction scenario. Formally, *$ i, 1 £ i £ m* $\mathbf{BM_i}$*|=* ***cpd***.

The evolution operators are meant to capture the safe, as well as intended, adaptation and evolution of the component under certification. Hence, the operators are applied to successive versions of the complete behavior matrices to determine whether a particular behavior, defined in terms of a process expression, is *introduced*, *prevented*, or *guaranteed*. Formally, the *prevent*, *introduce,* and *guarantee* operators are defined in terms of how the original satisfiability Boolean matrix, *SAT*, relates to the updated matrix, *SAT'*, which is obtained as a result of maintenance or evolution. Remember that both *SAT* and *SAT'* are the matrices derived for a particular expected causal process expression under a single test scenario.

- ***quarantee*** *($SAT_{1..m}$, $SAT'_{1..m}$, cpd)*: The formal definition of the *guarantee* operator, *guarantee* (*cpd*) *Û " S Î Scenarios, $ M'$_IB[S] and $ Õ' Î P' s.t. (M'$_IB, *Õ' |= cpd)* and *$ S Î Scenarios , $* M_{IB}*[S] and $ Õ Î P s.t. (*M_{IB}*, Õ |≠ cpd)*, indicates that (1) for every SAT_i' vector, *1£ i £ m*, denoted by a Boolean vector, *<b'*$_1$*, b'*$_2$*, …, b'*$_k$*>*, there exists a Boolean variable *1£ j £ k* b'_j*= 1* and (2) there exists a SAT_q, *1£ q £ m*, denoted by a Boolean vector, *<*b_1*,* b_2*, …,* b_k*>* such that *"j* 1*£ j £ k* b_j*= 0*.

- ***prevent*** *($SAT_{1..m}$, $SAT'_{1..m}$, cpd)*: The formal definition of the *prevent* operator, *prevent* (*cpd*) *Û " S Î Scenarios, " M'_{IB}[S] and " Õ' Î P' (M'_{IB}, Õ' |≠ cpd)* and *$ S Î Scenarios ,$ M_{IB} [S] and $ Õ Î P s.t. (M_{IB}, Õ |= cpd)* indicates that (1) for every SAT_i' vector, 1£ *i* £ *m*, denoted by a Boolean vector, <$b'_1, b'_2, \ldots, b'_k$>, *" j* 1£ *j* £ *k* b'_j= *0* and (2) there exists an SAT_q, 1£ *q* £ *m*, denoted by a Boolean vector, <b_1, $b_2, \ldots, b_k$> such that *$ j* 1£ *j* £ *k* b_j= *1*.

- ***introduce*** *($SAT_{1..m}$, $SAT'_{1..m}$, cpd)*: The formal definition of the *introduce* operator, *introduce* (*cpd*) *Û $ S Î Scenarios , $ M'_{IB}[S] and $ Õ' Î P' s.t. (M_{IB}', Õ' |= cpd)* and *" S Î Scenarios " M_{IB}[S] " Õ Î P (M_{IB}, Õ |≠ cpd)* indicates that (1) there exists an SAT_i' vector, 1£ *i* £ *m*, denoted by a Boolean vector, <$b'_1, b'_2, \ldots, b'_k$>, such that *$ j* 1£ *j* £ *k* b'_j= *1* and (2) for every SAT_q, 1£ *q* £ *m*, denoted by a Boolean vector, <$b_1, b_2, \ldots, b_k$>, there does **not** exist a Boolean variable b_j=*1*for *" j 1£ j £ k.*

CHAPTER 6

VERIFICATION ENGINEER'S ASSISTANT REFERENCE PROTOTYPE and CASE STUDIES

Software engineers have begun to understand the importance of reasoning about the interactions and the causal process of the components that they design and implement. In particular, one serious problem they face is the necessity of understanding the fitness of reused components in the context of the collaboration under development. Chapters 3, 4, and 5 outlined the fundamental principles and procedures for understanding the compatibility of the interaction assumptions and obligations of the reused components, as well as the safety of design extensions and enhancements for compatibility. These chapters also consider whether the actual implementation conforms to expected interaction policies and intended causal processes.

This chapter discusses lessons learned during the development of a reference prototype that implements the developed algorithms and framework. It then illustrates the techniques using three simple case studies to demonstrate and evaluate the utility of the approach under a reuse-oriented collaboration-based software design methodology.

The chapter is organized as follows. Section 1 introduces the utilized component development language and testbed used to demonstrate the developed methods. Section 2 uses the developed reasoning and verification methods to outline architecture and design issues involving the main components of the reference prototype environment. Section 3 discusses three simple case studies that demonstrate the application as well as the utility of the developed methods

6.1 A TESTBED FOR SYNCHRONOUS APPLICATIONS: SYNCHRONOUS JAVA

Synchronous Java (Sjava) introduces a concept of active object that corresponds to light processes. An active object is an instance of an active class declared as any Java class, with the keyword *class* preceded by *activ*e. Creation and destruction of active objects appear exactly alike in Java by direct declaration or by using the *new* and *delete* statements. What differentiates active objects from passive ones is that their methods are accessed in mutual exclusion: only one caller can access a method at a given moment. Interactions through methods between objects occur synchronously as rendezvous. Communication may be bi-directional. An active object has an extra method *@classnam*e, where *classname* represents the object class name, called its body. This method executes the concurrent behavior of the active object. This body may call other active objects methods (through rendezvous) with the usual Java syntax, or through *accept* calls using its own methods. The simultaneous waiting of several calls and acceptances is expressed with the *select* statement (close to ADA statement). Timeouts may be specified with the *waituntil* statement, and they may occur inside a *select* as an event. All synchronizing events, acceptances, calls, and timeouts may be guarded by particular conditions introduced by the *when* statement.

Inheritance may be defined between active and passive objects. Most popular modeling methodologies deploy object, dynamic, and functional models of components. The object model specifies the data components pertinent to the application, their attributes, and the operations that characterize them. The dynamic model uses state diagrams to specify operation sequence. Through dataflow diagrams, the functional model specifies data inputs and outputs, sources and sinks, and the location of transmission and reception. Basically, the object model describes the data, while the other two models describe aspects that are usually related to concurrency. The aspects captured in the functional model are closely related to message passing and parallelism. OO languages such as C++ and Java provide effective support for implementing the object model, but their support for implementing the dynamic and functional models falls short. In fact, it requires significant programmer involvement. *SJava* makes the implementation

of these two models much more straightforward. In Figure 6.1 w a simple example of an active object and its corresponding finite state interaction model is provided. The detailed discussion of the features of the language can be found in [Petitpierre 1998].

```
active class Act {
   virtual void a() {printf('a\n");}
   virtual void b() {}
   virtual void c() {}
    @Act() {
        for (; ;) {
              select  {
                accept a;
                ||
                accept b;
               }
               accept c;
}}}
```

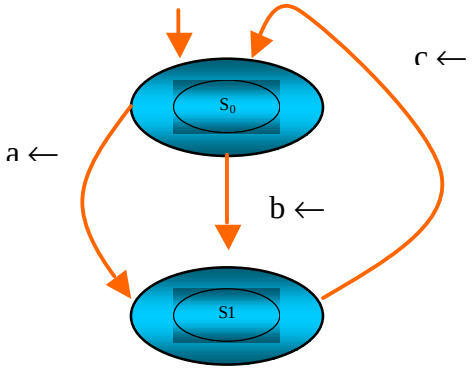


Figure 6-1: Active Class Finite State Model

6.2 VERIFICATION ENGINEER'S ASSISTANT TOOL REFERENCE PROTOTYPE

The model-based reasoning and interaction/causal process analysis methods developed in the previous chapters are so complex as to require tool support for enabling safe and systematic application. Therefore, a simple reference prototype that implements the abstract reasoning methods, as well as the interaction test suite generation, testing, and causal process analysis techniques is developed.

6.2.1 VEAT Reference Prototype Architecture and Certification Process Overview

The component certification process is augmented with tool support to guide the designer's performance of required verification and testing procedures. First, the process is described and then the components of the tool architecture that augment each step of the process are outlined.

The prototype consists of three main subsystems: (1) abstract interaction model analyzer, (2) test suite generator, executor/logger, and violation detector, and (3) causal process model generator and analyzer. These subsystems are utilized as depicted in the following process.

The Certification Process: For the role descriptions of each component under development, the designer starts with a set of abstract interaction policies. The first policy ensures that (1) substitutable roles integrate correctly and that (2) association role protocols are compatible with role interaction policies. As the abstraction is refined, the role policies for the provided roles, required roles, the mediating protocol, and the collaboration design of the component evolve with the new details. At each step in the evolution, the designer verifies whether the refined role interaction policies are safe refinements of the original policies and remain compatible with its context. The model-based analysis for checking the safe refinement and mutual policy compatibility is handled by *the abstract interaction model analyzer* subsystem. The component implementation process takes the least abstract model in the lattice to produce a concrete implementation of the component abstraction. The *test suite generator* component derives interaction test sequences for generating executable models of the component's expected environment. Then the test engineer initializes the test execution infrastructure components, exercises the component under certification with the environment emulator, and logs the pre-selected interactions using the *interaction logger framework*. The *violation detector* components that implement the interaction-sequence constraint-checking mechanism detect any violations of interaction policies. Finally, after run-time testing, post analysis on the interaction logs is performed to derive a causal process model. This model determines whether causal process constraints are satisfied.

The subsystems and constituted components of the reference prototype are implemented in the context and funding of a Small Business Innovation Research (SBIR) Phase I grant from Naval Surface Warfare Center. Figure 6.3 denotes the overall architecture of this environment containing the superset of components of the certification environment VEAT, developed as part of this dissertation. Figure 6.2 denotes only those components of the environment related to VEAT.

Figure 6.2b depicts components in the overall architecture of an interaction and synchronization-oriented dependable analysis framework for upgrading. First these components are discussed and then the reference prototype is related to the proposed framework under the SBIR work. A complex system is considered to have three

architectural levels: (1) Operational Architecture, (2) Systems Architecture, and (3) Technical Architecture. The definitions and detailed discussions of these architectural viewpoints can be found in Yilmaz and Tankersley (2000). The operational architecture part of the framework captures high-level, abstract synchronization and information flows among activities of a system, while the activities are modeled in terms of the interaction policies and coordination protocols investigated as part of this dissertation. The systems architecture is captured in terms of existing architecture definition and constraint languages. The technical architecture constitutes the implementation aspects of the system. The test suite generation, execution, causal interaction behavioral model derivation, and analysis form parts of the technical architecture dependable upgrade analysis.

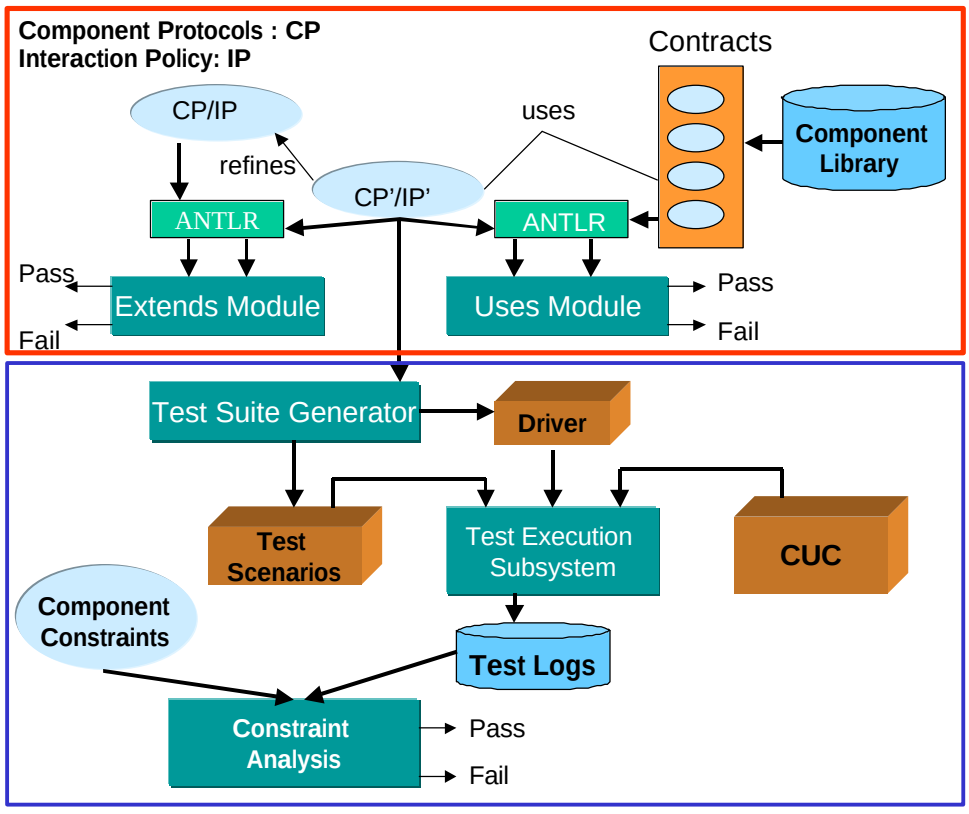


Figure 6-2: VEAT Component Layout

Figure 6.2 depicts the layout of those components that constitute the framework proposed in this research. The abstraction analysis subsystem is part of the operational architecture analysis component of the three-tiered analysis framework, while the test suite

generation, execution, causal behavioral model generation, and analysis are planned to be part of the technical architecture analysis component.

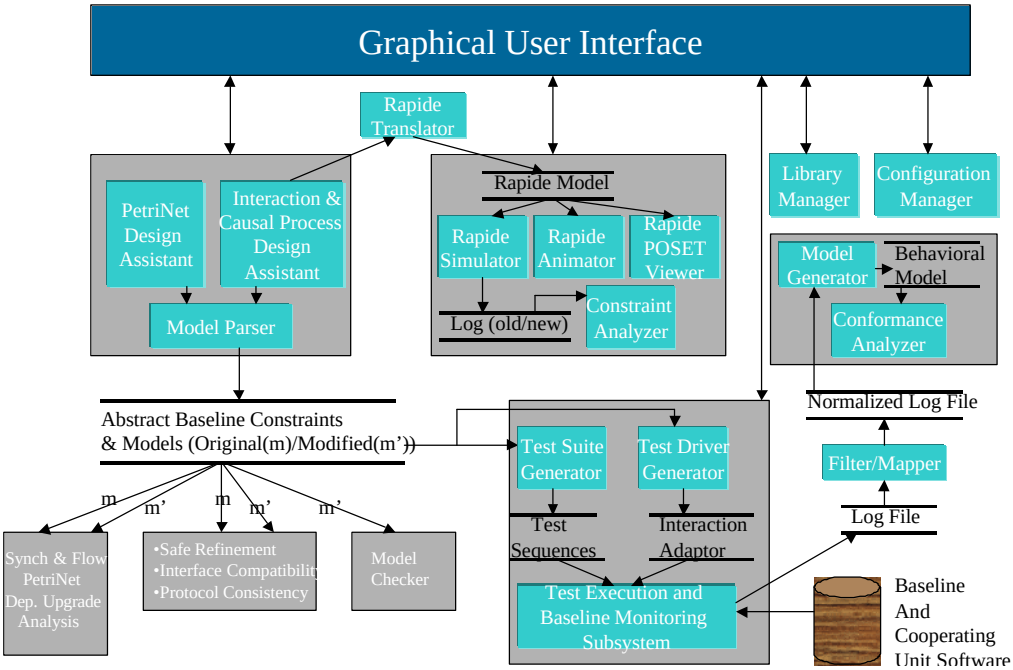


Figure 6-3: Operational/SystemArchitecture Interaction Analysis Reference Model

The abstract interaction and protocol analysis subsystem consists of three main components: the *interaction policy/protocol parser*, the *extends (safe refinement analysis)* module, and the *uses (mutual policy compatibility)* module. The protocol parser transforms the user-provided local interaction policies into internal data structures amenable to processing by the safe refinement and interaction compatibility analysis decision procedures. The component role interface interaction policies, the association role composition protocol, and the causal connections and constraints are represented using the public-domain compiler generator Java *ANTLR*, which generates the abstract component parser. The internal data structures generated by the translator are used by the compatibility and refinement analysis components. As discussed above, the *extends* module determines if a refined version of a protocol or a particular role interaction policy violates original interaction pattern constraints. On the other hand, the *uses* module determines if the associated role interaction policies are mutually compatible and whether the protocol is interaction-compatible with each of the role interaction policies.

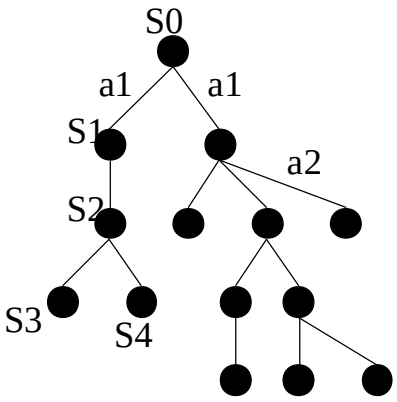
The *test execution, violation detector,* and *logging* subsystems facilitate three occurrences. The first involves component execution using the test sequences derived from the abstract component specification, while the second detects if the constituent components implementing the roles violate their expected interaction assumptions and obligations in run-time. Finally, the third collects a log of interactions as a sequence of message invocations observed during computations that occur in response to test sequences initiated by the driver. The details of test sequence generation, as well as the test execution, violation detection, and logging framework designs, are introduced in Sections 6.2.3 and 6.2.4. From *interaction logs* and the causal event model, the causal process model generator and analysis subsystem derives a causal behavioral model of the observed events. The log file contains the event sequences generated and accepted by each thread. Using the causal event model proposed in Chapter 5, the log file is then decomposed, filtered, and mapped (i.e., pattern matching) to derive a DG-based causal process behavioral model. Along with the implementation of the causal process constraint analysis algorithms, the model generation constitutes the causal process analysis subsystem. The design details of this subsystem are discussed in Section 6.2.4.

6.2.2 Safe Refinement and Mutual Compatibility

The abstract component specification contains two main sections: (1) the role interface interaction policy finite state model and (2) the composition protocol finite state model. Here, we translate each interaction policy, to enable application safe refinement and mutual compatibility. Composition protocol related decisions are similar and are based on mutual compatibility analysis, followed by protocol projection. Each interaction policy is represented in terms of import and export events and services, along with pattern constraints. Given the original and the refined versions of an interface, or the interface descriptions of two interacting role policies, the non-deterministic finite state model-based reachability analysis is used to determine if the conditions introduced in Chapter 4 are satisfied.

6.2.3 Interaction Test Suite Generator, Driver, and Violation Detector

The concrete component implementing the least abstract component at the bottom of the abstraction refinement lattice is tested using the test sequence patterns derived from the *environment* role interface, as defined in the augmented UML collaboration abstraction. The test execution and violation detection subsystem constitutes test suite generator, test execution, and violation detection components.



Test Derivation Tree

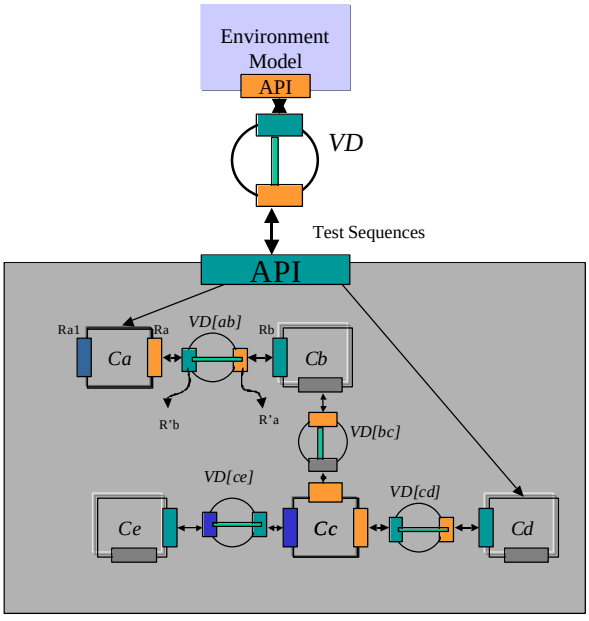
As shown in figure 6.4, the *test derivation tree* is the reachability tree derived from the interaction policy rule base. Each node and edge in the tree represents a state and permissible interaction, respectively. The test engineer is responsible for identifying the semantic constraints on the parameter domain and state space in deriving the complete test sequences. Once the test sequences are identified, they are encoded into test objects. Each test object is an active synchronous Sjava object that embeds the identified interaction sequence.the semantic constraints on the parameter domain and state space.

Figure 6-4: Test Sequence Generation

Test Suite Generation: Chapter 5 discusses the test suite generator algorithm. The *environment* role interaction policy generates the test sequences and is implemented by the test driver. Given the finite automata representing the interaction policy of the environment, the test suite generator runs a graph DFS algorithm to derive a *test derivation tree* encoding the test sequences. Each root to leaf path identifies a test sequence that must be implemented. Using domain knowledge of the implementation, the test engineer transforms the sequence into a new sequence with parameter values and type information. Each branch in the test derivation tree represents a decision made by the choice of interaction type, along with constraints on the types, destination (i.e., input or output), and state space of the interaction parameters.

Test Suite Execution Framework: Each test object initiated by the test driver behaves both like a client and server of the component under certification. The test execution framework addresses the need to implement the basic test execution and violation detection infrastructure devised in Chapter 5. The pattern, as depicted in Figure 6.5, contains the test sequence generator module, the component under certification, and violation detector/interaction mediator components. The same pattern is reused between each component implementing a provided role and each required component implementing a required role.

The structure of the violation detection pattern is depicted in Figure 6.6. The *target component* represents the component under certification and implements its functional interface. The *interaction controller* defines the composite interaction state transitions and dispatches the events to the *composite interaction adaptor*. The interaction state transition model embedded in the *interaction* controller is the composite interaction state model. It was derived from the individual interaction policies using the composite automata derivation algorithm introduced in Chapter 5.



We used a simple pattern for a subsystem that, based on a set of external events, moves from state to state. The subsystem implementing the pattern remembers its current interaction policy state and, with each new event, performs a corresponding mediating action and changes its composite interaction state. The goal is to devise a pattern to facilitate reuse of the mechanism for each associated component implementing connected roles. Therefore, the intent is to develop a set of collaborating classes so the functionality can be uncoupled from state management and violation detection.

Figure 6-5: Interaction Test Infrastructure

The pattern must be customized and reconfigured constantly to allow for different associations among the implemented roles. Figure 6.6 visualizes the structural class diagram to illustrate components of the developed mechanism. Together, the test driver, interaction controller, generic interaction controller, interaction, composite interaction adaptor, and the component under certification classes constitute the pattern, which enables customizable interaction testing and policy violation detection

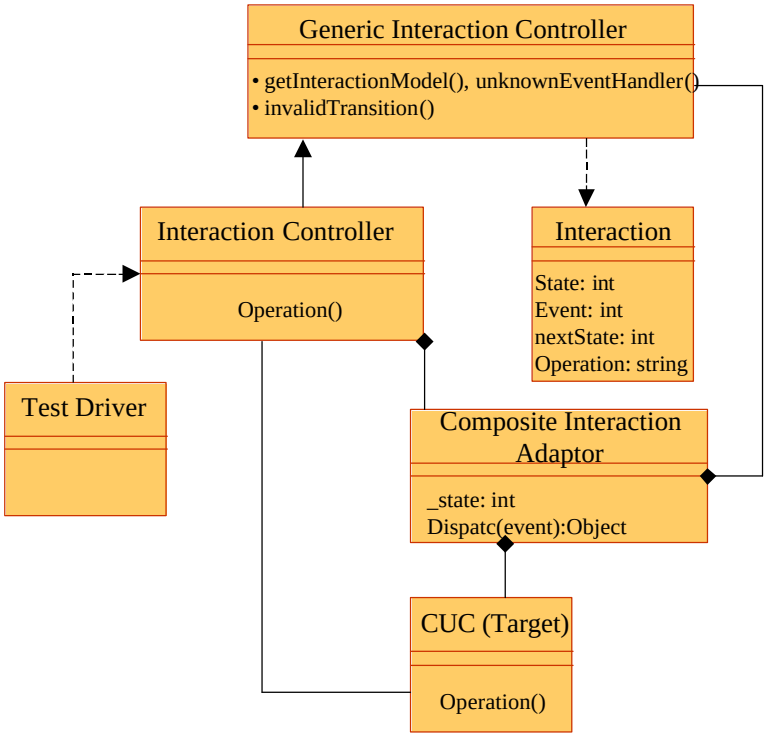


Figure 6-6: Interaction Violation Detection Design Pattern

The *test driver* is responsible for initializing the target component under certification before using the test objects to exercise it. After constructing the target component, the test driver instantiates each test object. The test object that is a member of a specific test class for the target component sends/receives interactions from the target. The *composite interaction adaptor* maintains state information and invokes methods on the target. The *interaction* defines state transition in terms of current state and input event. The generic interaction controller provides an interface to be

implemented by the *interaction controller*. This action permits the composite interaction adaptor to perform callbacks for valid transitions, as well as unknown events.

The following mechanism, shown in figure 6.7, facilitates the localization of composite interaction. The derived composite interaction state model is stored in one object (i.e., controller). This enables one to add new transitions or change the existing transitions without modifying the target. Furthermore, the mechanism isolates state-specific behavior from actual states, allowing a different implementation of target to be used with the controller. It permits the evolution of the target component; the only change is a required update of the finite state interaction model in the controller component.

The pattern facilitates interface violation detection through the following steps:

- The test driver creates the target component under certification.
- The test driver creates an interaction controller object.
- The interaction controller creates interaction objects based on the previously generated composite interaction model.
- Interaction controller creates the interaction adaptor object and specifies its initial state and target object.
- The interaction adaptor creates an interaction state adjacency matrix by requesting interaction objects from the interaction controller.
- Each time a test driver invokes a method due to the message sequences generated the message is forwarded to the interaction controller.

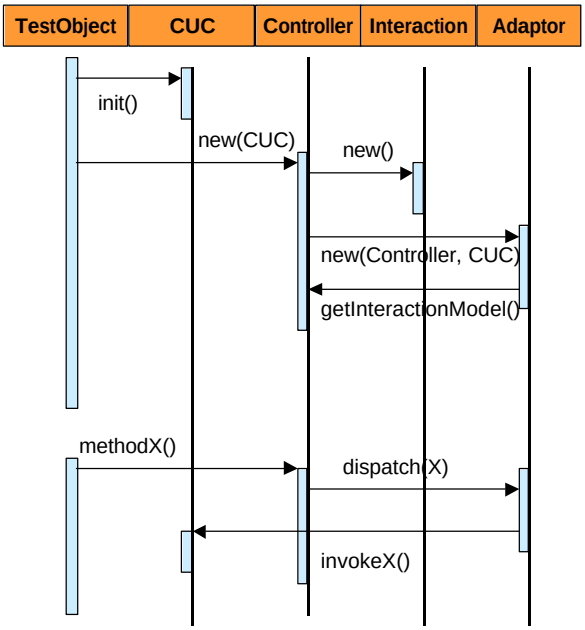


Figure 6-7: Violation Detection Protocol

6.2.4 Causal Process Analysis: Interaction Logging and Causal Model Generation

Section 6.2.3 discussed the design issues involved with test suite generation, execution, and interaction policy violation detection. The assessment of whether the implementation conforms to expected interaction policies considers such local integration issues. On the other hand, the causal process underlying the computation that results due to the

interaction sequences enforced by the environment model (i.e., test driver) facilitates observation of the effects of the emergent global behavior due to horizontal integration of constituent components. That is, deriving a causal behavior model of the constituent components in the context of the target component enables the designer to make decisions about fitness and suitability of reused components.

The fundamental processes that enable such analysis in the context of interaction perspective, as laid out in Chapter 3, were formally discussed in the previous chapter. Here, the design issues and mechanisms used during the reference prototype development are briefly mentioned. The three main subsystems developed in the context of reasoning about causal processes are (1) monitoring and logging interaction histories, (2) causal behavioral model derivation, and (3) causal process analysis.

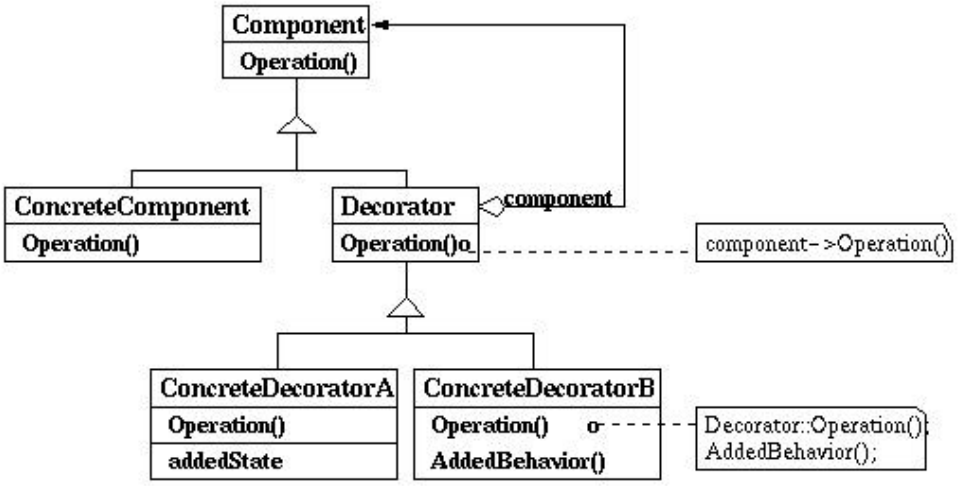


Figure 6-8: Decorator Pattern for Interaction Logging - Intent: Attach additional responsibilities to an object dynamically. **Applicability:** We use the *Decorator* to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects for responsibilities that can be withdrawn when extension is impractical.

The monitoring and logging subsystem is responsible for capturing the interaction histories among the constituent components. The causal behavior model derivator utilizes the logged interaction histories, along with the causal event model introduced in Chapter 5, to generate a DG-based causal event model for constraint analysis. The model derivation entails *log filter/decompose, causal event connection, mapping, clustering, and*

linearizing(serializing) functions that transform the original raw interaction history into a model based on the causal event model.

As illustrated in Figure 6.8, for each violation detector component, the *logging* subsystem deploys a parallel architecture utilizing the decorator pattern. This violation detector is augmented with the logging framework via decorator design pattern. That is, calls upon message receive and send operations, the decorator wraps violation detector APIs to introduce logging.

```
L4|14:35:31:787|Thread-3|0>>Philosopher.start
L4|14:35:31:791|Thread-5|1>>Philosopher.start
L4|14:35:31:794|Thread-7|2>>Philosopher.start
L4|14:35:31:797|Thread-9|3>>Philosopher.start
L4|14:35:31:801|Thread-11|4>>Philosopher.start
L4|14:35:31:841|Thread-3|Philosopher.startHungry
L4|14:35:31:844|Thread-7|Philosopher.startHungry
L4|14:35:31:845|Thread-3|Fork.get:ForkId=1
L4|14:35:31:847|Thread-7|Fork.get:ForkId=3
L4|14:35:31:848|Thread-5|PhilosopherReceive.please:ThreadId=Thread-3
L4|14:35:31:849|Thread-3|PhilosopherSend.please:CompId=1
L4|14:35:31:849|Thread-3|Fork.get:ForkId=0
L4|14:35:31:849|Thread-3|Philosopher.startEat
L4|14:35:31:850|Thread-9|PhilosopherReceive.please:ThreadId=Thread-7
```

Figure 6-9: Sample Output of Logging Calls for Dining Philosophers Problem

Figure 6.9 depicts the structure of the generated events during the execution of the component used in one of the case studies. Each event contains the time, level of monitoring, thread identifier, and event information, such as the name of the action taken. The filtering eliminates all events or messages generated by a component that are not in its interface constraint representation. This facilitates reduction of the event log to those events that will be analyzed relative to the provided causal event and process specification.

The *decompose* operation splits the interaction histories into individual thread interaction histories that incorporate the sequence of input output channel usage for each thread. Next, the *causal event connection* sub-process connects each thread service access point (i.e., receive/send message or event) to service access points of threads that interact with the thread under consideration. The *linearize* operation computes each root to leaf path, denoting a causal sequence of events either via thread execution sequence or connection-based ordering. To identify each expected causal event sequence specified by

the expected causal event description, the *map* operation first *linearizes* the causal event description that is in DAG format.

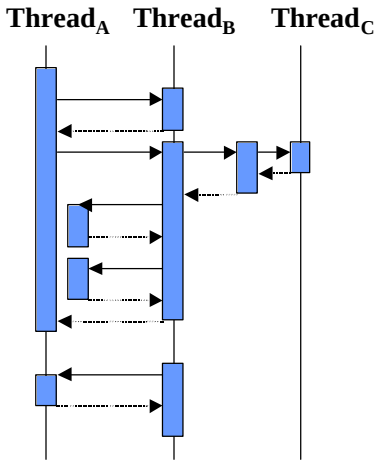


Figure 6-10: Causal Ordering of Logged Events: Two types of causal ordering are considered. As shown in figure 6.10, by utilizing the timing information, each thread develops its own ordering. First, the event log is decomposed into a set of disconnected event sequences for each thread. Each thread event sequence is then traversed, and corresponding send and receive events are identified and connected. This correspondence is achieved using a connection specification that describes which events and calls are associated with each other, as defined in the UML collaboration graph. Furthermore, to facilitate connections, each event receive log contains the thread identifier of the originating component. Each send operation records the component, as well as the thread to which it sends the message. Similarly, as components receive messages they also record the thread identification of the sending component. Every send operation on each thread order is visited and linked to the associated receive event on the receiving thread. Furthermore, every receiving event on every thread is checked to ensure that it is linked.

Then an event pattern-matching algorithm is applied to identify all occurrences of each causal event sequence in each of the observed causal event sequences. The result is a set of lists recording more abstract, coarse granular composite causal events. Each list contains the *mapped* composite causal events on a particular causal event sequence. Each observed composite causal event component should be contained in a causal process description specified in the collaboration abstraction.

The *clustering* operation processes the set of composite causal event patterns to detect complex causal process descriptions as they are specified in the abstraction. That is, each linearized causal path occurring in a causal process description is searched and clustered with other linearized causal paths, which constitute the causal process, into a new component. Each causal process constitutes a set of causal event sequences. Each set that represents a causal process is disjoint from the sets associated with the other causal

processes. The set of listed causal events is searched to detect the components of the causal process. A matrix is used to store each serialized and mapped sequence.

n# / s#	CS_1	CS_2	CS_3	….	CS_n
1	ced_{11}	ced_{21}	ced_{31}		ced_{n1}
2	ced_{12}	ced_{22}	ced_{32}		ced_{n2}
3		ced_{23}			
…… …… ….					
	ced_{1k}				

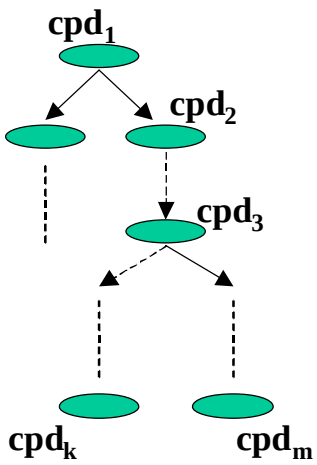
Figure 6-11: Causal Sequence Representation

The matrix, shown in Figure 6.11, lists the sequences (s#) as columns and nodes (n#) in each causal event sequence. Given a causal process identifier,

$$cpd_i = \{ced_a, ced_b, \ldots ced_m\},$$

the clustering process searches for the components of the causal process and creates a new graph containing the clustered component. As new clustered components are created, the relationships between these components are established by constructing new edges in the graph.

The cluster generation process for a given causal process identifier involves finding all composite causal event nodes constituting the causal process. Starting from the top of each list, the components of the causal process from each list are extracted to form the new causal process node. Information designating the causal sequence to which the clustered causal event node belongs is associated with the new causal process. This is required to facilitate the detection of interactions among the causal processes. The clustering of causal event nodes for the causal process under consideration ends when the lists become empty or when it is no longer possible to extract all constituent components of the causal process from the sequence lists. The same process is iterated until all causal process descriptions are identified.



The hypothetical causal process graph, depicted in figure 6.12, represents a derived behavioral model that will be checked against the user-provided constraints. According to this model, the component construction is followed by the observation of causal process, cpd_1, which causes the enactment of cpd_2 followed by $cpd3$. Cpd_3 enacts a sequence of processes which concludes with the cpd_k and cpd_m. Each causal process enacts zero or more other causal processes leading to a set of sequences of causally related processes that can be derived from the model.

Figure 6-12: Causal Process Graph

6.2.5 Constraint Analysis Subsystem

The constraint analysis subsystem implements those causal process constraint checking rules introduced in Chapter 5. To determine if the model satisfies the constraints, the subsystem takes the graph-based causal process behavioral model and the constraints as input.

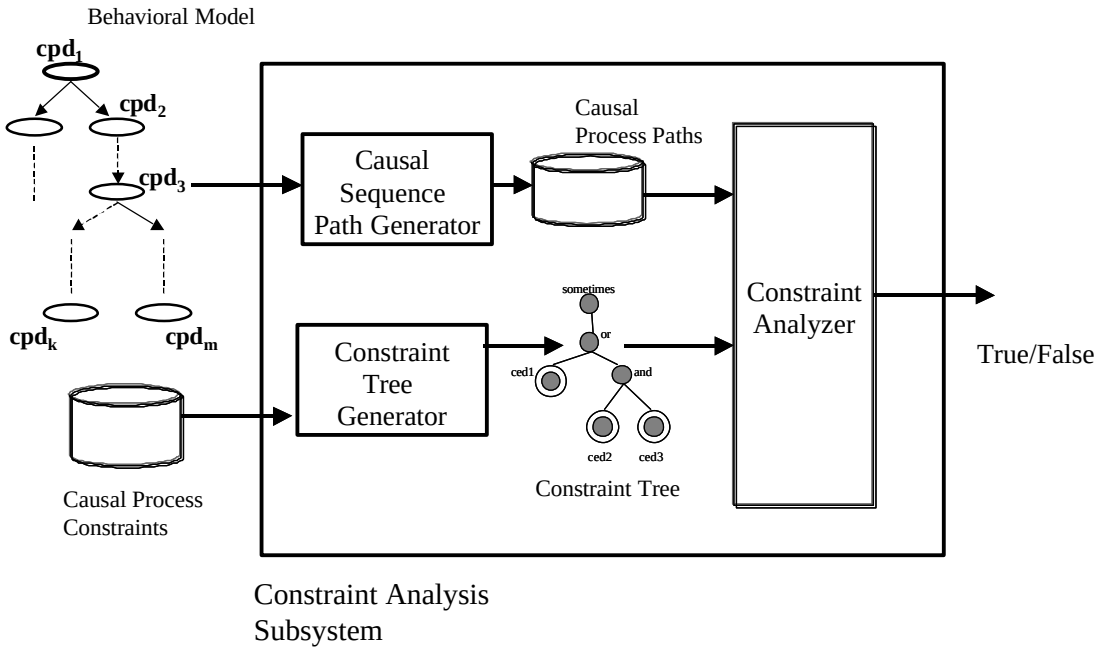


Figure 6-13: Constraint Analysis Subsystem

The constraint analysis subsystem, shown in Figure 6.13, contains three main components: (1) the causal sequence path generator, (2) the constraint tree generator, and (3) constraint analysis. From the directed causal process graph generated by the interaction history logs, the causal-sequence path-generator module derives non-circular paths of causal process sequences. The constraint tree generator module takes the causal process constraint, specified in the abstract specification, as input and constructs a tree data structure that models the user-defined constraint. The generator constructs an *abstract syntax tree*, called *constraint tree*. The constraint tree can be thought of as representing a shorthand notation called *abstract syntax*, much as a parse tree represents the structure of ordinary syntax. The leaf and internal nodes of the constraint tree are the operands (i.e., causal process descriptors) and operators, respectively. Using the causal process paths and the rules associated with the operators of the constraint grammar, the

constraint analysis module evaluates the constraint tree in a bottom-up fashion. The result is a Boolean representation depicting the satisfaction of the constraint against the observed behavioral model.

6.2.5.1 Causal Process Sequence Path Generator Algorithm

The observed causal process behavior model is an ordered tree and can be represented as a directed acyclic graph, $G_M = (V_M, E_M)$, with the property $|E_M| = |V_M| - 1$ and a specific node, called the *root*, that has no incoming edges. Each *root* to *leaf* path in the model denotes causally dependent processes observed during the computation, with respect to the particular causal interaction history from which the model is derived.

Algorithm: DFSG(G)
Input: *G(V,E):* Input graph

Output: *T*: Depth First Tree
for each $u \hat{I} V$ do
  visited[u]=false
for each $u \hat{I} V$ do
  if *visited[u*]=false then *DFS(u)*
end.

DFS(u)
  visited [u] =true
  for *each* $v \hat{I}$ *Adj[u]* do
    if *visited[v]*=false then
      parent(v) = u
      DFS(v)
end.

The above algorithm derives a depth-first tree of the behavior model. In the resultant tree every node except the root points to its parent. Hence, for every leaf node of the tree, a traversal from the leaf to the root, following the parent link of each node, leads to the reverse of the required sequences. Finally, each sequence obtained by the traversal is simply reversed to obtain the desired sequence denoting the causal process sequence path in the derived causal behavioral model.

6.2.5.2 Constraint Tree Generation and Data Structure Representation

The constraint tree is a key data structure used to evaluate the user-provided causal process constraints against the causal process sequence paths derived from the observed behavioral model. The tree is obtained by transforming the abstract syntax tree of the parsed constraint statement. The leaf nodes denote the primitive causal event sequences and the internal nodes depict the operators. As the abstract syntax tree is traversed in pre-order, the constraint tree data structure is constructed in a top-down fashion. A left child-

right sibling oriented tree implementation is chosen to facilitate compact and flexible representation of the constraint tree. A one-dimensional matrix that has as many rows as the number of nodes in an AST is created to maintain the constraint tree nodes and child/sibling relationships. Each row of the matrix contains a node that stores its type (i.e., internal or external), denoting if the node is an operator or a casual process identifier, data record associated with the operator or causal process, its parent node, leftmost child, and the right sibling.

6.2.5.3 Constraint Analysis Module

Constraint analysis involves using the semantics of operators to evaluate the constraint tree from the bottom up. The evaluation is performed against the causal process sequences derived from the observed behavioral causal process model. Each causal process model and an associated set of derived sequences represent the observed causal processes of a particular interaction scenario..

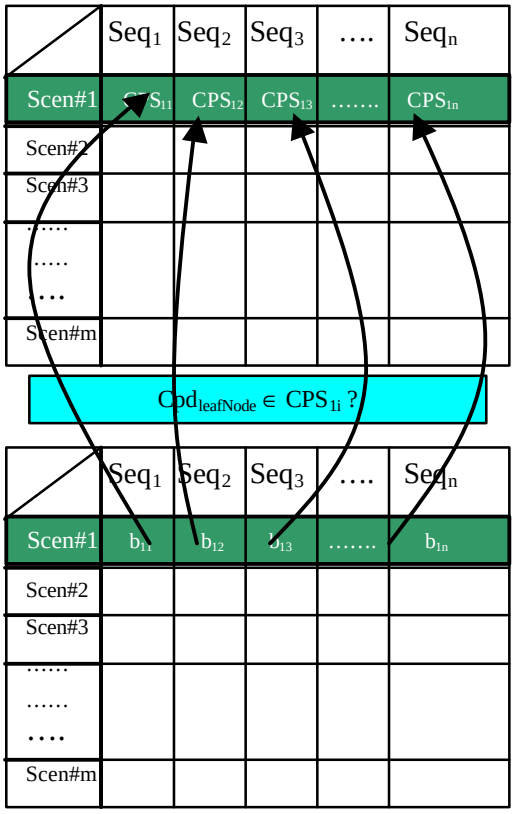


Figure 6-14: Satisfiability Notion for Constraint Tree Leaf Nodes

Each *Causal Process Sequence*, CPS_{ij}, denotes the jth causal process sequence observed under the ith scenario. Each sequence is stored as a list of causal process nodes detected during the interaction pattern match and cluster operations discussed before. Each causal process record carries information regarding relative order of the detection of its initial/final event sequence.The constraint tree evaluation starts from the leaves of the constraint tree. The causal process identifier associated with each leaf node, $cpd_{leafNode}$, is checked for membership in the causal process sequences of the interaction scenario under consideration. The evaluation returns a boolean vector $<b_{11},b_{12},\ldots, b_{1n}>$.

Each interaction scenario refers to the message sequences derived from the interaction policy of the considered component's environment. The generated causal process sequences for each interaction scenario are stored in a behavior matrix. Each row of the matrix denotes the interaction scenario under which the causal processes are observed. Each column denotes the causal process sequence identifier. The result of evaluating the causal process identifier residing in the leaf node of the constraint tree is a Boolean vector $<b_{i1},b_{i2},\ldots, b_{in}>$, where each b_{ij} denote the satisfiability of the causal process under consideration with respect to sequence j of the scenario i. Constraint evaluation starts by evaluating the leaf nodes. To identify the leaf nodes of the tree, the constraint tree is simply scanned and checked if a node left child pointer is null. Once every leaf node in the tree is processed, a Boolean vector is associated with each leaf node. Then an internal node denoting an operator is selected to evaluate the Boolean vectors associated with the child nodes. As each internal node is evaluated, its parent node is notified for its processing status. A parent internal node that has all its child nodes processed becomes a candidate for the next round of operator selection. This process continues until the root node is evaluated.

To evaluate the Boolean vectors associated with its child nodes, each operator in the grammar is associated with a specific rule. Each basic Boolean operator (i.e., *or, and, implies*) application takes as input the Boolean vectors of the operator nodes' child node and returns a single Boolean vector. The composite Boolean vector operation repeatedly applies the atomic Boolean operators to the set of Boolean values residing at the same index of the input operand Boolean vectors.

The operators *until, unless,* and *since* utilize additional information regarding each causal process node. The *until (cpdexprA, cpdexprB)* constraint operator requires *cpdexprA* to be satisfied **until** *cpdexprB* is satisfied in a causal process sequence. As each causal process is applied, the start and end times of the causal process identifiers are used to derive new start and end times for the composite expression. For each Boolean value computed in the Boolean vector that propagates up during the constraint evaluation, this information is propagated up to the parent. The *since* (*cpdexprA, cpdexprB*) operator is similar: it requires *cpdexprA* to be satisfied only after *cpdexprB* is satisfied. The *unless*

(*cpdexprA, cpdexprB*) operator requires *cpdexprA* to be satisfied only if *cpdexprB* is not satisfied. Individual members of the two Boolean vectors are checked for *exclusiveor* to decide satisfaction of this composite expression. This results in a new Boolean vector that reflects the causal process sequence paths in which *cpdexprA* occurs iff *cpdexprB* does not occur.

The *always* and *sometimes* operators apply to the complete matrix, each row of which designates the Boolean vector derived from evaluating constraints against the behavioral model generated from a single interaction test scenario. If all Boolean vectors at each row of the matrix contain at least one Boolean variable with a *true value*, the *Always* operator evaluates to true. The *sometimes* operator evaluates to true a row of the complete matrix exists that has at least one Boolean variable with a *true* value. All of these operators are based on simple matrix traverse and condition checking algorithms that determine whether the constraint expression is satisfied.

The evolution (i.e., *introduce, prevent, quarantee*) operators are implemented to compare the resultant complete constraint evaluation Boolean matrices for two successive versions of the component. That is, two behavioral models associated with the original and new designs under the designated interaction scenario are analyzed to determine if the intended purpose of the modifications is achieved. Hence, the analysis requires derived causal process sequence paths, as well as the results of constraint evaluations on each model. Evaluation of the constraint against each model results in a Boolean vector utilized by the evolution operators devised in this work.

The *introduce* operator scans the two resultant Boolean matrices to determine if at least one scenario (i.e., depicted by a row) exists with at least one Boolean variable with a *true* value in the updated version, while all of the Boolean variables in the original vector have the value *false*.

The *prevent* operator scans the two resultant Boolean matrices to determine that no Boolean variables exist under any scenario in the modified version that returned a *true* value, while there exists at least one Boolean variable that returned *true* in at least one scenario.

Finally, the *guarantee* operator determines if under all scenarios at least one Boolean variable exists in the modified version that returns *true*, while the causal process is satisfied only under some of the scenarios of the original version.

6.3 THE CASE STUDIES

To illustrate the introduced concepts, two case studies are considered: (1) a semaphore-based parameterized synchronization pattern template and (2) CORBA-Based auctioneer bidder agents in an e-commerce negotiation subsystem The first scenario focuses on reasoning about abstract interaction policies, safe refinement, mutual interaction policy compatibility, and interaction policy testing. The second scenario emphasizes analysis involving abstract interaction and causal process.

6.3.1 Semaphore-Based Parameterized Synchronization Pattern Template

Semaphores are well-known constructs that facilitate mutually exclusive access to shared resources. A client that needs to access and use a data resource has to lock it so that no other process or client can access the resource until it is released. This case study is founded on a *reuseable* parameterized and generic synchronization collaboration pattern that embodies a semaphore and a set of client components.

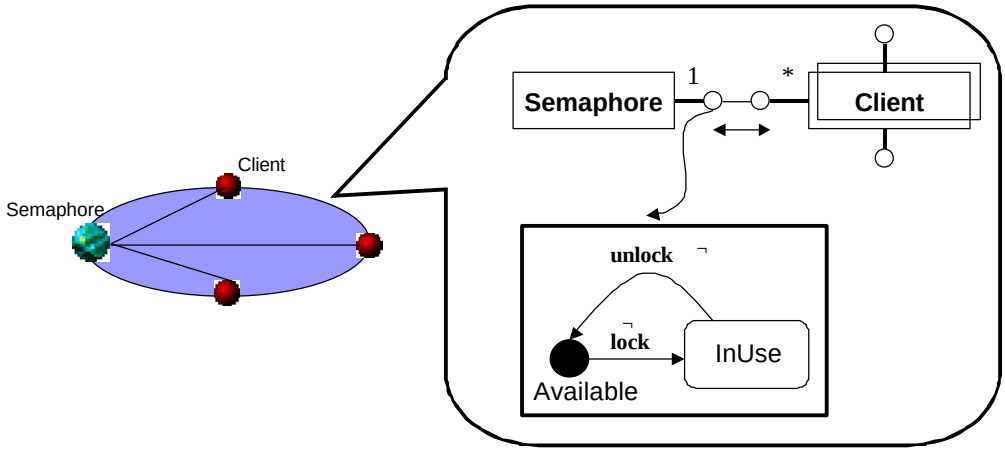


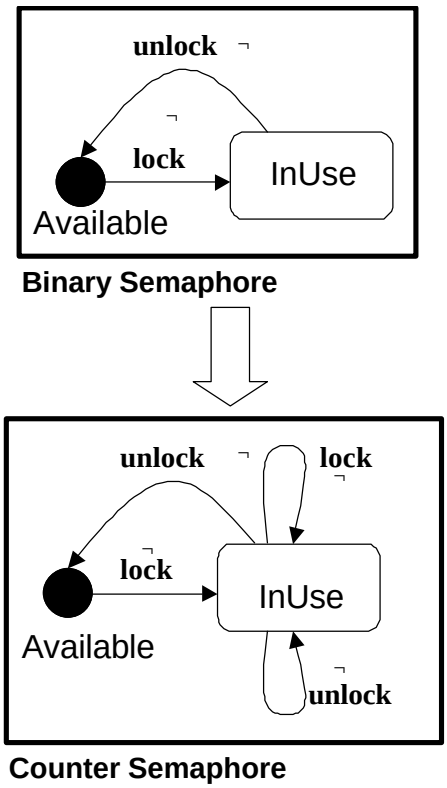
Figure 6-15: Semaphore Collaboration Pattern

The *semaphore class role* interface provides *lock* and *unlock* features used by the client components to provide mutually exclusive access to the data resource protected by the

semaphore. Given the parameterized collaboration pattern, the goal in scenario one, is to determine if the generic pattern could be instantiated with the available *counter semaphore* component. In particular, the goal determine whether the available *counter semaphore* interface discussed in Section 6.3.1.1 can be substituted for the binary semaphore. In scenario 2, it is demonstrated that the client role is compatible with the utilized semaphore role. Note, however, that as shown in Chapter 4, role compatibility is closed under the substitutability relationship. This demonstration will attempt to illustrate that the role used for the client class role is indeed compatible with the substituted semaphore role.

6.3.1.1 Scenario 1: Binary Semaphore vs. Counter Semaphores

The original semaphore role defined by the parameterized collaboration pattern associates two abstract states, a state in which only a *lock* is accepted, and another state in which an *unlock* is allowed



The counter semaphore, initially, enables only a *lock* request from clients. Then both *lock* and *unlock* are enabled. Further *lock* requests do not change the availability of the requests. However, an *unlock* request may bring the state of the semaphore to the initial state. This indicates the cases in which all the clients that requested a *lock* release the semaphore by *unlocking*; hence, the semaphore returns to the initial state denoting the condition that there exist no clients that are accessing the resource protected by the semaphore. Note that this version of the counter semaphore does not impose any upper bound on the client that can concurrently access the resource.

Figure 6-16: Counter Semaphore Interaction Policy

Upon accepting a *lock* or an *unlock* request, semaphore changes states. In this scenario we demonstrate how to determine, from the collaboration and interaction policy points of view, if an available component with a particular interaction policy is reusable in the context of the component under certification.

Method: The utilization of a *counter semaphore* and checking if it can be used in the context of the reused collaboration graph constitutes the problem

Using the method of *safe role refinement* analysis, the fitness and suitability of reusing the counter semaphore in place of the binary semaphore is demonstrated. The safe refinement of the binary semaphore would indicate that the substituted component could participate in the collaborations required by the parameterized collaboration pattern.

Results: In the following demonstration, states *A* and *B* represent the states *Available* and *Inuse*, respectively. The steps of applying *role refinement* on these two policies are as follow:

Step 1: Originally, both policies are at their initial states. The only possible transition for the binary semaphore is due to a *lock* request

[{A}, {A_1}, {lock}]

Step 2: An operation (i.e., *lock*) is selected and removed from the list and derive the reachable states using this operation to obtain the following list:

[[{B}, {B_1}, {unlock}],[{A}, {A_1}, Æ]]

Step 3: Since **[{B}, {B_1}, {unlock}]** is derived the first time, the algorithm continues to generate the reachability graph and checks for identified conditions using the transitions of the binary semaphore to obtain the following list:

[[{A}, {A_1,B_1}, {lock}], [{B}, {B_1},Æ],[{A}, {A_1}, Æ]]

Step 4: Applying the *lock* transition to both policies yield **[{B}, {B_1}, {unlock}]**, which has already been processed. The resulting list is as follows:

[[{A}, {A_1,B_1}, Æ], [{B}, {B_1},Æ],[{A}, {A_1}, Æ]]

The algorithm stops generating the reachability graphs for both semaphores and determines that counter semaphore is substitutable for the binary semaphore.

Experience and Discussion: The first step in assessing the fitness and suitability of available components involves determining if a role interaction policy is substitutable for the required role of a parameterized collaboration graph. For each participant role defined and required by the generic collaboration graph, the component engineer either develops a new component or reuses one that provides the required role.

Safe role refinement and substitutability analysis enable a designer to decide if a component fits into the context of the component under development without considering the internal dynamics of the associations between the introduced component and the rest of the pieces. Note, however, that components are usually multi-faceted and need to comply with the implicit design decisions. Therefore, the safe refinement method discussed here is not a guarantee that the component will fit into its context without any problems. It only indicates that the component will deliver the required interaction behavior as long as the interaction assumption constraints are satisfied. Before interaction policy substitutability is assessed, syntax and semantic compatibility at the level of signature and functional per-operational conformance need to be established.

The interaction policy analysis enables interaction policy compatibility and safe refinement assessment in cases where signature and behavioral compatibility are not demonstrated, mapping individual API features of the refined policy onto original features and assuming that these features are syntactically and behaviorally compatible. Failure to demonstrate the desired interaction behavior would require further adaptation. However, practically, it is not feasible for each available component or design to meet the requirements of the collaboration pattern template. Therefore, the designer usually needs either to redevelop or wrap the specification to make it compatible with the context, develop a new role, or use the available component role as it is by attaching new collaboration association roles that connect the role to the existing collaboration context. The method used to detect role incompatibilities is illustrated in the next scenario.

6.3.1.2 Scenario 2: Synchronization Role Compatibility between Bounded Semaphore and Clients

Role compatibility analysis forms an essential component of the framework, especially in cases where available components' interaction policies cannot substitute for the participants of the parameterized generic collaboration patterns. The following example illustrates the use of a *bounded semaphore* in place of the *counter semaphore*.

Method: We demonstrate the role compatibility of the *bounded semaphore* and *client*, under the assumption that the *bounded semaphore* is not immediately substitutable for the original semaphore. The *bounded semaphore* interaction policy does not allow more than the specific number (upper bound) of clients simultaneous access to the shared resource. Figure 6.17 depicts the interaction policy of the semaphore. In the following demonstration, states *A*, *B*, and *C* represent the states *Available, InUse*, and *Full*, respectively. State *A* of the interaction policy requires a client to issue a *lock* request, followed by a sequence of *locks* or *unlocks* enabled by state *B*. When the last client accessing the resource unlocks the semaphore, the interaction policy returns to state *A*.

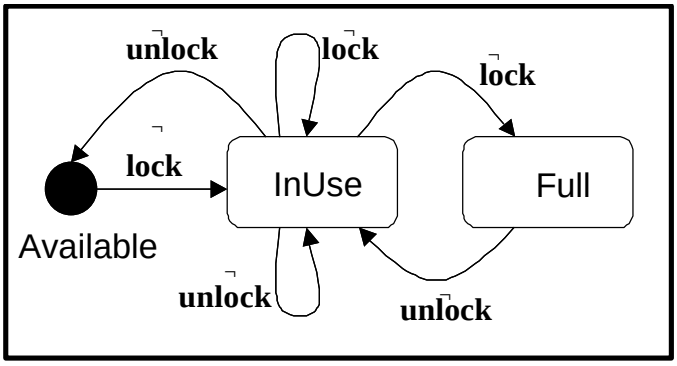


Figure 6-17: Bounded Semaphore

On the other hand, if the number of clients concurrently accessing the data resource exceeds *n*, which is the upper bound on the number of allowed concurrent accesses, the policy enters into state *C*. At state *C*, a client is allowed only to request an *unlock*. Receiving an *unlock* request, the semaphore returns to state *B*, representing the fact that the number of clients in the critical area is less than the upper bound allowed by the semaphore.

Results: The role compatibility algorithm is applied to the interaction policies of the *client* and *bounded semaphore*. The client interaction assumption policy is similar to the original *semaphore*, but the directions of the messages are reversed. That is, while the *lock* and *unlock* messages are the incoming methods calls for the semaphore, the same messages are outgoing (send) calls for the client component. In the finite state model, state *A* enables issuing only a *lock* request, while state *B* enables only a *unlock* request from the client. The steps involved in applying role compatibility are similar to those of role refinement. Note, however, that the obligations of the client are checked against the assumptions of the semaphore. That is, it is essential to make sure that the semaphore is able to accommodate the incoming requests.

Step 1: Both policies are at their initial states. The only possible transition for the binary semaphore is caused by a *lock* request. The semaphore is either at state *A* or state *B*. That is, there might already be clients in their critical region.

$$[\{A\}, \{A_1, B_1\}, \{lock\}]$$

Step 2: The algorithm selects and removes a permissible operation (i.e., lock) from the list of operations of the *client* and using this operation derives reachable states to obtain the following list:

$$[[\{B\}, \{B_1, C_1\}, \{unlock\ \}], [\{A\}, \{A_1, B_1\}, Æ]]$$

Step 3: The unlock operation is applied to see if it can be processed by the semaphore, leading to the following list.

$$[[\{A\}, \{A_1, B_1, C_1\}, \{lock\}], [\{B\}, \{B_1, C_1\}, Æ\ ], [\{A\}, \{A_1, B_1\}, Æ]]$$

Step 4: The application of the lock operation to the states listed in $[\{A\}, \{A_1, B_1, C_1\}, \{lock\}]$ leads to $[\{B\}, \{B_1, C_1\}, \{unlock\ \}]$ that has already been derived and extended in the reachability analysis. Hence, the algorithm stops here and returns that the policies are compatible.

Experience and Discussion: Role compatibility analysis is similar to role refinement with a slight variation of the properties checked during reachability graph generation. In

particular, the algorithm verifies if the obligations (i.e., output stream constraints) of a role (i.e., client) with respect to a particular role's (i.e., semaphore) assumptions constraints can be satisfied.

Role refinement and compatibility analysis are based on simultaneous state space reachability analysis using the transition rules of the original component (in the case of refinement) or the transition rules of the role, the obligations of which are checked for compatibility with respect to the assumptions of another role. Therefore, both algorithms have similar overall structures, with similar properties being checked at each step. At every reachable state, pairs are identified due to the permissible transitions of the obligation constraints of a role, and the method determines whether a receiver state exists that permits all incoming messages.

Furthermore, it is determined whether policies are failure compatible. That is, if the state of the receiving role fails to handle a particular message, then as a result of the exchange message sequence, that message cannot be sent by any of the reachable states of the obligations policy.

6.3.2 Auctioneer - Bidder Agents in an E-Commerce Negotiation Subsystem

In this case study an auction negotiation subsystem that constitutes auction manager, auctioneer, and bidder components, is modeled as depicted in Figure 6.18.

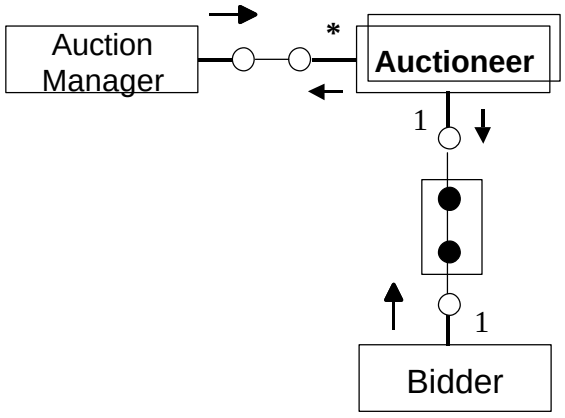


Figure 6-18: Auctioneer-Bidder Collaboration

When a customer uploads a task agent in the auction, a bidder component is instantiated to mediate the customer's requests to the auction subsystem. To facilitate managing the bidder and ensure that it complies with the rules of bidding, the auction manager instantiates an auctioneer object for each bidder.

The auction manager is responsible for instantiating an auctioneer for each bidder bound to the auction component and for managing the awareness of new incoming high bids by the auctioneer components. The bidder has two roles: one with respect to the customer; the other with respect to the auctioneer. The auctioneer is also assigned to two roles: one with respect to the auction manager and the other with respect to the bidder. In this case study the focus is on the **auctioneer and bidder** roles with respect to each other, as well as their association roles.

When the auction terminates or times out, the manager sends an *auctionOver* message to the auctioneer.

6.3.2.1 Scenario 1: Auctioneer and Bidder Role Compatibility Analysis

The first scenario identifies candidate components that will substitute for the auctioneer and bidder participants of the collaboration pattern depicted in Figure 6.18. Figure 6.19 shows the auctioneer and bidder interaction policies of the reused components. The figures on the left and right denote the auctioneer and bidder policies, respectively.

Although, ideally, the interaction policy compatibility analysis requires signature and per-operational conformance for individual features and services (i.e., methods), practically it is not feasible to achieve complete 1-1 mapping and compatibility on disparate components reused in the context of the component under development.

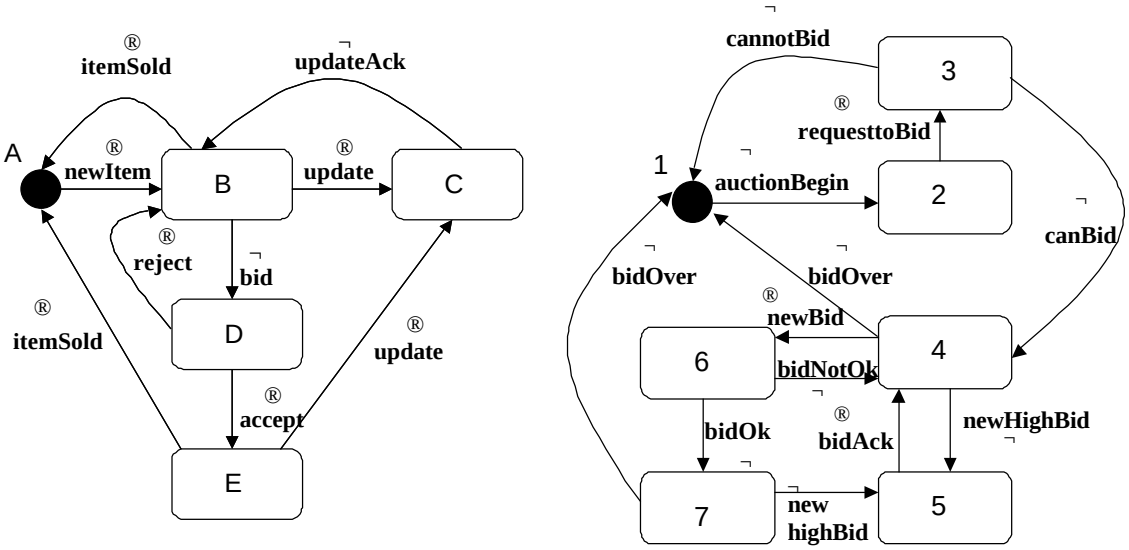


Figure 6-19: Auctioneer-Bidder Interaction Policies

Hence, in this scenario, it is demonstrated how interaction policy compatibility can be performed to detect protocol inconsistency between the associated components. Note, however, that even in protocols that are compatible, signature and functional compatibility are also necessary for components to be composed and function correctly.

To perform role compatibility analysis, first a mapping is defined between the individual messages of the role interfaces. In the case study, the messages sent by the *auctioneer* are *newItem, update, reject, accept,* and *itemSold*, whereas the messages received by the *bidder* are *auctionBegin, canBid, cannotBid, newHighBid, bidOk, bidnotOk, bidOver*. The outgoing messages of *auctioneer* are mapped onto the incoming services of the *bidder* as follow: (*newitem, auctionBegin*), (*update, newHighBid*), (*itemSold, bidOver*), (*accept, bidOk*), (*reject, bidnotOk*). In addition, the outgoing messages of the *bidder* are mapped to the incoming messages of the *auctioneer* as follow: (*newBid, bid*), (*highBidAck, updateAck*). Note that there exist messages that are not mapped on both sides. Next, the synchronization scenario between auctioneer and bidder components is discussed.

Synchronization of Auctioneer and Bidder Components: When the bidder component connects to the auctioneer component's active thread, the auction is initiated, and the bidder receives an *auctionBegin* message from the auctioneer interface including information about the item being auctioned. After receiving this message, the bidder indicates its intention to participate in the auction by sending the auctioneer a *requestToBid* message. The bidder then receives either a *canBid* or *canNotBid* response. Once granted permission to bid, the bidder can send a new bid to the auctioneer, be notified by the auctioneer that the bid is over, or be informed that there is a new high bid. In a case where the bidder sends a new bid message, the auctioneer is expected to respond by either a *bidOk* or *bidNotOk* message. In the former case, the bidder moves back to a state where it can make new bids; in the latter case, the bidder moves to a state where it owns the item. From that state the bidder can further inform the auctioneer of new bids in response to new reported high bids or can be signaled by the auctioneer that the bid is over.

Based on the above scenario and the given interaction policies, the role compatibility analysis method easily detects the inconsistency between these two policies. Basically, in the above scenario, the bidder has a two-phased interaction policy; that is, it first authenticates itself with the auctioneer in order to receive permission to bid. Therefore, the interaction states 2 and 3 of the bidder do not have a counterpart in the auctioneer interaction policy. To alleviate the problem, the component designer would develop an adaptor to mediate the mismatching or missing parameters, as well as the enactment sequence of interactions. Note, however, that the assumptions made by the protocol with respect to each role interaction policy need to be compatible with the constraints of each role interface.

The following scenario, discussed in Section 6.3.2.2, defines an association role between incompatible *auctioneer* and *bidder* components and illustrates the **protocol consistency** analysis method to determine if association end roles are compatible with the *auctioneer* and *bidder* roles.

6.3.2.2 Scenario 2: Association Role (Adaptor) Protocol Consistency

The main reason for the interaction policy mismatch between the *auctioneer* and *bidder* components is the two-phased policy of the *bidder*. In this section a mediator is developed in the form of a UML association role that adapts both polices to make them compatible with each other. The association role not only facilitates the involvement of *auctioneer* in a two-phase protocol but also performs the mappings among potential disparate message signatures, parameters, and domain mismatches.

Figure 6.20 illustrates the association role composition protocol developed during component design. The association role is developed to mediate interactions between auctioneer and bidder. The protocol not only bridges the interaction policy incompatibility but can also bridge the signature and potential parameter disorders, as well as domain differences. Remember that the bidder component has a two-phased interaction policy, which requires that it authenticate itself with the auctioneer. After receiving a *newItem* message from the auctioneer, the association composition protocol sends to the bidder an *auctionBegin* message. Being aware of the two-phased interaction policy of the bidder, the protocol expects from the bidder a *requesttoBid* message. Upon

receiving this request, the protocol immediately sends a *canBid* message to the bidder to bring both the auctioneer and bidder to a state in which they can synchronize.

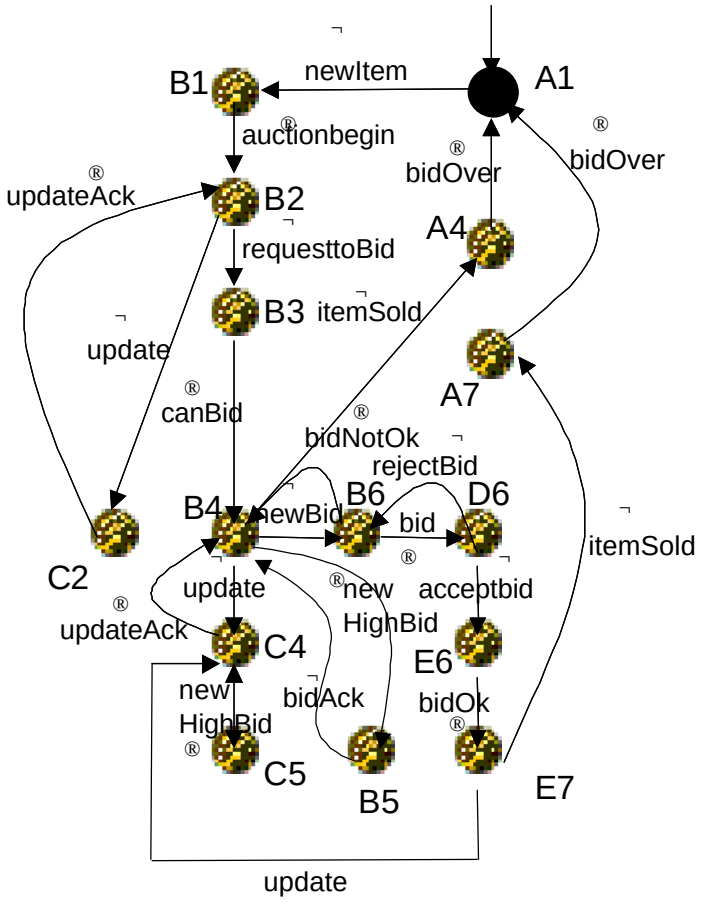


Figure 6-20: Association Role Composition Protocol

Due to the signature differences, the protocol acts as a mediator; that is, for instance, at state *B4*, when the bidder sends a *newBid* message, the protocol translates that message into a *bid* message and sends it to the auctioneer to enter state *D6*. Being aware of the interaction policy of the auctioneer, the protocol expects either an *acceptBid* or a *rejectBid* message from the auctioneer. In the case of the latter message, the protocol sends a *bidNotOk* message to the bidder. On the other hand, in the case of an *acceptBid* response from the auctioneer, the bidder receives a *bidOk* message from the mediator (i.e., association role) protocol

Association role protocol consistency is based on association end role compatibility with the participant role interaction policies. Therefore, in the next step, the

derivation of interaction policies of the protocol are demonstrated with respect to individual participants connected with the association role.

Basically, we utilize the method of reduced-order finite state model generation, introduced in Chapter 4, to project the protocol onto interaction policies that constitute assumption and obligation constraints of individual participants (i.e., auctioneer, bidder). For the sake of brevity, the approach is demonstrated by illustrating role compatibility with respect to the auctioneer role. The application for the bidder role is similar.

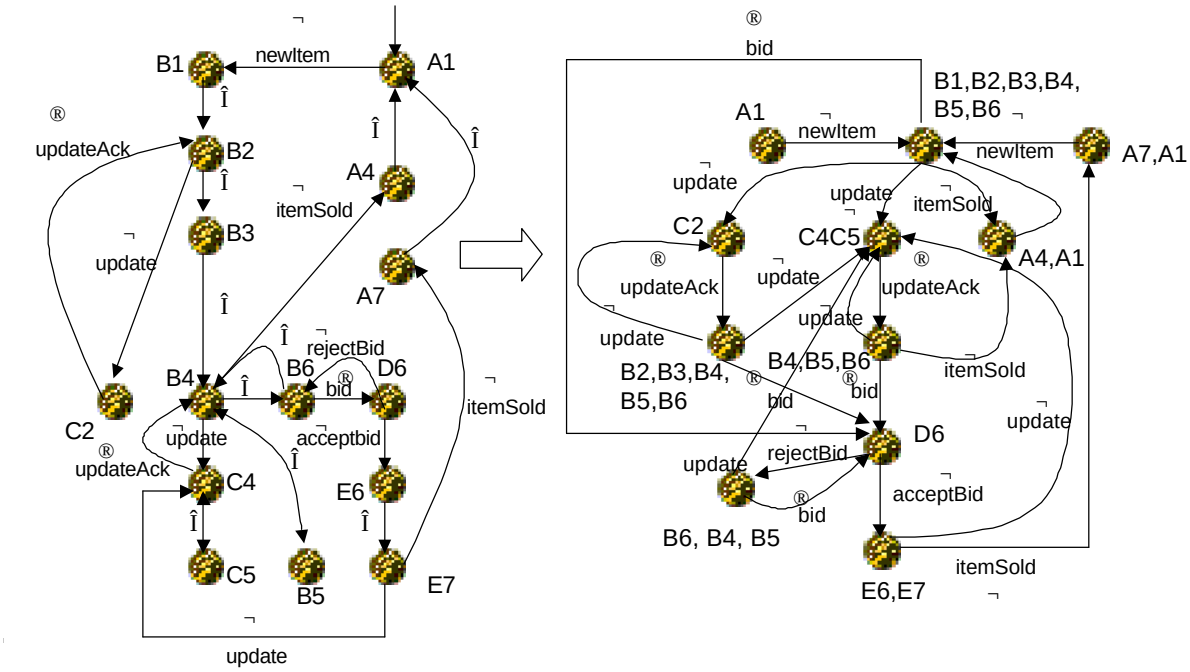


Figure 6-21: Composition Protocol Projection

Figure 6.21. Substitution of the irrelevant transitions of the protocol with $\hat{I}$*-transitions* is the first step.

Derivation of a new non-deterministic interaction policy without $\hat{I}$ transitions follows. The policy contains only those transitions relevant to the auctioneer role. The following rule is applied to obtain a new policy with $\hat{I}$ transitions:

$$M'_{protocol} = (Q_p, S_P \cup \{\hat{I}\}, d'_p, q_p) \text{ where}$$

$$\forall s \in Q_p \ \forall a \in S_P, \ d'_p(s,a) = d_p(s,a) \text{ and } \forall a' \in S_p - S_P \text{ s.t. } d_p(s,a')=s'$$

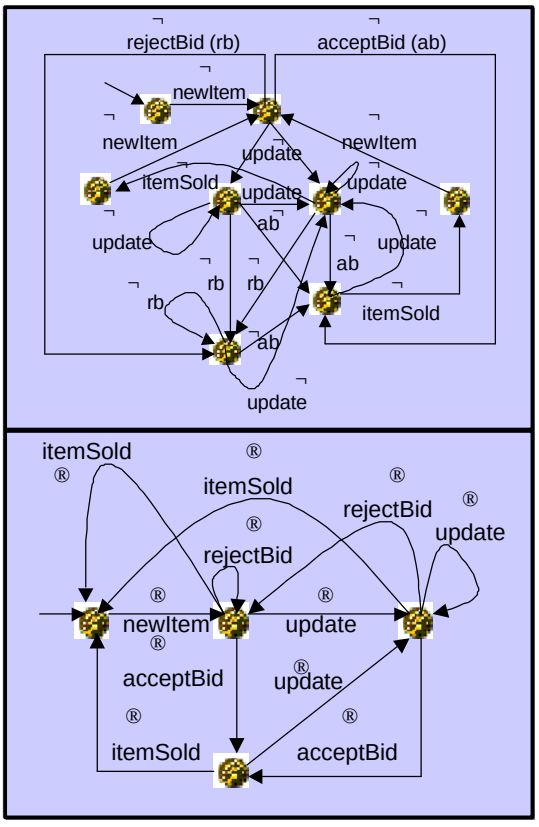
$$\text{define } d'_p(s,\hat{I})=s'$$

Once the new policy with $\hat{I}$*-transitions* denoted by ***M'$_{protocol}$*** is derived, the next step is to generate transitive closure based on $\hat{I}$ moves to find all reachable states through $\hat{I}$ transitions.

The following rules that are introduced in Chapter 4 are used to derive the new non-deterministic interaction policy:

1. $d_{\hat{O}}$**(q, $\hat{I}$) = $\hat{I}$-TransitiveClosure(q).**
2. **For *w* in S^*_P and *a* $\hat{I}$ S_P , $d_{\hat{O}}$*(q,wa)* = $\hat{I}$*-TransitiveClosure(P)*, where *P={p* | for some *r* in $d_{\hat{O}}$*(q,w)*, *p* is in d'_p*(r,a)}*.**

The third step entails analyzing role compatibility between the assumptions of the derived role (i.e., association end role) with respect to the obligations of the auctioneer role, as well as the obligations of the derived role with respect to the assumptions of the auctioneer role.



The projection of the association role composition protocol onto the auctioneer role constitutes both the assumptions and obligations of the reduced order policy with respect to the auctioneer. This policy is further projected onto the assumption constraints of the reduced order policy against the auctioneer. The resultant policy is shown in the upper half of Figure 6.22. The lower half of the figure depicts the projection of the auctioneer role onto its obligation (output channel) constraints.

Figure 6-22: Reduced-Order Policy vs Auctioneer - Association End Role Compatibility

In this scenario, the compatibility of the auctioneer's obligation constraints with respect to the assumptions of the auctioneer's association end role is considered. The compatibility check requires determining respective assumptions and obligations of association and the auctioneer. The same process need to be applied for the bidder as well.

The application of the role compatibility algorithm confirms the fact that every output trace of the auctioneer role is acceptable by the projected interaction policy denoting the assumptions of the associated end role. Furthermore, any failure of the projected policy to receive a particular message in a given state is also a failure of the auctioneer role to send that message at the corresponding state. To demonstrate the proof that the policies are indeed compatible, the steps involved in application of the role compatibility analysis method are illustrated. At each step, the necessary and sufficient conditions for compatibility are checked.

Step1: Originally policies are at their initial states.

[{A}, {A_1}, {newItem}]

Step 2: Applying the operation *newItem*, the algorithm obtains the following list.

[[{B}, {B_1}, {update, acceptBid, rejectBid}][{A}, {A_1},Æ]]

Step 3: The algorithm next applies the *update* operation at states *B and* B_1, respectively.

[[{B}, {B_1}, {acceptBid, rejectBid}], [{C},{D_1,E_1},{update,rejectbid,itemSold}],

[{A}, {A_1},Æ]]

Step 4: The algorithm then applies the *acceptBid* operation at states *B and* B_1, respectively.

[[{B}, {B_1}, {rejectBid}], [{D},{H_1},{update,sold}],

[{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ]]

Step 5: The last remaining operation applicable at state *B* is *rejectBid*. Applying the operation leads to the following extended list:

[[{B}, {G_1}, {rejectBid, update, acceptBid}], [{D},{H_1},{update,sold}],

[{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ]]

Step 6: Applying *rejectBid* at states *B* and G_1 leads back to states *B* and G_1*;* hence, the algorithm does not insert this pair of states into the list. Hence, we obtain

[[{B}, {G_1}, {update, acceptBid}], [{D},{H_1},{update,sold}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ]]

Step 7: Applying the *update* operation on states *B* and G_1, the algorithm generates

[[{B}, {G_1}, {acceptBid}], [{C},{E_1},{update, rejectBid, itemSold}], [{D},{H_1},{update,sold}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ]]

Step 8: Applying the last permissible operation, *acceptBid,* on state pairs *B* and G_1 the algorithm reaches states *D* and H_1, which are already in the list. Hence, the algorithm does not insert these state pairs to the list.

[[{C},{E_1},{update, rejectBid, itemSold}], [{D},{H_1},{update,sold}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ], [{B}, {G_1},Æ]]

Step 9: The application of operations *update* and *itemSold* leads to existing pairs of sets of states reachable by both interaction policies. Therefore, the list is not extended due to these operations. Application of *itemSold* operation leads to the following list.

[[{A},{C_1},{newItem}], [{D},{H_1},{update,sold}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ],[{B}, {G_1},Æ], [{C},{E_1},Æ}]]

Step 10: After receiving a *newItem* message, the auctioneer moves into state *B,* while the association end role moves into state B_1. Since this pair of states was already visited, it is not inserted to the list. Therefore, the resulting list is as follows.

[[{D},{H_1},{update,sold}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ],[{B}, {G_1},Æ], [{C},{E_1},Æ}], [{A},{C_1},Æ]]

Step 11: The application of *update* operation at states *D* and H_1 forces the interaction policies to enter states *C* and E_1, which already exist in the list. The application of the *itemSold* message leads to states *A* and F_1. Hence, the resultant list becomes as follows.

[[{A},{F_1},{newItem}], [{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ],[{B}, {G_1},Æ], [{C},{E_1},Æ}], [{A},{C_1},Æ], [{D},{H_1},Æ]]

Step 12: The application of *newItem* message, while the policies are at states *A* and F_1, leads to entrance into states *B* and B_1, which are already in the list. Therefore, the list becomes

[[{C},{D_1,E_1},{update,rejectbid,itemSold}], [{A}, {A_1},Æ], [{B}, {B_1}, Æ],[{B}, {G_1},Æ], [{C},{E_1},Æ}], [{A},{C_1},Æ], [{D},{H_1},Æ], [{A},{F_1},Æ]]

Step 13: Applying the *update* message to states *C* and $\{D_1, E_1\}$ would lead to *C* and E_1 that are already in the list. Then applying *rejectBid* would force the policies to enter into states *B* and G_1, which are also already in the list. Finally, applying *itemSold* message would lead to states *A* and C_1 that are also in the list. Hence, the resultant list becomes

[[{C},{D_1,E_1},Æ], [{A}, {A_1},Æ], [{B}, {B_1}, Æ],[{B}, {G_1},Æ], [{C},{E_1},Æ}], [{A},{C_1},Æ], [{D},{H_1},Æ], [{A},{F_1},Æ]]

Since all elements of the list have been traversed and there exist no other transitions to follow, the role compatibility analysis algorithm returns **TRUE**, indicating that the assumption constraints of the association role end are compatible with the obligations of the auctioneer role.

6.3.2.3 Scenario 3: Auctioneer Role Refinement: Two-Phased Auctioneer Policy

In this scenario, the interaction policy of the auctioneer role is extended in two ways. First, the auctioneer role is modified to be compatible with the authentication phase of the bidder. Then the semantics of the *update* operation is refined so that the auctioneer would periodically send the current bid status as opposed to reporting only the new high bids.

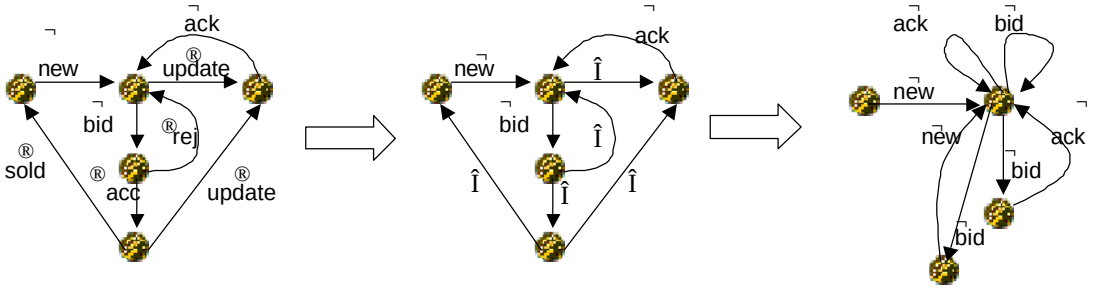


Figure 6-23: Original Auctioneer Role Assumption Constraint Derivation

Figure 6.23 depicts the three automata denoting, from left to right, the original complete interaction policy, the $\in$-instantiated policy that projects onto assumptions, and the reduced-order policy that exclusively represents the assumption constraints.

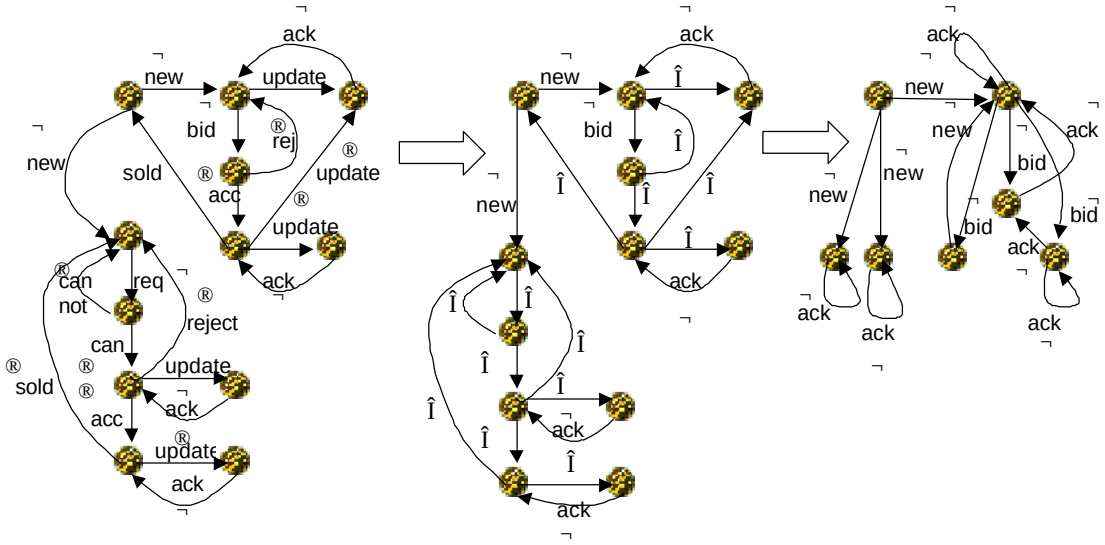


Figure 6.24: Extended Auctioneer Role Assumption Constraint Derivation

The message *new* denotes the construction operation, while the *bid* and *ack* messages denote a new incoming bid and the acknowledge message, due to a preceding update, respectively.

As shown in Figure 6.24, there are two major extensions to the auctioneer role. A construction message, denoted by *new*, depending on its parameter specification either constructs an auctioneer that requires authentication or one that requires a mediator to talk to the two-phased bidders. Furthermore, as a result of the extension, a periodic *update* leads the policy into two states depending on the *update* parameter. In the extended policy, a new state is introduced to distinguish between the cases where the bidder continues to be the owner and the bidder releases the ownership. Furthermore, the extension to make the auctioneer two-phase compliant introduces two new states that refer to cases where the bid request is accepted or rejected. Note, however, that errors are intentionally indiced to make the refined policy trace and failure incompatible.

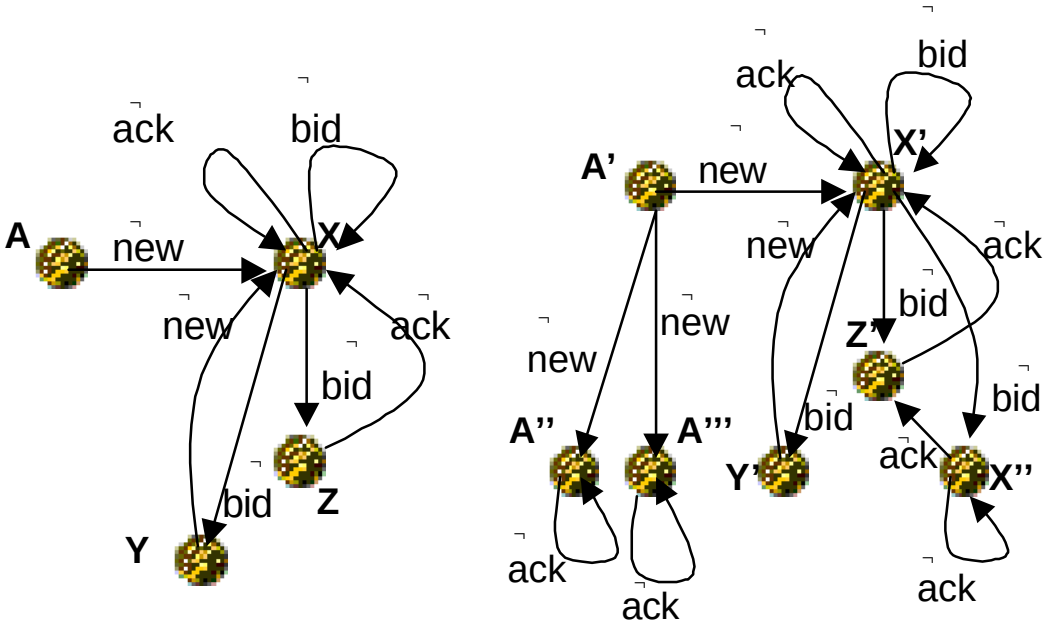


Figure 6.25: Extended Auctioneer vs. Original Auctioneer

First, after the authentication message sequence, we the *bid* operation that follows an *update* is intentionally omitted. Second, the designer omits the bidding message following periodic updates that do not necessitate changing the owner of the item. The resultant interaction assumption projections of the policies are shown in Figure 6.25. The figures on the left and right denote the original and refined assumption constraints, respectively.

To demonstrate how the role refinement algorithm determines the inconsistency of the refinement, as well as its lack of safety, several steps of the algorithm leading to the inconsistency are demonstrated. Basically, it is demonstrated that the sequence *<new, bid, ack>* would lead the original policy to enter into state *X*, while the same sequence could lead the refined policy into states *X'* or *Y'*. At state *X*, suppose the original policy accepts another *bid* message. The refined policy, however, would not be able to accept the same message unless the preceding sequence brings it into state *X'*. That is, in case the policy enters into state *Y'* due to the sequence *<new, bid, ack>* then the auctioneer would not be ready to accept a *bid* message. The following steps lead to the discovery of the failure:

Step 1: Both policies are originally at state *A* and *A'*. The reachability list is initialized to

[{A}, {A'}, {new}]

Step 2: Applying the *new* operation, the algorithm facilitates the derivation of the following list.

[[{X}, {A'', A''',X'}, {bid, ack}], [{A}, {A'}, Æ]]

Step 3: The extension of the list with the *bid* operation yields multi-states to which the policies could enter.

[[{X,Y,Z}, {X', X'',Y',Z'}, {bid,new,ack}],[{X}, {A'', A''',X'}, {ack}], [{A}, {A'}, Æ]]

Step 4: According to the formulated scenario and its associated sequence, <*new, bid, ack*>, the algorithm applies the *ack* operation to generate the reachable states for both policies.

[[{X}, {X', X'',Z'}, {bid,ack}], [X,Y,Z}, {X', X'',Y',Z'}, {bid,new}],

[{X}, {A'', A''',X'}, {ack}], [{A}, {A'}, Æ]]

[{X}, {X', X'',Z'}, {bid,ack}] element of the list leads us to the context where inconsistency emerges. In particular, the property that for every state, *s*, in ***{X', X'',Z'}*** there exists a state *s'* in ***{X}*** such that the transitions of *s'*, a subset of *s*, fail. Therefore, the policies are not failure equivalent. This results in the conclusion that a refined assumption policy of the auctioneer is not a safe extension; hence, it may lead to unexpected receptions.

6.3.3 CORBA Transport Mechanism Behavior Simulation for Causal Process Analysis using Synchronous Active Objects

The case study discussed in Section 6.3.1 focuses on reasoning and testing for local interaction policies among the composite component, its context, and constituent components. The case study under examination in this section complements the previous one to demonstrate the utility of reasoning about the causal and temporal constraints on *horizontal* causal processes that facilitate the delivery of the composite component's overall service. The interaction scenarios among the reused and developed components play a key role in reflecting emergent process, as well as the intended purpose of the reused components. Although the increasing incorporation of interaction scenarios and use cases during system development reflects the industry's acceptance that causal processes must be modeled in some form, the significance increases in the context of reuse. This is due to the fact that the fitness and suitability of reused components are confirmed only to the extent that they play the intended contextual roles. Furthermore, as opposed to development from scratch, in reuse-oriented settings, components are not designed and developed around the use cases and interaction scenarios of the context in which they are embedded.

To illustrate the utility of causal process analysis in the context of reuse, we provide a case study in which explicit understanding and reasoning about causal processes play a key role in correct functioning of the system. In the following case study, a lightweight CORBA emulation library is utilized. The library is compatible with the other CORBA packages at the IIOP level, but it can be used without any IDL compiler, special ORB, or narrowing features.

To demonstrate the approach, a simple CORBA calculator application offering a server with the four basic operations (addition, subtraction, multiplication and division) and a client invoking them is used. Each operation receives two integer arguments from the client and returns an integer. To implement the server, two different strategies are used: sequential server and concurrent server. In the original design, the server receives only a single request each time, and in the modified design, the server can handle several requests concurrently. The evolution constraints are tested on these two implementations.

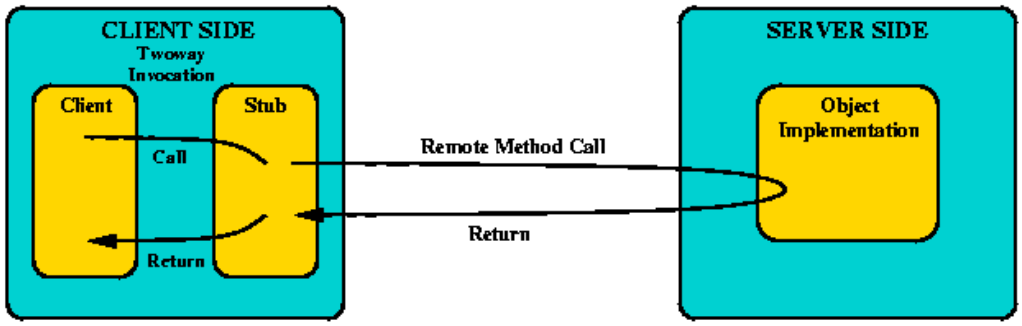


Figure 6-26: Corba RPC Implementation

A CORBA call is executed according to the following schema: the client can invoke the server methods using the two-way or deferred communication mode. In the two-way mode, as shown in Figure 6.26, the client's call is blocked until the server responds. The next figure represents this case.

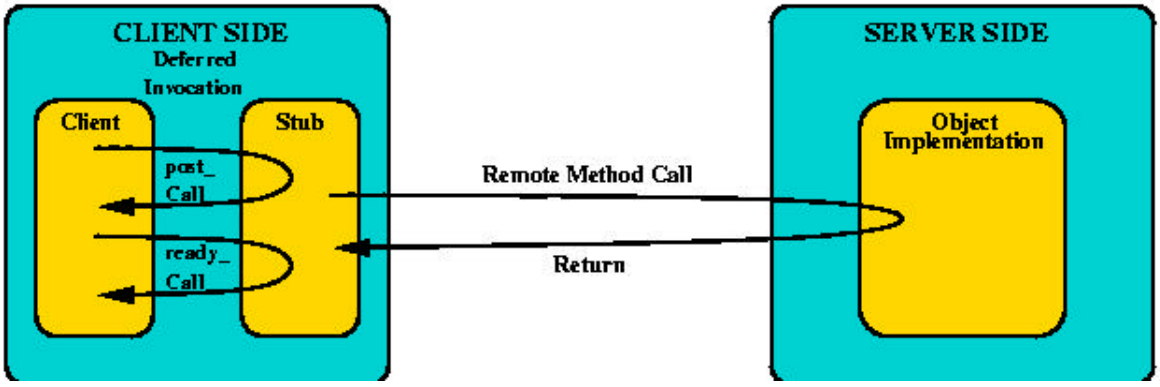
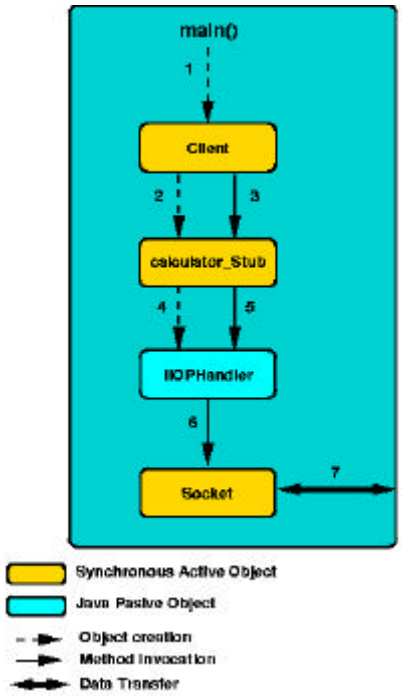


Figure 6-27: Proxy-Based Corba RPC Call (Deferred Call)

In the second mode, shown in Figure 6.27, the client makes the call and then continues in parallel with the transfer of the request. The reply can be collected at any later point. The CORBA specification also defines a third communication mode, a one-way mode that allows the client to invoke a remote method without having to wait for a response. It corresponds to the first phase of the deferred communication mode.

6.3.3.1 Scenario 1: Client-Side Causal Process Description and Analysis



The sequence involved in creating the different objects in the client side is illustrated in Figure 6.28. It is the same one for all communication modes. The main module creates the **Client** object (1) which instantiates the **calculator_Stub** object (2). The latter acts as a local representative for the remote calculator object. To invoke a method of the remote server, the client must call the equivalent method in the **calculator_Stub** (3) (if the invocation corresponds to a deferred call, the client calls a *post* method, and collects the reply by calling the *ready* method later), which creates a **IIOPHandler** object. The object encodes and decodes the arguments that will be sent and received (6/7) by the **Socket** object (previously created by the **IIOPHandler** object).

Figure 6-28: Client-Side Implementation

Method and Results: To generate the interaction logs that facilitate the generation of the behavior model, implementations of the client, calculator_stub, IIOPHandler, and Socket *Synchronous Java* components have been augmented with *logging framework* calls. Furthermore, to facilitate analysis for conformity to the intended collaboration scenario, the process has been decomposed into causal sub-processes, and the constraints are defined in terms of the processes' causal and temporal relationships. The connections among the components are defined in conformance with the collaboration graph concept introduced in Chapter 3 and discussed in Chapter 4. The first two causal processes that are defined refer to the creation of the *client* and *client stub* components:

$$cpd_1 : Main:\overline{createCalc()} \circledR\ Calc:createClient()$$

$$cpd_2 :\ Calc:\overline{createStub()} \circledR\ CalcStub:\ createStub()$$

The overlined calls in the above process description denote the outgoing events, whereas the others represent incoming requests. These causal processes capture the causally related initialization events in which the context sends a message to create the calculator client component. After receiving a construction call, the client sends a message to create

the stub component. The process ends with the client stub receiving this message, which leads to its construction and initialization.

The next three causal processes identify the two types of calls the stub can generate. These call types, discussed above, are defined in terms of causal processes as follow:

cpd_3 : Calc:$\overline{OP()}$ ® CalcStub:OP()® CalcStub:$\overline{IIOP()}$® IIOP:IIOP()®IIOP:$\overline{createSock()}$ ® Socket:CreateSock()® CalcStub:$\overline{Enc()}$

cpd_4 : Calc:$\overline{post\_OP()}$ ® CalcStub:OP()®CalcStub:$\overline{IIOP()}$® IIOP:IIOP()® IIOP:$\overline{createSock()}$ ® ClientSocket:createSock()® CalcStub:$\overline{Enc()}$

cpd_5 : Calc:ready_OP()®CalcStub:ready_OP()

The interaction between the *IIOP* handler and the *Socket* components are defined by the following causal process:

cpd_6 : IIOP:$\overline{sockSend()}$® ClientSocket:sockSend()® ClientSocket:$\overline{send()}$

The designer uses the above processes to derive constraints representing the intended temporal and causal patterns among these processes. For instance, the main component must always construct the *client* and *stub* components before initiating any operation. Furthermore, the client can make either synchronous or deferred calls, and each such call precedes the initiation of an IIOP handler object that constructs a socket each time a call is to be forwarded to the server. Basically, some of these simple constraints, along with other implied constraints, are defined in this case study:

Examined Constraints: *1. always (cpd_1 Þ cpd_2) 2. always [until ((cpd_1 and cpd_2), (cpd_3 or cpd_4))] 3. always (cpd_3 or (cpd_4 Þ cpd_5)) 4. always [(cpd_3 or cpd_4) Þ cpd_6)5. always (since(cpd6, (cpd_3 or cpd_4))*

The above constraints are tested against the behavioral model derived via observation, filtering, and mapping of interaction logs. Several modifications are introduced to detect violation of the constraints. One modification introduced to the original implementation constructed the *socket* immediately after the stub creation, thus violating the fifth constraint that requires socket creation after the detection of cpd_3 or cpd_4. The introduced method was able to discover this anomaly as well as the pattern irregularities introduced by other modifications.

6.3.3.2 Scenario2: Sequential Server Causal Process Analysis

A server that only handles a client connection at a time is called a sequential server. This kind of server does not have problems of simultaneous access and therefore does not need to address the concurrency control.

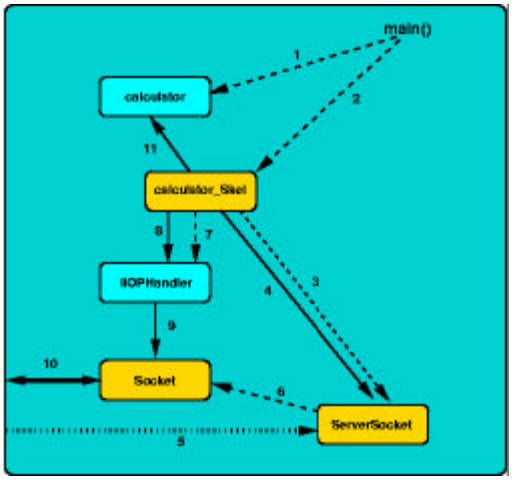


Figure 6-29: Sequential Server Implementation

As shown if figure 6.29, The **calculator** object, which implements the methods required by the client, is instantiated (1) in the main procedure. Then, the skeleton, **calculator_Skel** is created (2). It receives the reference of the **calculator**. The **calculator_Skel** object instantiates a Java **ServerSocket** object (3) and is blocked while calling its accept method (4) until a new connection (5) is established by a client. At that moment, the **ServerSocket** delivers the new **Socket** (6) to the **calculator_Skel**, which creates a **IIOPHandler** object (7) which offers several functions to decode and encode (8) the data read and sent by the **Socket** (9/10). It finally calls up the method in the **calculator** object (11).

Method and Results: Similar to the previous scenario, to simulate the above causal process and collect the interaction history, the original implementation of the components required by the sequential server scenario is augmented. The component connections conform with the UML collaboration *interaction* sequences defined for the component abstraction specification. For each relevant component, these connections denote associations among the features facilitating interaction or firing the events or methods of other components. As discussed before, such connections permit the identification of

causal relationships to derive a causal behavior model from the monitored raw interaction histories.

To demonstrate causal process constraint satisfiability or violation detection in the context of the above scenarios, the following causal processes are defined during the design stage:

cpd_1 : *Main:start* ® *Main:*$\overline{createCalc()}$ ® *Calc:createCalc()*
↓ *Main:*$\overline{createSkel()}$ ® *CalcSkel:createSkel()* ® *CalcSkel:*$\overline{ServerSocket()}$ ®
ServerSocket: ServerSocket() ® *CalcSkel:*$\overline{accept()}$ ®
ServerSocket: accept()

cpd_2: *ClientSock :*$\overline{connect()}$ ® *ServerSocket:*$\overline{Socket()}$ ® *Socket: Socket()*

cpd_3: *CalcSkel:*$\overline{IIOP()}$ ® *IIOP:IIOP()*

cpd_4: *CalcSkel:*$\overline{Decode()}$ ® *IIOP:Decode()* ® *IIOP:* $\overline{Receive()}$ ® *Socket:Receive()*
↓ *CalcSkel:*$\overline{OP()}$ ® *Calc:OP()*

cpd_5: *CalcSkel:*$\overline{Encode()}$ ® *IIOP:Encode()* ® *IIOP:* $\overline{Send()}$ ® *Socket:Send()*

Based on the above processes, the constraints imposed by the scenario are defined as follows:

Examined Constraints : *1.always (until (*cpd_1*,* cpd_2*)) 2. always(*cpd_2 Þ cpd_3*) 3. always(*cpd_4 Þ cpd_5*) 4. always (since (*cpd_3*,* cpd_2*))*

The above constraints have been checked and shown to be satisfied against the behavioral model derived from the raw interaction logs. A single deviation is induced by enabling the *calculator* component to initiate the *serverSocket* component as opposed to the *skeleton* component. Once the connection is made, the *calculator* component directs the *skeleton* to call the *IIOP:Encode()* routine. Hence, all constraints that incorporate cpd_1 and cpd_3 failed.

6.3.3.3 Scenario 3: Concurrent Server Causal Process Analysis

The last scenario, shown in Figure 6.30, intends to demonstrate the utility of the *evolution* operators defined in terms of the causal behavioral models of the original and upgraded component versions. As in the previous scenarios, the causal processes and constraints for the new version of the server are defined, and then analyzed for their satisfiability.

The new server component that must handle several client requests simultaneously is called a concurrent server. It requires more sophisticated synchronization and scheduling strategies than do iterative servers. The creation sequence for different objects in the implementation of the concurrent calculator server is shown in Figure 6.30.

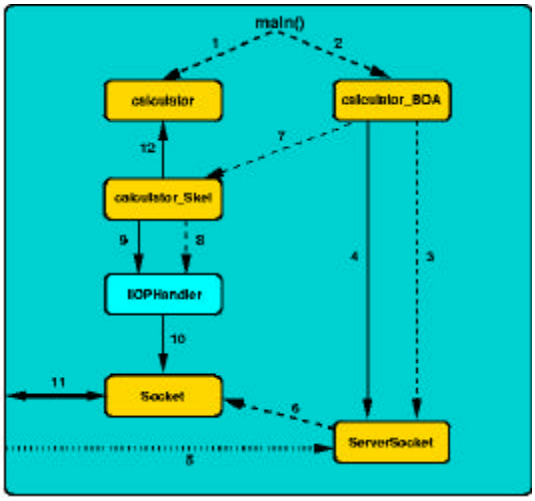
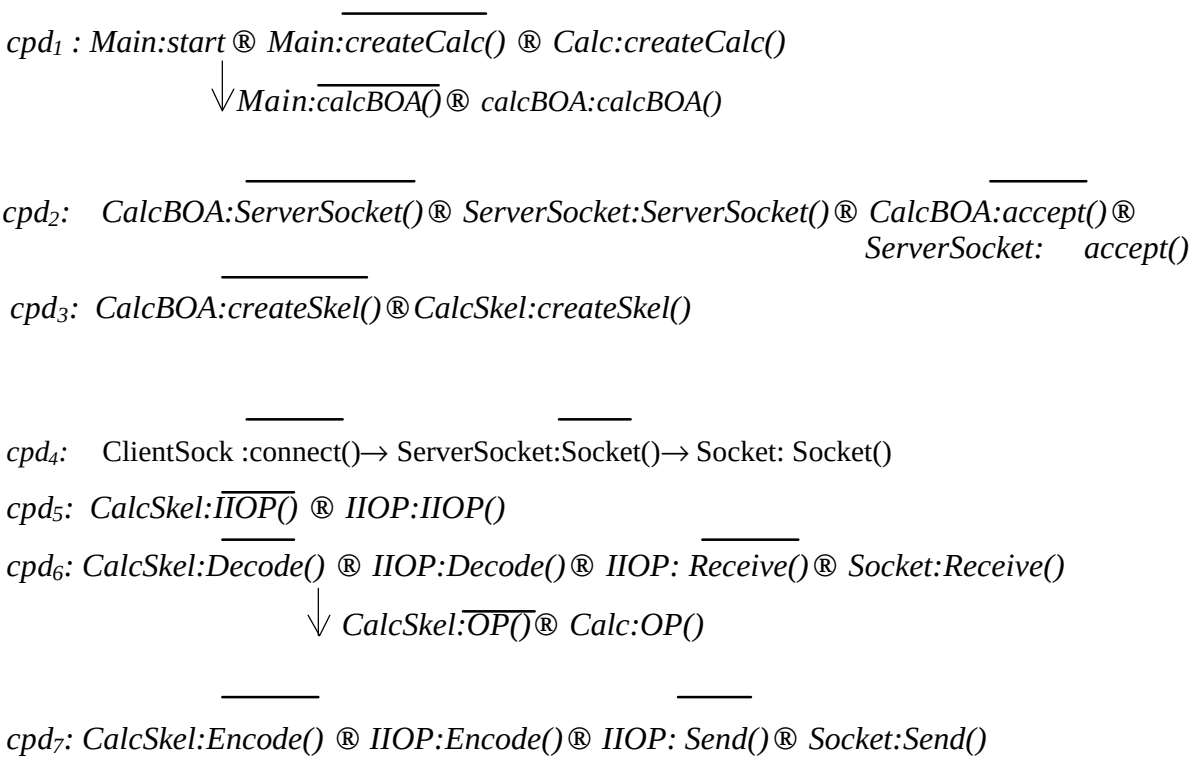


Figure 6-30: Concurrent Server Implementation

The **calculator** object is instantiated (1) in the **main** procedure. Then, the **calculator_BOA** object is created (2) with a reference to the **calculator**. The **calculator_BOA** object instantiates a Java **ServerSocket** object (3) and is blocked while calling its accept method (4), until a new connection (5) is established by a client, and a new **Socket** is created (6). At that moment, the **ServerSocket** delivers the reference of the new Socket to the **calculator_BOA**, which instantiates a new **calculator_Skel** (7). The latter creates a **IIOPHandler** object (8) which offers several functions to decode and encode (9) the data read and sent by the **Socket** (10/11). It calls the method in the **calculator** object (12). If a second connection is established, the sequence of object creations is the same as in the first case. This sequence automatically creates parallel access to the **calculator** object.

To demonstrate causal process constraint satisfiability and violation detection in the context of the above scenarios, during the design stage the following causal processes are defined:



Based on the above processes, the constraints imposed by the scenario are defined as follows:

Examined Constraints: *1. always(cpd_1 Þ cpd_2) 2. always ((cpd_2 and cpd_4) Þ cpd3) 3. always ((cpd_5 Þ cpd_6) and (cpd_6 Þ cpd_7)) 4. always (until(cpd_4, cpd_5))*

The above constraints have been checked and shown to be satisfied against the behavioral model derived from the raw interaction logs. By comparing the satisfying of an evolution constraint in the context of the original behavioral models for the sequential and concurrent server, the method tries to determine if the intended modification is encoded in the upgraded causal process as expected. Basically, it is ensured that the behavior, which enforces the creation of *ServerSocket* by the *skeleton* component is **prevented**, and that the creation of *ServerSocket* is delegated by the *main* component to the new *CalcBOA* component by **introducing** cpd_2.

6.3.4 Evaluation of The Scalability and Scope of the Certification Approach

The certification approach presented in this dissertation emphasizes a protocol-centered reuse perspective in the context of collaboration-based designs. *Frameworks* that are defined as components that provide the skeleton of some business or application function, but do so in a manner which permits "plugging-in" of customization pieces to adapt the component to a specific use provides a practical application domain for the approach. Frameworks represent not only software organizations but also facilitate the capture of the business and domain models around specific design patterns. Hence, the utility of the approach could easily extend to large-scale software deployment and verification around such reused frameworks as long as they can be modularized across small scale design patterns with low degree of coupling and to facilitate tractable reasoning.

Customizing frameworks with component substitution via framework hook points require components to offer and use services in conformance with the framework's structural as well as behavioral assumptions and obligations. Type checking and behavioral conformance analysis means via behavioral API reuse contracts and invariants via axiomatic and algebraic methods are preconditions for successful framework instantiation. The presented perspective assumes the application of such existing type and API behavioral compatibility analysis methods before establishing the synchronous interaction pattern protocol compatibility of the *collective* component features (API) as well as the framework causal process correctness among utilized components of the collaboration design. We assert that the application of the presented approach would be a precondition for reuse-based *emergent* quality-of-service (i.e., performance) conformance analysis. The developed consistency conditions and decision procedures oblige restrictions on the request scheduling and interaction mechanisms of the components. The interaction mechanism is restricted to synchronous messaging and the utilized interaction representation formalism restrict message patterns to regular expressions. Extending the framework to asynchronous message exchange and large (i.e., internet) scale distributed systems would require relaxing the assumptions of local and global consistency analysis methods as well as rules that govern the causal behavioral model derivation.

CHAPTER 7

CONCLUSIONS

In Chapter 1 of this dissertation, it is argued that a critical challenge faced by component designers and developers is understanding whether the roles implemented by (re)used software suitably fit and correctly integrate with the new context. While type theory and model-based functional specifications have provided substantial assistance in determining mismatched static functional properties, much work remains to be accomplished in the context of non-functional aspects—in particular, dynamic interaction, collaboration, and coordination. An approach for modeling the interaction policies and causal processes, along with associated constraints, is formulated in such a way as to facilitate reasoning about the interaction aspects of components, throughout their development cycle and in the context of reuse. This chapter is organized as follows. Section 7.1 summarizes the research and briefly reviews the significance of the issues raised and solutions developed. Section 7.2 discusses the original contributions. Finally, Section 7.3 concludes by focusing on potential new future research issues raised by the presented results.

7.1 SUMMARY

This research presents a method involving role-based interaction specification templates that use UML collaboration graphs and associated certification methods to address three parts of the development lifecycle of reusable components. The first part designates the specification of roles, their interconnection, imposed constraints, and abstract level reasoning about behavioral inconsistencies. The second covers the validation of role interaction policy implementations. Finally, the third part considers the compositional reasoning used to verify that a collection of components indeed satisfy system (composite component) causal process specifications.

In specifying the interaction behavior of the component under certification, explicit *roles* for the provided and required components are defined, along with enforced interaction assumptions and obligations regarding these roles. These roles denote the *compositional laws* that define how components would be assembled to form the composite component. That is, the reused and developed primitive components are assembled according to their roles within the context of the *organizational rules* (i.e., composition protocol). Roles are undertaken by individual components, and they denote the channels through which interaction should occur according to the patterns of interchange specified as *interaction policies*.

Accompanying the roles are the *organizational rules* specified as the composition protocol, which defines coordination and control of the role enactment. The interaction aspect of UML collaboration graphs is utilized to define the *behavioral laws* that designate how a component's behavior depends upon the composition and configuration of its constituent components. The connections among the roles and the causal dependencies among interactions that signify how the functionality is delivered constitute the main components of the causal process. The causal process constraints impose restrictions on the feasible processes through which the desired functionality is provided.

The key problem during the abstraction and design of a composite component or system from reusable components involves detection of inconsistencies between the assumptions and obligations of associated components. It is believed that inconsistencies can be successfully managed through analysis techniques that not only reveal and characterize them but also provide an analytical scheme to facilitate reasoning about them. To this end, safe refinement, mutual compatibility, and protocol consistency analysis algorithms are developed to support the evolutionary development and certification of component abstractions at successive levels of abstractions. For each role, a specification is given for its interaction behavior. From this specification, a component implementing the role automatically makes assumptions about and obligations regarding the external context. Methods are defined to perform compatibility checks between finite representations of a component's contextual assumptions and the expected interaction behaviors of the components with which it is intended to interact. A configuration is said

to be feasible if and only if a successful way of matching actual behaviors with assumptions for each interacting role can be found. Such methods enforce the refinement of each role specification in a way compatible with the horizontal composition.

The *safe refinement* analysis method formalizes the consistency relationships between existent role specifications at different levels of abstraction. Conversely, the *mutual compatibility* method is based on detecting interaction inconsistencies through pair-wise role interaction policy checks between the viewpoints designated by each role. Global consistency is monitored through a series of pair-wise local consistency checks. The *protocol consistency* builds on the *mutual compatibility* analysis and assures that rules governing enactment of each role, as imposed by the UML association roles, are compatible with the provided and required role specifications.

Once the roles and protocols are shown to be compatible and consistent, the implementation stage ensues. The resultant implementation must be validated to demonstrate that it interacts with the assumed context as intended and in conformance with the specified interaction policy. The context constitutes not only the environment of the component, but also the interactions between each connected role and the underlying expected causal process. The validation of the causal process is discussed in the next paragraph in the context of compositional analysis. The component is exercised by the test sequences derived from the interaction policy assumptions and obligations of the external context of the component under certification and embedded into a separate test manager component that drives the component. The interaction policies between every pair of connected roles is validated by a mediator and a violation detector design pattern that flexibly connects to both ends and controls the interaction while facilitating the logging of local interactions. The embedded composite interaction policy, derived automatically from the individual policies, determines interaction violations.

After utilizing the observed interaction logs to exercise the component, compositional analysis is performed. Causal process constraints, associated operators, and analysis methods founded on querying the satisfying of these constraints enable developers to better understand the collective behavior of components reused in a new context. The current state of the practice—which emphasizes visualization and debugging

to determine whether the interaction history evolves as expected—does not progress to understanding the complex interactions that arise from the sheer mass of interactive details dominating systems and components, particularly in the context of reuse. The *Boolean operators* on causal processes, along with *temporal operators* such as **since, unless,** and **until,** provide a means for automated reasoning and inquiry about how functionality is delivered rather than merely whether it is delivered correctly. *Evolution operators*, such as **introduce, prevent,** and **guarantee**, extend the possibilities by determining, as the designer adapts constituent components or fine-tunes the interactions, whether a component truly evolves as intended.

The application of the presented framework to *auctioneer-bidder* and to *lightweight Corba emulation* scenarios convinces us of the utility and vitality of the approach. An active marketplace is envisioned for off-the-shelf components, dynamically tested and assembled by consumers into composite components and systems.

7.2 ORIGINAL CONTRIBUTIONS

The main contribution of this dissertation is the development of a framework and associated suite of model checking and testing algorithms developed under VEAT that enable an engineer to reason about the interaction assumptions of components. In particular, the framework provides mechanisms for detecting behavioral inconsistencies among interaction assumptions and obligations, as well as the inconsistencies between a component's implementation and the published interaction policies and causal integration dependencies reflecting internal causal process constraints (i.e., internal rationale).

The certification model, which provides the context for such a framework, is based on four related principles:

(1) Interaction policies and coordination protocols need to be enforced,
(2) Enforcement needs to be decentralized,
(3) Coordination protocols, causal processes, and associated constraints need to be formulated explicitly, rather than being implicit in the code of the components involved, and

(4) It should be possible to develop and deploy policies and protocols incrementally in an evolutionary manner.

The certification framework is founded on the role-interaction perspective in the context of an abstraction-refinement model. The framework aims to facilitate reasoning about the synchronization behavior based on abstract interaction assumptions/obligations regarding component role specifications, to validate the correctness of integrated role implementations by detecting interaction violations, and to both formulate constraints and develop a means for determining deviations from intended causal processes. To achieve that objective we

- Developed reasoning methods founded on a reuse-oriented model of certification that allows engineers to predict and analyze the interaction dynamics behavior of their component interfaces at an abstract level at different levels of abstraction, given the models of its composites and the interaction model of its architectural context;
- Developed representation and reasoning methods and operators for capturing a particular aspect of internal rationale and knowledge of purpose (i.e., component internal constraints);
- Embodied the techniques developed in this research in an evolving prototype—"Verification Engineer's Assistant Tool" (VEAT). VEAT focuses on helping designers and verification engineers reason about refinement and the semantic correctness of interconnections at an abstract interface level. It also provides them at the concrete component level with methods for detecting interface violations of architectural assumptions. Finally, VEAT also provides a means for analyzing observed behavior with respect to knowledge of purpose and rationale representations and operators that are developed as part of this research.

Being founded on a clear conceptual view of component engineering, the certification model provides component engineers with systematic guidance for capturing and determining the correct realization of semantic dependencies among the abstract/concrete

components; hence, the model itself facilitates the development of various algorithms and capabilities for reusable component certification.

The certification model founded on precisely-defined semantic design dependencies such as *implements* (role interaction policy validation), *extends* (safe refinement), and *uses* (mutual compatibility) relationships among the interfaces on which the CUC depends is introduced. The interface model is represented in terms of regular grammars. The patterns of interactions are restricted to the languages that can be generated by regular grammars.

The underlying component interaction and request scheduling semantics is synchronous. The utilized role interaction policy representation formalism is based on finite automata. Hence, the expressions and coordination patterns that can be described by this formalism is constrained with the regular expressions. This restriction brings with it weaknesses in terms of expressing certain coordination mechanisms required by the domain, while it enables decidable analytical computation of consistency conditions for safe refinement and mutual role compatibility.

The necessary and sufficient conditions for preventing particular interaction policy level faults are identified (safe interface refinement, valid interface connections, interface violation detection) based on finite state models of interaction policies. The decision procedures are developed to detect extension, composition, and interface violations for architectural assumptions about the component under certification. Using these valid composition, extension, and interface violation detection procedures for verification will not guarantee that future compositions will work; rather, it will guarantee that the aspects modeled using the point of view represented by that interaction-oriented interface model will be honored.

With regard to causal processes, as mentioned in Chapter 1, there is wide agreement that design rationale presents valuable knowledge; however, there are few means for acquiring, representing, and verifying it. Furthermore, design rationale information is not only difficult to elicit from designers, but can also be fragmented, easily lost, and generally not captured in machine-intelligible form. The framework provides a representation mechanism for functional and obligation rationale in terms of

causal and temporal interactions among the used sub-components. Thus it allows engineers to capture whether the components play their intended roles and if the rationale and purpose of adaptations during the component evolution are satisfied. The certification strategy incorporates a method for detecting deviations from the expected internal integration dependencies, in addition to the public interface design dependencies discussed above. Such integration dependencies provide an account of how the desired goal of the component under certification is achieved; hence, they reflect the internal rationale of the component. To this end, a directed, graph-based, novel way of representing such expected causal dependencies, along with operators and reasoning mechanisms for their existence in the actual behavior is introduced. To be able to reason about this rationale and purpose, means for representing the causal processes, formulating constraints on them, and verifying whether the constraints are satisfied by the behavior of the component are provided. To this end, a new technique, which can be categorized as *hybrid simulation-verification*, is developed. This technique combines the best of both worlds by synthesizing an intermediate analysis method. The technique uses the execution of test scenarios to limit the generation of causal computation graphs for computations consistent with the simulation. This limited computation graph, called the *causal behavior model*, can involve orders of magnitude smaller than the potential graph that could be derived from analyzing the local interaction policies of roles.

The presented approach particularly complements other approaches that deal with the functional assumptions of the component. Components with per-functional conformance (functionally compatible) may not be protocol-compatible since they are not designed to interact under an expected protocol. Therefore, in order to achieve effective reuse, it is crucial to certify not only expected functional properties but also architectural properties.

7.3 FUTURE RESEARCH

There are several directions in which the work presented here can be extended. These extensions build on the presented framework and are consistent with the fundamental goal of supporting, in a practical way, the creation of protocol-centered correct composite components in a reuse-oriented context.

First, for practical application of the approach, it is essential that the methods are further implemented and automated to decrease and alleviate—at least to a large extent—manual intervention required from the component designer. A complete certification and development support system centered around the work presented here would require a repository search facility to identify the components and their role specifications, which are compatible with the contextual collaboration requirements of the component being developed. The current implementation of the test driver generator requires extensive manual intervention to carry out the test sequences as synchronous active object tasks. Creating driver templates from derives message sequences could enhance productivity by increasing the automation of test drivers, or by at least making them semi-automatic. Similarly, the composite-interaction state machine of the associated roles used by the interface violation detector controller is currently manually embedded. The semi-automatic generation of a controller component in the development of violation detectors would also increase productivity and reduce the potential for human errors.

The presented work emphasizes the significance of reasoning about a component's interaction assumptions and coordination aspects. Note, however, that the approach and its underlying framework are generic in nature and could be tailored for a variety of aspects, such as security, safety, and performance, as well as quality of service-related properties. Extending the problem to other quality aspects, widely known as 'ilities, is a quite fruitful research area as well. Reasoning about the composability and reusability of quality aspects of components in the context of reuse would require fundamentally different algorithms for assessing fitness and suitability. Investigation of a means for representing these aspects in such a way as to make them amenable to formal analysis would increase the utility and extent of the work presented in this dissertation. An interesting application of the role-based approach presented here would be to emerging *aspect-oriented* languages, where aspects are developed separately from the components and are interwoven as required for dealing with specific synchronization, security, and safety policies. The extension of the framework for a variety of communication mechanisms, such as callback, forwarding, acknowledgement, multicast and broadcasts, will also increase its utility and applicability.

Returning to the topic of interaction and coordination aspects, another area of interest involves multi-agent systems which, we believe, currently lack proper verification methods. Agent-based software engineering is emerging as a new paradigm [Jennings, 2000], based on the argument that it has the potential to significantly improve the theory and practice of modeling, designing, and implementing systems. An agent is advocated as an encapsulated component that is autonomous, proactive as well as reactive. Agents are situated in an environment and, in order to achieve their objectives, are capable of flexible, complex goal-oriented interactions. The interaction, coordination, and collaboration aspects of agent-based systems are pervasive. Hence, multi-agent systems designers are often interested in local agent interactions and the emergence phenomena resulting from them. The intent is to further investigate the agent-based modeling domain to understand the requirements, as well as to distinguish aspects of new agent-based verification and testing methods.

The framework introduced in this dissertation requires role interfaces and their connections to be defined in advance. For instance, in the auctioneer-bidders problem the connections are pre-defined to reason about interaction compatibility and protocol consistency. Note, however, that dynamically constructed entities and connections exist during the execution. Hence, proper extensions to the current framework would be essential for handling run-time reasoning about the compatibility of connections in the context of highly dynamic and potentially mobile execution architectures with flexible adaptive configurations. The causal behavioral model generation process constitutes several sub-processes that manipulate and transform the raw interaction log file. Current implementation of *filtering* and *mapping* processes require more flexible control to easily specify interesting and relevant events and aggregate events, causal event descriptions, and even causal processes at any level of abstraction desired by the component designer. Current aggregation and mapping strategies are hard-wired; that is, causal event sequences of causal processes are identified and fixed to detect individual causal processes. To move at different levels of abstraction and to develop associated reasoning concepts suitable for each level, flexible methods are essential.

BIBLIOGRAPHY

Addy E. (1997). "The Application of V&V within Reuse-Based Software Engineering," In Proceedings of the Eight Annual Workshop on Software Reuse.

Alfaro de L., T. A. Henzinger (2001). "Interface Automata," Software Engineering Notes, vol. 26, no.5, pp.109-129.

Allemang D. (1997). "Design Rationale and Reuse," In Proceedings of the Eight Annual Workshop on Software Reuse.

America P. (1989). "Issues in the Design of a Parallel Object-Oriented Language," Formal Aspects of Computing 1, 4, 366-411.

Arapis (1995). "A Temporal Perspective of Composite Objects," In Object-Oriented Software Composition (Nierstrasz and Tsichritzis Eds.). Prentice-Hall, UK

Barbey S. and A. Strohmeier (1994). "The Problematics of Testing Object-Oriented Software," Software Quality Management II Vol 2: Building Quality into Software, Computational Mechanics Publications, Boston, 411-425.

Beizer B. (1990). *Software Testing Techniques*, 2nd ed, Van Nostrand-Reinhold Company, New York, NY.

Berard E. (1992). *Essays on Object-Oriented Software Engineering*, Volume 1, Prentice-Hall, Englewood Cliffs, NJ.

Beugnard A., J.-M. Jezequel, N. Plouzeau, and D. Watkins (1999), "Making Components Contract Aware," IEEE Computer, vol. 32 no.7, 38-45.

Binder R. (1994). "Testing Object-Oriented Systems: A Status Report," American Programmer 7,4 (April), 22-28.

Binder R. (1995). "Trends in Testing Object-Oriented Software," IEEE Computer, October, 68-69.

Butkevich S., M. Raneco, G. Baumgartner, M. Young, "Compiler and Tool Support for Debugging Object Protocols," Software Engineering Notes, vol. 25 no. 6, pp.50-59.

Booch G. (1991). *Object-Oriented Design with Application*s, Benjamin Cummings, Menlo Park, CA.

Booch G. (1994). *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings, Redwood City.

Buchi M., M. Wolfgang (1997). "A Plea for Grey-Box Components," In Proceedings of the Foundations of Component-Based Systems Workshop, 39-49.

Campbell R. H. and A. N. Habermann. (1974). *The Specification of Process Synchronization by Path Expression.* vol 16. Springer-Verlag, Berlin, 89-102.

Cheatham J. T. and L. Mellinger (1990). “Testing Object-Oriented Software Systems,” Proceedings of the 1990 ACM 18th Annual Computer Science Conference, ACM Press, New York. 161-165.

Chen Y., B. H. C. Cheng (1997). “Formally Specifying and Analyzing Architectural and Functional Properties of Components for Reuse,” In Proceedings of the Eight Annual Workshop on Software Reuse.

Chen Y. T. and C. K. Low (1995). “Dynamic Data Flow Analysis for C++,” Proceedings of the Asia-Pacific Software Engineering Conference, December, IEEE Computer Society Press, Los Alamitos, CA. 22-28.

Chung C.-M. and M.-C. Lee (1992). “Object-Oriented Programming Testing Methodology,” Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering, June, IEEE Computer Society Press, Los Alamitos, CA. 378-385

C. IBM. (1992). “Evaluating Reusable Software Assets: Criteria and Procedures,” Tech. Rep. ASSET_A_252, ASSET, 1992.

Clarke E. M., O. Grumberg, D. E. Long (1994). “Model Checking and Abstraction,” ACM Transactions on Programming Languages and Systems, 16, 5, 1512-1542.

Coad P. and E. Yourdon (1991). *Object-Oriented Analysis*, 2nd ed., Prentice-Hall, Englewood Cliffs, NJ.

Cormen H. T., C. H. Leiserson, R. L. Rivest (199o). *Introduction to Algorithms*, MIT Electrical Engineering and Computer Science Series.

Cramer J., E. Doberkat, and M. Goedicke (1994). “Formal Methods,” In Software Reusability (Eds.Schafer *et al.*). Ellis Horwood.

Dellarocas C. (1997a). “Software Component Interconnection Should Be Treated as a Distinct Design Problem,” In Proceedings of the 8th Annual Workshop on Software Reuse (WISR), Columbus, Ohio, March 23-26, 1997.

Dellarocas C. (1997b). “Towards a Design Handbook for Integrating Software Conmponents,” In Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST'97), Pittsburgh, PA, June 2-5, 1997, 3-13.

Dijkstra W. E. (1976). *A Discipline of Programming*. Prentice-Hall.

D’Souza J. R. and R. J. LeBlanc (1994). “Class Testing by Examining Pointers,” Journal of Object-Oriented Programming 7, 4 (July), 33-39.

Doong R.-K. and P. Frankl (1994). “The ASTOOT Approach to Testing Object-Oriented Programs,” ACM Transactions on Software Engineering and Methodology 3, 4 (April), 101-130.

Dunn M. F. and J. C. Knight. (1996). “Certification of Reusable Software Parts,” Tech. Rep., University of Virginia, August 1992.

Edwards *et al.* (1998). A Framework for Interface Violations in Component-based Software,” In Proceedings of the Fifth international Conference on Software Reuse, 46-55.

Edwards S. (1997a). “Representation inheritance: A safe form of “white box” code inheritance,” IEEE Transactions on Software Engineering, 23,2, 83-92.

Edwards S., D. S.Gibson, B. W. Weide, and S. Zhupanov (1997b), “Software Component Relationships,” In Proceedings of the Eight Annual Workshop on Software Reuse.

Edwards S. (1995). “A Formal Model of Software Subsystems,” Ph.D. dissertation. Department of Computer and Information Science, Ohio State University, Columbus.

Engels G., J. M. Kuster, L. Groenewegen. (2001). “A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models,” Software Engineering Notes, vol. 26, no.5, pp.186-195.

Ernst, G. W., Navlakha J. K., and Ogden W. F. (1982). “Verification of Programs with Procedure Type parameters,” Acta Inf. 4, 2, 522-543.

Fahmy H. and D. Blostein. (1992). “A Survey of Graph Grammars: Theory and Applications,” In Proceedings of the 11th International Conference on Pattern Recognition. 294-298.

Fiadeiro J. L. and T. Maibaum. (1994). Verifying for Reuse: Foundations of Object-Oriented System Verification,” In Proceedings of Theory and Formal Methods of Computing 94, 235-257.

Fiedler P. S. (1989). “Object-Oriented Unit Testing,” Hewlett-Packard Journal, 40, 2 (April), 69-75.

Firesmith G. D. (1993). “Testing Object-Oriented Software,” Proceedings of the 11th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA’93), Prentice-Hall, Englewood Cliffs, NJ. 407-426.

Firesmith G. D. (1996). “Pattern Language for Testing Object-Oriented Software,” Object Magazine 5, 9 (January), 32-38.

Fisler K. and S. Krishnamurti (2001). “Modular Verification of Collaboration-Based Software Design,” In Proceedings of the 8th European Software Engineering Conference, 152-163.

Floyd W. R. (1968). “Assigning Meanings to Programs,” In Proceedings of the Applied Mathematics. American Mathematical Society. 19, 19-32.

Freedman S. R. (1991). “Testability of Software Components,” IEEE Transactions on Software Engineering, 17, 6, 553-564.

Gannon J., P. McMullin, R. Hamlet (1981). “Data Abstraction Implementation, Specification, and Testing,” ACM Transactions on Programming Languages and Systems 3, 3 (July), 211-223.

Garlan D., R. Allen, J. Ockerbloom (1995). “Architectural Mismatch: Why Reuse is So Hard,” IEEE Software, November, 17-26.

Gehani N. and A. D. McGettric (1986). *Software Specification Techniques*. International Computer Science Series, Addison-Wesley.

Gibson S. D. (1997). “Behaviral Relationships Between Software Components,” Ph.D. thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio.

Gibson S. D. and B. W. Weide (1997). “Semantic Spaces for Specifications and Templates,” In Proceedings of the Foundations of Component-Based Systems Workshop.

Gluch P. D. and C. B. Weinstock (1998). “Model-based verification: A Technology for Dependable System Upgrade,” CMU/SEI - 98-Technical Report-009.

Goguen J. A. (1986). “Reusing and Interconnecting Software Components,” IEEE Computer, (February), 16-27.

Goguen J. A. and J. Meseguer. (1986). “Extensions and foundations of object-oriented programming,” SIGPLAN Notices, 10, 21, 153-162.

Graham A. J., A. C. T. Drakeford, and C. D. Turner (1993). “The Verification, Validation, and Testing of Object-Oriented Systems,” BT Technology Journal 11, 3. 79-88.

Guaspari D., Marceau C., and Polak W. (1990). “Formal Verification of Ada Programs,” IEEE Transactions of Software Engineering 16, 9, 1058-1075.

Harrold J. M., J. D. McGregor, K. J. Fitzpatrick (1992). “Incremental Testing of Object-Oriented Class Structures,” Proceedings, 14th International Conference on Software Engineering, May, IEEE Computer Society, Los Alamitos,CA. 68-80.

Harrold J. M. and G. Rothermel (1994). “Performing Data Flow Testing on Classes,” Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, December, ACM Press, New York. 154-163.

Hartman J., C. Imoberdorf, M. Meisinger (2001). “UML-Based Integration Testing,” Software Engineering Notes, vol. 25, no.5, pp. 60-70.

Hayes H. J.(1994), “Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach,” Proceedings, Object-Oriented Methodologies and Systems, E. Bertino, S. Urban (Eds.), LNCS 858. Springer Verlag, Berlin. 205-220.

Hegazy W. A. (1989). “The Requirements of Testing a Class of Reusable Software Modules,” Ph.D. dissertation. Department of Computer and Information Science, Ohio State University, Columbus.

Hoare A. C. (1969). “An Axiomatic Basis for Programming,” Communications of the ACM, 12, 10 (October).

Hopkins J. E. and M. Sitaraman (1993). “Software Quality is Inversely Proportional to Potential Verification Effort,” In Proceedings of the 6th Annual Workshop on Software Reuse.

Hopcroft E. J., J. D. Ullman (1978). *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.

Hughes M. and D. Stotts (1996). “Daistish: Systematic Algebraic Testing for OO Programs,” Software Engineering Notes, 21, 3 (May), 53-61.

I. Software Productivity Solutions (1995). “Operational Concept Document for the Automated Certification Environment (ACE)”, volume 1, tech. rep., Rome Laboratory, September 1995.

Jacobson Ivar (1992). Object-Oriented Software Engineering: A Use-Case Driven Approach, ACM Press/Addison Wesley, New York, NY.

Jennings R. N., (2000). “On Agent-Based Software Engineering,” Artificial Inteligence 117 (2000) pp. 277-296.

Jorgensen C. P. and C. Erickson (1994). “Object-Oriented Integration Testing,” Communications of the ACM 37, 9 (September), 30-38.

Kandelin A. N. and D. E. O’Leary (1995). “Verification of Object-Oriented Systems: Domain Dependent and Domain-Independent Approaches,” Journal of Systems and Software 29, 261-269.

Kendall A. E. (2000). “Role Modeling for Agent System Analysis, Design, and Implementation,” IEEE Concurrency, April-June, pp. 34-41.

Kirani and Tsai. (1994). “Method Sequence Specification and Verification,” JOOP, (September).

Korson T. and J. D. McGregor (1990). “Understanding Object-Oriented: A Unifying Paradigm,” Communications of the ACM 33,9 (September), 40-60.

Kramer J. and A. L. Wolf (1996). “Succeedings of the 8th International Workshop on Software Specification and Design,” ACM Software Engineering Notes 21, 5 (September), 21-35.

Krone J. (1988). “The Role of Verification in Software Reusability,” Ph.D. dissertation. Department of Computer and Information Science, Ohio State University, Columbus.

Kung David J., J. Gao, P. Hsia, Y. Toyoshima, and C. Chen (1993). “Design Recovery for Software Testing of Object-Oriented Programs,” Proceedings of the Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos, CA. 202-211.

Kung David J., N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen (1994). “On Object State Testing,” Proceedings, The 18th International Computer Software & Applications Conference, IEEE Computer Society Press, Los Alamitos, CA. 222-227.

Kung David J., J. Gao, P. Hsia, J. Lin, and Y. Toyoshima (1995a). “Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs,” Journal of Object-Oriented Programming 8, 2 (May), 51-65.

Kung David J., P. Hsia, Y. Toyoshima, C. Chen, Y. Kim and Y. Song (1995b). “Developing an Object-Oriented Software Testing and Maintenance Environment,” Communications of the ACM 38, 10 (October), 75-86.

L. D. Systems-East. (1996). “Product Line Asset Support Concept of Operations,” Tech. Rep. STARS-VC-K017R1/001/00, Electronic Systems Command, 1996.

Lea D. (1999). “Complex Components Create New Challenges,” In Proceedings of the Nineth Annual Workshop on Software Reuse.

Lea D. (1995). “PSL: Protocols and Pragmatics for Open Systems,” http://gee.cs.oswego.edu/dl/ psl/psl.html.

Leavens T. G. (1991). “Modular Specification and Verification of Object-Oriented Programs,” IEEE Software, 8, 4 (July), 72-80.

Lee J. White (1987). “Software Testing and Verification”, In Advances in Computers, 26, 335-391.

Liskov B. and J. M. Wing (1993). “Specification and Their use in Defining Subtypes,” In Proceedings of OOPSLA’93, 16-28.

Luckham D. C., J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. (1995). "Specification and Analysis of System Architecture using Rapide," IEEE Transactions on Software Engineering, 21, 4, (Apr.) 336-355.

Luckham D., von Henke F., Krieg-Bruckner B., and Owe O. (1987). *ANNA: A Language for Annotating Ada Programs*. Springer-Verlag. Berlin.

Martin J. and J. J. Odell (1992). *Object-Oriented Analysis & Design*, Prentice-Hall, Englewood Cliffs, NJ.

McGregor D. J. and T. Korson (1994). “Integrating Object-Oriented Testing and Development Processes,” Communications of the ACM, 37, 9 (September), 59-77.

McGregor D. J. and D. M. Dyer. (1993). “A note on inheritance and state machines,” Software Engineering Notes 18, 4, 61-69.

Meyer B. (1988). *Object-Oriented Software Construction*, Prentice-Hall, Englewood-Cliffs, NJ.

Myers J. G. (1979). *The Art of Software Testing*. Wiley-Interscience, New-York, NY.

Nance, R.E., Keller, B.J. and D. Boldery (1989), "Document Production Under Next Generation Technologies," Technical Report SRC-89-001, Systems Research Center, Virginia Tech, Blacksburg VA, February

Nierstrasz O. (1995). “Regular Types for Active Objects,” In Object-Oriented Software Composition (Nierstrasz and Tsichritzis Eds.). Prentice-Hall, UK.

Niersatrasz O., J. G. Schneider, M. Lumpe (1997). “Formalizing Composable Software Systems - A Research Agenda,” In Formal Methods for OPen Object-based Distributed Systems (Najm and Stefani Eds.), Chapter 20, 271, 282.

Ostrand T. J. and M. J. Balcer (1988). “The Category-Partition Method for Specifying and Generating Functional Tests,” Communications of the ACM 31,6 (June).

Pande H., W. Landi, B. G. Ryder (1994). “Interprocedural def-use associations in C programs,” IEEE Transactions on Software Engineering, 20, 5 (May), 385-403.

Papathomas M. (1995). “Concurrency in Object-Oriented Programming Languages,” In Object-Oriented Software Composition (Nierstrasz and Tsichritzis Eds.). Prentice-Hall, UK.

Parrish A., R. B. Borie, D. W. Cordes (1993). "Automated Flow Graph-Based Testing of Object-Oriented Software Modules," Journal of Systems and Software 23, 2 (November), 95-109.

Parsons J. and Y. Wand (1997). "Choosing Classes in Conceptual Modeling," Communications of the ACM 40, 6 (June).

Perry E. D. and G. E. Kaiser (1990). "Adequate Testing and Object-Oriented Programming," Journal of Object-Oriented Programming, 2, 5 (January/February), 13-19.

Petitpierre C. (1998). "Synchronous C++: A Language for Interactive Applications," IEEE Computer, 31,9 (September), 65-72.

Pintado X. (1995). "Gluons and the Cooperation between Software Components," In Object-Oriented Software Composition (Nierstrasz and Tsichritzis Eds.). Prentice-Hall, UK.

Poston M. R. (1994). "Automated Testing from Object Models," Communications of the ACM 37, 9 (September), 48-58.

Purchase A. J. and R. L. Winder (1991). "Debugging Tools for Object-Oriented Programming," Journal of Object-Oriented Programming 4, 3 (June), 10-27.

Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen (1991). *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ.

Schafer W., R. Prieto-Diaz, M. Matsumoto (1994). *Software Reusability*. Ellis Horwood.

Schmidt D. (1995). "Active Object -- An Object Behavioral Pattern for Concurrent Programming," Proceedings of the Second Pattern Languages of Programs conference in Monticello, Illinois, September 6-8.

Schmidt D. (1998). "An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse", USENIX login magazine, Tools special issue, November, 1998.

Shaw M., ed. (1981). *ALPHARD. Form and content*. Springer-Verlag, New-York.

Shlaer S. and S. Mellor (1988). *Object Lifecycles: Modeling the World in States*, Prentice-Hall, Englewood Cliffs, NJ.

Sitaraman M. (Eds.) (1994). "Special feature: Component-Based Software Using RESOLVE," Software Engineering Notes, 19, 4, 21-51.

Sitaraman M. and S. H. Zweben (1997). "Guest Editorial, Introduction to Special Section," IEEE Transactions on Software Engineering, 23, 2, 65-66.

Smith D. M. and D. J. Robson (1990). "Object-Oriented Programming: The Problems of Validation," Proceedings of the 6th International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA. 272-282.

Smith D. M. and D. J. Robson (1992). "A Framework for Testing Object-Oriented Programs," Journal of Object-Oriented Programming 5, 3 (June), 45-53.

Subramanian S., W. T. Tsai, S. H. Kirani (1994). "Hierarchical Data Flow Analysis for O-O programs," Journal of Object-Oriented Programming, May, 36-46.

Turner D. C. and D. J. Robson (1993a). The Testing of Object-Oriented Programs. Technical Report TR-13/92. Durham, England: Computer Science Division, School of Engineering and Computer Science (SECS), University of Durham (UK).

Turner D. C. and D. J. Robson (1993b). "The State-Based Testing of Object-Oriented Programs," Conference on Software Maintenance, September, IEEE Computer Society Press, Los Alamitos, CA. 302-310.

Voas M. Jeffrey (1996). "Object-Oriented Software Testability," Achieving Quality in Software. Proceedings of the 3rd International Conference on Achieving Quality in Software, Chapman & Hall, London. 279-290.

Wallace D., R. Fujii (1989). "Software Verification and Validation: An Overview," IEEE Software, May, 10-17.

Weide W. B. and J. E. Hollingsworth (1992). "On Local Certifiability of Software Components," In Proceedings of the 5th Annual Workshop on Software Reuse.

Wing M. J., M. V. Farahani (1995). "Model Checking Software Systems: A case Study," SIGSOFT Software Engineering Notes, 20, 4, 128-39.

Wirfs-Brock R., B. Wilkerson, and L. Wiener (1990). Designing Object-Oriented Software, Prentice-Hall, Englewood Cliffs, NJ.

Wohlin C. and P. Runeson (1994). "Certification of Software Components," IEEE Transactions on Software Engineering 20, 6 (June), 494-499.

Yellin M. D. and R. E. Strom. (1997). "Protocol Specifications and Adaptors," ACM Transactions on Programming Languages and Systems, vol 19, no 292-333.

Yilmaz L., and L. Tankersley (1997). " DARWIN: A Model-Based Interaction Analysis and Recertification Environment for Dependable Upgradability and Interoperability Engineering,," Trident Systems Inc. Technical Report-S&S Division, (Submitted to US naval Surface Warfare Center).

APPENDIX A

UML COLLABORATION GRAPHS

This section provides a description of the semantics of the elements in the Collaboration graphs as depicted in the Object Modeling Group (OMG) standard. Collaboration graphs are divided into two parts: Collaboration and Interaction. The description of behavior involves two aspects: 1) the structural description of the participants and 2) the description of their communication patterns. The structure of Instances playing roles in a behavior and their relationships is described by a *collaboration.* The communication pattern performed by Instances playing the roles to accomplish a specific purpose is specified by an *interaction.*

Collaboration

Behavior is implemented by ensembles of instances that exchange stimuli to accomplish a task. To understand the mechanisms used in a design, it is important to see only those instances and their interactions that are involved in accomplishing the task or a related set of tasks, projected from the larger system of which they are parts of, and might be used for other purposes as well. Such a static construct is called a *collaboration.* The participants define roles that instances and links play when interacting with each other. The roles to be played by the instances are modeled as classifier roles, and by the links as association roles. Classifier roles and association roles, as shown in figure A.1., define a usage of instances and links, while the classifiers and associations specify all required properties of these instances and links.

The collaboration defines a context in which the behavior of the realized element can be specified. A collaboration specifies what properties instances must have to be able to take part in the collaboration, i.e. a role in the collaboration specifies the required set of features a conforming instance must have. Furthermore, the collaboration also states

what associations must exist between the participants, as well as what classifiers a participant, like a subsystem, must contain.

A classifier role is a description of the features required in a particular collaboration, i.e. a classifier role can be seen as a projection of a classifier which is called the base of the classifier role. These classifier roles specify different roles played by (possibly different) instances of the same classifier. A collaboration which describes e.g. a use case, references classifiers and associations in general, while a collaboration describing an operation includes only the parameters and the local variables of the operation, as well as ordinary associations attached to the classifier owning the operation. The interactions defined within the collaboration specify the communication pattern between the instances when they perform the behavior specified in the operation or the use case. Each association role represents the usage of an association in the collaboration, and it is defined between the classifier roles that represent the associated classifiers. The represented association is called the base association of the association role. As the association roles specify a particular usage of an association in a specific collaboration, all constraints expressed by the association ends are necessarily required to be fulfilled in the specified usage.

A collaboration instance set references a collection of instances that play the roles defined in the collaboration instance set's collaboration. An instance participating in a collaboration instance set plays a specific role; i.e. conforms to a classifier role, in the collaboration. The number of instances that should play one specific role in a collaboration is specified by the classifier role's multiplicity. Different instances may play the same role but in different collaboration instance sets. Since all these instances play the same role, they must all conform to the classifier role specifying the role. Thus, they are normally instances of one of the base classifier of the classifier role, or one of their descendants. The only requirement on conforming instances is that they must offer operations according to the classifier role, as well as support attribute links corresponding to the attributes specified by the classifier role, and links corresponding to the association roles connected to the classifier role. They may, therefore, be instances of any classifier meeting this requirement.

Collaborations (but not collaboration instance sets) may have generalization relationships to other collaborations. This means that one collaboration can specify a specialization of another collaboration's task. This implies that all the roles of the parent collaboration are also available in the child collaboration; the child collaboration may, of course, also contain new roles. The former roles may possibly be specialized with new features; i.e. the role defined in the parent is replaced in the child by a role with the same name as the parent role.

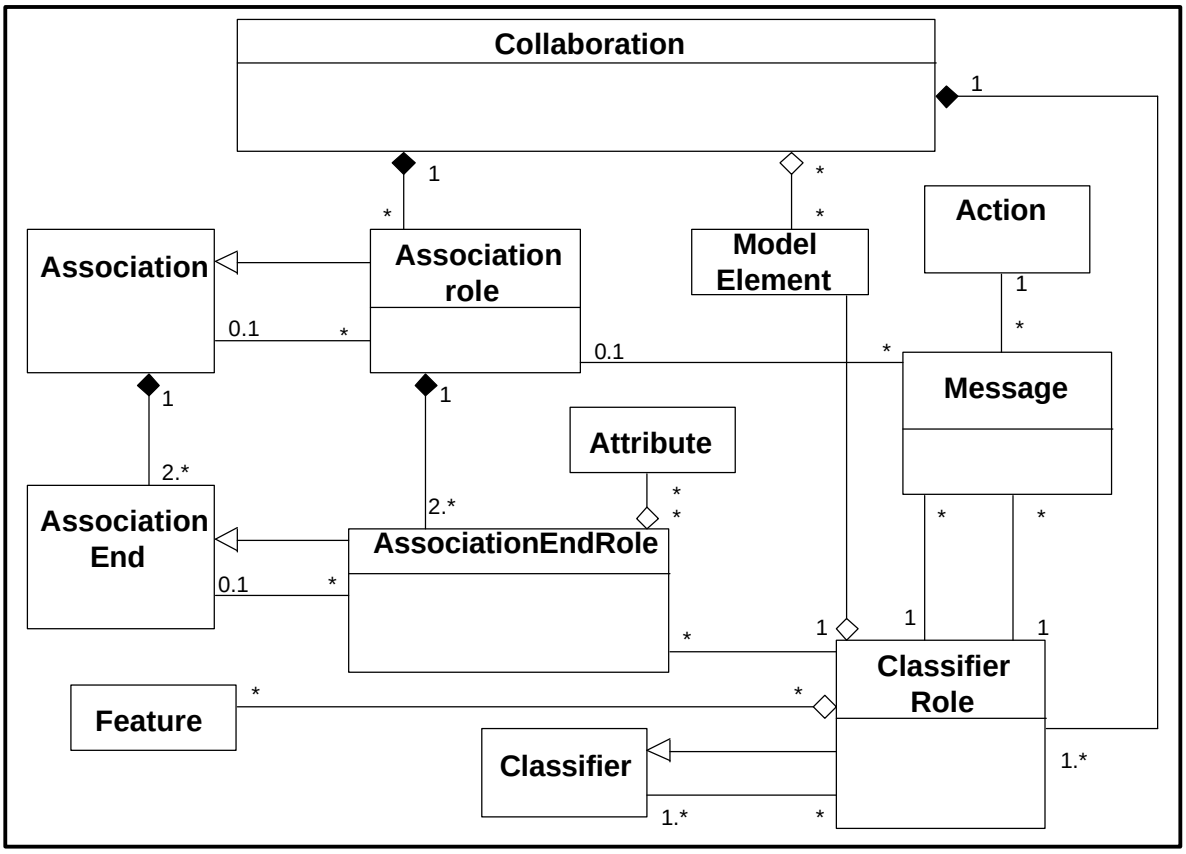


Figure A.1: Collaborations - Roles

The role in the child must reference the same collection of features and the same collection of contained elements as the role in the parent, and may also reference some additional features and additional contained elements. In this way it is possible to specialize a collaboration both by adding new roles and by replacing existing roles with specializations of them. The specialized role, i.e. a role with a generalization relationship

to the replaced role, may both reference new features and replace (override) features of its parent.

Two or more collaborations may be composed to form a new collaboration. For example, when refining a super ordinate use case into a set of subordinate use cases, the collaborations specifying each of the subordinate use cases may be composed into one collaboration, which will be a (simple) refinement of the super ordinate collaboration. The composition is done by observing that at least one instance must participate in both sets of collaborating instances. This instance has to conform to one classifier role in each collaboration. In the composite collaboration these two classifier roles are merged into a new one, which will contain all features included in either of the two original classifier roles. The new classifier role will, of course, be able to fulfill the requirements of both of the previous collaborations, so the instance participating in both of the two sets of collaborating instances will conform to the new classifier role. A parameterized collaboration represents a design construct that can be used repeatedly in different designs. The participants in the collaboration, including the classifiers and relationships, can be parameters of the generic collaboration. The parameters are bound to particular model elements in each instantiation of generic collaboration. Such a parameterized collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects).

Interaction

An interaction is defined in the context of a collaboration. It specifies the communication patterns between its roles. More precisely, it contains a set of partially ordered messages, as shown in figure A.2., each specifying one communication, e.g. what signal to be sent or what operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively. The purpose of an interaction is to specify the communication between an ensemble of interacting instances performing a specific task. An interaction is defined within a collaboration, i.e. the collaboration defines the context in which the interaction takes place. The instances performing the communication specified by the interaction are included in a collaboration instance set, i.e. they conform to the classifier

roles of the collaboration instance set's collaboration. An interaction specifies the sending of a set of stimuli. These are partially ordered based on which execution thread they belong to. Within each thread the stimuli are sent in a sequential order while stimuli of different threads may be sent in parallel or in an arbitrary order. An interaction instance set references the collection of stimuli that constitute the actual communication between the collection of instances. These instances are the collection of instances that participate in the collaboration instance set owning the interaction instance set. Hence, the interaction instance set includes those stimuli that the instances communicate when performing the task of the collaboration instance set. The stimuli of an interaction instance set match the messages of the interaction instance set's interaction.

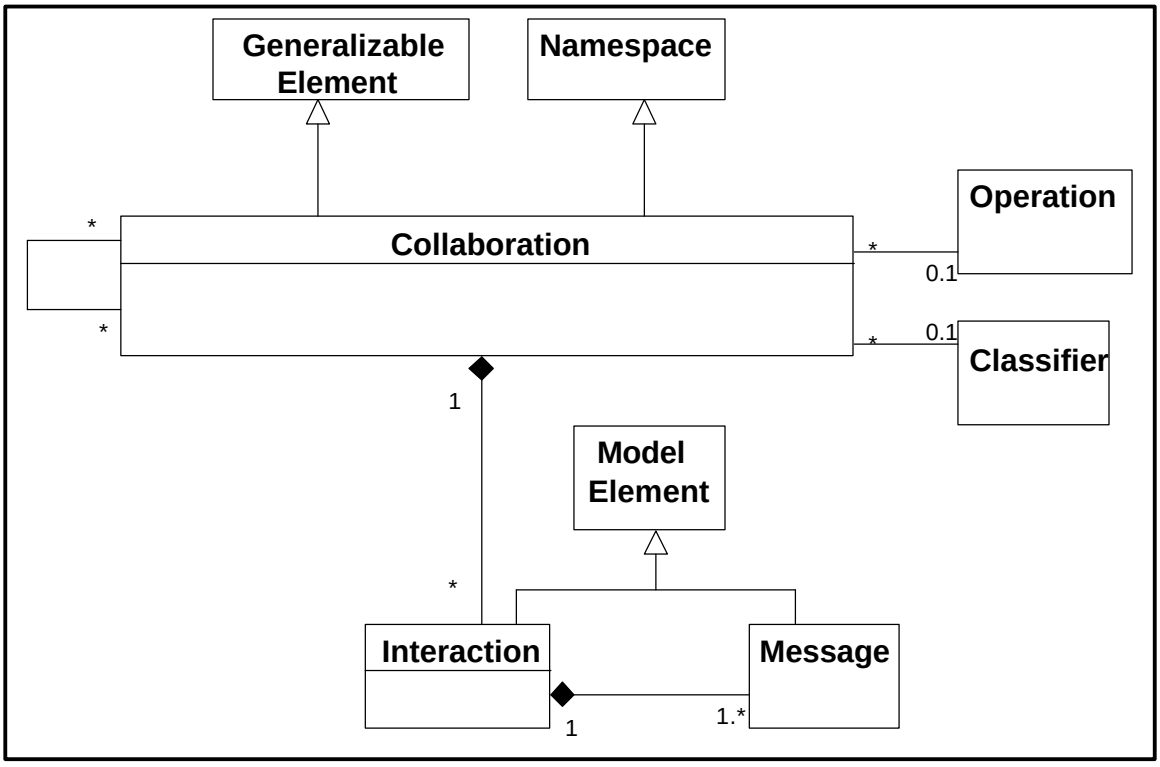


Figure A.2: Collaborations - Interactions

A message is a specification of a communication. It specifies the roles of the sender and the receiver instances, as well as which association role specifies the communication link. The message is connected to an action, which specifies the statement that, when executed, causes the communication specified by the message to

take place. If the action is a call action or a send action, the signal to be sent or the operation to be invoked in the communication is stated by the action. The action also contains the argument expressions that, when executed, will determine the actual arguments being transmitted in the communication. Moreover, any conditions or iterations of the communication are also specified by the action. Apart from send action and call action, the action connected to a message can also be of other kinds, like create action and destroy action.

In these cases, the communication will not raise a signal or invoke an operation, but cause a new instance to be created or an already existing instance to be destroyed. In the case of a create action, the receiver specified by the message is the role to be played by the instance which is created when the action is performed. The stimuli being sent when an action is executed conforms to a message, implying that the sender and receiver instances of the stimuli are in conformance with the sender and the receiver roles specified by the message. Furthermore, the action dispatching the stimulus is the same as the action attached to the message. If the action connected to the message is a create action or destroy action, the receiver role of the message specifies the role to be played by the instance, or was played by the instance, respectively. The interaction specifies the activator and predecessors of each message.

The activator is the message that invoked the procedure that in turn invokes the current message. Every message except the initial messages of an interaction thus has an activator. The predecessors are the set of messages that must be completed before the current message may be executed. The first message in a procedure of course has no predecessors. If a message has more than one predecessor, it represents the joining of two threads of control. If a message has more than one successor (the inverse of predecessor), it indicates a fork of control into multiple threads. Thus, the predecessor's relationship imposes a partial ordering on the messages within a procedure, whereas the activator relationship imposes a tree on the activation of operations. Messages maybe executed concurrently subject to the sequential constraints imposed by the predecessors and activator relationship.

VITA

LEVENT YILMAZ

1993 .....**B.S. Computer Engineering,**
Bilkent University.

1996 .....**M.S. Computer Science,**
Virginia Polytechnic Institute
and State University.

1994 - 1999 .....**Graduate Teaching Assistant,**
Virginia Polytechnic Institute
and State University.

1999 - present .....**Senior Research Engineer,**
Simulation and Software Division,
Trident Systems Incorporated.

Research Grants

- Principal investigator (PI) for "Force Level Automated Certification of Downward Compatible Baseline Software," Naval Surface Warfare Center, Small Business Innovation Research (SBIR) Phase I, II research and development grants.
- PI for "Upward Compatible Baseline Support Framework for Effective Force Level System Regression Testing and Certification," Naval Surface Warfare Center, SBIR Phase I, II research and development grants.
- PI for "Processor Technology Insertion Recertification," Naval Air Warfare Center, SBIR Phase I research grant.
- PI for "Synthetic Natural Environment Interoperability," Army Simulation Training and Instrumentation Command (STRICOM), SBIR Phase I research grant.

Publications and Technical Reports

- Levent Yilmaz (2001). "Automated Object-Flow Testing of Dynamic Process Interaction Models," In Proceedings of the Winter Simulation Conference.
- Levent Yilmaz and Leland Tankersley (2001). "Processor Technology Insertion Recertification," Trident Systems Inc. Technical Report-S&S Division, April 2001 (Submitted to US Naval Air Warfare).

- Levent Yilmaz (2000). "Issues in the Certification of Reusable Object-Oriented Model Components: The Coordination Perspective," In Proceedings of the 2000 International Conference on Web-Based Simulation and Modeling.
- Levent Yilmaz, Leland Tankersley, and Matthew Andrews (2000). “FIREWALL: A Customizable, Front-End Parser Retargatable Baseline Recertification Support Environment For Downward Baseline Compatibility”, Trident Systems Inc. Technical Report-S&S Division November 2000. (Submitted to US Naval Surface Warfare Center).
- Levent Yilmaz and Leland Tankersley (2000). “DARWIN: A Model-Based Interaction Analysis and Recertification Environment for Dependable Upgradability and Interoperability Engineering,” Trident Systems Inc. Technical Report-S&S Division, (Submitted to US Naval Surface Warfare Center).
- Levent Yilmaz (1999). "Certification for Reuse: A Reuse-Oriented Temporal Perspective for Composite Object-Oriented Component Certification," Doctoral student poster session at the International Symposium on Software Reliability Engineering.
- Levent Yilmaz and Thomas Green (1999). “ASSET: HLA Advanced Synthetic System Environment Interoperability Analysis Tool,” Technical Report-S&S Division, April 2000.
- Levent Yilmaz, Leland Tankersley, and Brian Breslin (1999) "Alternate Time Management Mechanisms for Use in Defense Modeling and Simulation Office High Level Architecture Federate Simulation," Trident Systems Inc. Technical Report-S&S Division, June 99 (Submitted to US Navy).
- Levent Yilmaz (1998)."Verification and Validation Guidelines for Object-Oriented Simulation Models", In Proceedings of the 1998 Summer Computer Simulation Conference, July 19, 22 1998. The Society for Computer Simulation International, Reno Nevada.
- Levent Yilmaz (1998). "A Taxonomical Review of Object-Oriented Simulation Model Verification and Validation Techniques," Technical Report TR-98-6. Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Va. (March).
- Levent Yilmaz and Osman Balci (1997). "Object-Oriented Simulation Model Verification and Validation", In Proceedings of the 1997 Summer Computer Simulation Conference, July 13, 17 1997. The Society for Computer Simulation International, San Diego, California

.