A Foundation for Fault Tolerant Components

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the
Graduate School of The Ohio State University

By

William Leal, B.A., M.S., M.S.

* * * * *

The Ohio State University

2001

Dissertation Committee:

Anish Arora, Adviser
William Ogden
Neelam Soundararajan
Bruce Weide

Approved by

Adviser
Department of Computer
Information Science
Components for data abstraction have been useful building blocks in the construction of large systems. Rather than design and reason about systems monolithically, components can be composed into a hierarchy so that at each level the properties of the lower components are used to develop new components and to reason about their behavior. This simplifies construction and reasoning as a whole. To date, however, this has generally been conducted with the implicit assumption that the system being constructed is free from faults. Modular approaches to fault tolerance, on the other hand, have been generally limited to narrow disciplines. Either the lower levels hide the effect of faults, or layers are composed such that lower-levels are oblivious to higher ones, thus permitting tolerance at a lower level without interference from a higher one. Both of these approaches are too restricted. The requirement that fault affect be hidden is too demanding in some situations, particularly distributed ones. Likewise the requirement that a subcomponent be oblivious to its environment does not suffice for components that may achieve tolerance only if higher-level components invoke it correctly.
This study provides a foundation for the study of data abstraction components that are combined with faults in a compositional framework. We can reason in isolation about components with respect to their specification, both in absence and presence of faults, and we can reason about composition in terms of the specifications. Hence we achieve modular reasoning in a compositional setting.

This study adopts a unified view: specifications, rather than being different kinds of objects, are also components. We define the behaviors of a component and in terms of these behaviors define what it means for one component to be “good enough” for another one; that is, for one component to refine another, usually by exhibiting less nondeterminism. Key features of the component definition are that they can model distributed components, and that they can be affected by a fault environment. Additionally, component methods can be spontaneous, reflecting background processes.

Refinement is hard to work with directly since it involves reasoning over infinite computations; simulations provide a more local way to reason. We give two kinds of simulations, state-level (reasoning over the states of clients of the components) and value-level (reasoning over the values that make up the states). These reasoning approaches are shown to be sound, and state-level simulations to be relatively complete. Value-level simulations are less complex, generally preferred, and have been thought to be complete; but we show that they are not unless the components involved are severely restricted. We investigate the behavior of components in the presence of faults, regarding a component as fault tolerant if, in the absence of faults, it refines some specification, and in the presence of faults, it refines some tolerance specification. This general approach permits the definition of the three standard tolerance specifications of masking (never showing the effect of faults), failsafe (never doing anything
wrong) and nonmasking (temporarily showing the effect of faults; stabilization is an extremal case). We show that refining a specification does not mean refinement in the presence of faults of the tolerant specification. The converse is also false even in the case of masking. Hence when a component is designed for tolerance, it is necessary to verify separately that it refines its specification and that, in the presence of faults, it refines the tolerance specification.

We investigate stepwise composition by considering clients that use tolerant components. A client of a masking or failsafe component will also be masking or failsafe, respectively, but this is false for nonmasking. Hence in general we must check the behavior of the client not only with respect to the original specification but also with respect to the tolerance specification. This gives important implications for synthesis approaches. For example, tolerance compilers have been generally regarded as producing refinements of the source; but in the case of nonmasking, the compiler must also produce a concrete component that is a refinement of the nonmasking specification in the presence of faults.

The setup is illustrated by a case study of mutual exclusion using a fault-tolerant vector clock subcomponent. It shows, among other things, that access patterns by the client are critical not only for tolerance of the subcomponent but also for normal, fault-free behavior.

The major contributions of this study are as follows.

• We have a rich component model that accommodates distributed fault tolerance.

• Refinement can be used to define component fault tolerance.

• We have composition and locality of reasoning through refinement and substitution.
• We demonstrate that ideal and tolerance specifications must be separately satisfied.
So many people have contributed into my life and the accomplishments represented in this thesis that it would be difficult to acknowledge them all. Here are some of the ones that have been most prominent.

Professor Anish Arora, whose kindness, wit, friendship, and unbounded patience, combined with rigorous scholarship, has made him both a mentor and a friend. I am privileged to know him.

Professor William Ogden, with whom I have passed many pleasant hours wrestling with the issues of components.

Professor Neelam Soundarajan, for inspiring me to pursue theory. His enthusiasm, kindness and expertise has made semantics a joy.

Professor Bruce Weide, whose practical approach and dedication to the science helped keep me honest.

Otis and Mary Leal, my parents, who not only brought me into the world, but also brought me up in it. They showed me what a quality life could be.

Professor Dow Robinson, spiritual guide of many years standing, whose encouragement to understand God and his world was instrumental in my life.

Professor M. J. Niccolai, who mentored me into computer science and gave me such a good start.
Professor Joseph Grimes, whose love of computers and all they could do was infectious.

Jonathan Leal and Jennifer Leal, my children, for their cheerful encouragement over the years of study.

The staff of the CIS department at Ohio State, who helped me selflessly in so many ways and at so many times

Of all the people on the list, top billing goes to Dorothy Leal, wife, friend, and true companion for the journey. I couldn’t have made it without her.

I am also grateful for the various organizations who have so generously provided me with support during my studies. I have received help from the General Electric Foundation, from the Dean of the Graduate School at Ohio State, and from various grants. They made it possible for me to devote my time to studies.

Finally, I would be remiss if I did not acknowledge my God, whose strength has sustained me, who gave me insight and wisdom when I needed it, and who did not abandon me.

“Streams of mercy, never ceasing,

Call for songs of loudest praise.”
VITA

October 23, 1947 ........................... Born – Harlingen, Texas

1969 ........................................... B.A. in Mathematics
        University of California
        Berkeley, California

1969-1981 .................................. Linguistic Technician and Systems An-
        lyst/Programmer
        Summer Institute of Linguistics
        Nepal and Mexico

1979-1994 .................................. Sr. Systems Analyst / Task Leader /
        Site Manager for various organizations
        Oklahoma, Kentucky, Mississippi

1994 ........................................... M.S. in Computer Science
        University of South Alabama
        Mobile, Alabama
        Thesis title: Modeling Asynchronous
        Architecture Using Data Flow

2001 ........................................... M.S. in Computer Science
        The Ohio State University
        Columbus, Ohio

PUBLICATIONS

Research Publications

W. Leal and A. Arora. “State-Level and Value-Level Simulations in Data Type
FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Professor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Methodology &amp; Systems</td>
<td>Prof. Anish Arora</td>
</tr>
<tr>
<td>Theory &amp; Algorithms</td>
<td>Prof. Kenneth Supowit</td>
</tr>
<tr>
<td>Logic</td>
<td>Prof. Randall Dougherty</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td></td>
<td>viii</td>
</tr>
<tr>
<td>List of Results</td>
<td></td>
<td>xiv</td>
</tr>
<tr>
<td>List of Definitions</td>
<td></td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td></td>
<td>xvi</td>
</tr>
<tr>
<td>1. Introduction</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1 The Locality Problem</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.2 Component Fault Tolerance</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1.3 Components as Data Abstractions</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>1.4 Thesis Goals</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>1.5 Contributions</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1.6 Related Work</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
1.6.1 Modeling Data Abstraction Components 10
1.6.2 Component Refinement 13
1.6.3 Component Verification 15
1.6.4 Models for Fault Tolerance 16
1.6.5 Models for Modular Tolerance 16
1.7 General Overview and Road Map 17

2. Components 20

2.1 Relational Components 21
2.1.1 Client, Spontaneous and Fault Methods 24
2.1.2 Component Behavior 25
2.2 Behavioral Components 31
2.3 Component Refinement 33
2.4 Program Components 36
2.5 Monotonicity under Substitution 39
2.6 Conditional Refinement and Monotonicity 40

3. State-Level and Value-Level Component Verification 43

3.1 State-Level Verification 44
3.1.1 State-Level Forward and Backward Simulations 44
3.1.2 Soundness of State-Level Simulations 46
3.1.3 Completeness of State-Level Simulations 49
3.2 Value-Level Verification 49
3.2.1 Value-Level Forward and Backward Simulations 50
3.2.2 Soundness of Value-Level Simulations 51
3.2.3 Incompleteness of Value-Level Simulations .......................... 52
3.2.4 Completeness of Monadic Value-Level Simulations ............... 58

4. Tolerant Components ............................................ 60
   4.1 Component Tolerance ........................................... 63
   4.2 Refinement vs. Tolerance ....................................... 66
   4.3 Clients of Tolerant Components .................................. 69
   4.4 Synthesizing Tolerance ............................................ 73

5. Case Study: Resettable Vector Clock .................................. 75
   5.1 The Components .................................................. 78
   5.2 Modular Reasoning for Resettable Vector Clock ..................... 84

6. Conclusions and Future Work ........................................ 88
   6.1 Conclusions ....................................................... 88
   6.2 Discussion ......................................................... 89
   6.3 Future Work ....................................................... 91
      6.3.1 Component Frameworks ...................................... 93
      6.3.2 Model-Based Stabilization .................................. 93
      6.3.3 Tolerance-Preserving Atomicity Refinement .................... 94
      6.3.4 Tolerance Compilers ......................................... 95
      6.3.5 Seuss As a Platform ......................................... 96
      6.3.6 Extending to Hybrid Components ............................... 96

Bibliography .......................................................... 97

Appendices:

xii
A. Notation 100

B. Proofs 102

C. Program Syntax and Semantics 126
   C.1 Program Syntax 126
   C.2 Program Semantics 127
List of Results

Theorem 2.1: Refinement is a Preorder .................................. 36
Theorem 2.2: Client Substitution is Monotonic ..................... 39
Theorem 2.3: Conditional Client Substitution is Monotonic ........... 41
Theorem 3.1: Forward State-Level Simulations Are Sound ........... 46
Theorem 3.2: Backward State-Level Simulations Are Sound ........... 48
Theorem 3.3: State-level Simulations Are Relatively Complete ........ 49
Theorem 3.4: Value-Level Forward Simulations Are Sound ........... 51
Theorem 3.5: Value-Level Image-Finite Backward Simulations Are Sound ............................................. 51
Theorem 3.6: Value-level Simulations are Incomplete ................. 54
Theorem 3.7: Completeness of Monadic Value-Level Simulations ........ 58
Theorem 4.1: Ideal Refinement Does Not Imply Tolerance Refinement .... 66
Theorem 4.2: Tolerance Refinement Does Not Imply Ideal Refinement .... 67
Theorem 4.3: Masking and Failsafe Inherited by Clients ................ 70
Theorem 4.4: Conditional Masking and Failsafe Inherited by Clients .... 71
Theorem 4.5: Tolerance from Tolerant Subcomponent Specifications .... 72
Theorem 4.6: Tolerance from Cond. Tolerant Subcomponent Specs ........ 72
List of Definitions

2.1 Relational Component ................................. 22
2.2 Trace .................................................. 29
2.3 Scenario .............................................. 30
2.4 Behavioral Component ............................... 32
2.5 Component Refinement .................................. 35
2.6 Program Component ................................... 37
2.7 Conditional Refinement ................................ 41
3.1 State-level Forward Simulation Relation ......... 44
3.2 State-level Backward Simulation Relation ........ 45
3.3 Value-level Forward Simulation Relation .......... 50
3.4 Value-level Backward Simulation Relation ........ 50
4.1 Fault-Affected Component ............................. 61
4.2 Fault Tolerance ........................................ 63
4.3 Conditional Fault Tolerance .......................... 63
4.4 Standard Tolerance Specifications ................ 65
4.5 Tolerance Refinement .................................. 65
List of Figures

1.1 Locality of Design and Reasoning .............................. 3
2.1 Refinement under substitution does not imply refinement .......... 41
3.1 Diagram for Incompleteness of Value-Level Simulations .............. 53
3.2 Diagram for Incompleteness Proof (Forward-Backward) ............... 56
3.3 Diagram for Incompleteness Proof (Backward-Forward) ............... 57
4.1 Ideal Refinement Does Not Imply Tolerance Refinement ............... 66
4.2 Tolerance Refinement Does Not Imply Ideal Refinement ............... 68
5.1 Resettable Vector Clock (RVC) components ........................ 77
B.1 Diagram for monotonicity lemma .................................. 103
B.2 Hidden C and non-C method calls .................................. 104
B.3 Diagram for proof of monotonicity ................................ 105
B.4 Hidden events in simulation ........................................ 108
B.5 Diagram for Monadic Completeness: Value-Level Forward ........... 123
B.6 Diagram for Monadic Completeness: Value-Level Backward .......... 124
C.1 Unfolding a program statement ..................................... 128
CHAPTER 1

Introduction

1.1 The Locality Problem

Monolithic approaches to the design, implementation, verification and maintenance of systems tend to yield intractable systems since behavior in one part of the system can have unexpected consequences elsewhere. Compositional approaches with locality are popular since complex systems are reduced to the composition of smaller, more manageable components in which each component can be understood locally and the properties of the system can be derived from the composition.

At present, the compositional approach is suitable only for ideal (fault-free) components or for those that mask the effect of faults. If the effect of faults in a component is exposed then the system might no longer behave correctly. The principle of locality is violated since effect of the fault can ripple through the entire system, thus requiring us to deal with the system monolithically. This study gives an approach to fault tolerance that is both compositional and local, and accommodates tolerances other than masking.

To begin with, a component can be any identified part of a system. For software, this includes such things as data types, objects, servers, subroutines, dynamically linked libraries, and even sections of program code. For hardware, this can include
switches, circuits, workstations, IC chips, and the like. It only matters that the component can be understood in isolation, and its relation to the rest of the system is understood. For our purposes there is no essential distinction between hardware and software: discrete systems can be modeled as state transitions [2]; we set aside, for this thesis, the question of continuous or hybrid systems, leaving that as a topic for future work (Chapter 6).

We assume that components can be associated with specifications that abstract the essential characteristics of the component. Given a suitable definition of refinement, a system needing a particular service can be designed not in terms of a particular component but rather in terms of a component specification. The system can then be fielded as a concrete, running system using any component that refines (or implements) the specification. Since specifications are usually less complex than components, this simplifies system development and reasoning.

Though we use the terms “system”, “specification” and “component” as though they were distinct, we treat these as the same kind of object: components. The distinction is just a matter of how they are used. A “specification” is more abstract than a “component”; a “system” is some composition of other components. For the present we leave the notion of refinement undefined; intuitively, a component refines a specification if it is “as good as” the specification, usually with a reduction in nondeterminism. For example, a specification for a bag (multiset) could indicate that any item can be removed. A stack component would refine the bag specification by returning the item most recently added.

The use of specifications also gives us locality of development and reasoning. A component can be developed according to a specification and verified against that
specification without concern for any particular system in which it might be used. Similarly, a system can be developed and verified in terms of a component specification, without concern for any particular component that might be used in a concrete system. For example, a router may need to impose a minimum spanning tree on a set of connected nodes. The router program could be designed in terms of a specification for a minimum spanning tree component. Different components using Kruskal’s and Prim’s algorithms could be developed and verified in isolation, and either could be used when the router is fielded; one might be chosen over the other for non-functional reasons such as performance. This is illustrated in Fig. 1.1.

Figure 1.1: Locality of Design and Reasoning.

SpanTree is refined by Kruskal and by Prim. Either one could be used in a concrete system.
1.2 Component Fault Tolerance

As mentioned, a common approach in component fault tolerance is to mask the faults, using physical or temporal redundancy [33, p. 129]. Examples of physical redundancy are replication with voting; for temporal redundancy, rollback and recovery before commit. An approach that does not mask the faults is to use composition by layers [6]. Here, lower-level tolerance is achieved without interference from higher levels: the higher level can read but not modify the lower-level values, and the lower-level is oblivious to the existence of the higher level. A second non-masking approach is that of stabilizing data structures [14] in which a component repairs an arbitrarily faulty data structure, provided that a client accesses the component sufficiently often; the stabilization represented by this repair depends only on sufficient access and is achieved irrespective of the particular pattern of client access.

While promising, these solutions fall short in several ways. First, masking often does not suffice. This is especially true in distributed systems where masking faults would yield unacceptable performance impacts. Second, a component and a system using the component often have to interact; a subroutine accepts input and produces output. Third, stabilization of a component sometimes requires that it be accessed by the system according to some pattern; the resettable vector clock discussed in Chapter 5 is an example. Finally, we need to consider general tolerances, not just masking and stabilization.

The approach we take to the problem of dealing with component faults is to introduce a second specification, for component tolerance. Then a component in the absence of faults refines an ideal specification, as before, but in the presence of faults
the component refines the tolerance specification. Similarly we given two specifications for the system: an ideal specification as before, and a tolerance specification. This use of tolerance specification with refinement to express component tolerance is new, and lets us build on the approach of ideal specifications to gain locality in reasoning about component fault tolerance. To explain further, let us introduce some notation.

\[
\begin{align*}
C & \quad \text{a component} \\
C\text{Spec} & \quad \text{the component's ideal specification} \\
L & \quad \text{a system} \\
L\text{Spec} & \quad \text{the system's ideal specification} \\
L(C) & \quad \text{the ideal concrete system, using } C \\
L(C\text{Spec}) & \quad \text{the ideal concrete system, using } C\text{Spec} \\
F & \quad \text{a set of faults} \\
C \parallel f & \quad \text{the component in the presence of faults} \\
C\text{TolSpec} & \quad \text{the component's tolerance specification} \\
L\text{TolSpec} & \quad \text{the system's tolerance specification} \\
L(C \parallel f) & \quad \text{the fault-affected concrete system, using } C \text{ in the presence of faults} \\
L(C\text{TolSpec}) & \quad \text{the fault-affected system, using } C\text{TolSpec}
\end{align*}
\]

In this we see the idea of substitution: if \(L(C)\) is the ideal concrete system, then \(L(C \parallel f)\) is the concrete system in the presence of faults. In this fault-affected system we might expect different behavior and therefore require a different specification. For example, stabilization contains the idea that a component may, in the presence of faults, behave arbitrarily but that when the faults stop, eventually the component will behave correctly. This can be reflected in the component’s tolerance specification. From these definitions, whose details are given later, we are able to obtain the following results.
If $C$ refines $CSpec$ and $L(CSpec)$ refines $LSpec$
Then $L(C)$ refines $LSpec$.

If $C \sqsupset F$ refines $CTolSpec$ and $L(CTolSpec)$ refines $LTolSpec$
Then $L(C \sqsupset F)$ refines $LTolSpec$.

The first result expresses the locality of reasoning for systems using components, in the absence of faults: we can check in isolation that the component refines its ideal specification, and we can check in isolation that the system using the ideal specification refines its own ideal specification. From this we can conclude that the concrete system still refines its specification. The second result expresses the same locality of reasoning in the presence of faults: we can check in isolation that the component in the presence of faults refines its tolerance specification, and we can check in isolation that the system using the tolerance specification refines its own tolerance specification. From this we can conclude that the concrete system in the presence of faults still refines the tolerance specification.

For example, the IP protocol is a component used by the TCP protocol. IP is unreliable; that is, it is subject to faults that, among other things, drop message packets in a transient way. Thus in the absence of faults, the IP component refines an ideal IP specification, $IP-Spec$, in which all packets sent are received; in the presence of faults $F$, $IP \not\downarrow F$ refines a tolerance IP specification, $IP-TolSpec$, in which the packets received are a subset of the packets sent, but when the faults stop (or at least, when they stop long enough), the ideal specification holds. Hence the IP protocol is non-masking tolerant to dropped packets. The TCP protocol masks the IP faults. The ideal TCP protocol, $TCP(IP-Spec)$, refines the ideal TCP specification, $TCP-Spec$, in which all packets received are delivered in order. Since the fault-affected TCP
protocol, $TCP(IP-TolSpec)$, is masking, it refines the ideal TCP specification. That is, $TCP-TolSpec = TCP-Spec$.

### 1.3 Components as Data Abstractions

The discussion so far has not been precise as to the nature of components. In this thesis we focus on a broadly-applicable kind of component that is an abstract data structure. Like a traditional abstract data type, such a component may define a domain of values and give a set of methods that manipulate instances of that domain. To represent distributed components conveniently we also permit a component to have shared state that is persistent. For example, a distributed vector clock could be represented as array in shared state, where each element of the array corresponds to a clock value for some process.

Components always hide data, whether instances or shared state. The data can be accessed and manipulated only through methods. To model faults we permit designated methods to be accessed not by a client but by a fault environment. These methods can be used by the fault environment to change client instances or shared data, possibly corrupting them. To model internal actions by a component, such as to repair the effects of faults or to communicate between processes, a component can have methods that are designated as spontaneous.

This rich component, combined with use of specifications, refinement and client substitution, lets us take a simple, abstract view of data structures, both tolerant and ideal.
1.4 Thesis Goals

The primary goal of this thesis is to provide a way to reasoning about fault-tolerant data abstraction in a modular and compositional way. This permits us to design and verify tolerant components in isolation, to design and verify clients of these components in isolation, and to compose them into a system.

In accomplishing this goal, we maintain a unified approach. There is only one kind of entity: the component. There is only one relation between components: refinement. There is only one composition among components: that of client with subcomponents. This permits us to give a simple, semantic characterization to components in the presence of faults and to give simple, semantic definitions of refinement and composition.

There is an additional goal. Directly confirming a refinement can be hard, since component behaviors can be infinite. An important contribution given here is in the area of verification. We give simulations, similar to [23], as a local way to reason about refinements. The contribution comes in the adaptation of simulations to fault tolerant components and in the distinction between two forms of simulation: state-level and value-level, with forward and backward versions for each. State-level simulations are relations over the relevant states of all possible clients, while value-level simulations are relations over the values that make up the states. Both state-level and value-level simulations are sound, and state-level simulations are complete relative to a finiteness condition. The value-level simulations are popular because they are less complex, and have been thought to be complete for components that do not have shared variables, but we show that this is not the case unless they are monadic: that is, there are
no shared variables and their methods are restricted in such a way that at most one argument of each method is over the domain defined by the component.

1.5 Contributions

1. Data abstraction components.
   - Definition of component behavior with respect to faults.
   - Definition of component refinement.

2. Composition.
   - Definition of client component with respect to subcomponents.
   - Result: Subcomponent substitution in clients is monotonic wrt refinement.
   - Use of “rely” condition to weaken substitution monotonicity.

3. Component verification.
   - Definition of forward and backward value-level and state-level simulations.
   - Result: Relative completeness of state-level simulations.
   - Result: Incompleteness of value-level simulations.
   - Result: Completeness of value-level simulations are complete with monadic restrictions.

4. Tolerant components.
   - Definition of fault-affected component.
   - Uniform definition of fault tolerance based on component refinement.
   - Definition of fault refinement.
• Definition of fault-affected client.

• Result: Neither ideal refinement nor tolerance imply the other.

• Result: Tolerant subcomponents do not imply tolerant clients.

• Result: Clients of tolerance specifications can use tolerant refinements.

• Observation: Tolerant compilers do not produce refinements.

5. Contributions to RESOLVE. (Some of these contributions were made in [20].)

• Strengthened Ogden’s proposed definition of component refinement to include conditions for service and safety.

• Result: The state-level correspondence relation is sound and relatively complete.

• Result: The value-level correspondence relation is sound but incomplete.

• Result: Finite invisible nondeterminism (fin) is an issue for RESOLVE.

1.6 Related Work

In this section we consider other related work. The topics considered are how data abstraction components are modeled, the various definitions of component refinement that have been proposed, approaches to component verification, models for fault tolerance, and modular approaches to tolerance.

1.6.1 Modeling Data Abstraction Components

Szyperski [33] defines a data abstraction component as something that can be independently deployed, whose implementation details are hidden from the user and that has no persistent state. The first condition, independent deployment, means
that it is separated from its environment and from other components: we can un-
derstand a component in isolation. The second condition, hiding implementation
details, presupposes a specification for every component that can be used by clients
of the component. The third condition, lack of persistent state, means that one copy
of a component cannot be distinguished from another. Nipkow [28], Leavens and
Pigozzi [21], Liskov and Wing [22] agree with this definition of component. These
definitions are all independent of a particular state space.

We depart from Szyperski’s definition in two ways. First, we make no distinction
between a component and a specification: both are components in our setup. We
define the implementation (or refinement) of one component by a another one, in-
formally referring to the first as a specification or as an abstract component and the
second as an implementation or concrete component, but not otherwise distinguishing
them. Second, we assume that components can define and access shared variables.
A distributed component, say for a token ring, can be defined such that the shared
variable contains the state of the token ring. A channel can be similarly defined.
However, to preserve information hiding, we ensure that if a subcomponent uses a
shared variable, no client can access this variable except through the methods of the
subcomponent.

de Roever and Engelhardt [10] define a data type in terms of a particular state
space. Not only is the data type fixed to a certain state space, it subsumes any
“subtypes” it may require. In this respect, it resembles the approach taken in the
refinement calculus [11, 7]. There, a block of code that defines and manipulates a
object (such as a stack) is viewed as a component. Although these approaches support
a certain kind of data abstraction, they are bound to particular state spaces. We are interested in components that can be used by a wide range of clients.

The components suggested by Garland and Lynch [12] contain state, but they do not provide data abstraction.

RESOLVE [31] components most closely resemble ours. They provide for data abstraction, can access shared variables, and can be used by different clients. However, RESOLVE is modeled for sequential, single thread of control only, and cannot accommodate our fault or spontaneous methods.

Seuss boxes [24] could potentially be adapted for our purpose. Boxes are instances of a category (in the Seuss terminology) and contain local data, methods and actions. Actions are invoked spontaneously and fairly by a scheduler. Categories call the methods of themselves or other categories. The semantics are given operationally.

A data abstraction in Seuss could be viewed as a category; the boxes are instances of the type. For shared data, another category could be given whose box contains the data. Fault methods for a component can be handled as addition of actions to a category. The idea of substitution in a client can be given by enforcing an acyclic dependency structure on a set of categories and then replacing one subcategory by another. The transformation of a behavioral component into a relational one (Chapter 2.2) could probably be adapted to Seuss. Since Seuss gives the developer the ability to reason sequentially about a system that is actually run concurrently, it offers a possible platform for our work.
1.6.2 Component Refinement

Our key relation among components is that of refinement, so here we consider definitions of data type and component refinement. In some treatments the term “refinement” is used, in others, “implementation” or “behavioral subtype.” An author’s definition is consistent with ours if whenever one component refines another according to the author’s definition, the refinement still holds when recast into our setup.

Nipkow [28] presented an early definition of data type implementation. Data types are defined over hidden and visible sorts. Data types are placed in a simple program language that includes input and output primitives, and handles termination. A stream sequence of a data type in a program is a maximal sequence of symbols that can be input and output according to the program. If a sequence is finite, the last symbol is termination, ⊥; otherwise the symbols are the visible type values input or output. A data type’s streams is the union of the sequences under all possible programs.

Data types are compared according to their behaviors. For stream sequences, Nipkow defines $i \preceq s$ as holding when $i = s$ or when $i$ is finite and $i$ up to $\perp$ prefixes $s$. Given streams $I$ for an implementation data type and $S$ for a specification data type, the following three definitions of “$I$ implements $S$” are given:

- **Loose**, if $I \subseteq S$.

- **Partial**, if for all $i \in I$ there is $s \in S$ s.t. $i \preceq s$.

- **Robust**, if for all $i \in I$ there is $s \in S$ s.t. $s \preceq i$.

Nipkow’s setup hides the operations executed; but exposing those is important for us. If Nipkow’s setup were suitably adjusted, the loose and partial implementation
relations would still not suffice since our service condition is not met (that is, the implementation may not terminate early). The third definition resembles ours in that it requires the implementation to behave like the specification until such time as the latter may terminate; after that, the implementation may behave arbitrarily.

Leavens and Pigozzi [21] note the weakness of considering only visible data, and extend visibility to include the procedures used. A behavior of a data type, set in their deterministic algebraic context, consists of an input, a program composed of operations, and an output. One data type subtypes another if they have the same behaviors. This has the advantage of exposing the operations, as we do, and of relating a data type’s behavior to a program. This definition is consistent with ours, but is in a more restricted setup.

Liskov and Wing [22] present a definition of behavioral subtyping based on an abstraction function over the relations of the operations. This definition is consistent with ours. Although this definition effectively makes methods visible, it is in fact a forward simulation in a deterministic setting. The authors handle the behavioral meaning of subtyping only informally.

Although technically quite different, our definition is philosophically similar to the definition of refinement given by Back and Wright using the refinement calculus [7, Chap. 1.4]. There, program $S$ is refined by $S'$ if $\sigma\{|S|\} q \Rightarrow \sigma\{|S'|\} q$: if starting from precondition $\sigma S$ can establish postcondition $q$, then so can the refinement $S'$. Hence $S'$ provides the same service as $S$, but may be more deterministic; and $S'$ may be able to establish the postcondition from a larger precondition so that it has more behaviors than $S$. Although this definition is consistent with ours, exposing the methods, including nondeterminism and allowing the refined concrete program
to have more behaviors than the abstract, there is an important difference. A data
type is viewed as embedded in a program and hence depends on the program’s states.
Data refinement means transforming an abstract section of a program (say, that
manipulates a bag) into program statements over something more concrete (say, a
stack). Thus a data type cannot be easily used in a different state space, and the
notion of a client makes little sense.

For de Roever and Engelhardt’s components [10], refinement is defined for partial
and total correctness. The partial correctness version is similar to Naumann [25].
The total correctness version is similar to the definition for Back and Wright, and is
consistent with ours.

Unpublished work in RESOLVE [29] by Ogden is closest to our definition of com-
ponent refinement. The realization relation, as originally proposed, is the same as our
refinement definition, but it only included the behavior condition. To model compo-
nent refinement appropriately, our definition extends it with the service and safety
conditions.

1.6.3 Component Verification

With the exception of the refinement calculus, verification of refinement for the
authors cited in the previous section all use some form of forward or backward sim-
ulation (See Chapter 3). Nipkow [28] uses a value-level forward simulation, as do
Leavens and Pigozzi[21]. Liskov and Wing’s abstraction function [22] is a nonden-
terministic forward simulation. RESOLVE [31] uses a backward simulation (called
the “correspondence”) and a restricted forward simulation; these are each defined for

In the refinement calculus [7], data refinement is accomplished by changing an abstract program (or specification) to a concrete program by a series of stepwise transformations that preserve refinement. These transformations can be viewed as verification that the concrete data-refines the abstract.

1.6.4 Models for Fault Tolerance

Historically, fault tolerance was studied in the context of specific technologies, architectures, and applications, with the consequence that a variety of apparently-unrelated subdisciplines emerged. Arora and Gouda [5, 2] proposed that faults be uniformly represented as actions of a fault environment that perturb the state of the system. This widely-cited view permitted them to unify the other fault-tolerance approaches. Kulkarni [16] showed that detectors and correctors are both necessary and sufficient for these standard tolerances.

We adopt this approach. Faults in components are represented as state perturbations by fault methods, and although we permit general tolerances, we work primarily with the standard ones.

1.6.5 Models for Modular Tolerance

Masking is frequently used to give tolerance to components [33, p. 129]. This is usually done by adding redundancy to the component: either spatially, as in multiple modular redundancy where copies of the component are executed in parallel and the results passed through a voting mechanism; or temporally, as in checkpoint and recovery. Yen, Bastani and Taylor [34], for example, define masking data structures
that perform efficiently in the absence of faults, perform correctly but less efficiently in the presence of faults and during recovery, and eventually return to full efficiency.

Herman [14] gives data abstraction components that, in the presence of faults, can stabilize over time if a client continues to access the component’s methods. In stabilizing 2-3 trees, for instance, a subtree is corrected when it is found to be other than a correctly-formed 2-3 tree. Other requirements are added to the component: method calls on faulty trees should not be too much slower than calls on good trees; and after the faults stop, the tree immediately respects its specification with respect to newly-added items, even if parts of the tree remain faulty. This kind of component qualifies as stabilizing in our sense.

In work by Arora and Gouda [6], a layered approach to tolerance is given in which upper layers may access but not change the values of lower layers, and lower layers are oblivious to the existence of upper layers. This approach could be adapted for specialized kinds of components in which the subcomponent uses spontaneous methods to handle faults, and invocations by clients do not affect the state of the subcomponent.

Breitling [9] focuses on modeling faults as changes to the control of a component, whose effect is to perturb state. He suggests wrapping a component with control code that can respond to faults, and explores composition with wrappers and verification of the composite behavior.

1.7 General Overview and Road Map

Here we give a general overview, and describe the contents of each chapter. In Chapter 2 we define relational components as our fundamental components and give
their behaviors. We define a behavioral component in terms of these behaviors and show that such a component can be equivalently given as a relational component. We define what it means for one component (relational or behavioral) to refine another. Finally we deal with program components that are clients of some configuration of components. The behavior of such a client depends on the behavior of the components in the configuration, so we can substitute one subcomponent for another in a configuration and obtain new behaviors for the client. We show that this substitution is monotonic: if the substitution in the configuration is by a refined subcomponent, then the new behaviors of the client will refine the old ones.

In Chapter 3 we deal with verification of refinement. We do this with simulations that are of two varieties: state-level and value-level. The state-level simulations are relations over all relevant client states, while the simpler value-level simulations are relations over the values that make up the states. We show that state-level simulations are complete relative to a finiteness condition but that, contrary to popular belief, value-level simulations are incomplete unless they are monadic.

In Chapter 4, fault-affected components are considered in detail. A component is defined to be fault-tolerant if, in the presence of faults, it refines some tolerance specification (which is itself a component). With this we define the three standard tolerance specifications of masking, failsafe and nonmasking; stabilization is an extremal case of nonmasking in which any state corruption can occur. We observe that if a component refines a specification it will not necessarily refine one of the standard tolerance specifications, and that the converse is also false. Next we take up the behavior of clients that use a tolerant subcomponent. We see that such a client can be masking or failsafe tolerant if the subcomponent is masking or failsafe, respectively,
but that this does not hold for nonmasking. But we see that if the client takes into account the subcomponent’s tolerance specification then it will be tolerant.

In Chapter 5 we give a case study of a stabilizing vector clock that uses bounded space. This is presented in the context of a client that implements the Ricart-Agrawala mutual exclusion algorithm. The case study shows how reasoning can be done in a modular and compositional way and illustrates the importance of “rely” conditions for conditional refinement and substitution.

Chapter 6 wraps up with conclusions and discussion of future work.
CHAPTER 2

Components

As the title indicates, this chapter deals with components: definition, behavior and usage. Subsequent chapters deal with verification and fault tolerance.

Three kinds of components are defined: relational, program and behavioral. Relational components are the elementary ones upon which the others are built, with methods defined as relations; from the methods, a set of operational behaviors can be calculated. The methods for program components are programs that invoke methods of subcomponents; the meaning of such a component varies according to the meaning of the subcomponents invoked. A behavioral component assigns no meaning to methods directly; the meaning of the component is given directly as a set of behaviors over the methods.

Each kind of component has its use. Relational components are the foundational ones, in which each method is given as a relation. For such a component an operational interpretation can be assigned according to its behavior with all possible clients.

To represent clients whose meaning varies according to the components selected we use program components. If $L$ is such a component, then we can contrast the meaning of $L(C)$ with that of $L(CSpec)$. Given that $C$ and $CSpec$ are relational then we can calculate equivalent relational components for $L(C)$ and $L(CSpec)$, respectively.
Behavioral components permit us to specify a component’s desired behaviors directly. Specifications are often given most readily as sets of behaviors. Consequently, the methods for these are not given as relations or programs; rather, the behaviors of the methods are given directly. This is the same as the operational interpretation assigned to a relational component. Additionally, a behavioral component can be given as an equivalent relational one.

2.1 Relational Components

At its simplest, a relational component is just a data type: a domain of data values, with methods over the domain that are invoked by some client. Data types are used to create data abstractions, letting us manipulate complex objects such as minimum spanning trees without being concerned about the details of the lower-level structures that make up the trees. We extend this in two ways. First, we let a component use one or more shared variables, so that a component has persistent state. The use of shared variables lets us model, for example, a linked-list component in which the shared data represents the available pool of memory. Second, we do not require that a component define a data type since sometimes we only want one instance of a component in a system.

In a relational component, the methods are given as relations. Later we define the client component in which the methods are given as programs. When a client component is put together with an appropriate configuration of relational components then we can calculate the relations of the program methods and treat the component as another relational component. This lets us place a single client component in different configurations and calculate different corresponding relational components.
This compositional ability is foundational to our goal of compositional and modular reasoning.

**Domain.** Assume a collection of domains, \( \text{Domains} \), that is closed under finitary cross product, includes the empty set and is pairwise disjoint.

**Definition 2.1 (Relational Component)** A relational component \( C \) is a tuple \((C.\text{cdom}, (C.\text{cshare}, C.\text{cshinit}), (C.\text{cmeth}, C.\text{crel}, C.\text{cvis}, C.\text{cctl}), C.\text{ctype})\) that is defined as follows.

- \( C.\text{cdom} \in \text{Domains} \) is a component domain.
  
  If the domain is not empty then we say that the component defines a domain. If the domain is empty then we say that the component does not define a domain. If there is a domain, we refer to it as \( \text{self} \).

- \( C.\text{cshare} \) is a possibly-empty tuple of shared variables.
  
  - \( C.\text{cshinit} \) is a function \( C.\text{cshinit} : \bigcup C.\text{cshare} \rightarrow \text{Domains} \) that maps shared variables to initial values s.t. \( C.\text{cshinit}(G) \subseteq C.\text{ctype}(G) \) and \( C.\text{cshinit}(G) \) is empty iff \( C.\text{ctype}(G) \) is.

- \( C.\text{cmeth} \) is a set of method names.
  
  - \( C.\text{crel} \) is a function
    
    \[
    C.\text{crel} : C.\text{cmeth} \rightarrow (\times (C.\text{ctype}(C.\text{cmeth}); C.\text{ctype}(C.\text{cshare})))^2
    \]
    
    that maps each method to a relation over the argument domains and those of the shared variables.

  - \( C.\text{cvis} \) is a function that maps each method name to a visibility value: “visible” or “hidden”.

22
○ \textit{C.cctl} is a function that maps each method name to a control type: “client-controlled”, “spontaneous” or “fault”.

○ If the component defines a domain then there is a distinguished visible client-controlled method \textit{New}(\textit{self}) that is used for creating or reinitializing client variables. The domain of the relation consists of all values in \textit{self}; the range consists of the initial values.

• \textit{C.ctype} is a sorted function relating variables and methods to their associated domains.

  ○ \textit{C.ctype} : \bigcup \textit{cshare} \rightarrow \textit{Domains} maps each variable to a domain.

  ○ \textit{C.ctype} : visible \textit{cmeth} \rightarrow \textit{Domains}^* maps each visible method name to a possibly-empty tuple of argument domains.

  ○ \textit{C.ctype} : hidden \textit{cmeth} \rightarrow \{\textit{self}\}^* maps each hidden method name to a possibly-empty tuple of \textit{self}.

As noted, references to \textit{cdom} are given as \textit{self}.

Notationally, where there is no confusion, \textit{M} refers both to the name of the method and to the associated relation. \textit{M}(\textit{d}, \textit{e}) means that \textit{M} is a method and \textit{ctype}(\textit{M}) = (\textit{d}, \textit{e}). Also, where the component is understood, the qualification is dropped:

+++ \textit{C.ctype}(\textit{C.cshare}) is given as \textit{ctype}(\textit{cshare}) We say that component \textit{D} is referenced or used by \textit{C} if \textit{D} is a domain other than \textit{self} and if \textit{D} is the domain for any shared variable, or is in \textit{ctype}(\textit{M}) for any method \textit{M} of \textit{C}. We say that \textit{c} \in \textit{C.New} if for some \textit{c}', \textit{c} \in \textit{C.New}[\textit{c}'].

23
The domains referenced in the methods of $C$, excluding $self$, are called the indicators of $C$ since they help indicate the behavior of the component. As discussed below under component behavior, indicators enter into the visible behavior of the component. Consequently, hidden methods are restricted to $self$ arguments. They cannot include any indicator arguments as this would be inconsistent with being hidden.

Component signature. For a component, the signature of the component is the set of method names ($M$), their associated type ($C.c$type$), control ($C.cctl$) and visibility ($C.cvis$). The signature for behavioral and program components is similarly defined.

2.1.1 Client, Spontaneous and Fault Methods

The definition of relational component allows three kinds of methods: client methods, spontaneous methods, and fault methods. A client method is invoked by a program client (Chapter 2.4); a spontaneous method is, as the name implies, invoked spontaneously by the component, and is used to model internal processes; a fault method is invoked by a fault environment and is used to model faulty behavior.

Client methods are typically partial relations. For a stack component, the $Add$ operation is defined only if the stack has space available to add new items. Since a well-formed client will not attempt to add more items to a stack than it can hold, this is appropriate.

Fault and spontaneous methods are required to be total relations. The assumption is that each of these methods can be invoked (by the fault environment or the component itself, respectively) at any time. Though these methods are total and hence
always enabled (see next section for discussion of enablement), they do not necessarily produce any changes to the component state. Spontaneous and fault methods are distinguished as regards liveness (or fairness). No assumptions of liveness are made with respect to the fault environment, but liveness may be required of spontaneous methods. This is discussed in more detail in Chapter 2.1.2.

2.1.2 Component Behavior

Consider the visible behavior of an integer queue component \( Q_1 \) that defines no domain, has a shared variable \( Q \) over tuples of integers, and has two visible client-controlled methods, \( Put(Int) \) and \( Get(Int) \). Intuitively, the desired behavior would be that integers added to the queue are removed in order. A sample visible behavior would be the following sequence of events (with the format simplified for presentation):

\[
\langle Put(3), Put(6), Get:3, Put(4), Get:6, \ldots \rangle.
\]

The relation for \( Put \) would include the pair \(((6, (3)), (6, (3, 6)))\) to indicate that \( Q \) is extended by the argument. Similarly, \( Get \) includes the pairs \(((n, (3, 6)), (3, (6)))\) for \( n \in \mathbb{N} \). This lets us extend the visible behavior to a complete behavior:

\[
\langle [Q = ()] Put(3) [Q = (3)] Put(6) [Q = (3, 6)] Get:3 [Q = (6)] Put(4)
[Q = (6, 4)] Get:6 [Q = (4)] \ldots \rangle.
\]

The complete behavior includes the information necessary to define the visible behavior in terms of the component’s relations. The transition \([Q = (3)]Put(6)[Q = (3, 6)]\) exists because \( Put \) is defined on \((6, (3))\) and is associated with \((6, (3, 6))\).
Now consider integer queue component $Q2$ that defines a domain and with the operations extended to include reference to the component’s domain: $Put(self, Int)$ and $Get(self, Int)$. A complete behavior would include

$$\langle \left[ \text{New}(X) \ [X = ()] \right], \left[ \text{Put}(X, 3) \ [X = (3)] \right], \left[ \text{New}(Y) \ [X = (3), Y = ()] \right], \left[ \text{Put}(Y, 6) \ [X = (3), Y = (6)] \right], \left[ \text{Get}(Y) ; 6 \ [X = (3), Y = ()] \right] \rangle.$$

The visible behavior of the component is

$$\langle \text{New}(X), \text{Put}(X, 3), \text{New}(Y), \text{Put}(Y, 6), \text{Get}(Y) ; 6 \rangle.$$

This exposes which variables are referenced and the values being added and removed, but hides the actual values of the queues, as is appropriate.

Now let us turn to the precise definition of component behavior. A weaker version of the definition and some of the terminology is due to Ogden [29]. We assume there is some infinite universal set of variable names.

**Event.** Let $M(\overline{dm})$ be a method of $C$ where $\overline{dm} = \text{ctype} (M)$ is the tuple of argument domains of $M$, and let $m = |\overline{dm}|$ be the number of domains in $\overline{dm}$. An event of $C$ is of the form

$$< (a'_0, \ldots, a'_{m-1}) \ M(h_0, \ldots, h_{m-1})(a_0, \ldots, a_{m-1}) >$$

where for $i < m$ the following holds.

- $a'_i$ and $a_i$ are in $\overline{dm}_i$ if $\overline{dm}_i$ is an indicator, else $a'_i = a_i = \circ$, a placeholder.

- $h_i$ is a variable if $\overline{dm}_i$ is not an indicator, else $h_i = \circ$.

$< (a'_0, \ldots, a'_{m-1}) \ M(h_0, \ldots, h_{m-1}) >$ is called the *invocation* of the event and $< (a_0, \ldots, a_{m-1}) >$ is called the *result*. An event gives the pre-values of the indicators,
the method name and the names of variables defined over \textit{self}, and the post-values of the indicators. For convenience we generally use a simpler, equivalent notation. Rather than write \(<(\circ, n)\text{Get}(X, \circ)(\circ, 6)>)\) for \(n \in \mathbb{N}\), for instance, we write instead \(<\text{Get}(X, \ast):6>)\), collapsing the invocation and letting \(*\) stand for any value in the domain. An event is visible if the method called is visible. It is a client, spontaneous or fault method if the method called is a client, spontaneous or fault method, respectively.

\textit{State.} A \textit{state} of \(C\) is of the form

\[ [G_0 = g_0, \ldots, G_{n-1} = g_{n-1}, H_0 = h_0, \ldots, H_{k-1} = h_{k-1}] \]

where for \(0 \leq i < n\), \(G_i\) are shared variables and \(g_i\) are values from the associated domains; for \(i < k\), \(H_i\) is a variable (called an \textit{client instance variable}) and \(h_i\) are values from \textit{self}. The variables in a state are unique. An \textit{initial state} of \(C\) is one that contains only shared variables s.t. for \(0 \leq i < n\), \(g_i \in C.cshinit(G_i)\).

\textit{Transition.} A \textit{transition} of \(C\) is a triple \((s', e, s)\) (optionally written \(s' \xrightarrow{C} s\)) in one of two forms.

- For methods other than \textit{New}, the form is

\[
\begin{align*}
{s'} &= [G_0 = g'_0, \ldots, G_{n-1} = g'_{n-1}, H_0 = h'_0, \ldots, H_{k-1} = h'_{k-1}], \\
{e} &= (a'_0, \ldots, a'_{m-1}) M(h_{p_0}, \ldots, h_{p_{m-1}});(a_0, \ldots, a_{m-1}), \\
{s} &= [G_0 = g_0, \ldots, G_{n-1} = g_{n-1}, H_0 = h_0, \ldots, H_{k-1} = h_{k-1}],
\end{align*}
\]

where \(M(dm)\) is a method of \(C\) and the following hold.

- For \(i < m\),
  - if \(dm_i\) is an indicator then \(a'_i \in dm_i\) else \(a'_i = \circ\), and
  - if \(dm_i\) is not an indicator then \(h_{p_i}\) is one of \(h_0, \ldots, h_{k-1}\), and
- if $dm_i$ is an indicator then $a_i \in t_i$ else $a_i = \circ$.

- Let $e'$ and $e$ be tuples of size $m$, and $f'$ and $f$ be tuples of size $n$ s.t. the following hold.
  - For $i < m$, if $h_{p_i}$ is variable $h_j$ then $e'_i = c'_j$, else $e'_i = a'_i$.
  - For $i < n$, $f'_i = g'_i$.
  - For $i < m$, if $h_{p_i}$ is variable $h_j$ then $e_i = c_j$, else $e_i = a_i$.
  - For $i < n$, $f_i = g_i$.

Then $(e; f) \in M[(e'; f')]$ \(^1\) and $x \in M[y]$ if $(x, y) \in M$.

- For $i < k$, if there is no $j$ s.t. $h_{p_j} = h_i$ then $h_i = h'_i$.

- The second form of a transition is one that creates a new variable with the domain self or initializes an existing self variable.
  - If the argument variable is not in $s'$ then
    \[
    \begin{align*}
    s' &= [G_0 = g'_0, \ldots, G_{n-1} = g'_{n-1}, h_0 = c'_0, \ldots, h_{k-2} = c'_{k-2}], \\
    e &= \text{New}(h_{k-1}), \\
    s &= [G_0 = g_0, \ldots, G_{n-1} = g_{n-1}, h_0 = c_0, \ldots, h_{k-1} = c_{k-1}],
    \end{align*}
    \]
  \[\text{where for } i < n, g_i = g'_i, \text{ for } i < k-1, c_i = c'_i, \text{ and } c_{k-1} \text{ is in the range of the relation of } C.New.\]

  - If the argument variable is in $s'$ then
    \[
    \begin{align*}
    s' &= [G_0 = g'_0, \ldots, G_{n-1} = g'_{n-1}, h_0 = c'_0, \ldots, h_{k-2} = c'_{k-2}], \\
    e &= \text{New}(h_j), \\
    s &= [G_0 = g_0, \ldots, G_{n-1} = g_{n-1}, h_0 = c_0, \ldots, h_{k-1} = c_{k-1}],
    \end{align*}
    \]
  \[\text{where for } i < n, g_i = g'_i, \text{ for } i \neq j, c_i = c'_i, \text{ and } c_j \text{ is in the range of the relation of } C.New.\]

\(^1\) $e'; f'$ is concatenation of tuples; $M[(e'; f')]$ is the set of all $(e; f)$ s.t. $((e'; f'), (e; f)) \in M$. See Appendix A for notation.
If $\alpha$ is a sequence of events of $C$ then we write $s' \xrightarrow{\alpha}^C s$ if there are $s_0, \ldots, s_{n-1}$ s.t. $s' = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \cdots \xrightarrow{e_{n-1}} s_{n-1} = s$ and $\langle e_1, \ldots, e_{n-1} \rangle = \alpha$.

**Computation.** A computation of $C$ is an alternating sequence $\langle s_0, e_1, s_1, \ldots \rangle$ of state and event, beginning with an initial state and ending with state if finite, s.t. for all $i$, $(s_{i-1}, e_i, s_i)$ is a transition of $C$.

**Enablement.** An event $e$ is enabled on a state $s'$ in $C$ if there is a state $s$ s.t. $(s', e, s)$ is a transition of $C$. An event $v$ is enabled on $s'$ if there is a result $r$ s.t. $e = (v, r)$ is enabled on $s'$.

**Reachability.** State $s$ is reachable by event sequence $\alpha$ in $C$ if there is a computation $\langle s_0, e_1, s_1, \ldots, e_{n-1}, s_{n-1} \rangle$ s.t. $s = s_{n-1}$ and $\langle e_1, \ldots, e_{n-1} \rangle = \alpha$. The states reachable by $\lambda$, the empty trace, in $C$, are the initial states.

**Visible Event Sequences.** If $\alpha$ is a sequence of events then $\hat{\alpha}$ is a projection to the visible events of $\alpha$.

**Definition 2.2 (Trace)** Let $\sigma = \langle e_0, e_1, \ldots \rangle$ be a sequence of visible events of $C$. $\sigma$ is a trace of $C$ if the following hold.

- $\sigma = \lambda$ or
- $\sigma = \sigma'; e$ and
  - $\sigma'$ is a trace of $C$ and
  - $e$ is enabled on all states reachable by $\hat{\sigma'}$ in $C$.

Note that enablement implies that there is a state reachable by $\sigma'; e$ in $C$. Also note that if $\sigma' = \langle e, f_1, \ldots, f_{n-1} \rangle$ where $e$ is a visible event and $f_i$ are hidden events then $e = \hat{\sigma'} = \langle \hat{e} \rangle = \langle \hat{e}, f_1 \rangle = \cdots = \langle \hat{e}, f_1, \ldots, f_{n-1} \rangle$. Hence if $e$ is enabled on all states
reachable by $\hat{\sigma}'$ in $C$ then $e$ must be enabled on all states reachable by $\langle e \rangle$, $\langle e, f_1 \rangle$, 
... and $\langle e, f_1, \ldots, f_{n-1} \rangle$.

Traces are defined in such a way that each event must be enabled on all the states resulting from the preceding event (or all initial states, for the first event). This is important for being able to substitute one component for another in a program (see Chapter 2.5).

**Definition 2.3 (Scenario)** A *scenario* of $C$ is a computation of $C$ that, when projected to events, is a trace of $C$.

The scenarios of a component give the complete behavior, including events and states, while traces give just the visible behavior. Since scenarios are not maximal, a scenario can consist of just a single state, giving the null trace, $\lambda$. If components $C$ and $A$ have the same traces then we write $C =_T A$, overloading $=_T$ so that either argument can alternatively be given as scenarios or traces.

**Component Liveness**

In defining the scenarios of a component we have not differentiated between the methods in terms of control. This reflects a neutral position on liveness (or fairness) with respect to a component: that is, no liveness is assumed. If appropriate for a given system, liveness can be imposed on the behaviors by adding a supplemental set of live computations. The view here is that liveness is a feature of spontaneous events only. Fault events are explicitly assumed not to have liveness, and for client-controlled events the notion is irrelevant since a client program explicitly determines the sequence in which they are used.
Before continuing, we briefly recall the topological nature of liveness, drawing on [1]. We can define a topological space as the set of all finite or infinite alternating sequences of states and events of a component $C$ that start with a state and, if finite, end with a state. Safe sets are topologically closed while live sets are dense. A metric for such a topology is one in which the distance between two computations is the inverse of the length of the longest common prefix that the computations share.

Let $\sigma = \langle s_0, e_1, s_1, \ldots \rangle$ is a scenario of $C$. Let $\hat{\sigma}$ be the projection to events that are client-controlled, retaining surrounding state, and using $X$ as a placeholder. For example, consider $\langle s_0, e_1, s_1, f_2, s_2, f_3, s_3, e_4, s_4 \rangle$ where the $e_i$ are client-controlled and $f_i$ are not. The projection yields $\langle s_0, e_1, s_1, X, s_3, e_4, s_4 \rangle$. The same thing can be done with spontaneous and fault events.

A supplemental set of computations is live if the following hold.

- The projection of computations for spontaneous events is live with respect to a topology that consists of alternating sequences of state and spontaneous events (along with $X$).

- The projection of computations for client-controlled or fault events is equal to the topological space that consists of all alternating sequences of state and client-controlled or fault events (along with $X$).

This definition reflects the fact that only spontaneous events can be live.

### 2.2 Behavioral Components

Behavioral components are straightforward. As with relational components, they may define a domain, and may use shared variables. They include methods, but no
relations for the methods. Instead, the behavior of the component is given by a set of traces. This gives a convenient way to define a specification.

Definition 2.4 (Behavioral Component) A behavioral component $C$ is a tuple $(C.cdom, (C.cshare, C.cshinit), (C.cmeth, C.cvis, C.cctl), C.ctype, C.ctrace)$ that is defined as follows.

- $C.cdom \in \text{Domains}$ is a \textit{component domain}. This is defined as for relational components.
- $C.cshare$ and $C.cshinit$ are defined as for relational components.
- $C.cmeth$ is a set of \textit{method names}.
  - $C.cvis$ and $C.cctl$ are defined as for relational components.
  - If the component defines a domain then there is a distinguished visible client-controlled method $\text{New}(self)$ that is used for creating or reinitializing client variables.
- $C.ctype$ is defined as for relational components.
- $C.ctrace$ is a set of \textit{traces}. $\sigma \in \text{ctrace}$ iff $\sigma$ is a finite or infinite sequences of events of $C$.

As we noted earlier, behavioral components can be used where relational components are required. It is consequently necessary to represent a behavioral component as an equivalent relational one.

Say that two traces are \textit{renaming-equivalent} if there is a renaming of variables s.t. the traces are identical. A renaming of variables is the application of a bijection from
the variable names of one trace to those of the other. Hence

\[ \langle \text{new}(1), \text{new}(2), M(1,a): b \rangle \]

and

\[ \langle \text{new}(X), \text{new}(Y), M(Y,a): b \rangle \]

are renaming-equivalent, where 1 and 2 are names of variables in the first trace.

Given behavioral component \( C \), construct relational component \( C' \) as follows. If \( C \) defines a domain then let \( C' \) define a domain over the integers and have a shared variable \( hist \), initially \( \lambda \), to track the history of method calls to the component, along with another shared variable \( vid \) over the integers, initially 0, to identify the variables.

The idea is to assign a unique value to a variable that does not change. Hence the value of a variable identifies the variable itself. A call to a method \( M(self, ind) \) of the form \( M(4,a) \) where \( ind \) is an indicator and \( a \in ind \) would be a call on variable number 4. \( hist \) keeps a history of these calls, so \( hist \) might contain, for example, \( \langle \text{new}(1), \text{new}(2), M(1,a): b \rangle \). This can be compared against a trace of \( C \) and checked for renaming-equivalence.

The relation for \( C.New(self) \) is defined as the set of all \( \langle (n', h', d'), (n, h, d) \rangle \) s.t. \( n = d = d' + 1 \) and \( h = h' \); \( New(d) \) is renaming-equivalent to a trace of \( C \).

The relation for a method such as \( M(self, ind) \) where \( ind \) is an indicator type is defined as the set of all \( \langle (n', a', h', d'), (n, a, h, d) \rangle \) s.t. \( n = n' \), \( d = d' \), and \( h = h' \); \( M(n', a')a \) is renaming-equivalent to a trace of \( C \).

2.3 Component Refinement

Now we consider what it means for one component to refine another: that is, it behaves compatibly but perhaps more deterministically. Refinement is the way
we reason about components, to determine whether one component is “as good as another” and hence can be used in its place.

When we say that $C$ refines $A$, we mean intuitively that $C$ behaves like $A$, but perhaps more nondeterministically. However, $C$ can also have some additional behaviors, as the following examples illustrate. Suppose we have the following three components, each of which defines a domain and none of which have shared variables. All methods are visible client methods.

- Component $B_2$, defining a bag (multiset) of size 2 of integers with methods:
  - $add(B_2, Int)$ that adds the integer to the bag, provided the bag has no more than two items in it; the method is otherwise undefined.
  - $remove(B_2, Int)$ that removes some integer from the bag provided the bag is not empty; the method is otherwise undefined.

- Component $S_2$ defining a stack of size 2 of integers with methods:
  - $add(S_2, Int)$ that pushes the integer onto the stack, provided the stack has no more than two items in it; the method is otherwise undefined.
  - $remove(S_2, Int)$ that removes the top integer from the stack, provided the stack is not empty; the method is otherwise undefined.

- Component $S_1$ defining a stack of size 1 of integers with the same methods as $S_2$ but limited to one item in the stack.

Suppose some client (a notion we leave informal for now) were to use the $B_2$ component. Would it be possible to substitute component $S_2$, replacing references
of $B_2$ to $S_2$? We argue that the answer is “yes” since the difference in the client’s behavior would be merely a reduction in non-determinism.

Could component $S_1$ be used instead of $B_2$? Here the answer is “no” since with $B_2$ the client could legitimately do two successive $add$ methods to the same bag, but an attempt to do that with the same stack would fail. $S_1$ fails to provide the necessary service.

Suppose now the client were to use $S_2$. Could $B_2$ be used instead? Again the answer is “no” since $B_2$ exhibits more nondeterminism than $S_2$. For example, if the stack contained $(1, 2)$ then a subsequent $remove$ on that stack would always return 2, whereas $B_2$ executing a $remove$ could return either 1 or 2. $B_2$ fails behavioral conformity.

Finally, suppose the client were to use $S_1$. Could $S_2$ be used instead? The answer is “yes” since any client using $S_1$ would behave identically if $S_2$ were used instead. That is, $S_2$ permits behaviors that are not exploited by the client.

Now we turn to the definition of component refinement. Given a trace of the form $\sigma'; e$ where $e = (v, r)$, $\sigma'; v$ is an extension of $\sigma$. The definition we give next of component refinement is a strengthening of that due to Ogden [29].

**Definition 2.5 (Component Refinement)** For components $C$ and $A$, $C$ refines $A$ if the following hold.

- Whenever $\sigma'; v$ is an extension of $A$ and $\sigma'$ is a trace of $C$,
  - (Refinement Service) $\sigma'; v$ is an extension of $C$.
  - (Refinement Behavior) Every trace $\sigma'; <v:r>$ of $C$ for some $r$ is a trace of $A$. 

35
• (Refinement Safety) If $\sigma$ is an infinite trace of $C$ and every finite prefix of $\sigma$ is a trace of $A$ then $\sigma$ is a trace of $A$. \hfill \square

As alternative language we can also say that $C$ meets $A$ or satisfies $A$. If $C$ refines $A$ then we write $C \subseteq A$. Overloading $\subseteq$, the arguments can alternatively be given as scenarios or traces.

If $C$ does not satisfy the service condition of $A$ then this occurs at some particular event $e$ in a trace of $A$. Not only can $C$ not engage in $e$, it cannot engage in any of the successors of the trace. Hence failure of the service condition means failure of the entire suffix of the trace.

**Theorem 2.1 (Refinement is a Preorder)** The refinement relation is a preorder: transitive and reflexive.

The proof can be found in Appendix B. Transitivity lets us write $C \subseteq B \subseteq A$ for $C \subseteq B$ and $B \subseteq A$.

### 2.4 Program Components

Refinement gives us a way to reason about components in terms of their behavior. Program components give us one of our key goals, a way to compose components into a system. As with relational components, program components have methods, but the methods are given not by relations but by programs that invoke methods of other components. A relational component has no meaning in itself, but only as it is composed with other components whose meaning is known. Hence the meaning of a program component will vary according to the meaning of the components it uses.
A program component is defined in terms of a set of subcomponents that are properly configured. We next give the definition of a program component, followed by the definition of configuration. In the definition, assume we have a set of programs, \( \text{Programs} \); programs are defined in Appendix C.

**Definition 2.6 (Program Component)** A program component \( L \) for a configuration \( \text{cfg} = \{C_0, \ldots, C_{n-1}\} \) is a tuple

\[
(L.\text{cdom}, (L.\text{cshare}, L.\text{cinhshare}, L.\text{cshinit}), (L.\text{cmeth}, L.\text{cprog}, L.\text{cvis}, L.\text{cctl}), L.\text{ctype}),
\]

where every indicator domain of \( L \) is a domain of a component in the configuration.

- \( L.\text{cdom} \in \text{Domains} \) is a **composite component domain** that is a cross product of indicator domains: \( L.\text{cdom} = C_0.\text{cdom} \times \ldots \times C_{k-1}.\text{cdom} \) for some \( k \geq 0 \) s.t. \( 0 \leq i \leq k, \ 0 \leq l_i < n \). As with relational components, \( L.\text{cdom} \) can be empty, in which case we say that the component does not define a domain.

- \( L.\text{cshare} \) is defined as for a relational component.

  - \( L.\text{cinhshare} \) is a tuple of shared variables that are inherited from the configuration components: \( \text{cinhshare} = C_0.\text{cshare}; \ldots; C_{n-1}.\text{cshare} \).

  - \( L.\text{cshinit} \) is a function \( L.\text{cshinit} : \bigcup(\text{cshare}; \text{cinhshare}) \to \text{Domains} \) that maps shared and inherited variables to initial values s.t.

    - \( L.\text{cshinit}(G) \subseteq L.\text{ctype}(G) \),

    - \( L.\text{cshinit}(G) \) is empty iff \( L.\text{ctype}(G) \) is, and

    - \( L.\text{cshinit} \) agrees with the initialization function for each inherited shared variable.
• \textit{L.cmeth} is a set of \textit{method names}. As with relational components, if the component defines a domain then \textit{New}(\textit{self}) is used to create or reinitialize instances.

- \textit{L.cprog} is a function \textit{cprog} : (\textit{cmeth}, \textit{cfg}) \rightarrow \textit{Programs} that maps each method in the context of a configuration to a program. The set of programs is defined below.

- \textit{C.cvis} is defined as for relational components.

- \textit{L.cctl} is defined as for relational components.

• \textit{L.cctype} is defined as for relational components but includes the \textit{ctype} function from each of the configuration components.

The domain consisting of tuples of indicator values is similar to the record types of [7, Chap. 10.3]. A program component is often referred to as a \textit{client component} or just \textit{client}. The shared variables are inherited from the configuration into the client to account for the fact that invocations of a configuration component can modify those variables.

The programs of a program component are written in a simple imperative language consisting of sequences of calls on methods of relational subcomponents along, with control statements \textbf{if} \ldots \textbf{then} \ldots \textbf{fi} and \textbf{do} \ldots \textbf{od}. Details on program syntax and semantics are in Appendix C.

The meaning of a program component is given with respect to a configuration of components. Components in a set can reference indicator domains (that is, the
domains of other components). This referencing induces a dependency graph, so that $C_1$ depends on $C_2$ if the domain of $C_2$ is an indicator for $C_1$. A component in the set is top-level if all dependencies are outgoing and is bottom-level if all dependencies are incoming.

**Configuration.** A *configuration* is a set of components whose dependency graph is acyclic and whose bottom-level components are relational or behavioral. The domains defined by the configuration components are called *configuration domains*.

Requiring the bottom-level components to be relational or behavioral lets us calculate the meaning for any program component in the configuration. We simply proceed from the bottom up, converting behavioral components to relational ones as necessary, and calculating the meaning of program components based on the meaning of lower-level components.

### 2.5 Monotonicity under Substitution

Combined with refinement, program components give us another key goal, locality in reasoning. Refinement permits us to replace a component by a refinement in a system while ensuring that the system continues to meet its specification. Recalling the notation from the Introduction, it permits us obtain the result that If $C$ refines $CSpec$ and $L(CSpec)$ refines $LSpec$ then $L(C)$ refines $LSpec$.

**Theorem 2.2 (Client Substitution is Monotonic)** Let $L$ be a client component and $C$ and $A$ be relational components. Let $cnf_C$ be a configuration and $cnf_A$ be the same configuration but with $C$ replaced by $A$. Let $C$ be a top-level component for $cnf_C$, $A$ a top-level component for $cnf_A$, and assume that $L$ does not have the domains of $C$ or $A$ as indicators.
Suppose $C \subseteq A$. Then $L(C) \subseteq L(A)$. □

The proof is in Appendix B. This monotonicity result is restricted to clients that do not reference $CSpec$ (or $C$) as an argument domain in any method. If this were not so, then the domain of $CSpec$ would be an indicator for $L(CSpec)$, while $C$ would be an indicator of $L(C)$. Since these domain values would be exposed in the respective traces, there would be no refinement. It is also restricted to substitution of top-level components in the configuration since the proof depends upon being able to extract a scenario for $C$ from an operational unrolling of the programs of $L(C)$. If $C$ were not top-level, some other component could modify values of $C$ in the unrolling and we could not extract the trace.

The converse of this theorem is not true because refinement can include unreachable indicator values. To see this, consider Fig. 2.1. Here, $C.M$ has a single argument defined over component $D$. $D$’s domain is \{1, 2, 3, 4\}. $D.New(X)$ initializes the argument variable to the value 1. Only the value 2 is reachable by methods of $D$; values 3 and 4 are unreachable. Since these values are unreachable, no client program will attempt to invoke $A.M$ with an argument of 3. However, $C$ contains the trace $\langle M(3):3 \rangle$, which is not a trace of $A$. Accounting for the reachability of indicator values would make reasoning more complex, and the converse of the theorem is not necessary for our goals.

### 2.6 Conditional Refinement and Monotonicity

Now we deal with the case in which we assert monotonicity not for every possible client, but for a restricted set of them. Suppose $C$ does not refine $A$ but some subset of the traces $C$ does. This means that if $C$ is used properly by a client then we can
substitute $C$ for $A$ with the assurance that the client will still behave properly; that is, that $L(C) \subseteq L(A)$. There is an example of this in the case study for resettable vector clocks (RVC), Chapter 5, where the RVC component does not refine its specification, but a restricted set of the behaviors of RVC does.

**Definition 2.7 (Conditional Refinement)** Let $Ry$ be a set of event sequences for component $C$. Then $C$ refines $A$ with respect to $Ry$ if $C \cap Ry \subseteq A$.

From a client’s point of view, if $C \cap Ry \subseteq A$ then we are assured that $L(C) \subseteq L(A)$ only if $L(C)$ is restricted to $Ry$ in the sense that the unfolding traces of $L(C)$ (see Appendix C.2) with respect to $C$ subset $Ry$. In this case, we say that $L(C)$ satisfies the rely condition.

**Theorem 2.3 (Conditional Client Substitution is Monotonic)** Assume the same restrictions as for regular monotonicity: $C$ and $A$ are top-level in their configurations and $L$ does not reference either one.

If $\text{traces}(C) \cap Ry \subseteq A$ then for any configuration containing $C$ and any program client $L$ s.t. $L(C)$ satisfies $Ry$, $L(C) \subseteq L(A)$.
**Proof.** The proof of this theorem is similar to that of Theorem 2.2. In that theorem we extracted traces of $C$ from the parallel executions of $L(C)$ and $L(A)$ on a trace and used the fact that $C \sqsubseteq A$ to get the result. For this proof we merely observe that the conditions of the theorem give us that each extracted trace of $C$ is in $Ry$ and, since $\text{traces}(C) \cap Ry \sqsubseteq A$, we know that $C$ is a trace of $A$. \[\Box\]

Suppose a client uses two subcomponents, each of which comes with a rely condition. The client must be able to satisfy both rely conditions simultaneously. If the client cannot simultaneously guarantee these rely conditions, then the behavior of the subcomponents with respect to their specifications cannot be guaranteed.
CHAPTER 3

State-Level and Value-Level Component Verification

Reasoning directly over sets of traces is hard since traces can be infinite. A more practical approach that involves only local reasoning over component methods is that of using simulation relations [23]. We define two simulation relations, forward and backward. The backward simulation is similar to the “correspondence” in [31]. Both simulations include a condition to account for the service condition in the definition of component refinement given in Chapter 2.

The simulations are given in two forms: state-level and value-level. As the names imply, a state-level simulation relates concrete states to abstract states while a value-level simulation relates the values that make up the states. For example, suppose we wish to refine a component Bag with the domain of bags of integers and with methods add(self, Int) and remove(self, Int) by another component Stack with the domain of stacks of integers and with the same methods. A state-level simulation would relate a given state of the stack to the corresponding state(s) of the bag; a value-level simulation would instead relate a particular stack to those bags containing the same elements and would relate the integers with an identity.

Value-level reasoning has two attractions. In the first place, this kind of reasoning is in accord with the notion of data abstraction: components define domains (bags,
queues), not state spaces. By contrast, much of the treatment in the literature on
data refinement supposes a particular state space and does not deal with the issue of
using a component in a different state space.

In the second place, the complexity of value-level reasoning is typically less than
that of state-level. To verify that the stack refines the bag we would have to show,
for the state-level simulation, that the relation holds for all events in all states. For
the value-level simulation, we need to show only that the relation holds for the com-
binations of values mentioned in the methods, a process that is less complex.

Even though value-level reasoning is preferable, it is not always possible. We
give an example showing that value-level reasoning is impossible although state-level
reasoning will still work.

3.1 State-Level Verification

Now we give the state-level definitions of forward and backward simulation rela-
tions. Two components are compatible if both define a domain or if neither do, and if
have the same signature. Compatibility ensures that the events of both components
are the same. In the following, assume that $C$ and $A$ are compatible components.

3.1.1 State-Level Forward and Backward Simulations

First we define the forward simulation relation and then the backward one. Both
are preorders.

**Definition 3.1 (State-level Forward Simulation Relation)** A state-level for-
ward simulation relation from $C$ to $A$ is a relation $f$ over the states of $C$ and $A$ s.t.
the following hold.
• (SF Init) If $s$ is an initial state of $C$ then $f[s]$ intersects the initial states of $A$.

• For each state $s'$ and visible invocation $v$ of $C$, suppose $v$ is enabled in $A$ on some $u' \in f[s']$. Then the following hold.
  - (SF Service) $v$ is enabled in $C$ on $s'$.
  - (SF Visible Behavior) For all $r$, if $e = (v, r)$, $s' \xrightarrow{e} C s$ and $u' \in f[s']$ then for some $\tilde{\alpha} = e$ there is $u \in f[s]$ s.t. $u' \xrightarrow{\tilde{\alpha}} A u$.

• (SF Hidden Behavior) For each state $s'$ and hidden event $e$ of $C$, if $s' \xrightarrow{e} C s$ and $u' \in f[s']$ then $u' \in f[s]$.

\[ \Box \]

**Definition 3.2 (State-level Backward Simulation Relation)** A state-level backward relation from $C$ to $A$ is a total relation $b$ over the states of $C$ and $A$ s.t. the following hold.

• (SB Init) If $s$ is an initial state of $C$ then $b[s]$ subsets the initial states of $A$.

• For each state $s'$ and visible invocation $v$ of $C$, suppose $v$ is enabled in $A$ on all $u' \in b[s']$. Then the following hold.
  - (SB Service) $v$ is enabled in $C$ on $s'$.
  - (SB Visible Behavior) For all $r$, if $e = (v, r)$, $s' \xrightarrow{e} C s$ is a transition of $C$ and $u \in b[s]$ then for some $\tilde{\alpha} = e$ there is $u' \in b[s']$ s.t. $u' \xrightarrow{\tilde{\alpha}} A u$.

• (SB Hidden Behavior) For each state $s'$ and hidden event $e$ of $C$, if $s' \xrightarrow{e} C s$ and $u \in b[s]$ then $u \in b[s']$.  

\[ \Box \]
The service condition in these definitions distinguish them from the standard ones [23]. The service condition guarantees that if the abstract component can make progress, then so can the concrete, thus preventing a simulation from a bag of size 1 to a bag of size 2.

\( b \) is \textit{image-finite} if for all \( s \) where it is defined, \( b[s] \) is finite. We write \( C \leq_f A \), \( C \leq_b A \), \( C \leq_{ib} A \) to indicate state-level forward, backward, and image-finite backward simulations from \( C \) to \( A \), respectively. These relations are pre-orders: transitive and reflexive.

### 3.1.2 Soundness of State-Level Simulations

Here we show that state-level component simulations are sound with respect to component refinement. We begin with the forward simulation.

**Theorem 3.1 (Forward State-Level Simulations Are Sound)** If there is a state-level forward simulation from \( C \) to \( A \) then \( C \sqsubseteq A \).

The proofs are in Appendix B.

Backward simulations are sound only if they are \textit{image-finite}, where a binary relation \( r \) is image-finite if \( \{ b : (a, b) \in r \} \) is finite.

To see this, consider the following example. Define \( A \) having shared variables \( x \) defined over natural numbers, with initialization being any value; and \( y \) also defined over natural numbers, with initialization \( \{0\} \). \( C \) has method \( M(Nat) \) defined as follows.

<table>
<thead>
<tr>
<th>( arg' )</th>
<th>( x' )</th>
<th>( y' )</th>
<th>( arg )</th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>( m' &gt; 0 )</td>
<td>( n' )</td>
<td>( n )</td>
<td>( m' - 1 )</td>
<td>( n' + 1 )</td>
</tr>
</tbody>
</table>
In $A$, $x$ is initialized to any natural number while $y$ is initialized to 0. Calls to $M$ decrease $x$ and increase $y$ until $x$ becomes 0. The traces of $A$ are as follows.

\[
\begin{align*}
\langle \rangle \\
\langle M(*):0 \rangle \\
\langle M(*):0, M(*):1 \rangle \\
\langle M(*):0, M(*):1, M(*):2 \rangle \\
\vdots
\end{align*}
\]

However, the infinite sequence $\langle M(*):0, M(*):1, M(*):2, \ldots \rangle$ is not a trace of $A$.

Define $C$ as having shared variable $y$ defined as for $A$. $A$ has method $M(Nat)$ defined as follows.

\[
\begin{array}{cc}
\text{arg} & y' \\
\ast & n' \\
\end{array}
\]

In $C$, $y$ can increase without bound. Hence $C$ has the traces of $A$ along with the infinite sequence:

\[
\begin{align*}
\langle \rangle \\
\langle M(*):0 \rangle \\
\langle M(*):0, M(*):1 \rangle \\
\langle M(*):0, M(*):1, M(*):2 \rangle \\
\vdots
\end{align*}
\]

$C$ does not refine $A$ since it fails the safety condition: $C$ has an infinite trace $\sigma$ and every prefix of $\sigma$ is a trace of $A$, but $\sigma$ is not a trace of $A$.

Despite the lack of refinement, there is a backward simulation $b$ from $C$ to $A$. Let $b[(y = j)] = \{(x = k, y = j) : b \geq 0\}$. $b$ is total and satisfies the initialization condition. Let $e = M(*):n$ for some natural number $n$. For any state $(y = j)$ of $C$, $e$ is enabled in $A$ on every $(x = k, y = j) \in b[(y = j)]$ iff $j = n$. Let $(y = n')$ be a state of $C$; we must show satisfaction of service and behavior. By the definition of $C$, $e$ is enabled on $(y = n')$ in $C$, satisfying the service condition. Let $((y = n'), e, (y = n))$ be a transition of $C$. By the definition of $C$, $n = n' + 1$. Let $(x = k, y = n' + 1) \in
We must show that there is \((x = k', y = j') \in b[(y = n')]\) s.t. \(((x = k', y = j'), e, (x = k, y = n' + 1))\) is a transition of \(A\). This is so if \(k' = n'\) and \(j' = b + 1\).

The problem has to do with \textit{finite invisible nondeterminism} (fin). \(A\) has fin if the set of initial states is finite and if for every state \(u'\) and every event \(e\), the set of all \(u\) forming transitions \((u', e, u)\) of \(A\) is finite. Here, \(A\) has an infinite set of initial states and so does not have fin. This is what permits \(A\) to have all finite traces of \(C\), but not the infinite trace.

If we restrict \(b\) to being image-finite then there is no backward simulation from \(C\) to \(A\) in our example. Suppose there were, and suppose for any state \(s\) of \(C\), the size of \(b[s]\) is 1. Let \(b[(y = n)]\) have the single state of the form \((x = k_n, y = n)\) for some \(k_n\). For \(((y = 0), M(*):0, (y = 1))\), we must have \(k_0 = 1\) and \(k_1 = 0\) to satisfy the behavior condition. Similarly, for \(((y = 1), M(*):1, (y = 2))\), we must have \(k_1 = 1\) and \(k_2 = 0\). Hence we must have that \(b[(y = 1)] = \{(y = 1, x = 0), (y = 1, x = 1)\}\), violating the assumption that \(b[(y = 1)]\) only has a single state. We may proceed inductively for the image of \(b\) having any finite size and arrive at a contradiction.

To attain soundness, we restrict \(b\) to being image finite. Note that this is a sufficient condition for soundness, not a necessary one. Now we show the soundness of state-level image-finite backward simulations.

\textbf{Theorem 3.2 (Backward State-Level Simulations Are Sound)} \textit{If there is a state-level image-finite backward simulation from \(C\) to \(A\) then \(C\) refines \(A\).}

The proof is in Appendix B.
3.1.3 Completeness of State-Level Simulations

As with standard simulations, the individual state-level simulations are incomplete, but are jointly complete provided, as discussed above, the abstract component has fin. The fin condition ensures that we can find an image-finite backward simulation.

Theorem 3.3 (State-level Simulations Are Relatively Complete) If $C$ refines $A$ and $A$ has fin then there is $B$ s.t. $A \leq_f B \leq_{stb} C$.

The proof is in Appendix B.

3.2 Value-Level Verification

Now we turn to the less-complex value-level simulation relations. These are defined not over states, but over the values that make up the states. As with state-level simulations, these are preorders: transitive and reflexive.

Value-level simulations can be considered when the components involved do not have hidden methods, and where either both have shared variables or neither has. If they have shared variables, then they must both have the same number of variables and they must have the same names.

A domain-sorted relation $r$ over domains $D_0, \ldots, D_{n-1}$ is a collection of binary relations s.t. for $i < n$, $r_{D_i} \subseteq D_i \times D_i$. Since domains are pairwise disjoint, $r[d]$ is unambiguous. Given a tuple $\overline{d}$ of values from these domains, $r[\overline{d}] = r[d_0] \times \ldots \times r[d_{|\overline{d}|}]$. Let $\overline{a} \xrightarrow{M} C \overline{a}$ stand for $\overline{a} \in C.M[\overline{a}]$. 

49
3.2.1 Value-Level Forward and Backward Simulations

Assume that the shared variables of \( C \) and \( A \) have the same names, though they may be defined over different domains. For simplicity (and without loss of generality), we assume that for any method \( M(\text{args}) \) of \( C \), the argument domains may be given as \( \text{ind}; \text{hid} \) where \( \text{ind} \) is a tuple of the indicator domains and \( \text{hid} \) is a tuple of the domains \( \text{self} \). The tuple of the domains of the shared variables is \( \text{glbl} \).

**Definition 3.3 (Value-level Forward Simulation Relation)** A value-level forward simulation relation from \( C \) to \( A \) is a domain-sorted relation \( \text{vf} \) over the domains referenced in \( C \) s.t. \( \text{vf} \) is identity for indicator domains, \( \text{vf} \) relates the domain of \( C \) to that of \( A \), \( \text{vf} \) relates the domain of \( C.G_i \) to that of \( A.G_i \), and the following hold

- (VF Init) For each initial value \( c \) in the domains of the shared variables \( C.G_i \), \( \text{vf}[c] \) intersects the initial values of the domains of \( A \) and \( A.G_i \), respectively.

- Let \( M(\text{ind}; \text{hid}) \) be a method of \( C \). Let \( \overrightarrow{d} \in \times \text{ind} \) and \( \overrightarrow{c} \in \times (\text{hid}; \text{glbl}) \). Suppose for some \( \overrightarrow{a} \in \text{vf}[\overrightarrow{c}] \) that \( A.M[\overrightarrow{d}; \overrightarrow{a}] \neq \emptyset \). Then the following hold.
  - (VF Service) \( C.M[\overrightarrow{d}; \overrightarrow{c}] \neq \emptyset \).
  - (VF Behavior) If \( \overrightarrow{d}; \overrightarrow{c} \xrightarrow{M} C \overrightarrow{d}; \overrightarrow{c} \) and there is \( \overrightarrow{a} \in \text{vf}[\overrightarrow{c}] \) then there is \( \overrightarrow{a} \in \text{vf}[\overrightarrow{c}] \) s.t. \( \overrightarrow{d}; \overrightarrow{a} \xrightarrow{M} A \overrightarrow{d}; \overrightarrow{a} \).

**Definition 3.4 (Value-level Backward Simulation Relation)** A value-level backward simulation relation from \( C \) to \( A \) is a domain-sorted relation \( \text{vb} \) over the domains referenced in \( C \) s.t. \( \text{vb} \) is total, \( \text{vb} \) is identity for indicator domains, \( \text{vb} \)
relates the domain of $C$ and $C.G_i$ to that of $A.G_i$; $vb$ relates the domain of $C$ to that of $A$, $vb$ relates the domain of $C.G_i$ to that of $A.G_i$, and the following hold

- **(VB Init)** For each initial value $c$ in the domains of the shared variables $C.G_i$, $vb[c]$ subsets the initial values of the corresponding domain of $A.G_i$.

- **Let $M(\text{ind}; \text{hid})$ be a method of $C$. Let $\vec{d}, \vec{c} \in \times \text{ind}$ and $\vec{c'} \in \times (\text{hid}; \text{glbl})$. Suppose for all $\vec{a} \in vb[\vec{c}]$ that $A.M[\vec{d}; \vec{a}] \neq \emptyset$. Then the following hold.**
  - **(VB Service)** $C.M[\vec{d}; \vec{c}] \neq \emptyset$.
  - **(VB Behavior)** For all $\vec{c} \in vb[\vec{c}]$, if $\vec{d}; \vec{a} \xrightarrow{M} \vec{d}; \vec{c}$ and $\vec{a} \in vb[\vec{c}]$ then there is $\vec{a} \in vb[\vec{c}]$ s.t. $\vec{d}; \vec{a} \xrightarrow{A} \vec{d}; \vec{a}$.

A value-level backward simulation $vb$ from $C$ to $A$ is image finite for each $c$ in the domain of $C$ or in the domain of a shared variable, $vb[c]$ is finite.

We write $C \leq_{vf} A$, $C \leq_{vb} A$, $C \leq_{ivb} A$ to indicate state-level forward, backward, and image-finite backward simulations from $C$ to $A$, respectively. These relations are pre-orders: transitive and reflexive.

### 3.2.2 Soundness of Value-Level Simulations

We show that component simulations are sound with respect to component refinement. As with state-level, value-level backward simulations are sound only if they are image-finite.

**Theorem 3.4 (Value-Level Forward Simulations Are Sound)**

*If there is a value-level forward simulation from $C$ to $A$ then $C$ refines $A$.***
Theorem 3.5 (Value-Level Image-Finite Backward Simulations Are Sound)

*If there is an image-finite value-level backward simulation from C to A then C refines A.*

The proofs are in Appendix B. The definitions for value-level simulations require that the shared variables be the same in both abstract and concrete components, though the domains may vary. In cases where they are not the same, a mixed kind of simulation can be carried out. In this approach, the states of the shared variables are treated as a domain of values, and then the value-level kind of reasoning is used.

### 3.2.3 Incompleteness of Value-Level Simulations

These results, in a different setup, were given by Leal and Arora in [19]. Here we see that value-level simulations for components are incomplete even if the abstract has fin. Note that the value-level simulations depend on both the abstract and concrete components having the same shared variables, even if over different domains. The incompleteness of the simulations could be trivially shown just by giving different shared variables. However, the counter-example used to show incompleteness does not depend on shared variables. Hence it applies not only to our setup, but also to other setups in which components only define domains.

Consider Fig. 3.1. The abstract data type *EagerToss* models an ambitious but incompetent juggler that tries to toss two ready plates (*R*) so that they are both right side up (*Up*) or upside down (*Dn*); he notes which way they came up (*tt* if both were up, *ff* if both were down, and either *tt* or *ff* if they don’t match) but he can’t catch them, so they smash at the end (*H*). A programmer implementing this (*LazyToss*) observes that deciding how they came up can be postponed, so takes two available
plates \((L)\), makes them match \((M)\), and then nondeterministically decides whether they were right side up or not \((tt\ or\ ff)\) before they are broken \((K)\). In this example, plates are hidden sorts. For \textit{EagerToss}, the domain for plates is \(\{R,\ Up,\ Dn,\ H\}\) while for \textit{LazyToss}, the domain is \(\{L,\ M,\ K\}\). For both, the visible values are three-state Booleans \(\{tt,\ ff,\ ?\}\), where “?” indicates an undefined value.

<table>
<thead>
<tr>
<th>New</th>
<th>EagerToss</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>(R,R) (\rightarrow) (Up,Up) (\rightarrow) (Dn,Dn)</td>
<td>(H,H,tt)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>New</th>
<th>LazyToss</th>
<th>Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L)</td>
<td>(L,L) (\rightarrow) (M,M)</td>
<td>(K,K,tt)</td>
</tr>
</tbody>
</table>

Figure 3.1: Diagram for Incompleteness of Value-Level Simulations

Component \textit{EagerToss}, which we will call \(A\), defines a domain of \(\{R,\ Up,\ Dn,\ H\}\) with initial value \(R\). Component \textit{LazyToss}, which we will call \(C\), defines a domain of \(\{L,\ M,\ K\}\) with initial value \(L\). Besides \textit{New}, there are two methods: \textit{MakeMatch}(\texttt{self, self})\ and \textit{Check}(\texttt{self, self, Bool}). The relations for the operations are as given in Fig. 3.1.
Since state-level forward and backward simulations are individually incomplete but jointly complete, we might suppose that value-level forward and backward simulations are also jointly complete. However, this is not the case, as the following theorem shows.

**Theorem 3.6 (Value-level Simulations are Incomplete)** There exists $C$ and $A$ s.t. $C$ refines $A$ and $A$ has fin, but for no component $B$ is it the case that $A \leq_{vf} B \leq_{ivb} C$ or $A \leq_{ivb} B \leq_{vf} C$. □

The proof is in Appendix B.

**Proof.**

**Refinement.** First we argue that LazyToss, $C$, is a refinement of EagerToss, $A$. Consider a trace $\sigma$ of $C$. Note that in any such $\sigma$ every variable is matched at most once and is checked at most once. Let $s$ be reachable by $\sigma$ in $C$. We have the coupling invariant that there if state $u$ is reachable by $\sigma$ in $C$, then for each variable $Y$, if $u(Y)$ is Up or Dn then $s(Y) = M$. It is clear that, if the Check operation were removed from the components, then they would be trace equivalent. So let us consider invocations involving Check. Assume that $\sigma'$ is a trace of both components, and consider the extension $\sigma';(\text{Check}(X,Y,?))$. Consider two cases.

- Case 1. $X$ and $Y$ were matched together. For $A$ the matching will nondeterministically give us abstract states $u'_1$ and $u'_2$ where $u'_1(X) = u'_1(Y) = Up$ and $u'_2(X) = u'_2(Y) = Dn$. In $C$ the matching gives us a state $s'$ in which the variables are both $M$. Given the invocation $\text{Check}(X,Y,?)$. Then we have the following transitions of $A$: $(u'_1, Check(X,Y,?):tt, u'_1), (u'_1, Check(X,Y,?):ff, u'_1), (u'_2, Check(X,Y,?):tt, u'_2), (u'_2, Check(X,Y,?):ff, u'_2)$. We have the following
transitions of $C$: $(s', \text{Check}(X, Y, ?): tt, s')$ and $(s', \text{Check}(X, Y, ?): ff, s')$. Hence the outcomes are visibly identical and we have the same trace.

- Case 2. $X$ and $Y$ were not matched together; that is, each was matched with other variables. Then we will have an abstract state $u'$ where $u'(X)$ is $Up$ and $u'(Y)$ is $Dn$ or vice versa. In $C$ the matching gives us a state $s'$ in which the variables are both $M$. Given the invocation $\text{Check}(X, Y, ?)$. Then we have the following transitions of $A$: $(u', \text{Check}(X, Y, ?): tt, u')$ and $(u', \text{Check}(X, Y, ?): ff, u')$. We have the following transitions of $C$: $(s', \text{Check}(X, Y, ?): tt, s')$ and $(s', \text{Check}(X, Y, ?): ff, s')$. Hence the outcomes are visibly identical and again we have the same trace.

**Forward-Backward.** Let $B$ be any data type such that $vf$ is a value-level forward simulation from $C$ to $B$. We show that there is no value-level backward simulation $vb$ from $B$ to $A$. For the sake of contradiction, suppose that $vb$ is such a value-level backward simulation. Figure 3.2 shows the commutativity relationships for the example. From this figure we may read off several facts. First, for all $c \in \{L, M, K\}$ and for all $b \in vf[c]$, $vf[c] \neq \emptyset$ and $vb[b] \neq \emptyset$. Second, there is $e$ in $vf[L]$ and for all such $e$, $vb[e] = \{R\}$.

Next we observe that since there are $f_1, f_2 \in vf[M]$ s.t. $(d, d) \xrightarrow{B.\text{MakeMatch}} (f_1, f_2)$ and since $vb_{mm}[(d, d)] = \{(R, R)\}$, then $vb[f_1]$ and $vb[f_2]$ both subset $\{Up, Dn\}$. Now we consider the three cases for $vb[f_1]$.
Quantifications shown are implied by the definitions of value-level simulation. For instance, \( \forall (d, d) \in vf[(L, L)] \exists (f_1, f_2) \in vf[(M, M)]: (d, d) \rightarrow^B (f_1, f_2) \rightarrow^A (H, H, \text{tt/ff}) \).

Figure 3.2: Diagram for Incompleteness Proof (Forward-Backward)

- \( vb[f_1] = \{ Up \} \). We have \((f_1, f_1, ?) \rightarrow^A (g_1, g_2, \text{ff}) \) and \((H, H, \text{ff}) \in vb_{\text{ck}}[(g_1, g_2, \text{ff})] \). We have \( vb[(f_1, f_1, ?)] = \{(Up, Up, ?)\} \) but this and (VL Backward Behavior) would falsely imply that \((Up, Up, ?) \rightarrow^B (H, H, \text{ff}) \).

- \( vb[f_1] = \{ Dn \} \). The reasoning is similar to the preceding case.

- \( vb[f_1] = \{ Up, Dn \} \). Suppose \( vb[f_2] \) contains \( Dn \). Then we have \((d, d) \rightarrow^B (f_1, f_2) \) and \((Up, Dn) \in vb[(f_1, f_2)] \). We have \( vb[(e, e)] = \{(R, R)\} \) but this and (VL Backward Behavior) would falsely imply that \((R, R) \rightarrow^B (Up, Dn) \).

The case for \( vb[f_2] \) containing \( Up \) is similar.

Since all possibilities for \( vb \) have failed, we conclude there is no value-level backward simulation from \( B \) to \( A \).

**Backward-Forward.** Let \( B \) be any data type such that \( vb \) is a value-level backward simulation from \( C \) to \( B \). We show that there is no value-level forward simulation
vf from B to A. For the sake of contradiction, suppose that vf is such a value-level forward simulation. Figure 3.3 shows the commutativity relationships for the example. From this figure we may read off several facts. First, for all $c \in \{L, M, K\}$

![Diagram for Incompleteness Proof (Backward-Forward)](image)

Figure 3.3: Diagram for Incompleteness Proof (Backward-Forward)

and for all $b \in vf[c]$, $vb[c] \neq \emptyset$ and $vf[b] \neq \emptyset$. Second, for all $d \in vb[L]$, $R \in vf[d]$.

Next we observe that since there are $f_1, f_2, f_3, f_4 \in vb[M]$ (not necessarily unique) as shown, $vf[f_1]$, $vf[f_2]$, $vf[f_3]$, and $vf[f_4]$ must contain Up or Dn (or both). We have the following two facts.

- Either $vf[f_1]$ or $vf[f_2]$ does not contain Dn. If both do, then we have $(f_1, f_2, ?) \rightarrow_B (g_1, g_2, tt)$ and $(Dn, Dn, ?) \in vf[(f_1, f_2, ?)]$, but for no $(m_1, m_2, tt) \in vf[(g_1, g_2, tt)]$ is it the case that $(Dn, Dn, ?) \rightarrow_A (m_1, m_2, tt)$, violating (VL Forward Behavior).

- Either $vf[f_3]$ or $vf[f_4]$ does not contain Up. The reasoning is similar to the preceding case.
Now we show the contradiction. Suppose $vf[f_1]$ contains $Up$ but not $Dn$ and $vf[f_3]$ contains $Dn$ but not $Up$. Since $f_1, f_3 \in vb[M]$, there is $(d_1, d_2) \in vb[(L, L)]$ s.t. $(d_1, d_2) \xrightarrow{\text{MakeMatch}} B (f_1, f_3)$. We have $(R, R) \in vf[(d_1, d_2)]$ but since $vf[(f_1, f_3)]$ does not contain $(Up, Up)$ or $(Dn, Dn)$, we do not meet the (VLB Forward Behavior) condition for $\text{MakeMatch}$. The case for $vf[f_1]$ containing $Dn$ but not $Up$ and $vf[f_3]$ containing $Up$ but not $Dn$ is similar. So we conclude there is no value-level forward simulation from $B$ to $A$.

Hence value-level simulations are incomplete, and state-level simulations must be used instead. Note that the example does not depend on any particular definition of termination. If our notion of traces were replaced by ordinary computation traces, or if different termination semantics were chosen, the example would still apply.

### 3.2.4 Completeness of Monadic Value-Level Simulations

Although value-level simulations are incomplete, there is a special form that is complete. For *monadic components*, methods are restricted s.t. the method type includes at most one `self` argument and no shared variables.

For the completeness of state-level simulations, an intermediate data type was created that contained traces as a shared variable. The states of the concrete and the states of the abstract can be related by these traces. For value-level simulations with no shared variables, we use the same idea but apply it to the single `self` values that appear in the method relations. We can describe a value as reachable by a trace if the trace is monadic in $X$: that is, if the events of the trace reference at most the client variable $X$. 

58
Theorem 3.7 (Completeness of Monadic Value-Level Simulations) Suppose $C$ and $A$ are monadic components satisfying the conditions for value-level simulation. If $C$ refines $A$ and $A$ has fin then there is a component $B$ s.t. $C \sqsubseteq^v fB \sqsubseteq^{v^b} A$.

The proof is in Appendix B.
Tolerant Components

In this chapter we are concerned with fault-tolerant components. We associate faults with particular components. Hence we may talk about the faults $F$ of a component $C$. When a fault-affected component, defined below, is used by a client, the effects of the faults can propagate to the behavior of the client, but the faults remain at the level of $C$. Given a configuration of components, faults will, intuitively, become “coarser” the higher a component is in the hierarchy. For example, a stack component may depend on an integer component. Faults in the integer component will only affect integers and will, indirectly, affect the behavior of the stack component. The stack component may also have faults, which affect entire stacks. Hence the faults of the stack component deal with more complex data structures than do the faults at the integer level.

In our setup, $C$ can have shared variables. If $C$ has a domain and is used by a program of a client component, the program can define variables over instances of the component’s domain, if any. Since we view faults as state corruption, we permit the faults of $C$ to change any shared variable or any program variable over $C$. This is
accomplished by means of the fault methods that are permitted for relational, behavioral and program components. Representing faults as change of state is consistent with [5].

Component methods are atomic. This means that if method $M$ of a particular component is executed, there is no interleaving with any other method of the component. Suppose we have program component $L$ with faults $F$, and we want to model the effect of one of the faults during execution of $M$. This can be accomplished if we define an auxiliary shared variable and then let the fault set the variable before $M$ executes. The program of $M$ can be defined in such a way it’s behavior varies according to the value of the auxiliary variable, thus modeling the occurrence the fault during $M$’s execution.

A fault-affected component $C$ (or, a component in the presence of faults) is a component to which fault methods $F$ have been added, represented by $C \triangleright F$. Note that this is a new component. We say that a component is tolerant to faults if, in the absence of faults, it refines a given ideal (fault-free) specification, and in the presence of faults, it refines a tolerance specification. Usually, but not necessarily, the tolerance specification is derived in some way from the ideal specification.

We begin by defining a fault-affected component.

**Definition 4.1 (Fault-Affected Component)** Suppose $C$ is a component and $F$ is a set of faults methods that do not appear in $C$. Then the fault-affected component $C \triangleright F$ is $C$ with fault methods $F$, provided that $C \triangleright F$ satisfies the definition for a component.

The study of fault tolerance usually assumes that only a finite number of fault events occur in the traces of $C \triangleright F$. In our definition of component we require that
fault methods be total, so that faults are always enabled. In this sense, faults can always occur infinitely. However, on some states a fault may have no effect (that is, might not change the state). Let us ignore this situation and consider the case that faults could occur infinitely, changing the state each time. If we want to suppress this, so that the fault effect occurs only finitely, we can add an auxiliary shared variable to the component that is defined over the integers. Let the initial values for the variable be all integers, so initially the variable can have any value. Define a fault s.t. it makes state changes only if the auxiliary variable is positive, and each time the fault is invoked, the variable is decremented. Then the fault will have effect only finitely many times.

Since the application of $F$ by the fault environment is nondeterministic, there will be scenarios of $C$ in which the fault actions are not executed. Hence the traces of $C$ subset those of $C \triangleright F$. However, $C$ may not refine $C \triangleright F$, as we see in an example below.

In the rest of this chapter we give a general definition of fault tolerance, and then handle the three standard tolerances of masking, failsafe and nonmasking (with stabilizing an extremal instance). We show by example that refinement of a specification in the absence of faults does not imply refinement of a tolerance specification in the presence of faults, nor does the converse hold. We show how to reason about clients that use tolerant subcomponents, with the result that if a client is designed with both the ideal specification and the tolerance specification then the client can use any tolerant refinement. Finally, we give implications for synthesis tools such as tolerance compilers.
4.1 Component Tolerance

Fault tolerance for a component means that in the absence of faults it refines an ideal specification, and in the presence of faults it refines a tolerance specification, reflecting the fact that for fault tolerance it is in general necessary to have two specifications and two refinements. This can be extended to the conditional case as well, by including a rely condition. As we see in the example in Chapter 5, conditional tolerance is necessary.

The standard tolerance specifications are masking (fault effect is hidden), failsafe (safety is maintained) and nonmasking (fault effect is exposed), in fact any component can be a tolerance specification, so we begin with the general definitions.

**Definition 4.2 (Fault Tolerance)** $C$ is $TolSpec$-tolerant to faults $F$ if the following hold, where $Spec$ is the ideal specification.

- $C$ refines $Spec$.
- $C \triangleright F$ refines $TolSpec$.

**Definition 4.3 (Conditional Fault Tolerance)** $C$ is $TolSpec$-tolerant to faults $F$ with respect to rely $R_y$ and fault rely $R_y_F$ if the following hold, where $Spec$ is the ideal specification.

- $C \sqsubseteq Spec$ wrt $R_y$.
- $C \triangleright F \sqsubseteq TolSpec$ wrt $R_y_F$. 

63
Note that if $Ry$ and $Ry_F$ are arbitrary event sequences then this definition reduces to the previous one.

These general definitions can be specialized to the three standard tolerances, masking, failsafe and failsafe, which we now discuss.

**Masking Tolerance.** Masking means “in the presence of faults the program continues to behave correctly.” This kind of tolerance is easy to characterize: if $C$ is masking tolerant to faults with respect to a specification then $C$ meets the specification, and in the presence of the faults, $C$ will still meet the specification.

**Failsafe Tolerance.** Failsafe means “in the presence of faults the program behaves safely but does not necessarily make progress.” Failsafe tolerance means that the safety of the component is preserved although liveness may not be. If the specification is not safe then there are some infinite sequences of events such that all finite prefixes are traces of the specification. These finite prefixes mean that some goal (such as termination) is ultimately achieved, but that it can take indefinitely many events to get there. The safety of the specification eliminates the goal in an infinite trace but maintains the steps leading to the goal. Hence we are justified in saying that for failsafe, nothing bad ever happens, but the desired good may not happen. Given $Spec$, we write $\text{safety}(Spec)$ to indicate the safety specification.

**Nonmasking Tolerance.** Nonmasking means “in the presence of faults, the component will eventually start behaving correctly,” referred to as convergence. What “behaving correctly” means could vary: usually it means traces of $Spec$, or suffixes of those traces.

Let $\Sigma$ be the set of events of a fault-affected component Let $\hat{Spec}$ stand for the desired traces where the component in the presence of faults behaves correctly. As
the fault-affected component converges, it’s behavior before it begins to behave like \( \hat{\text{Spec}} \) will be some set of sequences. Rather than specify these sequences precisely, let \( T \subset \Sigma \) be some superset of the events of those sequences. Then we can give the nonmasking specification as \( T^*; \hat{\text{Spec}} \). \( T \) gives a bound on the events that can be engaged during convergence, and is called the fault span.

Stabilization is an extremal case in which any state value can be perturbed and in which the fault effect is finite. Since these conditions mean that any state can be reached, it means that the component in the presence of faults could potentially engage any event. Hence we give the stabilizing specification as \( \Sigma^*; \hat{\text{Spec}} \).

Now we define the standard tolerance specifications and the meaning of tolerance refinement.

**Definition 4.4 (Standard Tolerance Specifications)** Given specification \( \text{Spec} \),

- The masking tolerance specification is \( \text{Spec} \) itself.

- The failsafe tolerance specification is \( \text{safety}(\text{Spec}) \).

- The nonmasking tolerance specification is \( T^*; \hat{\text{Spec}} \), where \( T \) is some set of traces and the traces of \( \hat{\text{Spec}} \) are the set of desired eventually good behavior.

**Definition 4.5 (Tolerance Refinement)** We say that \( C \) is a masking (failsafe, stabilizing) tolerance refinement of \( \text{Spec} \) with respect to faults \( F \) if \( C \) refines \( \text{Spec} \) and \( C \upharpoonright F \) refines the masking (failsafe, stabilizing) tolerance specification.

When this definition holds, we also say that \( C \) is masking (failsafe, stabilizing) to \( C\text{Spec} \) for \( F \).
4.2 Refinement vs. Tolerance

As one might expect, the fact that a component satisfies an ideal specification does not mean that the component in the presence of faults will satisfy any particular tolerance specification. What may be less expected is that the converse is also true: in the presence of faults, the fact that a component satisfies a tolerance specification does not mean that in the absence of faults it satisfies the specification. This means that if a component is modified to be tolerant to faults, it is still necessary to verify that the modified component satisfies its specification when the faults do not occur. The following two theorems, whose proofs are by example, summarize these results.

**Theorem 4.1 (Ideal Refinement Does Not Imply Tolerance Refinement)**

*That a component refines an ideal specification does not imply that the component refines a standard tolerance specification in the presence of faults.*

![Figure 4.1: Ideal Refinement Does Not Imply Tolerance Refinement.](image)

66
Proof. The proof of this theorem is by a pair of examples shown in Fig. 4.1.

- Masking and failsafe.
  - Component $C_0$ has a hidden fault method, $F_0$. ($F$ is total, but we only show the case in which the fault changes state.) $C_0$ has a single trace: $\langle M(\ast):0 \rangle$. $C_0 \triangleright F_0$ has two traces: $\langle M(\ast):0 \rangle$ and $\langle M(\ast):1 \rangle$.
  - Let $Spec_0 = TolSpec_0 = C_0$. Since the traces of $Spec_0$ form a safe set, $Spec_0 = safety(Spec_0)$, so we can use $Spec_0$ for both masking and failsafe.
  - $C_0 \sqsubseteq Spec_0$ since refinement is reflexive.
  - $C_0 \triangleright F_0 \not\sqsubseteq Spec_0$ since the refinement behavior condition fails.

- Nonmasking.
  - Component $C_1$ has a hidden fault method, $F_1$. $C_1$ has a single trace: $\langle M(\ast):0, M(\ast):1 \rangle$. $C_1 \triangleright F_1$ has two traces: $\langle M(\ast):0, M(\ast):1 \rangle$ and $\langle M(\ast):2 \rangle$.
  - Let $Spec_1 = C_1$ and $TolSpec_1 = \Sigma^*; suffix(Spec)$.
  - $C_1 \sqsubseteq Spec_1$.
  - $C_1 \triangleright F_1 \not\sqsubseteq TolSpec_1$ since the service condition fails. \qed

Now we show that the converse is also false, even in the case of masking. The notable result here is that this holds in the case of masking.

**Theorem 4.2 (Tolerance Refinement Does Not Imply Ideal Refinement)**

That a component refines a standard tolerance specification in the presence of faults does not imply that the component refines the ideal specification.
The proof of this theorem is by a pair of examples shown in Fig. 4.2.

- Masking and failsafe.
  - Component $C_0$ has a visible fault method, $F_0$. $C_0$ has a single trace: $\langle M(*) : 0 \rangle$.
    
    $C_0 \vdash F_0$ has two traces: $\langle M(*) : 0 \rangle$ and $\langle F, M(*) : 1 \rangle$.
  
  - Let $Spec_0 = TolSpec_0 = C_0 \vdash F_0$.
  
  - $C_0 \vdash F_0 \sqsubseteq Spec_0$.
  
  - $C_0 \not\sqsubseteq Spec_0$ since the service condition fails. $C_0$ cannot engage in $F_0$ and, more importantly, it cannot engage in $M(*) : 1$.
• Nonmasking.

  ◦ Component $C_1$ has a hidden fault method, $F_1$. $C_1$ has a single trace:
    \[ \langle M(\ast):2, M(\ast):1 \rangle \] $C_1 \vdash F_1$ has two traces: $\langle M(\ast):2, M(\ast):1 \rangle$ and $\langle M(\ast):0, M(\ast):1 \rangle$. $Spec_1$ has a single trace: $\langle M(\ast):0, M(\ast):1 \rangle$.

  ◦ Let $TolSpec_1 = \Sigma^*; \text{suffix}(Spec)$.

  ◦ $C_1 \vdash F_1 \sqsubseteq TolSpec_1$.

  ◦ $C_1 \not\sqsubseteq Spec_1$ since the behavior condition fails. □

Since both theorems give a negative result, adding tolerance to an existing component means that we must check both ideal and tolerance refinement.

## 4.3 Clients of Tolerant Components

The point of a tolerant component is to for a client to be able to exploit the tolerance. A client might have been developed in terms of a certain subcomponent specification. What can we say about the tolerance of the client with some refinement of the specification if faults occur? This can be addressed in two ways.

First, suppose we only know that the behavior of the client in terms of the specification is satisfactory (that is, meets the client specification), and suppose the specification is replaced by a subcomponent that is tolerant to these faults. If the subcomponent is masking or failsafe then the client will also be masking or failsafe, respectively. If the subcomponent is nonmasking then the client may or may not be nonmasking, since the subcomponent may have converged but the client’s state may be inconsistent.

On the other hand, suppose we know not only that the behavior of the client in terms of the subcomponent specification is satisfactory but also that its behavior in
terms of the subcomponent tolerance specification is satisfactory. In this case, if the specification is replaced by a tolerant component then the client will be tolerant to the subcomponent faults.

These observations are summarized in the following theorems. If \( F \) is a \( C \)-level fault, then “\( L(C) \) is masking (failsafe) to \( F \)” means that \( L(C \triangleright F) \) refines its masking (failsafe) specification.

First we show that masking and failsafe tolerance are inherited by clients. For ease of presentation, this is presented as the following two theorems, with the second adding conditional refinement.

**Theorem 4.3 (Masking and Failsafe Inherited by Clients)** If

- \( L(C\text{Spec}) \sqsubseteq L\text{Spec} \),
- \( C \sqsubseteq C\text{Spec} \), and
- \( C \) is masking (failsafe) to \( C\text{Spec} \) for \( F \)

then \( L(C) \) is masking (failsafe) to \( F \).

**Proof.** For both cases we have to show that \( L(C) \sqsubseteq L\text{Spec} \). The result follows from monotonicity and transitivity: \( L(C) \sqsubseteq L(C\text{Spec}) \sqsubseteq L\text{Spec} \).

For the masking case, we have that \( C \triangleright F \sqsubseteq C\text{Spec} \). We need to show that \( L(C \triangleright F) \sqsubseteq L\text{Spec} \). This follows from monotonicity and transitivity: \( L(C \triangleright F) \sqsubseteq L(C\text{Spec}) \sqsubseteq L\text{Spec} \).

For the failsafe case, we have that \( C \triangleright F \sqsubseteq \text{safe}(C\text{Spec}) \). We need to show that \( L(C \triangleright F) \sqsubseteq \text{safe}(L\text{Spec}) \). By monotonicity we have that \( L(C \triangleright F) \sqsubseteq L(\text{safe}(C\text{Spec})) \). It suffices to show that \( L(\text{safe}(C\text{Spec})) \sqsubseteq \text{safe}(L\text{Spec}) \).
Observe that \textit{safety}(\textit{CSpec}) differs from \textit{CSpec} only by the possible addition of infinite traces. The same holds for \textit{safety}(\textit{LSpec}) and \textit{LSpec}. So by an operational unrolling of \textit{L(safety}(\textit{CSpec})), we can confirm that the finite traces of \textit{L(safety}(\textit{CSpec})) are the same as those of \textit{L(CSpec)}. Hence the service and behavior conditions of refinement are satisfied.

For the safety condition of refinement, consider an infinite trace \(\sigma\) of \(\textit{L(safety}(\textit{CSpec}))\) s.t. every finite prefix of \(\sigma\) is a trace of \textit{safety}(\textit{LSpec}). We must show that \(\sigma\) is a trace of \textit{safety}(\textit{LSpec}). This follows at once from the definition of \textit{safety}(\textit{LSpec}). \(\square\)

**Theorem 4.4 (Conditional Masking and Failsafe Inherited by Clients)**

Let \(R_y\) be a set of sequences of events of \(C\). If

- \(C \sqsubseteq \textit{CSpec}\) with respect to \(R_y\),
- \(C\) is masking (failsafe) to \textit{CSpec} for \(F\) with respect to \(R_y_F\),
- \(\textit{L(CSpec)} \sqsubseteq \textit{LSpec}\),
- \(\textit{L(C)}\) satisfies \(R_y\), and
- \(\textit{L(C |}\triangleleft F)\) satisfies \(R_y_F\)

then \(\textit{L(C)}\) is masking (failsafe) to \(F\).

Note that if \(R_y\) is the set of all event sequences then this reduces to the previous theorem.

**Proof.** For both cases we have to show that \(\textit{L(C)} \sqsubseteq \textit{LSpec}\). Since \(C \sqsubseteq \textit{CSpec}\) with respect to \(R_y\), by conditional monotonicity we have that \(\textit{L(C)} \sqsubseteq \textit{L(CSpec)}\). By transitivity we have that \(\textit{L(C)} \sqsubseteq \textit{LSpec}\).
For the masking case, we have that $C \triangleright F \subseteq CSpec$ with respect to $R_{yF}$. We need to show that $L(C \triangleright F) \subseteq LSpec$. By conditional monotonicity we have that $L(C \triangleright F) \subseteq L(CSpec)$. By transitivity we have that $L(C \triangleright F) \subseteq LSpec$.

For the failsafe case, we have that $C \triangleright F \subseteq safety(CSpec)$ with respect to $R_{yF}$. We need to show that $L(C \triangleright F) \subseteq safety(LSpec)$. By conditional monotonicity we have that $L(C \triangleright F) \subseteq L(safety(CSpec))$. It suffices to show that $L(safety(CSpec)) \subseteq safety(LSpec)$. The conclusion follows from the same argument in the Theorem 4.3.

In the more general case, when the tolerance is not masking or failsafe, then we can use the following results, which show us that if we have a client that is correct for a subspecification and for a tolerant subspecification then it is correct for tolerant refinements of the specification. Again, for simplicity, we present this as two theorems, the second with conditional relys added in.

**Theorem 4.5 (Tolerance from Tolerant Subcomponent Specifications)**

If

- $C \subseteq CSpec$, $C \triangleright F \subseteq CTolSpec$,
- $L(CSpec) \subseteq LSpec$, and $L(CTolSpec) \subseteq LTolSpec$

then $L(C) \subseteq LSpec$ and $L(C \triangleright F) \subseteq LTolSpec$.

**Proof.** The proof of this theorem follows from monotonicity and transitivity:

$L(C) \subseteq L(CSpec) \subseteq LSpec$. $L(C \triangleright F) \subseteq L(CTolSpec) \subseteq LTolSpec$. □

**Theorem 4.6 (Tolerance from Cond. Tolerant Subcomponent Specs)**

If
• $C \subseteq CSpec$ with respect to $R_y$, $C \triangleright F \subseteq CTolSpec$ with respect to $R_y_F$,

• $L(CSpec) \subseteq LSpec$, $L(CTolSpec) \subseteq LTolSpec$,

• $L(C)$ satisfies $R_y$, and $L(C \triangleright F)$ satisfies $R_y_F$

then $L(C) \subseteq LSpec$ and $L(C \triangleright F) \subseteq LTolSpec$.

Again, if $R_y$ and $R_y_F$ are set of all event sequences then this reduces to the previous theorem.

**Proof.** Since $C \subseteq CSpec$ with respect to $R_y$ and $L(C)$ satisfies $R_y$, by conditional monotonicity we have that $L(C) \subseteq L(CSpec)$, so by transitivity, $L(C) \subseteq LSpec$.

Likewise, since $C \triangleright F \subseteq CSpec$ with respect to $R_y_F$ and $L(C)$ satisfies $R_y_F$, by conditional monotonicity we have that $L(C \triangleright F) \subseteq L(CTolSpec)$, so by transitivity, $L(C \triangleright F) \subseteq LTolSpec$. $\square$

If we have a client that uses multiple tolerant components, each with their own convergence rely condition, we have to guarantee interference freedom between the convergence rely conditions that the subcomponents require. It must be possible for both subcomponents to converge without interfering with each other. If the client cannot simultaneously guarantee these fault-free and fault-affected rely conditions, then the behavior of the subcomponents with respect to their fault-free and fault-affected specifications cannot be guaranteed.

### 4.4 Synthesizing Tolerance

These results have implications for a theory of tolerance compilers. Tolerance compilers today produce refinements, perhaps with side effects. For example, the *MultiSpace* project [13] includes a compiler designed to mask communication faults.
The PORCH compiler [32] adds checkpointing to file i/o in such a way that a process can be restarted on another processor. These do not handle other kinds of tolerance in which what is needed is not just refinement but tolerance refinement as well. The tolerance compiler problem may be stated as follows.

- (Relational Component Synthesis) Given $CSpec$, $CTolSpec$, and a set of faults $F$, synthesize a relational component $C$ s.t. $C \sqsubseteq CSpec$ and $C \nvdash F \sqsubseteq CTolSpec$.

- (Client Synthesis) Given $CSpec$, $CTolSpec$, $LSpec$ and $LTolSpec$, synthesize $L$ s.t. $L(CSpec) \sqsubseteq LSpec$ and $L(CTolSpec) \sqsubseteq LTolSpec$.

In the first case, we want to obtain a tolerant component, given the fault-free and tolerance specifications along with the faults. In the second case, we are given the fault-free and tolerance specifications for the client as well as for a subcomponent. We want to obtain a client that satisfies both the fault-free and tolerance specifications.

There is a project underway by Kulkarni and others [17] that appears promising in terms of tolerance synthesis. The project includes synthesizing detectors and correctors for commonly-occurring kinds of tolerance. Also, the work by Nesterenko and others on atomicity refinement [27] involves composing a high-atomicity component with a low-atomicity controller to achieve a low-atomicity refinement while still preserving tolerance properties. This kind of composition can potentially be automated.
CHAPTER 5

Case Study: Resettable Vector Clock

In this case study we consider two major components: a resettable vector clock (RVC) component that is bounded space and is stabilizing tolerant; and a client (RA) that implements the Ricart-Agrawala mutual exclusion algorithm.

The purpose of the case study is to illustrate how the theory works out in a non-trivial example. Besides demonstrating that the theory has sufficient descriptive power to handle the example, it illustrates several important points. First and foremost, it shows how to use compositional reasoning in a modular way. We show that RVC refines its ideal specification and tolerance specifications and that RA, using those specifications, refines its ideal and tolerance specifications, respectively. Consequently, RA with RVC refines its ideal specification and RA with RVC in the presence of faults refines its tolerance specification. Second, it illustrates the need for two specifications for components: a tolerance specification and an ideal specification. Separate verifications are required for each. Third, the use of a rely condition for conditional refinement is critical. The ideal vector clock, RVC, depends on client access to its methods according to a particular sequence. The algorithm upon which this case study depends is due to Arora, Kulkarni and Demirbas [4].
Let us begin with a brief review of vector clocks and mutual exclusion. In the absence of global time in distributed systems, vector clocks permit us to establish a causality relationship between events. They are used in such applications as distributed debugging, checkpointing and recovery, and causal communication. Process execution consists of a sequence of events (which we will call pr-events, to avoid confusion with the events defined in terms of components; the latter we will call c-events in this discussion). Each pr-event is either local, message send or message receive. Lamport’s “happened before” relation \( [18], hb \), gives a partial order on a set of pr-events. It is the smallest transitive relation that satisfies the following.

- For any two pr-events \( e \) and \( f \), if \( e \) and \( f \) are pr-events on the same process and \( e \) occurred before \( f \), then \( e \, hb \, f \).

- For any two pr-events \( e \) and \( f \), if \( e \) is a send pr-event in one process and \( f \) is the corresponding receive pr-event in another process, then \( e \, hb \, f \).

Pr-events \( e \) and \( f \) are concurrent, denoted \( e \, co \, f \), if both \( e \, hb \, f \) and \( f \, hb \, e \) are false.

A vector clock \( vc \) for \( n \) processes is a vector of \( n \) integers, initially all 0. For a given process \( j \), \( vc(j) \) contains the current value of \( j \)'s clock. \( vc(k) \) for \( k \neq j \) contains the most current information that \( j \) has about \( k \)'s clock.

In mutual exclusion, only one process can have access to a critical section at a time. The Ricart-Agrawala mutual exclusion algorithm [30] is as follows. Whenever process \( j \) wants to enter the critical section, it sends a timestamped REQUEST message to all the processes. When a process \( k \) receives a REQUEST message from \( j \), it defers the REPLY message if \( k \) has already requested the critical section with a lower timestamp than \( j \)'s request; otherwise, \( k \) sends a REPLY message to \( j \). \( j \) enters
the critical section after it has received REPLY messages from all other processes. When $j$ exits the critical section, it sends any deferred REPLY messages.

The RVC example illustrates components in a hierarchy: RA uses RVC and Chan, a component that has one channel for each process. RVC uses VClk, a component that defines individual vector clocks, and Chan uses ChanJ, a component that defines individual channels. RA, RVC and Chan all use shared variables to represent data that is distributed across the system. This is shown in Fig. 5.1. Let srl-event refer to Send, Receive or Local-event.

Figure 5.1: Resettable Vector Clock (RVC) components.

RVC is both bounded in space and nonmasking tolerant to faults; in fact, it is stabilizing. The insight is that clients like RA do not need to compare all pr-events: they operate by phases and only need to compare pr-events within some range. (Recall that pr-events are events of the processes.) This is given by a comparison predicate, $R(p, q)$ that bounds the range of pr-events between resets.
For RA, a phase consists of requesting the critical section, entering it, and then releasing it. At each release, RA resets its vector clock and increments a phase variable. This new phase variable could itself be unbounded, defeating the purpose of bounded vector clocks. To avoid this, a communication pattern \( \text{comm}(M, l) \) is established s.t. (a) “that in any window of \( M \) distinguished pr-events of a given process \( j \), all the processes deliver a message that originated in that window, and all messages that are in transit at the beginning of the window are delivered before the end of the window” [4] and (b) between any two adjacent distinguished pr-events, the number of calls to \text{Send}, \text{Receive}, and \text{Local-event} that increment the clock is less than \( l \). This lets RVC bound the size of the phase variable.

Together, the \( p, q, M \) and \( l \) values, along with \( n \), the number of processes, parameterize RVC and give bounds on the size and number of the vector clock and phase variables. RVC can be used by a client that observes \( R(p, q) \) and \( \text{comm}(M, l) \). As shown in [4], RA observes \( R(2, 1) \) and \( \text{comm}(2, 2) \), and so is suitable as a client for RVC.

### 5.1 The Components

We begin by giving the components RA, RVC and their subcomponents. Before doing so, we adopt some assumptions and conventions that make representation less cumbersome.

Assume every component \( C \) has an \( \text{Assign}(C, C) \) method that sets the second argument to the value of the first. Also assume that every configuration of components contains Boolean \text{Bool} and integer \text{Int} components. The Boolean component contains
the methods \(\neg, \land, \lor\), with the usual meanings. The integer component contains
the methods \(<, \leq, \ldots, +, -, *, \text{Max}, \text{Min}\), with the usual meanings.

We use the following syntactic sugar.

- \(X := Y\) for variables over the domain of \(C\) stands for \(\text{C.Assign}(Y, X)\)

- Functions are special methods that are defined on every value of the last argument. For example, if \(F(C,D,E)\) is regarded as a function then it is written \(F(C,D):E\). Here, \(E\) is the return argument. A function is free from side-effects if the initial and final values of the non-return arguments are always the same. A function is Boolean if the domain of its return argument is Boolean. We permit expressions of Boolean functions that are free from side-effects, using parentheses and the Boolean operators \(\neg, \land, \lor\), with the usual order of precedence. These functions can always be linearized and given as programs, if necessary.

- \(\text{if } c \text{ then } s_1 \text{ else } s_2\) stands for
  
  \[
  \text{if } c \rightarrow s_1
  \hspace{1em} \Box
  \hspace{1em} \neg c \rightarrow s_2
  \hspace{1em} \text{fi}
  \]

  When \(\text{else}\) is omitted, it is assumed to be the null program.

- An expression of the form
  
  \[
  (\land k : 0 \leq k < n : (e_j[k] \leq f_k[k]))
  \]

  is the same as
  
  \[
  (e_j[0] \leq f_k[0]) \land \ldots \land (e_j[n] \leq f_k[n])).
  \]

- Tuple members are referenced as \(vc[j]\) for the \(j\)-th member.

- We permit local functions as illustrated below. They behave as macros.
Component RA /* Ricart-Agrawala */

/* Note: this is paramaterized by the number of nodes, n */

Shared
REQ-ts : n-tuple of VClk,
dummy : VClk, /* for dummy ‘don’t care’ arguments
deferred-set : n-tuple of Int,
REPLY : n-tuple of n-tuples of VClk,
RECVD : n-tuple of n-tuples of VClk,
CS : n-tuple of Bool,
hungry : n-tuple of Bool;

Method Request-CS(Int)
Args j;
if (¬hungry[j]) then /* If hungry, do nothing*/
VC.local-event(j, REQ-ts[j], tt);
hungry[j] := tt;
(∀ k : k ≠ j ::
VC.Send(j, dummy, REQ-ts[j], ff);
Chan.Send(k, j, 0, REQ-ts[j]) )
fi;

Method Receive-REQ(Int)
Args j;
Int.New(k);
Int.New(type);
Vclk.New(clock);
if (Chan.Avail[j]) then /* If nothing on the channel, do nothing */
Chan.Receive(j, k, type, clock);
if (type=0) then /* Receipt of Request */
VC.Receive(j, dummy, REQ-ts[k], ff)
REPLY[j][k] := REQ-ts[k]
if (hungry[j] ∧ ¬Less-than(REQ-ts[k], REQ-ts[j])) then
deferred-set[j] = union(deferred-set[j], { k })
else
VC.Send(j, dummy, REPLY[j][k], ff);
Chan.Send(k, j, 1, REPLY[j][k]);
fi;
else /* Receipt of Reply */
RECVD[j][k] := REPLY[k];
fi;
fi;

Function Enter-CS(Int) : Bool
Args j, entered;
if (hungry[j] ∧ (∀ k : k ≠ j :: REQ-ts[j] = RECVD[j][k]) then
CS\[j\] := tt;
entered := tt;
else
entered := ff;

Method Release-CS(Int) : Bool
Args j, released;
if (CS\[j\]) then /* If not in critical section, then do nothing */
(∀ k : k ∈ deferred-set\[j\] :
VC.Send(j, dummy, REPLY\[j\]\[k\], ff);
Chan.Send(k, j, 1, REPLY\[j\]\[k\]) );
deferred-set\[j\] := ∅;
hungry\[j\] := ff;
CS\[j\] := ff;
RVC.Local-reset\[j\];
released := tt;
else;
released := ff;
fi;

LocalFunction Less-than(int, VClk, int, VClk) : Bool
Args j, ej, k, fk, rslt;
rslt := VC.Happened-before(ej, ek) ∨
(Concurrent(ej, ek) ∧ (j < k));

LocalFunction Concurrent(VClk, VClk)
Args ej, ek;
¬VC.Happened-before(ej, ek) ∧ ¬VC.Happened-before(ek, ej);

Resettable Vector Clock Component

Component RVC /* Array of vector clocks */
/* Parameterized by phase-bound, and M */
Shared vc : n-tuples of VClk; init is n-tuple of init value of VClk
Method Send (Int, VClk, VClk, Bool)
Args j, ej, mj, flag;
if (flag) then VClk.Incr-clock(VCVec\[j\],j) fi;
(∀ k : 0 ≤ k < n :
mj[k][0] := vc[j][k][0];
mj[k][1] := vc[j][k][1];
ej[k][0] := vc[j][k][0];
ej[k][1] := vc[j][k][1]; )

Method Receive (Int, VClk, VClk, Bool)
Args j, ej, ml, flag;
(∀ k : k ≠ j ::
if ( (vc[j][k][0] < ml[k][0] \( \land \) vc[j][k][0] + M + 1 > ml[k][0] )
\( \lor \) (vc[j][k][0] > ml[k][0] \( \land \)
vc[j][k][0] ≥ ml[k][0] + phase-bound - M )) )
then vc[j][k][0] := ml[k][0];
vc[j][k][1] := ml[k][1];
elseif (vc[j][k][0] = ml[k][0])
then vc[j][k][1] := max(vc[j][k][1],ml[k][1])
/* Otherwise don’t update */
fi;
if (flag) then VClk.Incr-clock(vc[j],j) fi;
ej := vc[j];
Method Local-event (Int, VClk, Bool)
   Args j, ej, flag;
   if (flag) then VClk.Incr-clock(vc[j],j) fi;
(\( \forall \) k : 0 ≤ k < n ::
ej[k] := vc[j][k][1]; )
Function Happened-before(VClk, Vclk) : Bool
   Args ej, fk, ret
   ret := VClk.le(ej,fk)
Method Local-reset(Int)
   Args j
   vc[j][j][0] := (vc[j][j][0] + 1) mod phase-bound;
   vc[j][j][1] := 0;
Local-method Global-reset()
(\( \forall \) j, k : 0 ≤ j < n \( \land \) 0 ≤ k < n ::
vc[j][k][0] := 0;
vc[j][k][1] := 0; )
Hidden Environment Method F1(VClk)
   Args clock;
   (\( \forall \) i : 0 ≤ k < n ::
clock[k][0] := ?;
clock[k][1] := ?; )
Hidden Environment Method F2
(\( \forall \) j, k : 0 ≤ j < n \( \land \) 0 ≤ k < n ::
vc[j][k][0] := ?;
vc[j][k][1] := ?; )

Component VClk /* Individual vector clocks */
/* paramaterized by phase-bound */
Domain is n-tuples of pairs of bounded Int; init is all zero values
/* 0th value is phase, 1st is clock */
Method Incr-clock(self, Int)
Args VClk, J;
VClk[J][1] := VClk[J][1] + 1 mod phase-bound;

Function \text{le}(\text{self}, \text{self}) : \text{Bool}
Args ej, fk, ret;
ret := (\land i : 0 \leq i < n : (ej[i] \leq fk[i]));

Channel Component

Component Chan /* Send-receive with n channels */
Global ChanEnt : n-tuple of ChanJ
Method Send (int, int, int, VClk)
Args k, j, type, msg;
ChanJ.Add(ChanEnt[k], j, type, msg);
Function Avail (int) : Bool
Args j, rslt;
rslt := ChanJ.Ready(ChanEnt[j]);
Method Receive (int, int, int, VClk)
Args j, k, type, msg;
ChanJ.Remove(ChanEnt[j], k, type, msg);

Component ChanJ (Relational) /* Define non-fifo channel for J */
Domain : \{()\} \cup \text{ChanEnt} \cup \text{ChanEnt}^2 \cup \text{ChanEnt}^3 \ldots
Method Add (\text{self}, \text{int}, \text{int}, \text{VClk})
\{((c',f',y',m'),(c,f,y,m)) : 
  c = c'; (f',y',m'), f = f', y = y' and m = m' \}
Function Ready (\text{self}) : \text{Bool}
\{((c',r'),(c,r)) : 
  c = c' and if c' = () then r = \text{ff} else r = \text{tt} \}
Method Remove (\text{self}, \text{int}, \text{VClk})
/* Remove any item from the channel */
\{((c',f',y',m'),(c,f,y,m)) : 
  c' = c_0; (f,y,m); c_1 and c = c_0; c_1 \}
5.2 Modular Reasoning for Resettable Vector Clock

Now we show how modular reasoning works for both the ideal and the tolerant components, and their respective specifications.

For the ideal case, we give the specifications and show that \( RVC \) refines \( RVC\text{-Spec} \) wrt a rely condition, that \( RA(RVC\text{-Spec}) \) refines \( RA\text{-Spec} \), and that \( RA(RVC) \) satisfies the rely condition. Hence we may conclude that \( RA(RVC) \) refines \( RA\text{-Spec} \).

We do the same thing for the tolerant case. We give the specifications and show that \( RVC \) refines \( RVC\text{-TolSpec} \) wrt a rely condition, that \( RA(RVC\text{-TolSpec}) \) refines \( RA\text{-TolSpec} \), and that \( RA(RVC \) refines \( RA\text{-TolSpec} \). We begin with the ideal case.

- \( RVC \) refines \( RVC\text{-Spec} \) wrt a rely condition, \( RVC\text{-Ry} \).
  - \( RVC\text{-Spec} \)
    - \( RVC\text{-Spec} \) is the same as \( VC\text{-Spec} \).
  - Need for \( RVC\text{-Ry} \).

\( RVC \) was designed with the assumption that a well-formed client will satisfy the comparison predicate \( R(p, q) \) for \( p, q \in Z^+ \) and the communication predicate \( comm(M, l) \) for \( M, l \in Z^+ \). Fix some values for \( p, q, M \) and \( l \). Let \( RVC\text{-Ry} \) be the set of all c-event sequences that satisfy the predicates. Then \( RVC \) refines \( RVC\text{-Spec} \) wrt \( RVC\text{-Ry} \).
• RA(RVC-Spec) refines RA-Spec.

RA-Spec is the same as RA-Spec: mutual exclusion. RA is identical with RA except that it includes the local-reset call required by RA. Hence RA(RVC-Spec) refines RA-Spec.

• RA(VC) respects RVC-Ry.

According to [4], RA(VC) respects RVC-Ry if RVC-Ry was defined with the values $p = 3, q = 2, M = 2, l = 2$: $l = 2$ since between resets for process $j$ there is only one event whose timestamp is updated; $M = 2$ since the processes communicate each other sufficiently often and deliver messages fast enough (bounded by $M$ resets for each); $p = 3$ since at most three resets intervene between earlier events compared by $j$ with the happened-before method, and at most two resets intervene between later events.

Now we make use of modular reasoning. The facts above satisfy the conditions for Theorem 2.3, monotonicity of conditional client substitution, so we have that RA(RVC) refines RA-TolSpec. Next we handle the tolerant case.

• RVC+F refines RVC-TolSpec wrt a rely condition, RVC-TolRy.

  o RVC-TolSpec

In the presence of faults that arbitrarily change the vector clock values, it suffices that eventually RVC accurately report causality. Let $\text{suffixes}(RVC)$ be the suffixes of traces of RVC. Then $\text{VC-TolSpec} = \Sigma^*; \text{suffixes}(RVC)$.

  o Need for RVC-TolRy.
The fault methods for RVC are F1 and F2; collectively we can refer to them as F. We know from [4] that RVC\(\downarrow F\) stabilizes under two assumptions. The first is that the client will eventually start to re-satisfy the ideal rely, RVC-Ry. The second is that the client performs a reset sufficiently often. Hence we can let RVC-TolRy be the set of all c-event sequences \(\sigma\) such that \(\sigma\) contains reset c-events at a sufficient frequency, and \(\sigma\) has a suffix that is a sequence in RVC-Ry. Then RVC\(\downarrow F\) refines RVC-TolSpec wrt RVC-TolRy.

- RA(RVC-TolSpec) refines RA-TolSpec.

As with the tolerance specification for RVC, let RA-TolSpec = \(\Sigma^*;\text{suffixes}(RA-Spec)\). As with RA, each method of RA is total, so the service condition of refinement is met. For behavior, from [4] we know that RA(RVC-TolSpec) will stabilize if each trace of RVC-TolSpec eventually behaves like RVC-Spec. Since this is true by definition, the behavior condition is satisfied.

For the safety condition, let \(\sigma\) be an infinite trace of RA(RVC-TolSpec) and suppose every finite prefix of \(\sigma\) is a trace of RA-TolSpec. We need to show that \(\sigma\) is a trace of RA-TolSpec. We reason directly from what we know about the stabilization of RA and RVC. We know that the traces of RVC-TolSpec have suffixes that are suffixes of traces of RVC-Spec. Since traces of RVC-TolSpec have suffixes that are suffixes of traces of RVC-Spec, and since according to [4], traces of RA(RVC-Spec have suffixes that are suffixes of traces of RA-Spec (that is, RA(RVC) stabilizes if RVC stabilizes), traces of RA(RVC-TolSpec) have suffixes that are suffixes of traces of RA(RVC-TolSpec). Since RA-TolSpec = \(\Sigma^*;\text{suffixes}(RA-Spec)\), \(\sigma\) is a trace of RA-TolSpec.
• RA(VC\$F) respects RVC-TolRy.

The fact that RA(RVC\$F) respects RVC-TolRy is given in [4]: in the presence of faults, RA(RVC) will eventually satisfy the ideal RVC-Ry, and RA performs a reset sufficiently often.

Again we make use of modular reasoning. The facts above satisfy the conditions for Theorem 2.3, motonicity of conditional client substitution, so we have that RA(RVC) refines RA-TolSpec.

Note that we had to carry out the reasoning with respect to ideal specifications in the absence of faults and for tolerance specifications in the presence of faults. It did not suffice to work with a single specification and show a single refinement.
CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

Our goal was to give a fault-tolerant data abstraction that can be used in modular design and reasoning for distributed systems. We have done so in terms of a rich definition of component that models distribution and fault-tolerance.

Fault tolerance is modeled as refinement of a tolerance specification, based on component semantics. This permits a single relation to be used for refinement of both ideal specifications and tolerance specifications. Composition of a client program component with a configuration of relational components lets us give the behavior of the client in the configuration, and so lets us study the effect of subcomponent faults on a client.

By maintaining a unified approach and combining component composition (a client component with a configuration) with refinement, we are able to reason in a modular way about components and clients of those components in terms of ideal and tolerant behaviors.

We additionally gave simulation relations that can be used to verify refinement. We showed that the state-level simulations are complete relative to finite invisible
nondeterminism (fin) but that value-level simulations are incomplete unless a monadic restriction is imposed.

6.2 Discussion

Now we take up a number of questions concerning this work.

_Is the definition of component adequate?_

The components we specify define at most a single data type. It would be a straightforward extension for components to define multiple data types. The important distinction in component behavior concerns visibility. Multiple data types would simply mean variables of multiple types in states, but would not affect the issue of visibility.

_Is the definition of fault affect adequate?_

A fault method can modify client instances of the component’s domain (self) as well as shared variables of the component. The idea of fault could be extended. One way would be a special method that, when invoked, changes the self variables meet a certain condition. Another related way would be to give a fault as a binary relation over state. This fault would change those states that are in some particular configuration. Neither of these extensions would affect the basic setup.

Faults are viewed as entirely optional. This means that a fault-affected component will have behaviors that are fault-free. Sometimes, however, we want to place a component in a specific environment; for example to model the behavior of an object in space instead of on the earth. This can be modeled by using a shared variable to indicate mode, and letting the component methods return results depending on the mode.
How are representation invariants affected by faults?

A representation invariant is a value-level relation from a concrete component to an abstract one that relates a value in the concrete to one or more values in the abstract. Representation invariants are used in verification, and the forward and backward value-level simulation relations are examples. In general, a representation invariant does not need to hold for all concrete values but only for some superset of the ideal reachable ones.

Assume that such a representation invariant has been established for an ideal component. A question then arises. In the presence of faults, the component’s values may go outside the ideal reachable set. What representation should be assigned to these faulty values?

From the state-level point of view this is not an issue, since state-level simulations are complete. However, value-level verification is usually less complex and hence more practical. Accommodating the notion of a representation of a faulty concrete value in the abstract would lead to simpler verification. It is not clear whether such an extension to faulty values can be done in all cases, even if there is a representation invariant for the ideal values. This is a topic for future research.

Can programs be clients of components?

Yes. Given a configuration of components, we do not specify a “top level,” or ultimate environment for them. This can be provided by any state-based program.

Method invocations are atomic. Can atomicity be refined?

Though method invocations are atomic, we may want to model a situation where they are not. For example, we may want a client update method to send a message to a database component rather than execute immediately. This can be handled by
introducing a channel component between the client and the database. Note, however, that this refinement does not preserve the interface of the components. The client may have to be changed to include a method that receives a confirmation response back from the database.

*A client L of A cannot reference A in its methods. Is this a problem?*

A case in which one might wish to do this would be a client L with a configuration that contains a channel component, Ch, that does not impose any particular ordering. The client may wish to pass the channel to a subcomponent that will calculate and return some value based on the channel’s contents. In this case, the client would include reference to the channel component in a method. If the channel were implemented by, say, a queue component Q, then we could not verify that \( L(Q) \sqsubseteq L(Ch) \). There may be a technical solution to the problem, and it should be studied.

### 6.3 Future Work

Besides the technical issues mentioned above, part of the future work will seek to address developer issues. While the use of components can help simplify the design and verification of large, tolerant systems, development remains a complex process. There have been a number of efforts related to program tolerance that simplify the design of tolerant systems. These techniques will be adapted as appropriate to use with components. Some of the techniques are tools to help the developer make a component tolerant (component frameworks); some are ways to refine a specification so that tolerance is preserved (model-based stabilization); some are ways to
increase atomicity while preserving tolerance (tolerance-preserving atomicity refinement); some are ways to synthesize refinements (tolerance compilers); some are ways to simplify reasoning (Seuss).

In addition, extending this setup to accommodate hybrid components would make the results more applicable to embedded systems.

An important issue that touches several of the areas that follow is that of fault refinement and tolerance preservation for components. We would like to be able to refine both a component and the faults that affect it. The faults that affect an abstract component will in general be different than the faults that affect a concrete component. For example, recall the bag component \( B_2 \) from Chapter 2.3 that is implemented by the stack component \( S_2 \). Suppose the bag has a fault \( F_b \) that deletes an item from a bag. The traces of the bag in the presence of the faults reflect the fact that items extracted are at most those that were added.

A parallel fault \( F_s \) on the stack deletes an item from a stack (and shortens the stack appropriately). The traces of the stack in the presence of its faults reflect the fact that items extracted are at most those that were added, and form a reverse subsequence of those added items.

Since \( S_2 \) refines \( B_2 \) and \( S_2 |\triangleright F_s \) refines \( B_2 |\triangleright F_b \), we can intuitively say that the stack faults refine the bag faults. It is unclear, however, whether refinement is the correct relation for modeling fault refinement; its applicability in the projects mentioned below would be the test.

Given a suitable definition of fault refinement, we could define tolerance preservation along the following lines. Let \( C \) and \( A \) be components with faults \( F_C \) and
$F_A$, respectively s.t. $F_C$ is a fault refinement of $F_A$. Then $C$ would be said to be a tolerance-preserving refinement of $A$ wrt $F_C$ and $F_A$ if

- $A$ is masking (failsafe, nonmasking) tolerant to faults $F_A$,
- $C \subseteq A$, and
- $C$ is masking (failsafe, nonmasking) tolerant to faults $F_C$.

### 6.3.1 Component Frameworks

This would capitalize on work now underway by Kulkarni and others [17]. The work is focused towards adding tolerance to programs in the following ways. It seeks to establish templates whose instantiation provides detectors and correctors that occur commonly in fault-tolerant programs, and includes heuristics to help a developer of a fault-tolerant program choose an appropriate template based on the problem at hand. The project also addresses when detectors and correctors can be used in automatically transforming a fault-intolerant program into a fault-tolerant one; where the complexity of the transformation makes this unreasonable, it would provide heuristics to attempt to reduce the complexity, besides making automatic use of pre-computed templates.

This work could be adapted to our setup by similarly developing templates of detectors and correctors that occur frequently in tolerant components, and by seeking automatic transformations of intolerant components into tolerant ones.

### 6.3.2 Model-Based Stabilization

Arora, Demirbas and Kulkarni [3] have looked into the model-based stabilization problem. There are two questions, both dealing with wrappers. A wrapper for a
specification is an enhancement of the specification s.t. in the absence of faults the wrapped specification behaves like the original specification, and in the presence of faults is stabilizing. The first question is whether a stabilization wrapper can be designed for a specification in such a way that the same wrapper, when applied to any implementation of the specification, will yield a stabilizing implementation. The second question is whether, given a wrapper for a specification, the specification $S$ and the wrapper $W$ can be separately implemented as $S'$ and $W'$ in such a way that $W'$ is a wrapper for $S'$.

The idea behind this approach is that this can be done provided the concrete component refines the specification not just from initial states, but from every state. Hence even in faulty states the concrete will behave like the specification.

The technical setup specified in model-based stabilization is significantly different than ours: for instance, the problem is defined for systems with processes rather than components, and faults are assumed to apply directly to both the specification and the concrete implementations. Given an understanding of fault refinement, the ideas are promising as a way to extend our work, and may contribute to the study of tolerance compilers.

### 6.3.3 Tolerance-Preserving Atomicity Refinement

Nesterenko and Arora have investigated atomicity refinement [26]. In this approach, a program with high atomicity can be refined to one with low atomicity, without compromising dependability properties. The approach is illustrated with a high atomicity program that atomically updates its state based on that of its neighbors. The high-atomicity program is composed with a stabilizing implementation of
the dining philosophers problem to yield a low-atomicity program that is also stabilizing.

Applied to our setup, we would seek to develop templates of tolerance-preserving atomicity control components (like the implementation of dining philosophers) that can be composed with a suitable high-atomicity component to yield a low atomicity component (or components) that preserves tolerance.

### 6.3.4 Tolerance Compilers

There are few examples of tolerance compilers (ambiguously called “tolerant compilers”). Those that exist either produce masking refinements (to handle unreliable communication media, for example) [13] or else produce additional data such as checkpoints that can be used by other parts of the system to give tolerance [32].

The problem of tolerance compilers was described in Chapter 4.3, where synthesis of tolerant components and tolerant clients is discussed. Such a compiler could be used to approach the problem of adding tolerance to a multilevel system.

The problem as stated is general: any tolerant compiler should be an instance of this problem. However, a given tolerance compiler will typically be developed for certain platforms, be restricted to certain kinds of faults and tolerance classes, and so forth.

Additionally, the problem is stated semantically. In fact, what a compiler receives is syntax, and it has to produce is a refinement of the meaning of that syntax. For tolerance compilers there are issues relating to the degree in which a syntactic program would have to annotated with semantics by the program author in order to accomplish its task.
6.3.5 Seuss As a Platform

The Seuss methodology, mentioned earlier in Chapter 1.6.1, offers a potential platform for using the ideas contained in this study. Of special appeal is the fact that a Seuss program can be regarded by the developer as sequential when its execution in fact is concurrent.

A number of issues would have to be resolved. The semantics of a Seuss program would have to be adjusted to accommodate our distinction of state and event, as well as to distinguish between a computation (for which there may be no trace) and a scenario (which has a trace). Assuming a suitable definition of refinement, we would have to assure ourselves that program substitution is monotonic. Finally, we would have to verify that arbitrary behavioral specifications could be given as Seuss programs so that client behavior in terms of such a specification can be calculated.

6.3.6 Extending to Hybrid Components

Increasingly systems are being embedded in order to control and respond to physical systems. These systems often exhibit dynamic behaviors that are modeled using ordinary or partial differential equations, sometimes including discrete events (as in our setup here). Hybrid systems [8] are those that combine the two, so that a “location”, similar to a state, is a combination of a dynamic and a discrete value.

For modeling the tolerance of such embedded systems, it would be useful to extend the work here to include hybrid systems. An important aspect of this extension would have to do with nonmasking control: how to drive such a system to an equilibrium state.
BIBLIOGRAPHY


APPENDIX A

Notation

We make use of the following notational conventions.

• Given a binary relation $r \subseteq A \times B$, then $r[a] = \{b : (a, b) \in r\}$. Relation $r$ is defined for $a$ provided $r[a] \neq \emptyset$.

• The semicolon “;” is overloaded.
  o For tuples, $\langle a, b \rangle; \langle c, d \rangle = \langle a, b, c, d \rangle$.
  o For scenarios, $\sigma; e$ is the scenario consisting of $\sigma$ followed by $e$.
  o In programs, “;” is sequential composition.

• For a function $f : X \to Y$ and a tuple $\overline{x} = \langle x_0, \ldots, x_n \rangle \in x^n$, $f(\overline{x})$ is the point-wise application $\langle f(x_0), \ldots, f(x_n) \rangle$. There is a similar extension for relations.

• Similarly, for $X \subseteq Y$, $f(X) = \{y : \exists x \in X \text{ and } y = f(x)\}$. There is a similar extension for relations.

We make use of the following standard symbols.

• $A, B, C$ for components.

• $L$ for clients.
• $F$ for faults.

• $T$ for nonmasking fault span.
APPENDIX B

Proofs

Proof of Theorem 2.1: Refinement is a Preorder.

• The proof that $C \sqsubseteq C$ is immediate.

• Let $C \sqsubseteq B$ and $B \sqsubseteq A$. Show that $C \sqsubseteq A$.
  
  ◦ Let $\sigma; v$ be an extension of $A$.

    − By applying the definition of component refinement twice, it is an extension of $B$ and consequently of $A$, proving service.

    − Let $\sigma; (v, r)$ be a trace of $C$. Again, it is a trace of $B$ and consequently of $A$, proving behavior.

  ◦ Let $\sigma$ be an infinite trace of $C$ s.t. every finite prefix of $\sigma$ is a trace of $B$.

    Then by the definition, $\sigma$ is an infinite trace of $C$. Suppose every finite prefix of $\sigma$ is a trace of $C$. Then again by the definition, $\sigma$ is a trace of $C$, proving liveness.

  $\Box$
Proof of Theorem 2.2: Client Substitution is Monotonic.

Let $C$ and $A$ be components s.t. $C$ refines $A$.

Let $L$ be a client of $A$ that does not have $A$ as an indicator.

Let $L(C)$ be wrt a configuration containing $C$ as top-level.

Let $L(A)$ be wrt the same configuration but containing $A$ as top-level.
Figure B.2: Hidden C and non-C method calls.

- Let $\sigma'$ be a finite trace of $L(C)$ and $L(A)$, and let $\sigma';v$ be an extension of $L(A)$.
  - Prove refinement service. $\sigma';v$ is an extension of $L(C)$.
  - Prove refinement behavior: every trace $\sigma';(v,r)$ of $L(C)$ is a trace of $L(A)$.
- Let $\sigma$ be an infinite trace of $L(C)$ s.t. every finite prefix is a trace of $L(A)$.
  - Prove refinement safety. $\sigma$ is a trace of $L(A)$. 

104
We begin with a lemma that is used in the proof.

**Lemma.** If $\sigma'$ is a trace of $L(C)$ then every unfolding trace $\alpha'$ of $C$ from $\sigma'$ in $L(C)$ is an unfolding trace of $A$ from $\sigma'$ in $L(A)$.

The lemma makes use of Fig. B.1. An unfolding of $\sigma'$ in $L(C)$ involves calls to non-$C$ methods, to hidden methods of $C$, and to visible methods of $C$. The transition from $ls0$ to $ls0'$ is via non-$C$ methods and hidden $C$ methods. These methods are
independent in their execution: since \( C \) is top-level in the configuration, calls to non-
\( C \) methods do not change \( C \) variables; and since by definition hidden method of \( C \)
do not have non-\( C \) arguments, calls to methods of \( C \) do not change non-\( C \) variables.
Hence \( ls0 \) and \( lu0 \) agree on non-\( C \) variables, as to \( ls1 \) and \( lu1 \), and so forth, which
permits us to construct an unfolding of \( \sigma' \) in \( L(A) \) by using the same sequences of
non-\( C \) method calls. This is illustrated in Fig. B.2.

**Proof of Lemma.**

- Let \( \sigma' \) be a trace of \( L(C) \).
- There is an unfolding of \( \sigma' \) in \( L(C) \) that is a sequence of visible method calls,
  possibly interleaved with sequences of calls to non-\( C \) methods or to hidden
  methods of \( C \).
- From this we may construct a scenario computation \( \alpha' \) in \( C \) s.t. \( \widehat{\alpha}'_i = (v_i : r_i) \)
  and \( \alpha' = cs_0 \xrightarrow{\alpha_1} cs_1 \ldots cs_{n-1} \xrightarrow{\alpha_n} cs_n \).
- Since \( C \) refines \( A \), \( \text{trace}(\alpha') \) is a trace of \( A \).
- Hence we may construct an unfolding of \( \sigma' \) in \( L(A) \), so \( \sigma' \) is a trace of \( L(A) \). \( \square \)

This ends the proof of the lemma. Now let us prove the theorem itself. This is
done in conjunction with Fig. B.3.

- Let \( \sigma' \) be a trace of \( L(C) \) and of \( L(A) \), and let \( \sigma'; v \) be an extension of \( L(A) \).
  - To show service, we will find an \( r \) s.t. \( \sigma'; < vr > \) is a trace of \( L(C) \).
  - To show behavior, we will show that for any such \( r \), \( \sigma'; < vr > \) is a trace of
    \( L(A) \).
Let $\alpha'$ be an unfolding trace of $C$ from $\sigma'$ in $L(C)$. By the lemma, $\alpha'$ is an unfolding trace of $A$ from $\sigma'$ in $L(A)$.

Now consider the unfolding for $\sigma'; v$ in $L(A)$.

- Let $e_0 = (v_0;r_{0,0})$ be the first visible event of $A$ in the unfolding of $\sigma'; v$ after $\alpha'$.
- Since $C$ refines $A$, $\alpha'; v_0$ is an extension of $C$. Hence for some result, say $r_{0,1}$, $\alpha'; (v_0;r_{0,0})$ is a trace of $C$ as well as a trace of $A$.

Continuing in this fashion we can find $\beta$ s.t. $\alpha'; \beta$ is an unfolding trace of $A$ from $\sigma'; <v:r>$ in $L(C)$ for some $r$.

From $\alpha'; \beta$ and the unfolding of $\sigma'; <v:r>$ in $L(A)$ we may construct an unfolding of $\sigma'; <v:r>$ in $L(C)$ s.t. $\alpha'; \beta$ is an unfolding trace and hence $\sigma'; <v:r>$ is a trace of $L(C)$.

So $\sigma'; v$ is an extension of $L(C)$, satisfying refinement service. To show refinement behavior, let $\sigma'; <v:r>$ be a trace of $L(C)$. Reversing the construction of Fig. B.3, each $\alpha'; \beta$ that is an unfolding trace of $C$ from $\sigma'; <v:r>$ is a trace of $A$ since $C$ refines $A$. Hence we can construct an unfolding of $\sigma'; <v:r>$ in $L(A)$ s.t. $\alpha'; \beta$ is an unfolding trace.

Finally we prove refinement safety. Let $\sigma$ be an infinite trace of $L(C)$ s.t. every finite prefix of $\sigma$ is a trace of $L(A)$. Show that $\sigma$ is an infinite trace of $L(A)$. From the constructions given, there is an unfolding trace $\alpha$ of $C$ from $\sigma$ in $L(C)$. Since $C$ refines $A$, we may use $\alpha$ to construct $\sigma$ as a trace of $L(A)$. 

\[\boxed{}\]
Proof of Theorem 3.1: Forward State-Level Simulations Are Sound.

Let $f$ be a state-level forward simulation from $C$ to $A$. Show that $C \sqsubseteq A$.

![Figure B.4: Hidden events in simulation.](image)

- Proof of refinement service and behavior.

  Assume $\sigma'; v$ is an extension of $A$ and $\sigma'$ is a trace of $C$. The proof is by induction on the length of $\sigma'$.

  - Base case. Let $\sigma' = \lambda$.
    
    - Proof of refinement service. Let $v$ be an extension of $A$. Note that $v$ is visible. Show that $v$ is an extension of $C$.
      
      - Let $s'$ be an initial state of $C$.
        
        - By (SF Init) there is $u' \in f[s']$ s.t. $u'$ is an initial state of $A$.
          
          - By the definition of extension, $v$ is enabled in $A$ on $u'$.
            
            - By (SF Service), $v$ is enabled in $C$ on $s'$.
              
              - Since this holds for all initial $s'$ of $C$, by the definition of extension, $v$ is an extension of $C$. 

- Proof of refinement behavior. We also give an additional reachability result that is necessary for the inductive step. Let $e = <v:r>$ be a trace of $C$ and $v$ an extension of $A$. Show (a) $e$ is a trace of $A$ and (b) For all states $s$ reachable by $e$ in $C$ there is $u \in f[s]$ that is reachable by $e$ in $A$.

Proof of (a), behavior.

▷ By the definition of trace there is an initial $s'$ s.t. $s' \stackrel{e}{\rightarrow}_C s$.

▷ By (SF Init) there is $u' \in f[s']$.

▷ By (SF Visible Behavior) there is $\hat{\alpha} = e$ s.t. $u' \stackrel{\alpha}{\Rightarrow}_A u$.

▷ Hence $e$ is a trace of $A$.

Proof of (b), reachability.

▷ The set of states reachable by $e$ in $C$ are the set of all $s$ s.t. for some $\hat{\alpha} = e$, $s' \stackrel{\hat{\alpha}}{\Rightarrow}_C s$ where $s'$ is an initial value of $C$.

▷ Let $\hat{\alpha} = e$ and let $s' \stackrel{\alpha}{\Rightarrow}_C s$ for initial $s'$.

▷ $\alpha$ is of the form $\langle g_1, \ldots, g_n, e \rangle$, where the $g_i$ are hidden. See Fig. B.4.

▷ By (SF Init) there is $u' \in f[s']$ s.t. $u'$ is initial in $A$.

▷ By (SF Hidden Behavior), $u' \in f[s_1]$ and hence $u' \in f[s_2]$, and so on, so $u' \in f[s_n]$.

▷ By (SF Visible Behavior), there is $u \in f[s]$ s.t. $u' \stackrel{e}{\rightarrow}_A u$.

▷ Hence $u$ is reachable by $e$ in $A$.

- Inductive step. Let $\sigma'$ be a trace of $A$ and $C$.

- Proof of refinement service. Let $\sigma'; v$ be an extension of $A$. Note that $v$ is visible. Show that $\sigma'; v$ is an extension of $C$. 

109
Let $s'$ be a state reachable by $\sigma'$ in $C$. This exists by the inductive hypothesis since $\sigma'$ is a trace of $C$.

By the inductive hypothesis, there is $u' \in f[s']$ that is reachable by $\sigma'$ in $A$.

By the definition of extension, $v$ is enabled in $A$ on $u'$.

By (SF Service), $v$ is enabled in $C$ on $s'$.

Since this holds for all states reachable by $\sigma'$ in $C$, by the definition of extension, $v$ is an extension of $C$.

Proof of refinement behavior. Let $\sigma'; e$ be a trace of $C$ where $e = < vr >$ and $\sigma'; v$ an extension of $A$. Show (a) $\sigma'; e$ is a trace of $A$ and (b) For all states $s$ reachable by $\sigma'; e$ in $C$ there is $u \in f[s]$ that is reachable by $\sigma'; e$ in $A$.

Proof of (a), behavior.

By the definition of trace there is a $s'$ reachable by $\sigma'$ in $C$ s.t. $s' \xrightarrow{e} C s$.

By the inductive hypothesis there is $u' \in f[s']$.

By (SF Visible Behavior) there is $\hat{\alpha} = e$ s.t. $u' \xrightarrow{\hat{\alpha}} A u$.

Hence $e$ is a trace of $A$.

Proof of (b), reachability.

The set of states reachable by $\sigma'; e$ in $C$ are the set of all $s$ s.t. for some $\hat{\alpha} = e$, $s' \xrightarrow{\hat{\alpha}} C s$ where $s'$ is reachable by $\sigma'$ in $C$.

Let $\hat{\alpha} = e$ and let $s' \xrightarrow{\alpha} C s$ for $s'$ reachable by $\sigma'$ in $C$.

$\alpha$ is of the form $(g_1, \ldots, g_n, e)$, where the $g_i$ are hidden. See Fig. B.4.
By the inductive hypothesis there is $u' \in f[s']$ s.t. $u'$ is initial in $A$.

By (SF Hidden Behavior), $u' \in f[s_1]$ and hence $u' \in f[s_2]$, and so on, so $u' \in f[s_n]$.

By (SF Visible Behavior), there is $u \in f[s]$ s.t. $u \xrightarrow{e_{i}} u'$.

Hence $u$ is reachable by $\sigma'; e$ in $A$.

- Corollary: If $s$ is reachable by $\sigma$ in $C$ then there is $u \in f[s]$ that is reachable by $\sigma$ in $A$.

- Proof of refinement safety.

Let $\sigma$ be an infinite trace of $C$ and let every finite prefix of $\sigma$ be a trace of $A$.

Show that $\sigma$ is a trace of $A$.

- Let $\alpha_C = s_0 \xrightarrow{e_1} s_1 \ldots s_{i-1} \xrightarrow{e_i} s_i$ be an infinite computation of $C$ corresponding to $\sigma$.

- We may construct $\alpha_A = u_0 \xrightarrow{e_1} u_1 \ldots u_{i-1} \xrightarrow{e_i} u_i$ s.t. for all $i$, $u_i \in f[s_i]$. This exists since each $s_i$ is reachable by $\sigma_i = \langle e_1, \ldots, e_i \rangle$ in $C$, so by the corollary there is $u_i \in f[s_i]$ reachable by $\sigma_i$ in $A$.

- $\alpha_A$ is a scenario computation, so $\text{trace}(\alpha) = \sigma$ is a trace of $A$. 

\[\square\]
Proof of Theorem 3.2: Backward State-Level Simulations Are Sound.

Let $b$ be an image-finite backward simulation from $C$ to $A$. We must show that $C \subseteq A$.

- Proof of refinement service and behavior.

Assume $\sigma' ; v$ is an extension of $A$ and $\sigma'$ is a trace of $C$. The proof is by induction on the length of $\sigma'$.

  ◦ Base case. Let $\sigma' = \lambda$.

    - Proof of refinement service. Let $v$ be an extension of $A$. Note that $v$ is visible. Show that $v$ is an extension of $C$.

      ▷ Let $s'$ be an initial state of $C$.

      ▷ By (SB Init) each $u' \in b[s']$ is an initial state of $A$.

      ▷ By the definition of extension, $v$ is enabled in $A$ on $u'$.

      ▷ By (SB Service), $v$ is enabled in $C$ on $s'$.

      ▷ Since this holds for all initial $s'$ of $C$, by the definition of extension, $v$ is an extension of $C$.

    - Proof of refinement behavior. As with forward simulation, we also give an additional reachability result that is necessary for the inductive step. Let $e <<= v: r$ be a trace of $C$ and $v$ an extension of $A$. Show (a) $e$ is a trace of $A$ and (b) For all states $s$ reachable by $e$ in $C$, each $u \in b[s]$ is reachable by $e$ in $A$.

      Proof of (a), behavior.

      ▷ By the definition of trace there is an initial $s'$ s.t. $s' \xrightarrow{e} C s$. 

112
By totality of $b$ there is $u \in b[s]$.

By (SB Vis Behavior) there is $\hat{\alpha} = e$ s.t. $u' \xrightarrow{\hat{\alpha}}_A u$.

Hence $e$ is a trace of $A$.

Proof of (b), reachability.

The set of states reachable by $e$ in $C$ are the set of all $s$ s.t. for some

$\hat{\alpha} = e$, $s' \xrightarrow{\hat{\alpha}}_C s$ where $s'$ is an initial value of $C$.

Let $\hat{\alpha} = e$ and let $s' \xrightarrow{\hat{\alpha}}_C s$ for initial $s'$.

$\alpha$ is of the form $(g_1, \ldots, g_n, e)$, where the $g_i$ are hidden. See Fig. B.4.

Let $u \in b[s]$. This exists since $b$ is total.

By (SB Visible Behavior), there is $u' \in b[s_n]$ s.t. $u' \xrightarrow{e}_A u$.

By (SB Hidden Behavior), $u' \in b'[s']$.

By (SB Init), $u'$ is initial in $A$.

Hence $u$ is reachable by $e$ in $A$.

Inductive step. Let $\sigma'$ be a trace of $A$ and $C$.

Proof of refinement service. Let $\sigma'; v$ be an extension of $A$. Note that $v$ is visible. Show that $\sigma'; v$ is an extension of $C$.

Let $s'$ be a state reachable by $\sigma'$ in $C$. This exists by the inductive hypothesis since $\sigma'$ is a trace of $C$.

By the inductive hypothesis, each $u' \in b'[s']$ is reachable by $\sigma'$ in $A$.

By the definition of extension, $v$ is enabled in $A$ on $u'$.

By (SB Service), $v$ is enabled in $C$ on $s'$. 

113
Since this holds for all states reachable by \( \sigma' \) in \( C \), by the definition of extension, \( v \) is an extension of \( C \).

- Proof of refinement behavior and reachability. Let \( \sigma'; e \) be a trace of \( C \) where \( e = < v, r > \) and \( \sigma'; v \) an extension of \( A \). Show (a) \( \sigma'; e \) is a trace of \( A \) and (b) For all states \( s \) reachable by \( \sigma'; e \) in \( C \) \( b[s] \) is reachable by \( \sigma'; e \) in \( A \).

Proof of (a), behavior.

- By the definition of trace there is a \( s' \) reachable by \( \sigma' \) in \( C \) s.t. \( s' \xrightarrow{\sigma'} C s \).

- By the totality of \( b \) there is \( u \in b[s] \).

- By (SB Visible Behavior), there is \( \hat{\alpha} \) and \( u' \in b[s'] \) s.t. \( u' \xrightarrow{\hat{\alpha}} A u \).

- Hence \( e \) is a trace of \( A \).

Proof of (b), reachability.

- The set of states reachable by \( \sigma'; e \) in \( C \) are the set of all \( s \) s.t. for some \( \hat{\alpha} = e \), \( s' \xrightarrow{\hat{\alpha}} C s \) where \( s' \) is reachable by \( \sigma' \) in \( C \).

- Let \( \hat{\alpha} = e \) and let \( s' \xrightarrow{\alpha} C s \) for \( s' \) reachable by \( \sigma' \) in \( C \).

- \( \alpha \) is of the form \( \langle g_1, \ldots, g_n, e \rangle \), where the \( g_i \) are hidden. See Fig. B.4.

- By the inductive hypothesis there is \( u' \in f[s'] \) s.t. \( u' \) is initial in \( A \).

- Let \( u \in b[s] \). This exists since \( b \) is total.

- By (SB Visible Behavior), there is \( u' \in b[s'] \) s.t. \( u' \xrightarrow{e} A u \).

- By (SB Hidden Behavior), \( u' \in b[s'] \).

- By (SB Init), \( u' \) is initial in \( A \).
Hence \( u \) is reachable by \( \sigma'; e \) in \( A \).

- **Corollary:** If \( s \) is reachable by \( \sigma \) in \( C \) then every \( u \in b[s] \) is reachable by \( \sigma \) in \( A \).

- **Proof of refinement safety.**

  Let \( \sigma \) be an infinite trace of \( C \) and let every finite prefix of \( \sigma \) be a trace of \( A \). Show that \( \sigma \) is a trace of \( A \).

  We rely on the fact that \( b \) is image finite. It suffices to show that there is an infinite computation of \( A \) that, projected to events, is \( \sigma \).

  - Let \( \alpha = \langle s_0, e_1, s_1, e_2, \ldots \rangle \) be a scenario of \( C \) whose trace is \( \sigma \).

  - Construct the digraph \( G \) whose nodes are pairs \( (u, i) \) s.t. \( u \in b[s_i] \) (this is justified since \( b \) is total) and in which there is an edge from \( (u', i') \) to \( (u, i) \) exactly if \( i = i' + 1 \) and \( (u', e_i, u) \) is a transition of \( A \). This construction is given by [23].

  - \( G \) has finitely many roots (since the initial states of \( A \) are finite), each node has finite outdegree (since \( b \) is image finite) and each node of \( G \) is reachable from some root (since every finite prefix of \( \sigma \) is a trace of \( A \)).

  - This satisfies the conditions of by König’s Lemma[15], so there is an infinite path in \( G \) starting from some root. This is the infinite computation of \( A \) whose projection to events is \( \sigma \). □
Proof of Theorem 3.3: State-level Simulations Are Relatively Complete.

Assume $C \sqsubseteq A$ and $A$ has fin. Show that there is $B$ s.t. $A \preceq_f B \preceq_v C$.

Let $B$ be a relational component with the same signature as $C$ that has a shared variable $H$ over $\text{traces}(A) \cap \text{traces}(C)$ and that defines a domain over the integers.

The initial value for $H$ is $\lambda$.

Define the methods of $B$ s.t. $(H = \sigma') \xrightarrow{e_B} (H = \sigma)$ iff $\sigma = \sigma'; e$. For convenience we refer to $\sigma$ as a state of $B$, refering to any state s.t. $H = \sigma$.

Define $f$ and $b$ as follows.

- $f[s] = \{\sigma : s \text{ is reachable by } \sigma \text{ in } C\}$.
- $b[s] = \{u : u \text{ is reachable by } \sigma \text{ in } A\}$.

Show $f$ is a state-level forward simulation from $C$ to $B$.

- SF Init.
  - Let $s$ be an initial state of $C$.
  - Then $f[s] = \{\lambda\}$, so $f[s]$ intersects the initial states of $C$.

- SF Service and Visible Behavior.

  Let $s'$ be a state and $v$ a visible invocation of $C$ s.t. $v$ is enabled in $B$ on some $\sigma' \in f[s']$.

  - SF Service.
    - Since $v$ is enabled in $B$ on some $\sigma' \in f[s']$, for some $f$, $\sigma'; e$ is a state of $B$ where $e =< vr >$.
    - By definition of $B$, $\sigma'; e$ is a trace of both $C$ and $A$.  

116
Consequently $\sigma';v$ is an invocation of $C$.

- SF Visible Behavior.

Let $s' \xrightarrow{e} s$ and $\sigma' \in f[s']$. Show there is $\tilde{\alpha} = e$ and $\sigma \in f[s]$ s.t.

$$\sigma' \xrightarrow{\tilde{\alpha}} B \sigma.$$  

- Let $\alpha = e$.

- Since $\sigma' \in f[s']$, $s'$ is reachable by $\sigma'$ in $C$.

- Hence $s$ is reachable by $\sigma'; e$ in $C$.

- By definition of $f$, $\sigma'; e \in f[s]$.

- By definition of $B$, $\sigma' \xrightarrow{e} B \sigma'; e$.

- SF Hidden Behavior.

Let $s' \xrightarrow{e} s$ for hidden $e$ and $\sigma' \in f[s']$. Then $s$ is reachable by $\sigma'$ in $C$, so

$$\sigma \in f[s].$$

- Show $b$ is an image-finite state-level backward simulation from $B$ to $A$.

- Totality of $b$. $b$ is total by definition of $b$.

- Image finiteness of $b$.

  - Since $A$ has fin, the set of states reachable by any trace $\sigma$ in $A$ is finite.

  - Hence for any $\sigma$, $b[\sigma]$ is finite.

- SB Init.

  - Let $\sigma$ be an initial state of $B$.

  - By definition of $B$, $\sigma = \lambda$.

  - By definition of $b$, $b[\lambda]$ is the set of initial states of $A$.  

117
- Hence for each initial state $\sigma$ of $B$, $b[\sigma]$ subsets the initial states of $A$.

○ SB Service and Visible Behavior.

Let $\sigma$ be a state of $B$ and $v$ a visible invocation of $B$. Let $v$ be enabled in $A$ on all $u' \in b[\sigma']$.

- SB Service.

  ▷ Since $v$ is enabled on all $u' \in b[\sigma']$ and $C$ refines $A$ then for some $r$,
  
  $u' \xrightarrow{<v:r>} A u$ s.t. $\sigma'; <v:r>$ is a trace of both $C$ and $A$.

  ▷ Hence $\sigma' \xrightarrow{<v:r>} B \sigma'; <v:r>$ is a transition of $B$, and so $v$ is enabled in $B$ on $\sigma'$.

- SB Visible Behavior.

Let $e = <v:r>$, $\sigma' \xrightarrow{e} B \sigma'; e$ and $u \in b[\sigma'; e]$. Show there is $\tilde{\alpha} = e$ and $u' \in b[\sigma']$ s.t. $u' \xrightarrow{\tilde{\alpha}} A u$.

  ▷ Let $\alpha = e$.

  ▷ Let $u'$ be reachable by $\sigma'$ in $A$. This exists since $u$ is reachable by $\sigma'; e$ in $B$.

  ▷ By the definition of trace, $u' \xrightarrow{e} A u$.

○ SB Hidden Behavior.

Let $\sigma' \xrightarrow{e} B \sigma$ where $e$ is hidden and let $u \in b[\sigma]$. Show $u \in b[\sigma']$.

- By the definition of $B$, $\sigma' = \sigma'$.

- So $u \in b[\sigma']$. $\square$
Proof of Theorem 3.4: Value-Level Forward Simulations Are Sound.

The approach is to show that a value-level forward simulation induces a state-level forward simulation, and is a straightforward application of the various definitions.

- Let $vf$ be a value-level forward simulation.
- Let $s = (X_0 = c_0, \ldots, X_{n-1} = c_{n-1}, G_0 = c_{g_0}, \ldots, G_{g_{n-1}} = c_{g_{n-1}})$.
- Let $f[s] = \{u : (X_0 = a_o, \ldots, X_{n-1} = can - 1, G_0 = a_{g_0}, \ldots, G_{g_{n-1}} = a_{g_{n-1}})\}$
  where $a_i \in vf[c_i]$ and $ag_i \in vf[cg_i]$.

We must show that $f$ is a state-level forward simulation. Note that since by assumption all methods are visible, we do not need to prove the SF Hidden Behavior condition.

- SF Init. Let $s$ be an initial state of $C$. Show that $f[s]$ intersects the initial states of $A$.
  - $s = G_0 = c_{g_0}, \ldots, G_{g_{n-1}} = c_{g_{n-1}}$ for some initial $cg_i$ values.
  - By the definition of $vf$, for each initial $cg_i$ there is an initial $ag_i \in vf[cg_i]$.
  - There is $u$ s.t. $u = G_0 = a_{g_0}, \ldots, G_{g_{n-1}} = a_{g_{n-1}}$ be s.t. $ag_i \in vf[cg_i]$.
  - Then by the definition of initial states, $u$ is an initial state of $A$.

- Let $s'$ be a state and $v$ a visible invocation of $C$. Suppose $v$ is enabled in $A$ on some $u' \in f[s']$.
  - $s'$ is of the form $(X_0 = c'_0, \ldots, X_{n-1} = c'_{n-1}, G_0 = c_{g'_0}, \ldots, G_{g_{n-1}} = c_{g_{n-1}})$.
  - SF Service. Show that $v$ is enabled in $C$ on $s'$.
    - $v$ is of the form $M(d'_0, \ldots, d'_{m-1}, X_{k_0}, \ldots, X_{k_{l-1}})$.
\[- \text{Let } \overline{d}' = (d'_{0}, \ldots, d'_{m-1}).\]

\[- \text{Let } \overline{c}' = (s'(X_{k0}), \ldots, s'(X_{k_{l-1}}), s'(G_{0}), \ldots, s'(G_{g_{n-1}})).\]

\[- \text{Since } v \text{ is enabled in } A \text{ on some } u' \in f[s'] \text{ there is } u' = (X_{0} = u'_{0}, \ldots, X_{n-1} = u'_{n-1}, G_{0} = ag'_{0}, \ldots, G_{g_{n-1}} = cg'_{g_{n-1}}) \text{ s.t. } A.M[\overline{d}; \overline{a}] \neq \emptyset, \text{ where } \overline{a}' = (u'(X_{k0}), \ldots, u'(X_{k_{l-1}}), u'(ag_{0}), \ldots, u'(ag_{g_{n-1}})).\]

\[- \text{By (VF Service), } C.M[\overline{d}; \overline{c}] \neq \emptyset.\]

\[- \text{Hence } v \text{ is enabled in } C \text{ on } s'.\]

\[\bullet \text{ SF Visible Behavior. Let } e = < \text{vr} > , \; s' \xrightarrow{e} s \text{ and } u' \in f[s']. \text{ Show for some } \alpha = e \text{ there is } u \in f[s] \text{ s.t. } u' \xrightarrow{\alpha} A \; u.\]

\text{Assume } e \text{ is not an invocation of } New; \text{ or, if } e's \text{ invocation is } New(X) \text{ then } X \text{ is a variable that exists in } s'.

\[- \text{Let } \alpha = e.\]

\[- e \text{ is of the form } M(d'_{0}, \ldots, d'_{m-1}, X_{k0}, \ldots, X_{k_{l-1}}); (d_{0}, \ldots, d_{m-1}).\]

\[- \text{Let } \overline{c}' \text{ and } \overline{d}' \text{ be as defined above.}\]

\[- \text{By the definition of transition, } s = (X_{0} = c_{0}, \ldots, X_{n-1} = c_{n-1}, G_{0} = c_{g_{0}}, \ldots, G_{g_{n-1}} = c_{g_{g_{n-1}}}) \text{ s.t. } s' \text{ and } s \text{ agree except that } \overline{d}; \overline{c} \in C.M[\overline{d}; \overline{c}],\]

\[- \text{where } \overline{c} = (s(X_{k0}), \ldots, s(X_{k_{l-1}}), s(G_{0}), \ldots, s(G_{g_{n-1}})) \text{ and } \overline{d} = (d_{0}, \ldots, d_{m-1}).\]

\[- \text{Hence } \overline{d}; \overline{c} \xrightarrow{M} \overline{d}; \overline{c}, \text{ satisfying the first condition of (VF Behavior).}\]

\[- \text{Since } u' \in f[s'], \text{ } u = (X_{0} = a_{0}, \ldots, X_{n-1} = a_{n-1}, G_{0} = ag_{0}, \ldots, G_{g_{n-1}} = ag_{g_{n-1}}) \text{ where } a'_{i} \in vf[c_{i}] \text{ and } ag'_{i} \in vf[c_{g_{i}}].\]

\[- \text{Let } \overline{a}' = (u'(X_{k0}), \ldots, u'(X_{k_{l-1}}), u'(G_{0}), \ldots, u'(G_{g_{n-1}})).\]
Then by the definition of $f$, $\overline{a'} \in vf[c']$. This satisfies the second condition of (VF Behavior).

Hence by (VF Behavior) there is $\overline{a} \in vf[c]$ s.t. $d'; \overline{a'} \xrightarrow{M}d; \overline{a}$.

Let $u$ be the same as $u'$ except $(u(X_{k_0}), \ldots, u(X_{k_l-1}), u(G_0), \ldots, u(G_{g_n-1})) = \overline{a}$.

Then by the definition of step, $u' \xrightarrow{e} u$.

If $e$’s invocation is $New(X)$ then the proof is similar, except that $s$ contains the new, initialized variable. \qed

Proof of Theorem 3.5: Value-Level Image-Finite Backward Simulations Are Sound.

The value-level backward relation for $vb$ is defined as $b[s] = \{ u : (X_0 = a_0, \ldots, X_{n-1} = can - 1, G_0 = ag_0, \ldots, G_{g_n-1} = ag_{g_n-1}) \}$ where $a_i \in vb[c_i]$ and $ag_i \in vb[ cg_i]$. The proof is omitted as it is parallel to that for value-level forward simulations. \qed
Proof of Theorem 3.7: Completeness of Monadic Value-Level Simulations.

We begin with two definitions needed for the theorem.

- Say that $\sigma$ is monadic in $X$ if the set of client variables referenced by events of $\sigma$ is empty or the singleton $\{X\}$.

- Say that $c$ is reachable by $\sigma$ in $C$ for $X$ if $\sigma$ is monadic in $X$, $\sigma$ is a trace of $C$, and for some state $s$ reachable by $\sigma$ in $C$, $s(X) = c$.

Define $B$ as a data type with the same signature as $C$. We may assume that $C$ defines a domain, else the proof is vacuous. Let the domain of $B$ be the intersection of the traces of $C$ and $A$. Define each method of $B$ as follows.

\[
t' \xrightarrow{M(\text{ind},X):\text{ind}}_B t \quad \text{iff} \quad (M \text{ is not } \text{New} \text{ or if } X \text{ is in } t') \text{ and } t(X) = t'(X); M(\text{ind},X) : \text{ind}.
\]
\[
t' \xrightarrow{\text{New}(X)}_B t \quad \text{iff} \quad M \text{ is } \text{New}, X \text{ is not in } t' \text{ and } t \text{ agrees with } t' \text{ except that } s(X) = \langle\text{New}(X)\rangle.
\]

Define $vf$ and $vb$ as follows

- $vf[c] = \{\sigma : \sigma \text{ is monadic in } X \text{ for some client variable } X \text{ and } c \text{ is reachable by } \sigma \text{ in } C \text{ for } X\}$.

- $vb[\sigma] = \{a : \sigma \text{ is monadic in } X \text{ for some client variable } X \text{ and } a \text{ is reachable by } \sigma \text{ in } A \text{ for } X\}$.

Now we show that $vf$ is a value-level forward simulation. Refer to Fig. B.5.

- VF Init. Since there are no shared variables, this is vacuously true.
Figure B.5: Diagram for Monadic Completeness: Value-Level Forward.

- Let $M(\text{ind}; \text{self})$ be a method of $C$. Let $\vec{d}'$ be in $\times \text{ind}$ and $c'$ be in the domain of $C$. Suppose for some $\sigma' \in \text{vf}[c']$ that $B.M(\vec{d}'; \sigma') \neq \emptyset$.
  - VF Init. This is vacuously true since there are no shared variables.
  - VF Service. Show that $C.M(\vec{d}'; c') \neq \emptyset$.
    - By the definition of $\text{vf}$, $c'$ is reachable in $C$ by $\sigma'$ where $\sigma'$ is monadic in $X$ for some variable $X$.
    - Let $s'$ be a state reachable by $\sigma'$ in $C$. Then $s'(X) = c$.
    - Since $B.m(\vec{d}; \sigma') \neq \emptyset$, for some $\vec{d}, \vec{d}'; \sigma' \xrightarrow{M} \vec{d}; \sigma$ where $\sigma = \sigma'; M(\vec{d}', X): \vec{d}$.
    - Hence there is $s$ s.t. $s \xrightarrow{M(\vec{d}', X)} C; s$ where $s(X) = c$.
    - Then by the definition of step, $\vec{d}'; c \xrightarrow{M} \vec{d}; c$.
  - VF Behavior. Let $\vec{d}'; c \xrightarrow{M} \vec{d}; c$ and $\sigma' \in \text{vf}[c]$. Show that there is $\sigma \in \text{vf}[c]$ s.t. $\vec{d}; \sigma' \xrightarrow{M} \vec{d}; \sigma$.  

123
- By the definition of $vf$, $c'$ is reachable by $\sigma'$ in $C$ and in $A$, where $\sigma'$ is monadic in $X$ for some client variable $X$.

- Since $\overline{d'}; c' \xrightarrow{M} \overline{d}; c$, there is $s'$ s.t. $s \xrightarrow{M(\overline{d'}, X)} s$ and $s(X) = c$.

- Hence $c$ is reachable by $\sigma'; M(\overline{d'}, X); \overline{d'}$ in $C$.

- So $\overline{d}; \sigma \in vf[\overline{d}; c]$ and $\overline{d'}; \sigma' \xrightarrow{M} \overline{d}; \sigma$.

Now we show that $vb$ is an image-finite value-level backward simulation. Refer to Fig. B.6.

![Diagram for Monadic Completeness: Value-Level Backward](image)

Figure B.6: Diagram for Monadic Completeness: Value-Level Backward.

- Totality. Let $\sigma'$ be in the domain of $B$. Show that $vb[\sigma'] \neq \emptyset$.

Since $\sigma'$ is a trace of $A$, there is a reachable by $\sigma'$ in $A$.

- Image-finiteness. Let $\sigma'$ be in the domain of $B$. Show that $vb[\sigma']$ is finite.
Since $A$ has fin, the set of states reachable by any trace $\sigma$ is finite. Hence by the definition of $vb$, $vb[\sigma]$ is finite.

- VB Init. This is vacuously true since there are no shared variables.

- Let $M(\text{ind}; \text{self})$ be a method of $B$. Let $\overrightarrow{d'}$ be in $\times \text{ind}$ and $\sigma'$ be in the domain of $B$. Suppose for all $a' \in vf[\sigma']$ that $A.M[\overrightarrow{d'}; a'] \neq \emptyset$.
  
  - VB Service. Show that $B.M[\overrightarrow{d'}; \sigma'] \neq \emptyset$.
    
    - By the definition of $vb$, $a'$ is reachable by $\sigma'$ in $A$ by $X$ where $\sigma'$ is monadic in $X$ for some client variable $X$.
    
    - Hence there is $u'$ be a state reachable by $\sigma'$ s.t. $u'(X) = a'$.
    
    - Since $A.M[\overrightarrow{d'}; a'] \neq \emptyset$ there is $u$ s.t. $u' \xrightarrow{M(\overrightarrow{d'}, X; \overrightarrow{d})} A u$. Let $a = u(X)$.
    
    - Hence $a$ is reachable by $\sigma = \sigma'; M(\overrightarrow{d'}, X; \overrightarrow{d})$ in $A$.
    
    - So $B.M[\overrightarrow{d'}; \sigma'] \neq \emptyset$.

  - VB Behavior. Let $\overrightarrow{d'}; \sigma' \xrightarrow{M} B \overrightarrow{d}; \sigma$ and $a' \in vf[\sigma']$. Show that there is $a \in vf[\sigma]$ s.t. $\overrightarrow{d'}; a' \xrightarrow{M} A \overrightarrow{d}; a$.
    
    - $\sigma'$ is monadic in $X$ for some client variable $X$.
    
    - By the definition of $B$, $\sigma = \sigma'; M(\overrightarrow{d'}, X; \overrightarrow{d})$.
    
    - Hence there is $a'$ reachable by $\sigma'$ in $A$ s.t. $\overrightarrow{d'}; a' \xrightarrow{M} A \overrightarrow{d}; a$.
    
    - So by the definition of $vb$, $a' \in vb[\sigma']$.  

\[\square\]
APPENDIX C

Program Syntax and Semantics

C.1 Program Syntax

The set of programs, Programs, is defined as follows. Given a configuration of relational components $cfg$ for a program component $L$ and a method $L.N(dm)$, a program $pgm \in \text{Programs}$ is a pair $(parg, stmt)$ as follows.

- $L.N.parg$ is a tuple of argument variables of size $|dm|$ for the method arguments.
  s.t. for $0 \leq i < |dm|$, variable $L.N.parg_i$ is defined over $dm_i$.

- $L.N.stmt$ is a program statement, defined below.

In the context of a program $pgm$ for method $L.N(dm)$ and configuration $cfg = \{C_0, C_1, \ldots C_{n-1}\}$, a statement $stmt$ is defined recursively as follows. The set of accessible program variables is given initially as $L.N.pvars = L.cshare \cup L.N.parg$; this set is expanded as new variables are created with the $CNew$ command for some configuration subcomponent $C$. $L.N.BV$ is the subset of accessible program variables that are Boolean. The domain function $L.ctype$ is extended in the natural way to include all accessible program variables.
Here, \textit{methinv} is $C.M(\text{arg})$ for method $M$ of component $C \in \text{cfg}$, where $\text{arg} \in L.N.pvars^*$ and $L.ctype(\text{arg}) = C.ctype(C.M)$. If \textit{methinv} is $C.new(X)$ and $X$ is not in $L.N.pvars$, then $X$ is added to $L.N.pvars$.

\section*{C.2 Program Semantics}

The meaning of a method program is calculated by operationally unfolding the program. This is illustrated in Fig. C.1. Program-invoked methods are interleaved arbitrarily with spontaneous and fault methods from the subcomponents. This can result in infinite (that is, nonterminating) computations. However, we project away those sequences of spontaneous and fault methods that do not change the unfolding state. This permits us to calculate the relation operationally, based on initial final values.

Following de Roever and Engelhardt [10, p. 71], we use demonic nondeterminism, in which the nondeterministic possibility of nontermination is as bad as guaranteed nontermination. Hence the relation does not contain inputs for which termination is not guaranteed. Nontermination can occur either because a program statement at an unfolding state contains a subcomponent method invocation that is not enabled or because a computation is not finite.

Suppose we have a program component $L$ with a configuration $cfg$ as above, and suppose that $L.N(\text{args})$ is an invocation of $L$. 
States for $L$ are defined as for relational components. To recall, a state of $L$ is of the form

$$s = [L.G_0 = g_0, \ldots, L.G_{n-1} = g_{n-1}, L.H_0 = h_0, \ldots, L.H_{k-1} = h_{k-1}],$$

where the $G_i$ variables are shared, and the $H_i$ are client instance variables over $L.cdom$, created by $L.New$ methods.
Unfolding State. An unfolding state us of L.N is a state that has been extended to include argument and temporary variables. It is of the form

\[ us = [ \]
\[ L.G_0 = g_0, \ldots, L.G_{n-1} = g_{n-1}, \]
\[ L.N.U_0 = u_0, \ldots, L.N.U_{l-1} = u_{l-1}, \]
\[ ] \]

where L.N.U_i are the argument variables of L.N.cprog(L.N). Given state s of L, an initial unfolding state of L.N for the invocation L.N(args) at state s of L:

- Agrees with s as regards shared variables
- For 0 \leq i < l, L.N.U_i = s(args_i) if L.c_type(L.N)_i is self and otherwise L.N.U_i = \_args_i. Note that the invocation L.N(args), the indicator arguments are given literally while the self arguments are variables.

For the unfolded program statement (refer again to Fig. C.1), us_0 is the initial unfolding state and stmt_0 is the program statement of L.N.

Each unfolded program transition (us_{i-1}, stmt_{i-1}) \xrightarrow{ast_i} (us_i, stmt_i) is one of the following.

- \textit{stmt}_{i-1} = ast_i; stmt'. This is either an invocation of a client-controlled method or is a program control statement (if, do or skip).
  - ast_i is the client controlled method C.M(arg) that is either not New, or is New(X) where X is a variable in us_i. Then us_i and us_{i-1} agree except that us_i(arg) \in C.M[us_{i-1}(arg)], and stmt_i = stmt'.
  - ast_i is C.New(X) where X is not in us_{i-1}. Then us_i and us_{i-1} agree except that us_i(X) \in C.New, and stmt_i = stmt'.
  - ast_i is \textbf{skip}. Then us_i = us_{i-1} and stmt_i = stmt'.

129
• \textit{stp} is \texttt{do }\Box_{i<n}^n BV_i \rightarrow \texttt{stmt}_i \texttt{ od}, for some \( n > 0 \). If for any \( k < n \), \( f'.BV_k = \texttt{tt} \) then for some \( j < n \), \( f'.BV_j = \texttt{tt} \), \( us_i = us_{i-1} \) and \( \texttt{stmt}_i = \texttt{stmt}_k; \texttt{stmt}' \)). Otherwise, \( \texttt{stmt}_i = \texttt{stmt}' \).

• \textit{stmt}_i \textit{ is a spontaneous fault method, }C.M(\texttt{arg}). Then \( us_i \) and \( us_{i-1} \) agree except that \( us_i(\texttt{arg}) \in C.M[us_{i-1}(\texttt{arg})] \), and \( \texttt{stmt}_i = \texttt{stmt}_{i-1} \).

Now we can calculate the relation for L.N.

Let \( E \) be the set of unfolded program statements for the invocation \(< \texttt{ind'} > L.N(\texttt{instvar}) \texttt{ on state } s' \). For each unfolding in \( E \) we project away transitions that are of the form \((us', stmt') \xrightarrow{\texttt{ast}} (us', stmt')\), where \texttt{ast} is a spontaneous or fault method. That is, we ignore these transitions if they have no effect. Let \( \hat{E} \) be the projected set.

• Let \( \vec{a} = us_0(L.N.U_0, \ldots, L.N.U_{l-1}, L.G_0, \ldots, L.G_{n-1}) \). These are the input values to the relation.

• If any member of \( \hat{E} \) is infinite then L.N is undefined at \( \vec{a} \).

• If \((us_x, stmt_x)\) is the last unfolding state in some finite member of \( \hat{E} \) and \( stmt_x \) is not null then L.N is undefined at \( \vec{a} \).

• Otherwise L.N contains the pairs \((\vec{a}, a)\) where for some (finite) member of \( \hat{E} \), \((us_x, stmt_x)\) is the last unfolding state and

\[
\vec{a} = us_x(L.N.U_0, \ldots, L.N.U_{l-1}, L.G_0, \ldots, L.G_{n-1}).
\]
Finally, Fig. C.1 shows us how to calculate the unfolding of traces $\sigma'$ of $C$ in $L$ by projecting away from members of $\hat{E}$ the state and any methods that are not of $C$. 