The College of Wooster Libraries
Open Works

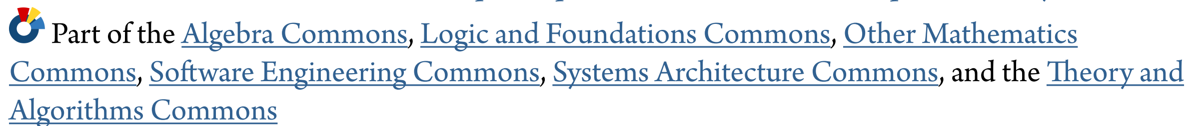
Senior Independent Study Theses

2018

Logic -> Proof -> REST

Maxwell Taylor
The College of Wooster, mtaylor18@wooster.edu

Follow this and additional works at: https://openworks.wooster.edu/independentstudy


Part of the Algebra Commons, Logic and Foundations Commons, Other Mathematics Commons, Software Engineering Commons, Systems Architecture Commons, and the Theory and Algorithms Commons

Recommended Citation
Taylor, Maxwell, "Logic -> Proof -> REST" (2018). *Senior Independent Study Theses.* Paper 8289.
https://openworks.wooster.edu/independentstudy/8289

This Senior Independent Study Thesis Exemplar is brought to you by Open Works, a service of The College of Wooster Libraries. It has been accepted for inclusion in Senior Independent Study Theses by an authorized administrator of Open Works. For more information, please contact openworks@wooster.edu.

© Copyright 2018 Maxwell Taylor

Logic → Proof → REST

Independent Study Thesis

Presented in Partial Fulfillment of the
Requirements for the Degree Bachelor of Arts in
the Department of Mathematics and Computer
Science at The College of Wooster

by
Maxwell Taylor

The College of Wooster
2018

Advised by:

Dr. Denise Byrnes

Dr. Robert Kelvey

Abstract

REST is a common architecture for networked applications. Applications that adhere to the REST constraints enjoy significant scaling advantages over other architectures. But REST is not a panacea for the task of building correct software. Algebraic models of computation, particularly CSP, prove useful to describe the composition of applications using REST. CSP enables us to describe and verify the behavior of RESTful systems. The descriptions of each component can be used independently to verify that a system behaves as expected. This thesis demonstrates and develops CSP methodology to verify the behavior of RESTful applications.

Acknowledgements

I would like to thank my advisors, Dr. Byrnes and Dr. Kelvey, who have been simply amazing to work with. I would also like to express my gratitude towards my parents, Don and Kerry. Pratistha Bhandari has been incredibly supportive throughout the process of creating this thesis. Last, but certainly not least, I wish to acknowledge my Grandma and the support that she has given me throughout my life.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1

2 RESTful Architectures 3

2.0.1 REST and the Web 4

2.1 RESTful Architectural Principles 4

2.1.1 Client-Server Architecture 4

2.1.2 Stateless Communication 6

2.1.3 Cache 7

2.1.4 Uniform Interface 8

2.1.5 Layered System 8

2.1.6 Code on Demand 10

2.2 REST Architectural Elements 10

2.3 Actors in a RESTful System 11

2.4 Conclusion 12

3 Introduction to Process Algebra 15

3.1 Sequent Calculus 15

3.2 Process Algebra 17

3.2.1 Semantics 17

3.2.2 Concurrent and Parallel Processes 20

3.3 Communicating Sequential Processes (CSP) 22

3.3.1 Basic CSP Terms 22

3.3.2 Communication 24

3.3.3 Extended Operators of CSP 27

3.4 Parallel Composition in CSP 30

3.5 Traces of CSP 32

3.5.1 Finding traces 33

3.5.2 Traces of Recursive Processes 33

3.5.3 Traces and Specification 37

3.5.4 Additional Trace Notation 38

3.6 Conclusion 41

4 Formalizing REST with CSP 43

4.1 Introduction to Terminology 43

4.1.1 Communication Channels 44

4.1.2 Set Semantics 46

4.2 Process Definitions 48

4.3 Conclusion 54

5 Concurrency Abstracted **57**
5.1 Trace Theory and Other Formalizations 58
5.1.1 Preliminary Definitions 58
5.1.2 The Trace Monoid 59
5.1.3 Hoare Structures 62
5.2 Trace Structures and Hoare Structures 63
5.2.1 Some Category Theory 63
5.2.2 The Relationship 67
5.3 Other Models of Concurrency 69
5.4 Conclusion 70

6 Description of Software **71**
6.1 Overview of Tools 71
6.2 Case Study of RESTful Application 73
6.2.1 Entrypoint 74
6.2.2 Creating a New Queue 74
6.2.3 Writing to a Queue 75
6.2.4 Retrieving Objects from a Queue 76
6.2.5 Deleting a Queue 76
6.3 Dynamic Error State Observations 77
6.4 Modeling Application with CSP and formalized REST 82
6.5 Conclusion 84

7 Future Work **87**

List of Figures

2.1 A diagram of a client-server network architecture. 5

2.2 A general diagram of a layered RESTful system. 9

4.1 Overview of REST modeled in CSP. 45

5.1 A diagram to visualize objects in a category. 64

5.2 Diagram demonstrating the properties of a functor. 65

5.3 Diagram describing adjunction. 66

5.4 Diagram of relationship between trace and Hoare structures. 69

List of Tables

3.1 Equation specifications for basic process algebra. 18

3.2 Equation specifications for CSP. 27

3.3 Overview of CSP's syntax. 31

4.1 The various sets of objects found in the CSP model of REST. 46

Listings

4.1 UserAgent process definition. 48

4.2 Intermediary process definition. 51

4.3 OriginServer process definition. 53

6.1 A simple Clojure web service using replay middleware. 80

6.2 Sample test generated by middleware. 80

1

Introduction

"But I don't want to go among mad people," Alice remarked. "Oh, you can't help that," said the Cat: "we're all mad here. I'm mad. You're mad." "How do you know I'm mad?" said Alice. "You must be," said the Cat, "or you wouldn't have come here."

– Lewis Carroll, *Alice in Wonderland*

Verifying that the behavior of a program meets some specification is a challenging task. In fact, the problem is unsolvable, in general. So why are we devoting time to a task that is both difficult and impossible? Because software is ubiquitous and has become necessary to sustain the modern world. This statement is so obvious that it has become trite, but the significance of the implications of this clear fact cannot be overstated. We must negotiate a world increasingly governed by information systems. Understanding the behavior of these systems must be a priority.

Meanwhile, data breaches occur daily. We are beginning to trust

automated systems in tasks such as transportation, grid infrastructure control, and decisions about healthcare. Clearly, we have a problem. Each new technology stands on the shoulders of those that have come before, yet not much attention is paid to the underlying process of composition that permits this phenomenon to safely manifest itself. At times, we suffer the consequences for this lack of respect.

Representational State Transfer, or REST, is one of the fastest growing approaches to application architectures. This thesis concerns itself with two tasks. First, we seek to provide an algebraic system to permit formal reasoning of RESTful architectures. Second, we demonstrate how the tools that we use to understand REST interact with other approaches to software verification and concurrency.

Like Alice, we are about to fall down a rabbit hole, albeit one that tests the limits of formal systems. We encounter all kinds of beasts in this wonderland: strange models of computation, bizarre nondeterminism, exotic abstractions in mathematics, all in a beautiful landscape of theory. With that said, enjoy the thesis.

2

RESTful Architectures

Roy Fielding proposed a new architectural style for network based software applications in [13]. This style, called *REST* (for "Representational State Transfer"), solves issues unique to the challenges of the then-emerging world wide web, albeit in a more general fashion. These problems include handling a large number of clients accessing shared resources simultaneously, managing large information systems involving ephemeral hypermedia, and coordinating the efforts of a large enterprise (specifically the functions of discrete entities with competing interests, like quality assurance, application support, and application development.) RESTful architectures are virtually omnipresent in modern enterprises, and its users include technology giants such as Google and Amazon [1] [3]. This section provides an overview of the RESTful paradigm, highlights the components of a RESTful system, and defines its actors.

2.0.1 REST and the Web

We frequently use the world wide web as an example when speaking of a RESTful system. Applications built on the world wide web are not necessarily RESTful, although it is possible and common to design applications on the web that are. In particular, it is important to understand that the protocol of the web, called the "hypertext transfer protocol" (HTTP), can be used to develop RESTful applications. REST itself is protocol agnostic; it is simply a set of architectural principles and elements that are useful when designing applications.

2.1 RESTful Architectural Principles

RESTful architectures are built on several key principles including a client-server design, stateless communication, data caches, uniform interfaces, layered design, and code on demand. Here, we define these principles, provide an analysis of their benefits and drawbacks in application development, and provide examples of their usage.

2.1.1 Client-Server Architecture

REST does not seek to be a peer-to-peer architecture (i.e. various nodes in a network communicate directly with one another.) The alternative to this approach is a client-server architecture. A client-server architecture has two agents operating:

1. *Server* - the node responsible for providing *services* (logically related

functional units) and that actively listens for requests.

2. *Client* - the node that is responsible for initiating communication with the server via *requests*.

A diagram of this architecture is presented in Figure 2.1. Here, each client communicates with a central server. In an alternative peer to peer application, each client also functions as a server and is therefore in direct communcation with other clients.

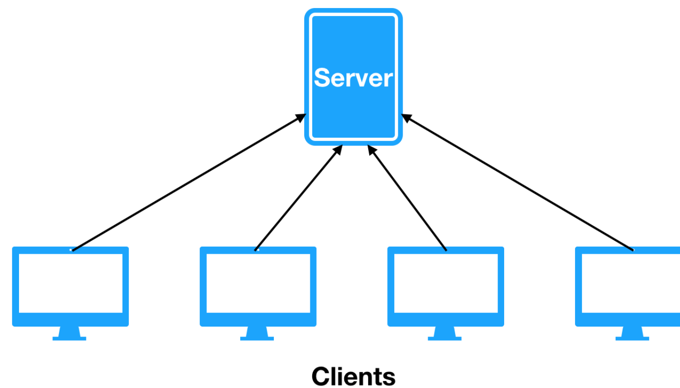


Figure 2.1: A diagram of a client-server network architecture.

The main advantage to the client-server approach lies in its separation of concerns. Consider the following example: applications on the world wide web may be viewed as a RESTful system. Since the user interface rendering is implemented on the client side (i.e. a web browser like Mozilla Firefox) and is therefore separate from the data storage and access concerns handled by the server, then the application is more portable. Observe that the server also does not need to utilize resources rendering an interface, thereby increasing the scalability of the application.

2.1.2 Stateless Communication

Arguably the most important principle to RESTful design, *Stateless Communication*, is the architectural principle that each request contains all information required by the server to create an appropriate response. This necessarily implies that all session state is stored on the client. A counter-example to this principle is the common usage of session identifiers in web applications. Web applications usually issue a *session id* that is stored on the client in the form of a cookie. Requests from the client to the server contain this session id, and the server associates the current state of the client in a key-value store internally.

Applications that emphasize stateless communication enjoy several benefits. First, *visability* improves. *Visability* is the degree that a response can be debugged given its corresponding request. Since each request contains all necessary information for the server to fulfill it, debugging is much simpler. The *reliability* of the application is improved since a partial failure of any particular server does not result in the loss of any client state. If a particular request fails, a client may simply retry. Finally, the application itself is more *scalable* since the server can quickly release any resource associated with a request, and the server does not need to manage any client resources. Drawbacks to this approach include decreased network performance. Since duplicate information (i.e. authentication information) is sent with each request, the network is unnecessarily strained. Also, the server loses control of client interactions. Clients can potentially enter states that the application developers did not consider as a possibility, resulting in scenarios that are

difficult to debug. For example, a website might require a user to complete several forms. If a user "bookmarks" a particular page in that sequence of forms and revisits it, previous form data might be missing, thus leading to undefined interactions.

2.1.3 Cache

Data in a response is implicitly or explicitly labeled as cacheable or non-cacheable by the server. If data is cacheable, then the client reserves the right to reuse the data for an identical request. For example, web servers commonly give the documents that they serve a lifetime. Clients (i.e. web browsers) request the headers for a particular resource. The server responds with the meta-data for the particular resource, including its last modified time. The client can then reuse the resource if it has a local copy within its lifetime. This has the positive effect of mitigating the poor network performance caused by stateless communication. Note that the client need not be associated with a particular end-user, but rather could be a component in a *layered system*. However, this principle can introduce new issues because data can become *stale*, or no longer valid. Application developers must take great care to ensure proper lifetimes are associated with each particular resource, and this potentially causes a complexity bloom. Different resources have different reasonable default lifetimes, and so managing the lifetime of all resources can become a significant challenge.

2.1.4 Uniform Interface

All components of a system must be exposed with an identical interface. For example, consider the world wide web. All resources are accessed using the same methods – GET, POST, PUT, DELETE, HEAD, PATCH, and OPTIONS. Applications that expose resources uniformly are simplified. Clients are more general, as they only need to implement several predefined operations to use a server. Furthermore, visibility is improved since each request is composed of well-understood methods with generalized semantics. A potential drawback to this approach is degraded efficiency. Custom protocols usually can exploit the format of its data to improve the network efficiency for a particular request.

2.1.5 Layered System

REST allows for the composition of separate services into a hierarchical model by adding an application-layer constraint that services can only communicate with layers immediately adjacent to them. This allows application developers to encapsulate legacy services (services that have been deprecated and replaced) and easily provide support for legacy clients (clients that rely on legacy services). Furthermore, since communication is stateless, layers can perform *load balancing* across services in layers below them. The main drawback associated with this approach is that unnecessary overhead and data processing is added, decreasing the overall efficiency of the system.

Consider Figure 2.2. Here, the client is communicating to a *load balancer*, a specialized piece of software or hardware designed to handle a large number of concurrent connections and forward them to a set of services based on some

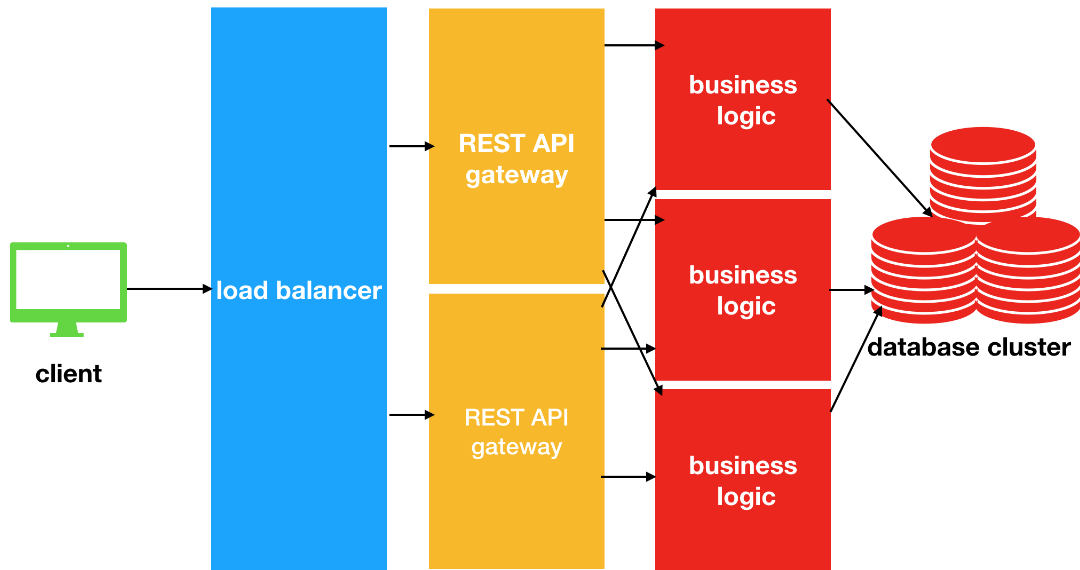


Figure 2.2: A general diagram of a layered RESTful system.

set of rules. In this case, the load balancer is distributing requests among *REST API gateways* – services that might provide features like API versioning, secure socket layer encryption (SSL), and expose additional services as a logical resource according to some scheme. These gateways expose functionality provided by the business logic layer that contain custom services to support the main functionality of the application. Notice that these gateways provide an additional layer of load-balancing across the services implementing the business logic layer. Architectures similar to this are widely used today and are a good example of layered design in REST.

2.1.6 Code on Demand

REST clients can be arbitrarily extended by downloading and executing code in the form of applets or scripts. As an example, consider a web browser. Servers often provide scripts that are executed inside of the browser. The client is free to request whatever code is necessary to accomplish its particular goals. This principle is perhaps one of the most difficult to implement in general, since it requires the client to implement a language or provide other mechanisms for arbitrary code execution, and consequentially it is often relaxed as a requirement.

2.2 REST Architectural Elements

REST defines several high-level elements of interest. These elements are deliberately designed to be vague so that REST can be employed in virtually any application involving networked communications. Here, we define these elements and provide examples. For a more detailed analysis of these elements, consult Fielding's original thesis [13].

1. *Resource* – the conceptual target of a *hypermedia* reference. *Hypermedia* refers to a particular resource (i.e. a document, image, mp3 file, etc.) or a reference to such a resource. A resource can also be viewed as any information that can be named, such as "today's weather." Note that such a relationship is only persistent in the sense that the concept itself is persistent, although the underlying data may vary temporally.

2. *Resource Identifier* – a way to refer to a unique resource. An example of

this is a URL for a website.

3. *Representation Metadata* – metadata for a resource's representation. For instance, a "Text-Encoding" header that represents how a textual resource is encoded.

4. *Resource Metadata* – metadata for a particular resource itself, such as source links or alternate versions.

5. *Control Data* – data used to control access to a resource. As an example, consider the "if-modified-since" hypertext transfer protocol (HTTP) header that requests a response only if a resource was modified after a particular time.

2.3 Actors in a RESTful System

Fielding describes several agents that are at work in any RESTful system. Here, we provide an overview and examples of these entities, and describe how they operate in conjunction with one another.

(a) *Origin Server* – an authoritative entity for resource representations. Origin servers govern the namespace of a requested resource. For example, Apache's HTTP daemon is a common origin server.

(b) *Gateway* – the entrypoint to an origin server. Consider NGINX and similar reverse proxies as an example.

(c) *Proxy* – an intermediary connector between a client and a gateway.

(d) *User Agent* – the client-side entity responsible for initiating requests. Consider a web browser like Mozilla Firefox as an example.

Furthermore, Fielding describes the concept of *connectors* that are responsible for encapsulating logically discrete activities involved in the act of accessing a hypermedia resource. These connectors include:

(a) *Client* – the connector responsible for initiating requests.

(b) *Server* – the connector that actively listens for and responds to client requests.

(c) *Cache* – the connector responsible for implementing the cache described in REST's architectural principles. This connector can reside on the client or the server.

(d) *Resolver* – the connector responsible for translating universal resource identifiers (URIs) into a format that is network-addressable. As an example, consider the domain name system (DNS).

(e) *Tunnel* – the connector that relays communication across the client and server boundaries. For instance, consider a firewall or SSL.

2.4 Conclusion

REST is among the most widely used application architectures today. It is designed to be highly scalable, both organizationally and functionally. By separating concerns to create entities that are as simple as required, applications increase in reliability and organizations can more easily

manage complex and evolving technologies. Finally, REST itself is protocol agnostic and therefore its applications range from internet-of-things devices (i.e. devices communicating with the "Constrained Application Protocol" or COAP) to full-featured web applications over HTTP.

In the next chapter, we introduce Process Algebra. Process Algebra is used to construct a formal model of REST. This model allows us to verify the behavior of RESTful systems with both component and system level granularities.

3

Introduction to Process Algebra

This chapter presents an overview of Process Algebra, particularly the variant we use to model and verify REST, called *Communicating Sequential Processes* (CSP). First, we describe a style of formal logic called *Sequent Calculus*, where each statement in the system is inferred by previous statements. Then, we use sequent calculus to describe process algebra. Finally, we present fundamental results of process algebra in general, as well as particular results of CSP.

3.1 Sequent Calculus

Sequent Calculus is an approach to proof theory developed by the logician Gerhard Gentzen between 1934 and 1935 [12]. A *sequent* is a sequence of formulae of the form $\Gamma \vdash \Delta$, where both Γ and Δ are formulae themselves. The operator $\vdash$, called the "turnstile operator," can be read as *entails*. The antecedent Γ is understood to be read conjunctively, while the consequent Δ is read as a disjunction. Hence, a sequent is derivable if and only if $\wedge\Gamma \implies \vee\Delta$

is a theorem in the logic of the particular sequent calculus. In the case of process algebra, we restrict ourselves to propositional calculus.

There is one axiom that is required in all sequent systems. This is called the *identity axiom*, which is described as:

$$\overline{a \vdash a} \cdot$$

The horizontal line here is referred to as an *inference line.* If all the *premises* above the inference line are true, then the *conclusion* below the inference line is also true. Notice that the definition of the identity axiom does not include any premises, hence it is an axiom.

As an example of more complex formulae, we look at classical logical operators as presented by Mark Tarver in *Logic, Proof, and Computation* [21]. Consider logical conjunction. To prove $A \wedge B$, we must prove both formulae A and B. We can easily describe this in sequent calculus:

$$\frac{\mathrm{A\ B}}{A \wedge B} .$$

We can read this rule as "Any sequent with a conclusion of $A \wedge B$ is provable if both A and B can be proved."

A more comprehensive treatment of sequent calculus can be found in Mark Tarver's "Logic, Proof, and Computation" [21]. An excellent overview of sequent calculus and its place in the timeline of the development of formal logic is provided in "A Dictionary of Logic" [12].

3.2 Process Algebra

Process algebra is the study of concurrent communicating processes in an algebraic framework. Hence, we treat the theory of concurrent communicating processes in an axiomatic way. We begin by presenting a theory of "Basic Process Algebra (BPA)," as described in [8] and [9].

Definition 3.2.1. *An **equational specification** is given by the tuple* (Σ, E)*. E is a set of equations in the form $t_1 = t_2$ where t_1 and t_2 are called terms and Σ is their signature; that is, the set of constant and function symbols that may appear in the equations. E is the set of axioms of an equational specification.*

For our basic process algebra, we have the equational specification $BPA = (\Sigma_{BPA}, E_{BPA})$. Here, $\Sigma_{BPA} = \{+, *, \lfloor\!\lfloor, \|, \delta, \epsilon\} \cup A$. The binary operators that can be applied to terms in basic process algebra are elements of the the set $\{+, *, \lfloor\!\lfloor, \|\}$. Often, we simply omit the $*$ symbol in equations, i.e. $a * b$ is equivalent to ab. The set A contains the *actions* that a process is able to perform, which are usually denoted by latin characters. The equational specification of basic process algebra, E_{BPA}, is shown in Table 3.1.

3.2.1 Semantics

We begin by presenting a precise definition of *atomic actions* for the set A in basic process algebra.

Definition 3.2.2. *An **atomic action** is an abstract step of computation that cannot be interrupted or subdivided.*

Table 3.1: Equation specifications for basic process algebra.

Equation	Axiom
$x + y = y + x$	A1
$(x + y) + z = x + (y + z)$	A2
$x + x = x$	A3
$(x + y)z = xz + yz$	A4
$(xy)z = x(yz)$	A5
$x + \delta = x$	A6
$\delta x = \delta$	A7
$x\epsilon = x$	A8
$\epsilon x = x$	A9
$x \parallel y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x$	A10
$a \lfloor\!\lfloor x = ax$	A11
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$	A12
$ax \lfloor\!\lfloor y = a(x \parallel y)$	A13

Examples of atomic actions include sending and receiving messages, or updating a memory location [8]. Since the base actions that a process can perform are atomic, we do not need to concern ourselves with the possibility that an action is somehow affected by another process. In other words, all atomic actions are executed independently of each other. Therefore, we need only worry about the composition of actions.

We have two basic operators responsible for sequentially composing processes in basic process algebra. First, $*$ is the *sequential composition* operator. The process $x * y$ first executes x, and upon completion of x executes y. Second, $+$ is called the *alternative composition operator.* $x + y$ is the process that either executes x or y, but never both.

A1 states that a choice between executing processes x and y is identical to the choice between executing processes y and x. This property is called the *commutativity* of $+$. Axiom A2 says that a choice between x and choosing

between y and z is identical to making a choice between x and y, and then choosing between that result and z. This property is known as *associativity*. Axiom A3 states that choosing between x and itself is the same as choosing x. This property is referred to as *idempotency*. Axiom A4 (called *right distributivity*) states that a choice between x and y followed by z is identical to choosing between xz and yz. Finally, axiom A5 states that executing x followed by yz is the same as executing xy and then z. This is the *associativity* property of $*$.

Axioms A6-A9 introduce specifications for operations on δ and ϵ. δ is used to represent a state of *deadlock*, while ϵ represents the *empty process*.

Definition 3.2.3. ***Deadlock****, denoted as the process δ, is a process in an error state, unable to proceed.*

Axiom A6 states that if an alternative to deadlock exists, then the process always "chooses" that alternative. A7 is a very intuitive specification; if an action is specified to occur after the process enters deadlock, then that action shall never take place.

Definition 3.2.4. *The* ***empty process****, denoted as ϵ, performs no actions.*

Axioms A8 and A9 state that prefixing or postfixing any action with the empty process is equivalent to only taking the original action. Therefore, the empty process serves as the unit element under sequential composition.

Notice that left distributivity is not present in this algebra, hence $x(y + z) \neq xy + xz$. This is because in the equation $x(y + z)$, first x is executed, and then a choice between y and z is made. In the equation $xy + xz$, first a choice between xy and xz is made, and then the term is executed. Thus, the behavior is not identical between these two expressions.

As an example of the difference between $x(y + z)$ and $xy + xz$, consider the following scenario. Imagine you are standing in a plane, preparing to parachute to the ground. There are two parachutes to choose from; one is going to fail, and the other is safe. Let x denote the act of jumping, y indicate that the parachute is safe, and z denote the parachute is unsafe. Then, this scenario is modeled with the process $xy + xz$. Since after we jump we are unable to retroactively change the parachute that we chose, this behavior is very different from $x(y + z)$, which first executes the jump and then chooses between the parachutes.

Although left distributivity is not present in this system in general, for a specific case this property is present. Specifically, the equation $a(b + b) = ab + ab$. The proof of this theorem is straightforward:

Proof. Observe that $a(b + b) = ab$ by A3. Also by A3, we have that $ab = ab + ab$. So, we have that $a(b + b) = ab = ab + ab$, as desired. □

3.2.2 Concurrent and Parallel Processes

Axioms A10-A13 provide a primitive model useful for describing concurrent processes using the *merge operator* ($\|$).

Definition 3.2.5. *The **merge operator** applied to processes x and y in the equation $x \| y$ arbitrarily interleaves the actions of x with the actions of y.*

This approach to modeling concurrency is often called *arbitrary interleaving* or *shuffling*.

In order to totally axiomize the merge operator, we add another operator to the system ($\lfloor\!\lfloor$) called *left merge.*

Definition 3.2.6. *The operator* $\lfloor\!\lfloor$ *applied to processes* $x = az$ *(where* a *is some atomic action and* z *is some process, possibly* ϵ*) and* y *in the equation* $x \lfloor\!\lfloor y$ *is equivalent to* $a(z \parallel y)$.

Left merge performs exactly the same operation as the merge operator, albeit the first action must come from the term on the left of the operator.

Axiom A11 states that an atomic action a left merged with some process x is equivalent to ax. Meanwhile, axiom A12 provides the right distributive law for the left merge operator. Finally, axiom A13 provides a formal specification of our assertion that left merge first executes an action from the leftmost term and then merges the remainder of both processes.

With A11 and A12, we provide the specification of the merge operator applied to two processes. Axiom A10 states that the merge operator on x and y is equivalent to the arbitrary choice between left merging x and y and left merging y and x. This matches our intuition fairly well; we do not care what term from what process is actually executed first.

Example 3.2.1. As an example application of the merge operator, consider a *spider* that is responsible for *crawling* over a fixed set of websites and indexing their contents. Let W denote the set of websites, and $crawl.w$ be an atomic action that crawls the site $w \in W$. We sequentially crawl W with the process $CRAWL = crawl.w_0 * crawl.w_1 * \ldots * crawl.w_n$ for $w_0, w_1, \ldots, w_n \in W$. However, in this specific scenario we are not concerned with the order of these operations. In practice, an implementor might decide to have each of these actions performed concurrently. To model this behavior algebraically, we use the merge operator. We write the concurrent crawl process as

$CRAWL_{CONCURRENT} = crawl.w_0 \parallel crawl.w_1 \parallel \ldots \parallel crawl.w_n$ for $w_0, w_1, \ldots, w_n \in W$.

3.3 Communicating Sequential Processes (CSP)

Tony Hoare's seminal 1978 article "Communicating Sequential Processes" presents one of the most common variants of process algebra [16]. Over the years, the theory he presented has been extended and modified in non-trivial ways. The majority of the modern theory of CSP is described well in [20]. Here, we present the background of the theory of CSP and connect it to the theory of basic process algebra presented earlier. Later in Chapter 4, we proceed to formally describe REST using CSP.

CSP has a unique view of computation as a form of communication. The alphabet of actions, Σ, defines allowed communications. A process might communicate an action to the global environment so that some computation is performed, or communication from one process can be redirected to another process. Actions can also come from the environment, as in the case of a user pressing a key. In short, action in CSP is equivalent to communication. This has a profound impact on the simplicity of this model with respect to interprocess communication.

3.3.1 Basic CSP Terms

Just as basic process algebra permits sequential process prefixing through the binary $*$ operator, CSP permits sequential prefixing through the $\rightarrow$ operator. So, given an action $a \in \Sigma$ and a process P, $a \rightarrow P$ is the process that is initially

willing to communicate a and then behaves like P. Notice that this is exactly identical to how $*$ behaves in basic process algebra. As an example, let $\Sigma = \{getup, breakfast, work, lunch, dinner, gotobed, STOP\}$. Then we can define the process Day given in example 3.3.1.

Example 3.3.1.

$$Day = getup \rightarrow breakfast \rightarrow work \rightarrow lunch \rightarrow work \rightarrow dinner \rightarrow gotobed \rightarrow STOP.$$

Note that the $STOP$ term in CSP corresponds directly to the δ symbol in basic process algebra.

We can also specify arbitrary choice between processes in CSP. The + operator from basic process algebra is written as | in CSP. The process described by $(a_1 \rightarrow P_1 | a_2 \rightarrow P_2 | \ldots | a_n \rightarrow P_n)$ therefore performs any one of $a_i \in \{a_1 \ldots a_n\}$ and then behaves like the corresponding P_i. This is known as the *guarded alternative* form for processes. Later in Section 3.3.3, we see that | is a specific case of a more generic *external choice* operator.

Notice that the Day process defined in example 3.3.1 concludes by communicating $STOP$ to the environment and then halts completely. We can remedy this situation by allowing recursive specifications such as:

Example 3.3.2.

$$Day = getup \rightarrow breakfast \rightarrow work \rightarrow lunch \rightarrow work \rightarrow dinner \rightarrow gotobed \rightarrow Day.$$

We consider such a specification valid in CSP.

Since resursive processes are permitted in CSP, we add the operator μ to

create anonymous (unnamed) recursive processes.

Definition 3.3.1. *The operator μ, provided with a variable x and a process P in the equation $\mu x.P$, is equivalent to $P[\![\mu x.P/x]\!]$ where $P[\![\mu x.P/x]\!]$ denotes the substitution of all instances of x that are not bound in another recursive expression in P with the expression $\mu x.P$.*

Consider the process $P = left \rightarrow right \rightarrow P$ over the alphabet $\Sigma = \{left, right\}$. Using the μ operator, we can rewrite this process as $\mu y.left \rightarrow right \rightarrow y$. After applying the definition of μ a single time (since this definition is recursive and there is no terminating behavior, we can apply this definition an unbounded number of times) we have $left \rightarrow right \rightarrow (\mu y.left \rightarrow right \rightarrow y)$. Notice that we can continue applying the definition of μ indefinitely to achieve an expansion of this anonymous expression that is equivalent to the original recursive definition provided in P.

3.3.2 Communication

We also have processes that can accept a variety of communications and then behave like some process. The ? operator, called *prefix choice*, permits a process to accept communication from its environment.

Definition 3.3.2. *Let Σ be a set of actions. Then, for $A \subseteq \Sigma$, the process $?\,x : A \rightarrow P(x)$ accepts any communication x of type A and then behaves like the corresponding $P(x)$.*

When the type of x is left unspecified, we assume that $x \in \Sigma$. There are several obvious results of prefix choice. First, observe that

$?\,x : \emptyset \rightarrow P(x) = STOP$. Since no communications are accepted by this process, it indefinitely waits for a communication that it cannot receive. Furthermore, $?\,x : \{a\} \rightarrow P(x) = a \rightarrow P(a)$ because there is only a single communication accepted by the process. As an example of the prefix choice operator in a useful process, consider the following definition of a process that accepts any communication x from its alphabet Σ and communicates x back to its environment:

Example 3.3.3.

$$Repeat = ?\,x : \Sigma \rightarrow x \rightarrow Repeat.$$

Communication is not confined to only occur between processes and their environments. CSP provides an abstraction called a *channel* that allows processes to direct their communication to a specific location. We require that all channels have a name.

Definition 3.3.3. *The string $c!\,x$ denotes communication of a datum x over channel c.*

We extend any alphabet Σ to allow communication over a channel c of objects of type T by defining $c.T = \{c!\,x | x \in T\}$ and unioning $c.T$ with the original alphabet.

CSP also provides facilities for receiving objects over a particular channel.

Definition 3.3.4. *The string $c?\,x : T \rightarrow P(x)$ denotes a process that receives and binds a datum of type T to a variable x over channel c and then behaves like the corresponding $P(x)$.*

The alphabet is further extended to allow a process to receive objects of type T over a channel c by defining $c?\,T = \{c?\,x | x \in T\}$ and unioning $c?\,T$ and

the original alphabet.

As an example of the communication features of CSP, consider a very simple buffer between channels $left$ and $right$ that holds only a single element of type T. We define this buffer as:

Example 3.3.4.

$$Buffer = left?x : T \rightarrow right!x \rightarrow Buffer.$$

This process first receives from $left$ some object of type T and binds x to that datum. Then, $Buffer$ communicates the datum x over channel $right$. Finally, $Buffer$ transitions back into itself with the recursive transition to $Buffer$.

All input over a channel is coordinated, so a process is unable to move on after placing an element into a channel until some process completes the action by removing that element from the channel. With that in mind, consider an example of a buffer than can store an infinite number of objects. We define this buffer as:

Example 3.3.5.

$$Buffer^{\infty}_{\langle\rangle} = left?x : T \rightarrow Buffer^{\infty}_{\langle x \rangle}$$
$$Buffer^{\infty}_{s\langle y \rangle} = (left?x : T \rightarrow Buffer^{\infty}_{\langle x \rangle s\langle y \rangle} | right!y \rightarrow Buffer^{\infty}_{s}).$$

Here, we use the notation $\langle y \rangle$ to denote a sequence containing only y, while s denotes the prefix of datums stored in the buffer. Placing two sequences adjacent to one another denotes their concatenation. The subscript parameter

of $Buffer^{\infty}$ signifies the sequence of objects currently in the buffer. When the sequence of objects in the buffer is empty, then $Buffer^{\infty}$ waits for some datum to be placed in channel $left$. Otherwise, $Buffer^{\infty}$ either receives another object and evolves into a new process containing that object in the stored sequence, or it succeeds in writing the earliest received datum in the sequence $s\langle y\rangle$ to channel $right$. This demonstrates two important features of CSP: (1) processes may have parameters associated with them, and (2) processes may have named identifiers, as is the case with the identifiers s and y.

3.3.3 Extended Operators of CSP

Table 3.2: Equation specifications for CSP.

Equation	Label
$P \Box P = P$	A14
$P \sqcap P = P$	A15
$P \Box Q = Q \Box P$	A16
$P \sqcap Q = Q \sqcap P$	A17
$P \Box (Q \Box R) = (P \Box Q) \Box R$	A18
$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$	A19

Table 3.2 presents the core axioms of the algebraic theory of CSP. We omit certain axioms of operators with direct equivalences in basic process algebra. Here, we provide an overview of how these operators function and provide examples of their applications.

Definition 3.3.5. *The **external choice operator**, written $P \Box Q$, offers the environment the choice between the first actions communicated by P and Q, and then behaves like the process that communicated the action. Further, if S is a finite indexing*

set over a collection of processes P_α, then $\square\{P_\alpha | \alpha \in S\} = P_\alpha \square P_\beta \square \ldots \square P_\gamma$ for $\alpha, \beta, \ldots, \gamma \in S$.

This external choice operator is provided in conjunction to the | operator. We see that external choice is a more general operator. First, observe that $(a \rightarrow P)\square(b \rightarrow Q)$ behaves like $(a \rightarrow P)|(b \rightarrow Q)$. External choice has the advantage over | in that $P\square STOP$ does not deadlock unless P deadlocks, since there are no actions offered by $STOP$ for the environment to choose.

Consider the formula $((a \rightarrow P)\square(a \rightarrow Q))$. Since the action a must be chosen by the environment, it is undefined whether this process behaves like P or Q after a. We intentionally leave this choice ambiguous. We define a *deterministic* process as one where the range of events offered to the environment depends only on the sequence of observed communications to this point. So, a process is *nondeterministic* if it is not deterministic. Since the trace (the sequence of communications) of the execution of $((a \rightarrow P)\square(a \rightarrow Q))$ depends on the implementation (and cannot be determined by the history of the process) it follows that this is a nondeterministic formulae. Since nondeterminism is present in CSP, we provide an operator to make this phenomenon simpler to reason about.

Definition 3.3.6. ***Nondeterministic*** *or* ***internal choice*** *between processes P and Q, written as $P \sqcap Q$, represents a choice between executing process P or Q. Further, if S is a finite indexing set over a set of processes P, $\sqcap\{P_\alpha | \alpha \in S\} = P_\alpha \sqcap P_\beta \sqcap \ldots \sqcap P_\gamma$ for $\alpha, \beta, \ldots, \gamma \in S$.*

The difference between internal ($\sqcap$) and external ($\square$) choice is highlighted by comparing the formulae $(a \rightarrow STOP)\square(b \rightarrow STOP)$ and

$(a \rightarrow STOP) \sqcap (b \rightarrow STOP)$. In the first formula, the process can proceed if either a or b is supplied by the environment. However, in the case of internal choice, if only one action is supplied by the environment then the process might not advance. Consider when a is provided to the environment, but not b. Then, the system might choose to execute the side $b \rightarrow STOP$, and block indefinitely waiting on b, which will never be provided. This is because the choice of what side of $\sqcap$ is executed occurs without regard of communication in the environment.

For an example of the internal choice operator applied, once again consider a buffer. The extended buffer provided in example 3.3.5 is somewhat unrealistic in the sense that it may store infinite data. We can model a buffer that stores an arbitrarily large amount of data before encountering some system fault using the internal choice operator. This process is given by:

Example 3.3.6.

$$Buffer = left?x : T \rightarrow Buffer_{<x>}$$
$$Buffer_{s<y>} = (STOP \sqcap left?x : T \rightarrow Buffer_{<x>s<y>}) \Box right!y \rightarrow Buffer_s.$$

This is similar to the definition provided in example 3.3.5 except that there is an option that the system enters a deadlock state if it is unable to write contents of the buffer to the *right* channel. In practice, this might signify something like an out of memory error.

CSP also provides conditional flow control. This is written in the form $P \triangleleft b \triangleright Q$ and is read as "If b then P, else Q." In general, we allow any computable expression to serve as the condition. For example, we define a

process that echoes all even numbers that it receives over channel $left$ as:

Example 3.3.7.

$$EVEN_ECHO = left?\,x : \mathbb{Z} \rightarrow (left!\,x \rightarrow EVEN_ECHO) \triangleleft (even?\ x) \triangleright EVEN_ECHO.$$

3.4 Parallel Composition in CSP

Recall that action in CSP is equivalent to communication. The interleaving $\|$ operator present in basic process algebra does have a direct analogue in CSP (the $|||$ operator), however $\|$ itself takes on a slightly new meaning.

Definition 3.4.1. *The process given by $P \|_X Q$ synchronizes P and Q on events $a \in X \subseteq \Sigma$.*

As an example of the parallel synchronization operator, consider the following:

Example 3.4.1. Let $\Sigma = \{left, right, work, sleep\}$. Then, the process $P = (left \rightarrow work \rightarrow right) \|_{\{left,right\}} (left \rightarrow right)$ first communicates "$left$", then communicates "*work*" since the communication "*right*" must be synchronized. Finally, "*right*" is communicated.

This parallel synchronization operator can introduce scenarios where the process deadlocks. For example:

Example 3.4.2. Let $\Sigma = \Sigma$ be as in example 3.4.1. Let $P = (left \rightarrow STOP) \|_{\{left,right\}} (right \rightarrow STOP)$. Because the left-hand side of the P communicates "$left$" initially, and the right-hand side of P communicates "*right*", P is unable to proceed, since these events must be synchronized.

Table 3.3: Overview of CSP's syntax.

Syntax	Description
$a \to b$	Sequentially communicate a and then b
$\mu x.P$	Execute P, with the variable x bound to P
$?\,x : A \to P(x)$	Accept $x \in A \subseteq \Sigma$, then behave like the corresponding $P(x)$
$c!\,x$	Communicate object x over channel c
$c?\,x : T \to P(x)$	Accept $x \in T \subseteq \Sigma$ over channel c, and then behave like $P(x)$
$P \square Q$	Offer the 1st actions of P and Q, behave like chosen term's process
$P \sqcap Q$	Non-deterministically choose between P and Q
$P \triangleleft a \triangleright Q$	If a then execute P, otherwise execute Q
$P \parallel_X Q$	Execute P and Q while sychronizing on events in X
$P \mid\mid\mid Q$	Arbitrarily interleave the communications of P and Q
$P[\mid X \mid]Q$	Execute P and Q communicating over channels in set X

Definition 3.4.2. *The process $P \mid\mid\mid Q$ arbitrarily interleaves the communications from processes P and Q. Hence, we have the law that:*

$$(x \to P) \mid\mid\mid (y \to Q) = (x \to (P \mid\mid\mid (y \to Q)))\square(y \to ((x \to P) \mid\mid\mid Q)).$$

Example 3.4.3. Let $\Sigma = \{a, b, c\}$. Then, the process $P = (a \to b \to c) \mid\mid\mid (b \to a \to c)$ has possible traces of communication $\langle a, b, c, b, a, c \rangle$ and $\langle a, b, b, a, c, c \rangle$, among other permutations of these communications.

Table 3.3 provides a reference for CSP's syntax. As we proceed through this dissertation, refer to it to refresh the meaning of the operators. CSP contains enough useful operators that we can adequately describe any computable process, given the proper atomic actions. Furthermore, as we will see, CSP equations can also be viewed as a *specification* for how a process ought to behave. Since actions correspond to communication, this begs the question: can we track the communication of processes? Can we verify that communication between processes complies to a formal specification? We

consider these questions in the next section.

3.5 Traces of CSP

A CSP environment can observe any of the communications between itself and an executing process. Imagine that an environment that keeps an ordered journal of communication with a process, P. We call such a journal a *trace* of P.

Definition 3.5.1. *The* ***traces*** *of a process P with alphabet* Σ*, denoted* $traces(P)$*, is the set of all possible sequences of P's communication. Furthermore,* $traces(P) \subseteq \Sigma^*$ *(the set of all strings over* Σ*).*

Definition 3.5.2. *Let P and Q be processes. If* $traces(P) = traces(Q)$*, then P and Q are* ***trace-equivalent****, written as* $P =_T Q$*.*

Example 3.5.1. Let $P = STOP$. Then, $traces(P) = \{\}$ since P communicates nothing.

Example 3.5.2. Let $P = (a \rightarrow b \rightarrow STOP)$. Then, $traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$.

Example 3.5.3. Let $P = (x \rightarrow STOP)\Box(y \rightarrow STOP)$. Then, $traces(P) = \{\langle\rangle, \langle x\rangle, \langle y\rangle\}$.

Observe that for any process P, $\forall t \in traces(P)$, t is a finite sequence. Furthermore, for any process P, $\langle\rangle \in traces(P)$. Finally, if s is the prefix of the trace st, then $s \in traces(P)$.

Definition 3.5.3. *Let P and Q be processes. We say that Q* ***trace-refines*** *P, written as* $P \sqsubseteq_T Q$*, if* $traces(P) \supseteq traces(Q)$*.*

3.5.1 Finding traces

In order for process traces to actually be useful, we must have laws to generate them. Here, we describe these traces rigorously, based on the descriptions provided by Hoare in [16], although we use sequent calculus. Often, trace laws are presented as equational specifications. Our approach has the advantage of making trace specifications a more obvious development from the rules needed to generate them.

Definition 3.5.4. *Let P, Q, and R be processes in CSP over Σ. Let $A \subseteq \Sigma$. Then, we have:*

- $\dfrac{P = STOP}{traces(P) = \{\langle\rangle\}}$.
- $\dfrac{P = a \to Q}{traces(P) = \{\langle\rangle\} \cup \{\langle a\rangle\, s | s \in traces(Q)\}}$.
- $\dfrac{P = ?\, x : A \to Q(x)}{traces(P) = \{\langle\rangle\} \cup \{\langle a\rangle\, s | s \in traces(Q[\![a/x]\!])\ and\ a \in A\}}$.
- $\dfrac{P = c?\, x : A \to Q(x)}{traces(P) = \{\langle\rangle\} \cup \{\langle c.a\rangle\, s | s \in traces(Q[\![a/x]\!])\ and\ a \in A\}}$.
- $\dfrac{P = Q \Box R}{traces(P) = traces(Q) \cup traces(R)}$.
- $\dfrac{P = Q \sqcap R}{traces(P) = traces(Q) \cup traces(R)}$.
- $\dfrac{b\ \ (P = Q \triangleleft b \triangleright R)}{traces(P) = traces(Q)}\ \dfrac{\neg b\ \ (P = Q \triangleleft b \triangleright R)}{traces(P) = traces(R)}$.

3.5.2 Traces of Recursive Processes

Finding the traces of recursive processes proves to be difficult, in general. Consider the example:

Example 3.5.4. Let $P_1 = up \rightarrow down \rightarrow P_1$, $P_u = up \rightarrow P_d$, and $P_d = down \rightarrow P_u$. We can show that P_1 and P_u are trace equivalent.

We do not yet have the tools to show this result. Here, we provide an overview of recursion theory to develop and understand the *Unique Fix-Point Theorem* described in [11] and [10].

Recursive Functions and Fix-Points

Consider the factorial function defined by:

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Now, hoist the recursive appearance of *factorial* as a function parameter:

```
factorialGenerator :: (Integer -> Integer) -> (Integer -> Integer)
factorialGenerator f 0 = 1
factorialGenerator f n = n * f (n - 1)
```

We are interested in examining the *fix-points* of $factorialGenerator$.

Definition 3.5.5. *A function f has a **fix-point** a if and only if $f(a) = a$.*

Observe that $factorialGenerator$ partially applied to $factorial$ gives the result:

```
factorialGenerator factorial 0 = 1
factorialGenerator factorial n = n * factorial (n - 1),
```

which is precisely the *factorial* function. Hence,
$factorialGenerator(factorial) = factorial$. Therefore, we conclude that $factorial$ is a fix-point of $factorialGenerator$.

In general, we can apply this "hoisting" procedure to any recursive function. The original recursive function is the fix-point of the new hoisted function. Does each hoisted recursive function have a solution? We consider this question after introducing an alternative model of computation.

One model of computing is called *lambda calculus.* Lambda calculus is an extremely simple function system that allows for function definitions with λ. Consider the example:

Example 3.5.5. $ID = \lambda x.x$. Then, $ID\ ID = ID$, $ID\ y = y$, etc.

The terms that follow λ are the parameters of the function. Following the function parameters, a "." is placed to separate the function body from the arguments. Finally, the function body can be any expression involving the function arguments or a previously defined and named lambda-term. To evaluate a function application, say IDy, simply replace each lexical occurrence of the function parameters with the supplied argument. This process is called *lambda-reduction.* For example:

Example 3.5.6. $(\lambda x.(\lambda x.x)x)y = (\lambda x.x)y = y$.

This knowledge of lambda calculus is sufficient to demonstrate that each function has a fix-point.

Theorem 3.5.1. *Let F be a function in untyped lambda calculus. Then, F has a fix-point.*

Proof. Let $X = \lambda x.F(xx)$ and $W = XX$. Then, we have:

$$\begin{aligned} W &= XX \\ &= (\lambda x.F(xx))X \\ &= F(XX) \\ &= F(W) \end{aligned}$$

Since $W = F(W)$, it follows that W is a fix-point of F. Since F is an arbitrary function, we conclude that every function has a fix-point. □

Now, consider example 3.5.4. We seek to provide a recursive function that generates the traces of the mutual recursion between P_u and P_d that hoists these functions in a similar way to $factorialGenerator$. Define $F_{ud}\left(\left\langle \vec{A}_1, \vec{A}_2 \right\rangle\right) = \left\langle up \rightarrow \vec{A}_2, down \rightarrow \vec{A}_1 \right\rangle$. Observe that $\langle P_u, P_d \rangle$ is a fix-point of F_{ud}.

Theorem 3.5.2. *The Unique Fix-Point (UFP) Theorem: If $Z = f(Z)$ is a fix-point equation generated by any recursion X on trace-sets, and Y is a process whose trace-set satisfies this equation (i.e. $traces(Y) = f(traces(Y))$), then $Y =_T X$.*

Now, we can conclude example 3.5.4. First, observe that $\langle C, D \rangle =_T \langle A, B \rangle \implies C =_T A \text{ and } D =_T B$. So, we seek to show that $\langle P_u, P_d \rangle =_T \left\langle P_1, \vec{A}_2 \right\rangle$ where $\vec{A}_2 = down \rightarrow P_1$. By the UFP, it is sufficient to

demonstrate $\left\langle P_1, \vec{A}_2 \right\rangle$ is a fix-point of F_{ud}. We have:

$$\begin{aligned} F_{ud}\left(\left\langle P_1, \vec{A}_2 \right\rangle\right)_1 &= up \rightarrow \vec{A}_2 \\ &= up \rightarrow down \rightarrow P_1 \\ &= P_1. \end{aligned}$$

Furthermore, the second component of this vector is fixed:

$$\begin{aligned} F_{ud}\left(\left\langle P_1, \vec{A}_2 \right\rangle\right)_2 &= down \rightarrow \vec{A}_1 \\ &= down \rightarrow P_1 \\ &= \vec{A}_2. \end{aligned}$$

So, we have that $F_{ud}\left(\left\langle P_1, \vec{A}_2 \right\rangle\right) = \left\langle P_1, \vec{A}_2 \right\rangle$. By the UFP, it follows that $\left\langle P_1, \vec{A}_2 \right\rangle =_T \langle P_u, P_d \rangle$. Therefore, we see that $P_1 =_T P_u$ as desired.

3.5.3 Traces and Specification

Now that we have laws that allow us to logically construct the traces of a given process, we can create a trace specification for a problem and demonstrate that a given implementation satisfies the specification. Rather than specifying the traces of a process, we can also give a specification in CSP. This approach involves creating a specification, S, and showing that an implementation P satisfies the expression $S \sqsubseteq_T P$.

3.5.4 Additional Trace Notation

In [16], Hoare provides additional notation describing common operations on traces. This includes:

- If t is a finite trace, then $\#t$ denotes the length of t
- If $t \in \Sigma^*$ and $A \subseteq \Sigma$, then $t \upharpoonright A$ denotes t restricted to A. We have that:
  - $\langle\rangle \upharpoonright A = \langle\rangle$
  - $s \langle a \rangle \upharpoonright A = (s \upharpoonright A) \langle a \rangle$ when $a \in A$
  - $s \langle a \rangle \upharpoonright A = (s \upharpoonright A)$ when $a \notin A$

  Example 3.5.7. Let $\Sigma = \{a, b, c, d\}$, $A = \{b, c\}$, and $s = \langle a, b, b, d, c, a \rangle$. Then, $s \upharpoonright A = \langle b, b, c \rangle$.

- If $t = \langle a \rangle s \in \Sigma^*$, then $t' = s$. So, t' denotes the tail of t.

  Example 3.5.8. If $t = \langle a, b, a, c \rangle$, then $t' = \langle b, a, c \rangle$.

- If $t \in \Sigma^*$, then $t \downarrow c$ denotes:
  1. If c is an event in Σ then $t \downarrow c$ is the number of times c appears in t. So, $t \downarrow c = \#(t \upharpoonright \{c\})$.
  2. If c is a channel, then $t \downarrow c$ denotes the sequence of values communicated along c in t.

     Example 3.5.9. Let c and d be channels and consider the trace $t = \langle c.1, d.2, d.3, c.4 \rangle$. Then, $t \downarrow c = \langle 1, 4 \rangle$.

- Let p, s, and t be traces. We write $p \leq t$ to denote that p is a prefix of t, i.e. $t = ps$
- Let P be some process. We write P **sat** $R(tr)$ if $\forall tr \in traces(P)$, $R(tr)$ is true

Specifications as a Logical Assertion

One common approach to process specification is to define a predicate R and demonstrate that it holds for each trace of a particular process P. This is generally accomplished by applying the definitions of traces of a process to deduce that the specific predicate holds.

We consider an early example of process specification provided by Hoare [16]. Foocorp is a vending machine manufacturer who creates wonderful machines that never run out of product. One of Foocorp's clients orders a vending machine that accepts a single quarter and provides the customer with a bar of chocolate. We might specify this vending machine as:

$$VMS = quarter \rightarrow chocolate \rightarrow VMS.$$

The client demands that the vending machine meet two requirements. First, the vending machine must not deposit more chocolate than the number of quarters that it has accepted. Second, the vending machine must be fair to the customer and deposit chocolates incrementally as it receives payment. We

codify these requirements as propositions *NOLOSS* and *FAIR* respectively:

$$NOLOSS = (tr \downarrow chocolate) < (tr \downarrow quarter)$$
$$FAIR = (tr \downarrow quarter) \leq (tr \downarrow chocolate + 1).$$

Finally, we combine these product requirements and simplify the resulting expression:

$$\begin{aligned} VMSPEC &= NOLOSS \wedge FAIR \\ &= (0 \leq (tr \downarrow quarter) - (tr \downarrow chocolate) \leq 1). \end{aligned}$$

Proof. We seek to demonstrate that the vending machine specification *VMS* meets the product requirement *VMSPEC* by induction. First, consider the *STOP* process. Observe that:

$$\frac{STOP \ \mathbf{sat} \ tr = \langle\rangle}{0 \leq (tr \downarrow quarter) - (tr \downarrow chocolate) \leq 1}\ .$$

Now, assume that some trace *X* of *VMS* satisfies *VMSPEC*. We show that after prefixing *X* with the sequence of communications offered by *VMS*, *VMSPEC* is still satisfied:

$$\dfrac{\dfrac{\dfrac{X \ \mathbf{sat} \ (0 \leq (tr \downarrow quarter) - (tr \downarrow chocolate) \leq 1)}{\langle quarter, chocolate\rangle \ X \ \mathbf{sat} \ tr \leq \langle quarter, chocolate\rangle}}{0 \leq ((tr'' \downarrow quarter) - (tr'' \downarrow chocolate)) \leq 1}}{0 \leq ((tr \downarrow quarter) - (tr \downarrow chocolate)) \leq 1}\ .$$

Notice that the last conclusion is precisely *VMSPEC*. Because *X* is an arbitrary trace of *VMS* and $\langle quarter, chocolate\rangle$ *X* satisfies *VMSPEC*, we conclude that all traces of *VMS* satisfy *VMSPEC*. □

3.6 Conclusion

This chapter distills the theory of Process Algebra into a palatable form. We presented sequent calculus, the logical tool used in describing the traces of processes. Then, we introduced a basic form of process algebra to serve as a foundation when considering traces and more advanced communication styles. Next, Hoare's Communicating Sequential Processes is described in great detail. In particular, we described the operations of CSP and showed how traces are generated from a CSP process. Finally, we described the procedure of specifying the behavior of more complex recursive processes.

4

Formalizing REST with CSP

Xi Wu and Huibiao Zhu provide the initial approach to the formalization of REST with CSP [22]. We extend this approach in a novel way. The approach of Wu et al consolidates the various resources into a single process. Because we desire to model microservices, we seek to model multiple resource providers. With this approach, the origin server can be thought of as an *API Gateway* that controls dispatch to various resources. We begin by introducing the work and notation of Wu et al that is used as a basis for our refinements. Then, we introduce the definition of the various components that we model, including our extension.

4.1 Introduction to Terminology

Our approach is to model the several distinct components of RESTful systems using CSP processes. For an overview of the role of each component, consult Chapter 2. These processes include:

1. *UserAgent*
2. *Intermediary*
3. *OriginServer*
4. Resources *Resource_0*, *Resource_1*, ..., *Resource_n* corresponding to multiple service providers
5. Caches corresponding to all processes except resources

4.1.1 Communication Channels

Each component of a RESTful system communicates with its corresponding cache over channels. Here, we explicitly name these channels as:

1. *InUAC*, *UserAgent's* internal channel allowing communication with its cache
2. *InIC*, *Intermediary's* internal channel allowing communication with its cache
3. *InOSC*, *OriginServer's* internal channel allowing communication with its cache

We model the communication of data through a network as data through channels. In accordance with the layered design of REST, we permit only adjacent layers in the system to communicate with one another. We define these channels as:

1. *ComUAI*, the channel that the User Agent and Intermediary use for communication

2. *ComIOS*, the channel that the Intermediary and Origin Server communicate over

3. $ComSR_0, ComSR_1, \ldots, ComSR_n$, the channels for communication between the Origin Server and n resources

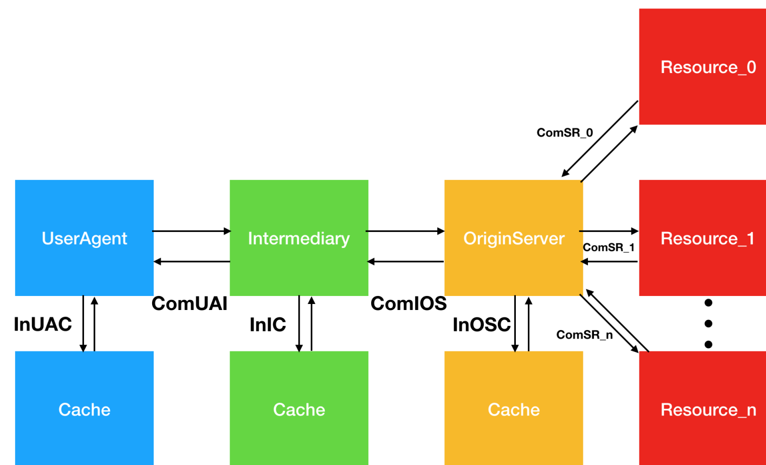


Figure 4.1: Overview of REST modeled in CSP.

Figure 4.1 provides a graphical illustration of this architecture to accompany these descriptions. Graphically, we visualize that each component in the system is only capable of communicating with immediately adjacent components. Furthermore, each of the component's caches is totally independent. Finally, we see that the Origin Server is responsible for distributing communication (presumably originating from the Intermediary)

over the various resources. This might indicate load balancing, an API gateway, etc.

4.1.2 Set Semantics

Table 4.1 describes the sets that are used by the CSP model of REST. These are useful when describing the construction of messages and trace predicates over RESTful systems, so we refer to this table throughout this chapter.

Table 4.1: The various sets of objects found in the CSP model of REST.

Set	Description	Set	Description
User	all user agents in system	*Server*	all origin servers in system
Representation	all resource representations	*SDInformation*	all self-descriptive messages
Cache	all caches in the system	*Intermediary*	all proxies and gateways
Operation	$\{get, put, post, delete\}$	*Hypermedia*	all hypermedia resources
Resource	all resources	*ID*	all resource identifiers

REST is concerned with how and what messages should be exchanged in a client-server architecture. We formally define the messages that are exchanged in a protocol-independent fashion. Any protocol that supports REST (i.e. HTTP, COAP) supports this abstract message definition.

Much of the context of a message is stored in its *interface* component. Indeed, the information most important to the application itself is stored in this location. Consult table 4.1 for descriptions of the sets referred to in the next few definitions.

Definition 4.1.1. *The **interface** component of a message is a 4-tuple* $(id, oper, data, link)$ *where* $id \in ID$, $oper \in Operation$, $data \in SDInformation$, *and* $link \subseteq HyperMedia$.

Definition 4.1.2. *A **request message** is defined as a 3-tuple* $(interface, sender, receiver)$*, where* $sender \in User \cup Intermediary \cup Server$*,* $receiver \in Intermediary \cup Server \cup \{Resource_0, Resource_1, \ldots, Resource_n\}$*, and* $interface$ *is a valid message interface. The set of all message requests is denoted as* MSG_{req}*.*

Note that for convenience, we usually write $msg_{req}.interface.sender.receiver$ in place of the usual notation for a tuple.

Definition 4.1.3. *The **content** component of a response message is defined as a 4-tuple* $(id, repr, data, link)$*, where* $id \in ID$*,* $repr \in Representation$*,* $data \in SDInformation$*, and* $link \subseteq HyperMedia$*.*

Definition 4.1.4. *A **response message** is defined as a 3-tuple* $(content, sender, receiver)$*, where content is the previously described content component of a message,* $sender \in Cache \cup Intermediary \cup Server \cup \{Resource_0, Resource_1, \ldots, Resource_n\}$*, and* $receiver \in User \cup Intermediary \cup Server$*. The set of all response messages is denoted as* MSG_{rep}*.*

Once again, we usually write something similar to $msg_{rep}.content.sender.receiver$ rather than use the tuple notation, as it appears more natural alongside terms in CSP.

Definition 4.1.5. *The set of all messages in a RESTful system is defined as:*

$$MSG = MSG_{req} \cup MSG_{rep}.$$

4.2 Process Definitions

In this section we present the definition of the various processes that compose a RESTful system. We begin by providing a top-level overview of the entire system, and then gradually provide the specific model for each component.

Definition 4.2.1. *The CSP model for a generic RESTful system is defined as:*

$$\begin{aligned} System = \; & UserAgent \; [|\{ComUAI\}|] \; Intermediary \; [|\{ComIOS\}|] \\ & OriginServer \; [|\{ComSR_0,\; ComSR_1,\; \ldots, ComSR_n\}|] \\ & (Resource_0, Resource_1, \ldots, Resource_n). \end{aligned}$$

The User Agent process is given an *interface* that is used to construct a request message. We use the notation $interface'$ to denote the next interface in the system. If $interface$ specifies that a *get* request is to be sent, then *UserAgent* first checks its cache to see if there is an existing saved reply that can be reused. When no such reply exists, then *UserAgent* communicates the request to *Intermediary* over *ComUAI*. Then, if the response from *Intermediary* indicates that the resource can be cached, then *UserAgent* updates its local cache before proceeding to the next interface in the system. Meanwhile, when $interface$ does not specify a *get* request, then *UserAgent* immediately communicates with *intermediary*, since other requests can not be cached. Listing 4.1 provides a definition of the UserAgent process. *U* denotes the UserAgent identifier, *C* denotes the Cache identifier, and *I* is the Intermediary identifier.

```
UserAgent = UserAgent(interface) [|{InUAC}|] Cache(interface)
UserAgent(interface) =
```

```
((InUAC ! request.interface.U.C →
   InUAC?reply.content.C.U →
   UserAgent(interface'))
     ◃ (content.id != NULL) ▹ ❶
 (ComUAI!request.interface.U.I →
  ComUAI?reply.content.I.U →
  ((InUAC!store_request.content.U.C →
    UserAgent(interface'))
      ◃ (cacheable? content.data.flag) ▹ ❷
    UserAgent(interface'))))
  ◃(interface.oper == get)▹ ❸
    (ComUAI!request.interface.U.I →
    ComUAI?reply.content.I.U →
    UserAgent(interface'))
```

❶ If the resource is present in the User Agent's cache, return the cached value

❷ Otherwise, request the resource from intermediary. If the response is cacheable, communicate the response to the cache for storage

❸ If the request's method is not *get*, then immediately communicate with Intermediary

Listing 4.1: UserAgent process definition.

We are interested in describing the cache for each of these components.

Each cache operates essentially the same way, so we describe the process in very general terms. *Cache* listens over the communication channel with its corresponding component. When a store request is sent, *Cache* simply stores the interface of the request. Otherwise, *Cache* checks its store for the interface in the request message. When *Cache* contains the requested resource, it communicates the resource with its corresponding process. *Cache* indicates a missing resource by responding with a *NULL* resource ID.

Definition 4.2.2. *Let InXC denote the communication channel between a REST component X and its corresponding cache, C. We define* ***Cache*** *as:*

```
Cache(interface) =
      (InXC?request.interface.X.C → store(interface))
       ◃ (request == store_request) ▹
      ((InXC!reply.content.C.X → Cache(interface))
       ◃ ((find interface.id) ∧ interface.oper == get) ▹
      (InUAC!reply.(content.id = NULL).C.X → Cache(interface)))
```

where store implements the cache storage functionality and $find$ locates a resource by its identifier and binds it to the lexical variable content.

The *Intermediary* process corresponds to components similar to application-level load balancers; that is, a load balancer that distributes traffic across a set of origin servers. Similarly, *Intermediary* could be viewed as a firewall or other gateway to the origin server. *Intermediary* operates by listening for requests from a *UserAgent* process. When *UserAgent* performs a *get* request, then *Intermediary* checks its cache for an appropriate saved resource. If that cache-check fails, *Intermediary* must slightly modify the

request message by changing the sender and receiver fields. The modified message is forwarded to *OriginServer*, and the response is cached if appropriate. Finally, the response is communicated to the original *UserAgent*. If the request does not perform a *get* operation, then *Intermediary* forwards the traffic along, similar to the failed cached lookup operation. Listing 4.2 provides the definition of the Intermediary process. Here, *U* denotes the UserAgent identifier, *I* denotes the Intermediary identifier, *C* denotes the identifier of the Intermediary's cache, and *S* is the identity of the origin server.

```
Intermediary = Intermediary(interface) [|{InUAC}|] Cache(interface)
Intermediary(interface) =
    ComUAI?request.interface.U.I →
    ((IncIC!request.interface.I.C →
      InIC?reply.content.C.I →
      ComUAI!reply.content.I.U →
      Intermediary(interface))
        ◃ (content.id != NULL) ▹ ❶
     ((ComIOS!changeFormat(request).interface'.I.S →
       ComIOS?reply.content.S.I →
       InIC!store_request.content.I.C →
       ComUAI!changeFormat(reply).content'.I.U →
       Intermediary(interface))
         ◃ (content.data.flag == cacheable) ▹ ❷
      (ComIOS!changeFormat(request).interface'.I.S →
       ComIOS?reply.content.S.I →
```

```
    ComUAI!changeFormat(reply).content'.I.U →
    Intermediary(interface))))
    ◃ (interface.oper == get) ▹ ❸
  (ComIOS!changeFormat(request).interface'.I.S →
   ComIOS?reply.content.S.I →
   ComUAI!changeFormat(reply).content'.I.U →
   Intermediary(interface))
```

❶ Receive a get request over *ComUAI*. If the cache contains a response, issue the cached resource as a reply to *UserAgent*

❷ If the resource is not cached, forward the request to *OriginServer*. If it is possible to cache the response, do so

❸ If we are not working with a get request, then we must immediately forward the request to *OriginServer*

Listing 4.2: Intermediary process definition.

We depart from the model described in [22] for the origin server. We permit resources to be composed of several processes, called *Resource*_0, *Resource*_1, ..., *Resource*_*n*. The origin server provides some function, $c_for(interface)$ that provides the appropriate channel to communicate over, and $r_for(interface)$ that provides the appropriate resource identifier for the given interface. This matches our intuition that routing is simply a function.

The basic functionality of the origin server is to route requests over

ComIOS to the corresponding resource. Because the origin server is ultimately the final destination and resources are simply implementation-details, we are not concerned with updating any fields in the message. Once a get request is received, the origin server checks its cache for the resource. Should the cache check fail, then the origin server communicates with the resource. Finally, the origin server passes the communication from the resource back to the intermediary. Listing 4.3 is the CSP specification of the Origin Server process. Here, *I* is understood to be the Intermediary identifier, *S* is the Origin Server identifier, and *C* is the origin server's cache.

```
OriginServer = Server(interface) [|{InUAC}|] Cache(interface)
Server(interface) = ComIOS?request.interface.I.S →
(InOSC!request.interface.S.C → InOSC?reply.content.C.S →
 ((ComIOS!reply.content.S.I → Server(interface))
     ◃ (content.id != NULL) ▹ ❶
  (c_for(interface)!request.interface.S.r_for(interface) →
   c_for(interface)?reply.content.r_for(interface).S →
   (InOSC!store_request.content.S.C →
    ComIOS!reply.content.S.I →
    Server(interface)))
      ◃ (content.data.flag == cacheable) ▹ ❷
    ComIOS!reply.content.S.I))
   ◃ (interface.oper == get) ▹ ❸
(c_for(interface)!request.interface.S.r_for(interface) →
 c_for(interface)?reply.content.r_for(interface).S →
```

```
ComIOS!reply.content.S.I → Server(interface)).
```

❶ If the cache holds the resource, respond with the cached value to *Intermediary*

❷ If the cache does not contain the resource, reach out to the resource provider. If it is possible to cache the response, do so. Regardless, respond to *Intermediary* with the resource

❸ If the request is not a get operation, then reach out to the resource provider for the appropriate response

Listing 4.3: OriginServer process definition.

There is flexibility as to how individual resources are modeled. So long as a *resource provider* (the process responsible for creating a response for a particular set of resources) is given a unique numeric identifier and communicates over the corresponding resource channel, we are not concerned with how the internals function specifically.

4.3 Conclusion

The principles of REST are modeled easily as CSP processes. We extend the approach provided by Wu et al in [22] to adequately express a multi-process environment. This approach permits us to employ the techniques presented in Chapter 3 to describe and verify the behavior of a RESTful application

composed of micro-services. Hence, we are able to create behavioral guarantees of *general* RESTful systems and particular applications.

5

Concurrency Abstracted

Besides Process Algebra, there are alternative formal theories that seek to model concurrency. Examples of these theories include Petri Nets and Asynchronous Transition Systems. Traces prove to be a fairly useful model for describing the behavior for each of these systems. By developing a sophisticated theory of traces we are able to generalize the behavior of concurrent systems in a model-independent fashion. Here, we begin by reviewing elementary algebraic concepts. Then, we discuss the theory of traces. The structures present in the behavioral-semantics of CSP are formally defined as *Hoare structures.* We present foundational concepts in category theory to demonstrate the important relationship between trace and Hoare structures.

5.1 Trace Theory and Other Formalizations

5.1.1 Preliminary Definitions

We begin by reviewing basic definitions of abstract algebra.

Definition 5.1.1. *Let M be a set closed under the operator $*$. M is a **monoid** if:*

1. *There exists an identity element $e \in M$ such that $\forall m \in M$, $e * m = m = m * e$*
2. *For all $a, b, c \in M$ we have that $(a * b) * c = a * (b * c)$*

Usually, we do not not include the $*$ symbol when writing expressions over a monoid.

Example 5.1.1. Consider the set $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ with the operator $+$. $\mathbb{N}_0$ is a monoid, with 0 functioning as the identity element.

Definition 5.1.2. *Let $f : M \to M'$ be a function over monoids M and M' with identity elements $e \in M$ and $e' \in M'$. If $\forall a, b \in M$ we have that $f(ab) = f(a)f(b)$ then f is a **homomorphism**.*

Example 5.1.2. Let M and M' be monoids, with respective identity elements e and e'. Define the operator $*$ on the *product monoid* on $M \times M'$ as $(m_1, m_1') * (m_2, m_2') = (m_1 m_2, m_1' m_2')$. The identity element of this monoid is (e, e'). Now, consider $\phi : M' \to M \times M'$ defined as $\phi(m') = (e, m')$. We see that ϕ is a homomorphism.

Definition 5.1.3. *Let $f : M \to M'$ be a homomorphism. If f is onto and injective, then f is an **isomorphism between monoids**.*

Definition 5.1.4. *Let Σ be a set. The* ***free monoid*** *on Σ is defined on all possible finite sequences under the operation of string concatenation. The identity element of the free monoid is the empty string, denoted ϵ.*

Example 5.1.3. Let $\Sigma = \{a, b\}$. The free monoid on Σ includes, among other elements, the subset $\{\epsilon, \langle ab \rangle, \langle bab \rangle, \ldots\}$.

Definition 5.1.5. *Let M be a monoid and ρ be an equivalence relation on M. Then, the quotient of M modulo ρ (denoted M/ρ) is the* ***quotient monoid of*** *M* ***with respect to*** *ρ. This induces the* ***natural homomorphism*** *$f : M \to M/\rho$ given by $f(x) = [x]$, i.e. $f(x)$ is defined as the equivalence class of x.*

Example 5.1.4. Consider $\mathbb{Z}/\mathbb{Z}_n$, the integers modulo n. The congruence *modulo n* places each integer into its equivalence class modulo n. Note that each integer has an inverse in this set, and so it follows that this set has a little more structure than a monoid, and it is called *group*. But all groups are monoids, so this is a simple example of a quotient monoid.

5.1.2 The Trace Monoid

Recall that a behavioral-semantics based approach to process verification holds that two processes are equivalent if they produce the same set of traces. This is reified by Unique Fixed Point theorem that states that if two processes satisfy the same recursive equation on their trace sets, then they are trace equivalent. To cope with the possibility of concurrent behavior, we acknowledge that a process might communicate a different sequence of events between different executions. The intuition is that if two primitive actions in

the process alphabet may be swapped in the traces of that particular process, then those actions must execute independently of one another. Here, we formalize this intuition and describe the result using traditional algebraic methods. The definitive resource discussing the theory of traces is [11], which should be consulted for further reference.

Definition 5.1.6. *Let A be a set with a relation D. If D is symmetric and reflexive, then it is called a* ***dependency relation*** *on A.*

Example 5.1.5. Let $\Sigma = \{a, b, c\}$. Then,
$D = (\{a, b\} \times \{a, b\}) \cup (\{a, c\} \times \{a, c\}) = \{(a, a), (b, b), (c, c), (a, b), (b, a), (a, c), (c, a)\}$ is a dependency relation on Σ.

Definition 5.1.7. *Let A be a set with a dependency relation D. The* ***independence relation induced by D on A*** *is defined as $I_D = (A \times A) - D$.*

Example 5.1.6. Let $\Sigma = \{a, b, c\}$ and $D = (\{a, b\} \times \{a, b\}) \cup (\{a, c\} \times \{a, c\})$ (as in Example 5.1.5.) The independence relation induced by D is $I_D = \{(b, c), (c, b)\}$.

Definition 5.1.8. *Let D be a dependency on Σ. We define* ***trace equivalence*** *as the least congruence $\equiv_D$ (the transitive, reflexive, and symmetric closure of D) in the free monoid on Σ such that $\forall a, b \in \Sigma$*

$$(a, b) \in I_D \implies ab \equiv_D ba.$$

Equivalence classes of $\equiv_D$ are called ***traces*** *over D.*

Example 5.1.7. Let $\Sigma = \{a, b, c\}$ and $D = (\{a, b\} \times \{a, b\}) \cup (\{a, c\} \times \{a, c\})$ (as above in example 5.1.5.) The trace over D given by the string $abbca$ is
$[abbca] = \{abbca, abcba, acbba\}$.

In other words, the independence relation states that two actions b and c are independent, and hence can be safely swapped.

Definition 5.1.9. *For an alphabet Σ and dependency relation D, the free partially commutative quotient monoid $M(D) = \Sigma^*/{\equiv_D}$ is called the* ***trace monoid****.*

When describing a concurrent system, usually not all possible strings over the trace monoid are permitted. So, we define a sub-collection of the trace monoid called a *trace structure.*

Definition 5.1.10. *A* ***trace structure*** *is a tuple $T = (M, \Sigma, I)$ where (Σ, I) describes an independence relation, and M is a subset of the trace monoid generated on (Σ, I) with the following properties for $t, t' \in \Sigma^*$ and $a, b \in \Sigma$:*

1. *consistency: if $t \equiv_{D_I} t'$ and $t' \in M$, then $t \in M$*
2. *prefix-closed: if $t \langle a \rangle \in M$, then $t \in M$*
3. *proper: if $t \langle a \rangle, t \langle b \rangle \in M$ and aIb, then $t \langle ab \rangle \in M$*

Example 5.1.8. Let $\Sigma = \{a, b, c\}$ and $D = (\{a, b\} \times \{a, b\}) \cup (\{a, c\} \times \{a, c\})$ (as in example 5.1.5.) Define $M = \{abc, acb, a, ac, ab\}$. Then, (M, Σ, I_D) is a trace structure.

We are also interested in describing behavior-preserving morphisms between different trace structures. First, we describe how to translate strings from one alphabet to another given a partial alphabet map λ.

Definition 5.1.11. *A function $\lambda : \Sigma \to \Sigma'$ extends over strings with the function:*

$$\widehat{\lambda}(s \langle a \rangle) = \begin{cases} \widehat{\lambda}(s)\lambda(a) & if\ \lambda(a)\ defined \\ \widehat{\lambda}(s) & if\ \lambda(a)\ undefined \end{cases}$$

Definition 5.1.12. *A **morphism** of trace structures* $(M, \Sigma, I) \to (M', \Sigma', I')$ *is induced by a partial function* $\lambda : \Sigma \to \Sigma'$ *that satisfies:*

1. *independence preservation: let* $a, b \in \Sigma$. *If* $\lambda(a)$ *and* $\lambda(b)$ *are defined, then* $\lambda(a) I' \lambda(b)$
2. *string preservation: if* $s \in M$ *then* $\widehat{\lambda}(s) \in M'$

Example 5.1.9. Let $\Sigma = \{a, b, c\}$ and $D = (\{a, b\} \times \{a, b\}) \cup (\{a, c\} \times \{a, c\})$ (as in example 5.1.5.) Further, let $\Sigma' = \{a', b', c'\}$ and $D' = (\{a', b'\} \times \{a', b'\}) \cup (\{a', c'\} \times \{a', c'\})$. Define $\lambda(t : \Sigma) = t'$. Since λ is simply a renaming of symbols of Σ, we have string and independence preservation. Hence, λ is a morphism between trace structures.

This example demonstrates the fundamental mode of trace morphisms: symbol renaming. In certain instances, a trace morphism may also *embed* a trace structure into another, in the sense that the image of the domain is strictly finer than the range.

5.1.3 Hoare Structures

In chapter 3, we used the term *trace* to refer to objects that do not quite meet the trace criteria we have established. The primary difference is that they lack an explicit dependency relation between the various atomic actions. We formalize what we have been calling traces using the notion of *Hoare structures* described in [11].

Definition 5.1.13. *A **Hoare structure** is a tuple* (H, Σ) *where* Σ *is a set of atomic actions and* H *is a nonempty, prefix-closed subset of the free monoid* Σ^*.

Once again, we are interested in defining behavior-preseving morphisms between Hoare structures.

Definition 5.1.14. *A **morphism** between Hoare structures* $(H, \Sigma) \to (H, \Sigma')$ *is induced by a partial function* $\lambda : \Sigma \to \Sigma'$ *so that for all strings in* H *we have* $\widehat{\lambda}(s) \in H'$.

It is easier to reason about trace structures because the explicit dependency relation captures the intent regarding the dependency of particular actions. Intuitively, we might guess that trace structures and Hoare structures have a special relationship. Indeed, this is the case. To discuss this, we introduce some fundamental ideas of *category theory* and demonstrate how trace structures may be viewed as an abstraction over Hoare structures.

5.2 Trace Structures and Hoare Structures

5.2.1 Some Category Theory

Here, we introduce a few fundamental concepts in category theory. For more specific details, [17] provides a fair introduction to category theory and is reasonably accessible. We seek to rigorously construct some notion of an *abstract model*. Specifically, we want to be able to construct morphisms between different theories of concurrency. We introduce the ideas of *adjoints* and *coreflectors* to accomplish this.

Definition 5.2.1. *A **directed graph** is composed of a set* O *of objects and* A *of arrows, alongside two functions. These functions are domain, which maps arrows to their domain object, and codomain, which maps arrows to their corresponding codomain.*

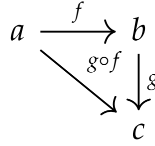


Figure 5.1: A diagram to visualize objects in a category.

Definition 5.2.2. *A **category** is a directed graph (O, A) with two additional properties. First, for each $o \in O$, there is an identity arrow $id_o : o \to o$. Second, let $a, b, c \in O$ with $a \xrightarrow{f} b \xrightarrow{g} c$. Then, there exists an arrow $a \xrightarrow{g \circ f} c$.*

Figure 5.1 provides a diagram of some category. Here, there are objects a, b, c and arrows $f, g, g \circ f$. The identity arrows are omitted, as is common in category diagrams. Those arrows would just be loops on each object.

Example 5.2.1. Consider the category of all sets. Arrows in this category are simply set-maps.

Example 5.2.2. Consider the category of all monoids. Arrows in this category are homomorphisms.

Definition 5.2.3. *A **functor** is a morphism between categories. Particularly, if C and B are categories, a functor $T : C \to B$ consists of:*

1. ***object function** - for each $c \in C$, there is a corresponding $b \in B$ so that $Tc = b$ and $T(id_c) = id_b$*
2. ***arrow function** - consider two arrows in C, $f : c \to c'$ and $g : c' \to c''$. Then, we have $Tf : Tc \to Tc'$ and $Tg : Tc' \to Tc''$ so that $T(g \circ f) = Tg \circ Tf$.*

Figure 5.2 is a diagram of a functor T between two categories. We visualize quite easily that T preserves the composition of g and f across the categories.

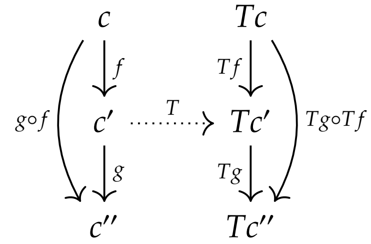


Figure 5.2: Diagram demonstrating the properties of a functor.

This matches our intuition that a functor (being a morphism between categories) should generally preserve the properties associated with *all* categories.

Example 5.2.3. Consider the category of all monoids M and the category of all sets S. There is a functor called the *forgetful functor* $U : M \to S$ that "forgets" the underlying structure of its domain. In this case, U sends each monoid to the set containing the elements.

We seek to generalize the notion of isomorphisms between objects in arbitrary categories. If the objects in the category are not sets, our previous definition does not apply. Observe that isomorphisms between monoids are behavior-preserving bijections. Generally, we can encapsulate this behavior by examining the inverses of arrows and their composition.

Definition 5.2.4. *Let C be a category with objects a and b. An arrow $e : a \to b$ is **invertible** if there is a morphism $e' : b \to a$ so that $e'e = id_a$ and $ee' = id_b$. These arrows are considered **isomorphisms**.*

Definition 5.2.5. *Let C be a category with objects a and b. We say that a and b are **isomorphic** (denoted $a \cong b$) if there is an invertible arrow $e : a \to b$.*

$$\begin{array}{ccc} m_o & \xrightarrow{f_0} & R(m_1) \\ \downarrow u & \nearrow R(f_1) & \\ R \circ L(m_0) & & \end{array}$$

Figure 5.3: Diagram describing adjunction.

We introduce the concept of adjunctions to describe how to embed categories into each other.

Definition 5.2.6. *Let M_0 and M_1 be categories, with functors $L : M_0 \to M_1$ and $R : M_1 \to M_0$. L and R form an **adjunction** if for any object $m_0 \in M_0$, there is a morphism $u : m_0 \to R \circ L(m_0)$ (called the **unit** at m_0) such that for any object m_1 of M_1, if there is a morphism $f_0 : m_0 \to R(m_1)$, then there is a unique morphism $f_1 : L(m_0) \to m_1$ so that $f_0 = R(f_1) \circ u$.*

Figure 5.3 is a diagram describing the concept of adjunction. We see that the central idea is that each object m_o in M_0 has an arrow to $R \circ L(m_0)$. Then, if m_1 is an object in M_1, and there is some arrow from m_0 to $R(m_1)$, there is a unique arrow f_1 in M_1 so that the diagram commutes.

Example 5.2.4. Consider the category of all sets S and the category of monoids M. Arrows in S are set-maps, while arrows on M are homomorphisms. We have the functor $F : S \to M$ and $U : M \to S$, with U being the familiar forgetful functor. We define F as:

$$F\ X = X^*$$

$$F\ f = \hat{f}$$

where $\hat{f}(s\langle a\rangle) = \hat{f}(s)f(a)$. So, F maps sets to their corresponding free monoid. Now, if $u : X \to U \circ F(X)$, define $f_1 : F(X) \to M$ as $f_1 = F(f) = \hat{f}$. This yields the commutative diagram:

$$\begin{array}{ccc} X & \xrightarrow{\;f\;} & U(M) \\ \downarrow u & \nearrow U(f_1) & \\ U \circ F(X) = X* & & \end{array}$$

which indicates that S and M form an adjunction. Notice that the unit must be the canonical embedding of X into X^*.

Definition 5.2.7. *If each unit of an adjunction is an isomorphism, then the adjunction is a* ***coreflection****.*

Intuitively, an adjunction (particularly a coreflection) describes how objects can be expanded and collapsed in a formulaic method. Our example demonstrates that free monoids and sets can be expanded and collapsed into one another. A similar result applies to Trace and Hoare structures.

5.2.2 The Relationship

Now, we have a well-defined general model of concurrency with the trace monoid and trace structures. Indeed, this model serves as an abstraction for several popular approaches to describing concurrent systems, including Petri nets and asynchronous transition systems [11]. Here, we formally describe trace structures as an abstraction over the Hoare structures that are used when analyzing CSP using behavioral semantics.

Definition 5.2.8. **T** *denotes the category of trace structures with their behavior-preserving morphisms.* **H** *signifies the category of Hoare structures with their behavior-preserving morphisms.*

We can define functors between these categories $ht : \mathbf{H} \to \mathbf{T}$ and $th : \mathbf{T} \to \mathbf{H}$ on objects as:

$$ht(H, \Sigma) = (H, \Sigma, \emptyset)$$
$$th(M, \Sigma, I) = (M, \Sigma).$$

Similarly, these functors are defined on morphisms λ as:

$$ht(\lambda) = \lambda$$
$$th(\lambda) = \lambda.$$

Notice that th is simply the forgetful functor. Meanwhile, Hoare structures are mapped by ht as a trace structure lacking an explicit dependence relation.

Theorem 5.2.1. *Trace and Hoare structures are coreflective.*

Proof. Suppose that (H, Σ) is an arbitrary Hoare structure. As Figure 5.4 demonstrates, we have an arrow $id_{(H,\Sigma)}$ from (H, Σ) to $(th \circ ht)(H, \Sigma)$. Clearly, this is an isomorphism. Now, let $f_0 : (H, \Sigma) \to th(M, \Sigma', I)$. We want to find some morphism $f_1 : ht(H, \Sigma) \to (M, \Sigma', I)$. Observe that $ht(H, \Sigma) = (H, \Sigma, \emptyset)$. Because f_0 is a morphism on Hoare structures, it follows that it preserves strings. Since there is no independence relationship to preserve, then $ht(f_0)$ adequately forms the commutative diagram in Figure 5.4. Hence, **T** and **H**

$$\begin{array}{ccc} (H,\Sigma) & \xrightarrow{f_0} & th(M,\Sigma',I) \\ \big\downarrow id_{(H,\Sigma)} & \nearrow_{th(f_1)=f_0} & \\ (th \circ ht)(H,\Sigma) = (H,\Sigma) & & \end{array}$$

Figure 5.4: Diagram of relationship between trace and Hoare structures.

form an adjunction. Furthermore, all units are isomorphic since units are identity morphisms. Thus, **T** and **H** are coreflective. □

5.3 Other Models of Concurrency

A system is concurrent when there are possibly two or more events happening simultaneously. Note that we need not restrict this behavior to computer systems. For instance, we might be interested in describing the behavior of chemical reactions. To accomplish this, we use Petri nets. It is demonstrated in [11] that the trace monoid is an abstraction of Petri nets.

As another example, a software engineer might describe a simple application with a diagram consisting of a set of states and asynchronous actions that trigger the transition between these states. Such a diagram is called a *labeled transition system.* Once again, trace structures are adequate to describe labeled transition systems.

Since trace structures are coreflective with several models of concurrency, it is possible to translate from one concurrency model to another. This result enables each of the models to share results and provides a common framework to describe concurrent systems. Furthermore, when one model is more appropriate to use when solving a particular problem, it may be safely

incorporated into a solution employing heterogeneous concurrency systems.

5.4 Conclusion

The trace monoid and trace structures are an abstract model that is useful when analyzing the behavior of concurrent systems. The traces of CSP are formalized as a Hoare structure, and we have the result that the categories of trace and Hoare structures are coreflective. This implies that the theory of Hoare structures can be embedded into trace structures. There are several other common models of concurrency that are also coreflective to trace structures, thus demonstrating that these models are roughly equivalent.

6

Description of Software

There are two subcomponents of the software portion of this IS. A case study of a RESTful system is demontrated by a queue application. We explore the usage of the REST architectural constraints in this system, and present a CSP model. Second, we exploit the structure of RESTful systems to dynamically capture error states in running RESTful applications and create new unit tests. We present that system as a library that can easily be included in existing systems. Finally, we discuss potential future work for this project.

6.1 Overview of Tools

All software components of this project are written in the Clojure programming language [6]. Clojure is a member of the Lisp family of programming languages with a special emphasis on functional programming through immutable data structures. In particular, Clojure employs a shared-memory model with destructive operations coordinated through a

highly sophisticated software transactional memory system. This permits us to create software composed of sequences of "atomic" data mutations, placing us close to the model of CSP.

Clojure software is generally built using Leiningen ("Lein") [18]. Lein is similar to tools such as Make and Maven [15] [14]. Make is traditionally used to build C and C++ software, while Maven is a popular choice as a build tool for Java software. Since Clojure is hosted on the Java Virtual Machine, it is actually possible to generate Maven's project object model using Lein, thus permitting existing Java applications to make use of Clojure. Similarly, Lein is capable of taking libraries from Maven repositories and integrating them into new Clojure applications.

Two major Clojure libraries are employed to construct the software. Compojure [2] is a project that enables the declarative configuration of web systems. By this, we mean that Compojure invokes forms corresponding to matched URLs according to a stateless specification. The actual request handling is performed by Ring, a library that abstracts the details of HTTP into a simple API [5].

All of the complexities involved with managing the installation of the Java Virtual Machine and Lein are handled with a tool called Docker [7]. Docker is a *container engine* that executes processes in an isolated environment. Docker is capable of bundling the various components of a complete software system into a portable *image* that is easily shared across heterogeneous computer systems. We use Docker to package and deploy our applications.

6.2 Case Study of RESTful Application

We consider a queue application as an example of a RESTful application. We walk through the story of a client interacting with the server to demonstrate an important emergent property of REST, *Hypertext as the Engine of Application State* (HATEOAS) [19]. In applications satisfying HATEOAS, a client interacts with a server through an initial request to a widely-known root resource (i.e. "/") and progresses through the various states of an application by following hyperlinks. Notice that this architectural style generally corresponds with how many web applications are built; the website's homepage is visited and the user navigates through different application states by following hyperlinks.

The client and server use JavaScript Object Notation (JSON) as their data-interchange format. JSON is a simple data format that is a subset of the JavaScript programming language [4]. JSON supports several primitive data types, including:

- Unicode strings, i.e. "A String"
- Numbers, i.e. 2.5 (note that there are no specific floating point types, just a generic Number type)
- Booleans, true or false
- Null, or no value
- Objects, i.e. {"Key": "Value", "otherKey" : 2}
- Heterogeneous arrays, i.e. [1, 2, "foo"]

We include snippets of JSON throughout this chapter to demonstrate potential interactions between the client and server.

6.2.1 Entrypoint

Clients begin any interaction by first making a GET request to "/api/1.0/queues". The server response takes the form of:

```
{"Version": "1.0",
 "Queues": [{"QueueName": "queue0",
             "QueueURL": "/api/1.0/queues/queue0"}, ...],
 "NewQueueURL": "/api/1.0/queues"}
```

The client has several options with how to proceed. It has enough information to either create a new queue, place or receive messages over an existing queue, or delete an existing queue. Notice that we always include a *version* field in our responses that allows the client to be made aware of any modifications to the data interchange format. Supporting legacy clients is easily achieved by routing the client's request based off the version field in the URL, which is commonly accomplished with load balancers.

6.2.2 Creating a New Queue

After initial successful contact with the server, the client might wish to create a new queue. The "NewQueueURL" field informs the client what URL to use to perform this operation. Hence, the client is not tied to the particular implementation details of the server. The well-defined semantics of REST

indicate to the client that in order to create a new queue it must POST to this URL.

The client POSTS to the value of "NewQueueURL" a message in the format:

```
{"QueueName": <queue-name>}.
```

If the queue does not exist, then the server responds with a message:

```
{"Status": "Success", "Version": "1.0",
 "QueueURL": <queue-url>}.
```

Meanwhile, if the queue already exists, the server does not modify the existing resource. Instead, a message is returned to the client indicating this condition:

```
{"Status": "QueueExists", "Version": "1.0",
 "QueueURL": <queue-url>}.
```

6.2.3 Writing to a Queue

Once the URL of a queue is discovered (either by creation or through a GET request to the entrypoint) the client enqueues by performing a POST request directed at the URL. The client POSTs to this URL a message similar to:

```
{"Object": <json-object>}.
```

Assuming the request is to a valid queue, the server responds with a message in the form:

```
{"Status": "Success", "Version": "1.0",
 "QueueURL": <queue-url>}.
```

If the queue does not exist in the system, a 404 status code is returned with an appropriate error message:

```
{"Status": "DNE", "Version": "1.0"}.
```

6.2.4 Retrieving Objects from a Queue

GET requests are sent to a queue URL to dequeue objects. Once again, this URL is discovered dynamically by the client through its sequence of interactions. When there is an object in the queue, the server responds with a success message containing the message:

```
{"Status": "Success", "Object": <json-object>,
 "Version": "1.0"}
```

If no object is present in the queue, this information is conveyed through an appropriate response:

```
{"Status": "QueueEmpty", "Version": "1.0"}.
```

6.2.5 Deleting a Queue

HTTP provides a DELETE method that is used to remove resources. In order to delete a queue, the client first discovers the queue's corresponding URL through its sequence of interactions with the server. Then, the client sends a DELETE request to the discovered URL and the queue is removed. The server responds with a success message in the form:

```
{"Status": "Success", "Version": "1.0"}.
```

If the queue does not exist, then the server responds with a 404 status code with a body of:

```
{"Status": "DNE", "Version": "1.0"}.
```

6.3 Dynamic Error State Observations

To support rapid debugging of RESTful applications using the Ring library in Clojure, we develop a piece of middleware that captures error states and automatically generates unit tests. *Middleware* is an interesting concept in Ring. Consider an application that employs JSON as its data interchange format. When handling a request, the application might explicitly call a library to transform the JSON object into a native Clojure object. Performing this task quickly becomes tedious. Ring allows us to define a function that accepts a handler – a function that corresponds to the next piece of middleware to be invoked in the processing pipeline – that returns a function that deals with the response, presumably invoking the next handler in that returned function. Similar approaches can be used to handle authorization to resources, parsing URL encoded structures, and similar repetitive tasks.

Note that the choice of Clojure is very deliberate. Being a dialect of Lisp, Clojure recognizes its own source code as a data structure. Similarly, Clojure is capable of easily transforming its data structures into source code. This enables us to dynamically generate human-readable test cases while avoiding painful and error-prone workarounds.

What error states ought we be concerned about? There are two approaches

to this problem: either provide an abstract specification for the construction of responses for any given request, or leverage the existing error system in the Java Virtual Machine. We proceed with the second approach, primarily due to its simplicity and performance. Any specification system provides a verification overhead to processing client requests. Our approach detects errors by waiting for unhandled exceptions and can be safely integrated into existing systems.

Once the program enters into an error state, how do we recreate the error? This depends on the way that the individual resources interact:

- *Resource Indepedence* - if all resources in the system are totally independent (i.e. operations on different resources only entail local state change) then only the traces of requests to the failed resource need to be considered
- *Resource Dependence* - if modifications to a pair of resources may entail non-local state changes, then the traces of requests to that pair of resources must be replayed to recreate the error state

In the resource independence model, there are two further considerations:

- *Stateful Resources* - a trace of all interactions is required to recreate an error state
- *Stateless Resources* - a single request is all that is necessary to recreate an error state

Note that resources that actually encapsulate state can safely be considered stateless if they are not modified over the course of a client's interaction with

them.

Our replay middleware provides support for stateless and stateful resource independent systems. Once an error state is discovered, a meaningfully named test is created in the project's ”replays/” directory. This test can be integrated into a unit-test system. Furthermore, an application might wait for changes in the ”replays/” directory. Once a new test is discovered, the application could create a new ticket in a system similar to Jira and assign the task of correcting the error to an engineer.

Stateful resources are enabled and disabled by invoking the functions “state-on!” and “state-off!” respectively. Similarly, logging all traces is enabled with calls to the “traces-on!” and “traces-off!” functions. Logging traces is complicated business. When trace logs are enabled, the system consumes significantly more memory. Consequently, this feature does not make sense in a production facing system.

However, there are alternative use-cases that render this feature useful. Generally speaking, total trace logging provides more useful generated test-cases. So, for local development, logging traces is a sane default. By making use of environment variables, this functionality can be disabled when the application is deployed.

Moreover, with the advent of platform-as-a-service technologies that abstract infrastructure concerns away from the process of software deployment, customer support agents are now in a position to deploy software in a sandboxed environment. If a customer is experiencing an error, the customer support agent simply deploys an instance of the application with traces enabled that only that specific customer is allowed access to. The

customer then recreates their error, and the generated test contains enough information for a developer to successfully resolve the fault.

Example 6.3.1. Consider a very simple Clojure web service using Compojure and the replay middleware:

```
(ns my-namespace
  (:require [replay-middleware.core :as replay] ...))
(defroutes app-routes
  (GET "/" _ (response "Hello, world!"))
  (GET "/this-will-error" _
    (throw (ex-info "SomeError" {:message "A problem!"}))))
(replay/state-on!)
(def app
  (-> (wrap-defaults app-routes
        (assoc-in site-defaults
                  [:security :anti-forgery] false))
      replay/wrap-traces))
```

Listing 6.1: A simple Clojure web service using replay middleware.

Obviously, the sequence of GET requests to "/" and "/this-will-error" causes the program to enter a fault state, as the handler for "/this-will-error" is defined as an unhandled exception. Under the "replays" directory in the software, a time-stamped test named "replay-this-will-error(timestamp)" is created. This file contains:

```
(clojure.core/ns replay (:require [clj-http.client]))
(def replay
```

```
(clojure.core/fn []
  (do (clj-http.client/get
         "http://localhost:3000/"
         {:cookies {},
          :headers {"accept" "*/*", "connection" "close",
                    "accept-encoding" "gzip, deflate",
                    "user-agent" "curl/7.51.0",
                    "content-type" "text/plain; charset=UTF-8",
                    "cookie" "",
                    "content-length" "0",
                    "host" "localhost:3000"},
          :body ""})
      (clj-http.client/get
         "http://localhost:3000/this-will-error"
         {:cookies {},
          :headers {"accept" "*/*",
                    "connection" "close",
                    "accept-encoding" "gzip, deflate",
                    "user-agent" "curl/7.51.0",
                    "content-type" "text/plain; charset=UTF-8",
                    "cookie" "",
                    "content-length" "0",
                    "host" "localhost:3000"},
```

```
            :body ""}))))
```

Listing 6.2: Sample test generated by middleware.

To use this test, a user opens a Clojure read-eval-print loop (REPL) and loads the file. Then, by invoking the function "replay/replay," the sequence of requests that led to the error are replayed.

6.4 Modeling Application with CSP and formalized REST

We model a component of the queue application in CSP and verify one of its properties to demonstrate how this task is performed in general. Let S denote an identifier of a server and R denote the queue's resource identifier (consult Chapter 4 for more details.) Furthermore, we define messages *empty_message* denoting an empty queue, *enqueue_message* indicating a successful enqueue operation, a *404_message* describing a missing resource, and finally an *object_message* that encapsulates the representation of some object. Observe that a single queue can adequately be expressed as the following CSP process:

```
QUEUE⟨⟩ = (ComSR ? request.interface.S.R →
             (ComSR ! reply.empty_message.R.S →
              QUEUE⟨⟩)
               ◃ (interface.oper == GET) ▹
             (ComSR ! reply.enqueue_message.R.S →
              QUEUE_interface.message.object)
```

```
         ◁ (interface.oper == POST) ▷
       (ComSR ! reply.404_message.R.S → QUEUE_⟨⟩))


QUEUE_⟨a⟩b = (ComSR ? request.interface.S.R →
       (ComSR ! reply.object_message(a).R.S →
        QUEUE_⟨⟩)
         ◁ (interface.oper == GET) ▷
       (ComSR ! reply.enqueue_message.R.S →
        QUEUE_⟨a⟩b  interface.message.object)
         ◁ (interface.oper == POST) ▷
       (ComSR ! reply.404_message.R.S → QUEUE_⟨⟩))
```

Define a *ghost object* as an object returned to satisfy a GET request on an empty queue. Obviously, such an object should never exist, so it would indeed be spooky if one is detected in a trace of the system! This behavior is observed by counting the number of successful enqueue and dequeue operations and comparing. We can codify this requirement as the trace proposition:

$$NOGHOSTS = (QUEUE \downarrow enqueue_success) - (QUEUE \downarrow dequeue_success) \geq 0.$$

Consult Chapter 3 for information regarding the semantics of trace propositions. Here, we slightly abuse notation and write *enqueue_success* to stand in for a sequence of actions corresponding to the successful enqueue of an object. Similarly, *dequeue_success* denotes a sequence of actions corresponding to a successful dequeue operation. It is left as an exercise to the reader to write out what sequences of actions in particular these names refer to.

Proof. We seek to demonstrate that no ghost objects exist in the traces of a queue process. We have for the trivial process:

$$\frac{STOP \ \mathbf{sat} \ tr = \langle\rangle}{(tr \downarrow enqueue) - (tr \downarrow dequeue) = 0 \geq 0}.$$

Now, assume that $X_{\langle p \rangle_s}$ is a trace-set of a queue process that satisfies the *NOGHOSTS* specification. Let $a = ComSR.request.interface.S.R$ and $b = ComSR!reply.object_message(p).R.S$. Then, we have that:

$$\frac{X_{\langle p \rangle_s} \ \mathbf{sat} \ (tr \downarrow enqueue) - (tr \downarrow dequeue) > 0; \ interface.oper = GET}{\dfrac{\dfrac{\langle a, \ b \rangle \, X_s \ \mathbf{sat} \ tr \leq \langle a, \ b \rangle}{0 < ((tr'' \downarrow enqueue) - (tr'' \downarrow dequeue))}}{0 \leq ((tr \downarrow enqueue) - (tr \downarrow dequeue))}}.$$

A similar but simpler argument holds for when the queue is empty. Hence, we see that the trivial process satisfies the trace proposition. Furthermore, prefixing an arbitrary trace of the queue with the additional communications of a queue process does not cause degenerate behavior to manifest itself. We conclude that there are no ghost objects returned by the queue. □

6.5 Conclusion

We discuss the software developed for this project. This includes a queue application that provides a case-study pertaining to the behavior and verification of RESTful systems. We provide a model of an aspect of this system and demonstrate that a trace proposition holds for this model. We also discuss the development of instrumentation for the facilitation of debugging in RESTful applications built in the Clojure programming language. This includes an investigation into the various strategies of trace collection and

error state exploration. Use-cases of this component are explored and the utility value of the middleware is discussed.

7

Future Work

We have discussed the general problem of verifying concurrent systems through an algebraic approach. REST has been formalized using CSP, and with this formalization new implementations of REST components can be integrated into existing architectures. Moreover, these implementations are guaranteed to compose correctly if they trace-refine their corresponding CSP descriptions.

There are exciting directions to take this work. The software can be extended to provide support for resource dependence. This can be accomplished by creating a declarative domain specific language that configures resource dependencies. These dependencies can be used to form a graph structure that can easily be used to generate replays for failed requests. The replay middleware can be improved by providing mechanisms for specifying trace retention. Furthermore, it is interesting to consider the possibility of reducing the size of the error space that needs replayed.

Finally, consider the behavior of the system, as a whole. There are many

communicating components where failures can "bubble up" and cause ironic problems. Perhaps the constraints of REST allow us to easily consider the total possible failure states of the system. Adding timing constraints to the model could help achieve this and also further improve its utility value.

Bibliography

[1] Amazon REST API Reference.
https://docs.aws.amazon.com/apigateway/api-reference/.
Accessed: 2018-01-30.

[2] Compojure. https://github.com/weavejester/compojure. Accessed: 2018-02-25.

[3] Google Compute Engine API Reference.
https://cloud.google.com/compute/docs/reference/latest/.
Accessed: 2018-01-30.

[4] Introducing JSON. https://www.json.org. Accessed: 2018-02-21.

[5] Ring. https://github.com/ring-clojure/ring. Accessed: 2018-02-25.

[6] The Clojure Programming Language. https://clojure.org. Accessed: 2018-02-20.

[7] What Is Docker? https://www.docker.com/what-docker. Accessed: 2018-02-20.

[8] J.C.M Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, New York, NY, 1990.

[9] J. A. Bergstra. *Handbook of Process Algebra*. Elsevier Science Inc., New York, NY, USA, 2001.

[10] Chi Tat Chong. *Recursion theory: a generalized point of view*. De Gruyter, Berlin, Germany, 2015.

[11] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, PO Box 128, Farrer Road, Singapore, 1995.

[12] Thomas Macaulay Ferguson and Graham Prieset. *A Dictionary of Logic*. Oxford University Press, 2016.

[13] Roy Thomas Fielding. *Architectural Styles and the design of Network Based Software Applications*. PhD thesis, University of California, Irvine, 2000.

[14] The Apache Software Foundation. What is Maven? `https://maven.apache.org/what-is-maven.html`. Accessed: 2018-02-26.

[15] GNU Operating System. GNU Make. `https://www.gnu.org/software/make/`. Accessed: 2018-02-26.

[16] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[17] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, New York, NY, 2000.

[18] Phil Hagelberg. Leiningen – for automating Clojure projects without setting your hair on fire. `https://leiningen.org`. Accessed: 2018-02-20.

[19] Pivotal Software, Inc. Understanding HATEOAS. `https://spring.io/understanding/HATEOAS`. Accessed: 2018-02-20.

[20] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[21] Mark Tarver. *Logic, Proof, and Computation*. Fastprint Publishing, 2nd edition, 2014.

[22] Xi Wu and Huibiao Zhu. Formalization and analysis of the rest architecture from the process algebra perspective. *Future Generation Computer Systems*, 56:153–168, March 2016.