University of California

Los Angeles

Robust Service Composition

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Jeffrey M. Fischer

2008

The dissertation of Jeffrey M. Fischer is approved.

Jens Palsberg

Todd Millstein

Uday Karmarkar

Rupak Majumdar, Committee Chair

University of California, Los Angeles

2008

For Qingning and Jason.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank the following people:

- Qingning, for all her support and encouragement,
- Milos Ercegovac and Verra Morgan, for encouraging me to return to UCLA for my doctorate,
- Rupak Majumdar, for his guidance in the world of academia,
- the newly-hired professors in the UCLA software area: Eddie Kohler, Rupak Majumdar, Todd Millstein, and Jens Palsberg, for their enthusiasm and excellent classes,
- my dissertation committee, for their feedback on this dissertation and support for my work,
- the other students of the TERTL lab, for technical discussions, humor, and commiseration,
- and to Jason, for providing a welcome distraction.

Collaborators This dissertation is based on work that I have done with my advisor and other collaborators. Specificially:

- Chapter 2 is based on joint work with Rupak Majumdar and Todd Millstein
- Chapter 3 is based on joint work with Rupak Majumdar and Francesco Sorrentino
- Chapter 4 is based on joint work with Rupak Majumdar
- Chapter 5 is based on joint work with Ajay Chander that I performed while a reseach intern at DoCoMo USA Labs

These collaborations were quite enjoyable and I gained much insight through them.

Vita

1991	B.S. (Electrical Engineering), UCLA, Los Angeles, California.
1993–1998	Principal Member of Technical Staff, Oracle Corporation.
1999	M.S. (Computer Science), UCLA, Los Angeles, California.
1998–2003	Senior Manager, Siebel Systems Inc.

Publications

J. Fischer and R. Majumdar. *A Theory of Role Composition*, ICWS, September 2008.

J. Fischer, R. Majumdar, and F. Sorrentino. *The Consistency of Web Conversations*, ASE, September 2008.

J. Fischer and R. Majumdar. *Ensuring Consistency in Long Running Transactions*, ASE, November 2007.

J. Fischer, R. Majumdar, and T. Millstein. *Tasks: Language Support for Event-driven Programming*, PEPM, January 2007.

M. Emmi, J. Fischer, R. Jhala, and R. Majumdar. *Lock allocation*, POPL, January 2007.

J. Fischer, R. Jhala, and R. Majumdar. *Joining dataflow with predicates*, FSE, August 2005.

J. Fischer and M. Ercegovac. *A component framework for communication in distributed applications*, IPDPS, May 2000.

Abstract of the Dissertation

Robust Service Composition

by

Jeffrey M. Fischer

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2008

Professor Rupak Majumdar, Chair

The development of the HTTP protocol and web services have made it convenient to build large-scale systems out of loosely-coupled services. Key advantages of this approach include the ability to leverage existing applications in new contexts, the incremental evolution of systems, and better scalability/availability through replication of services.

Unfortunately, service-oriented architectures have a number of drawbacks. Implementations must account for differing data representations or protocols, remote system failures, and asynchronous processing. New failure modes are introduced, such as hangs, deadlocks, and data inconsistencies. Securing a service-oriented architecture is frequently more difficult than for a monolithic architecture due to the larger attack "surface area" and differing security frameworks across the individual services. Finally, testing and debugging service-oriented architectures is difficult due to the lack of a global perspective, non-determinism, and the challenge of building tools which work across heterogeneous systems.

This dissertation investigates new approaches to address these issues. I show that service-oriented architectures can be made more robust through better abstractions together with lightweight, compositional tools to automatically enforce the abstractions. Four specific problems are addressed: lost messages in asynchronous programs, the consistency of long running transactions, reconciliation of access control policies, and the trust by end users of composed applications. For each issue, I develop a operational model which captures the salient aspects of service-oriented systems. From the models, I define new abstractions, which accomplish the goals of the original system while avoiding the issues under consideration. Practical tools to enforce the abstractions are then derived from the models, either by construction or through the lightweight verification of developer artifacts. Finally, each tool is evaluated using performance tests and case studies.

CHAPTER 1

Introduction

The development of the HTTP protocol and web services have made it convenient to build large-scale systems out of loosely-coupled services. Examples include e-commerce web applications, such as Amazon [DHJ07] and eBay [SP06], collections of enterprise applications within an organization, and "mashups", which compose independently-developed websites into entirely new applications.

Advantages of service-based architectures This approach is attractive for three key reasons. First, it enables existing applications to be used in new contexts. In Chapter 3, we describe a real-world scenario where customer data from a mainframe application is made available to newer applications. Second, this approach enables services to be partitioned or replicated, to improve both availability and scalability. For example, Amazon uses Dynamo, a partitioned and replicated data storage service, to maintain its customers' shopping cart data [DHJ07]. Third, service-based systems enable components to evolve independently, reducing the cost and availability impact of upgrades. This style of evolution is practiced by most large web applications, allowing new functionality to be delivered while providing continuous availability. It is also used to evolve the software in an enterprise: legacy systems can be incrementally replaced without disrupting the normal operation of a company.

Drawbacks Unfortunately, service-based architectures have a number of drawbacks when compared to centralized systems. The implementation of each component must handle the more complex semantics of remote calls. If services were developed independently, then glue code may be needed to reconcile differing data representations and protocols. In a centralized system, if a callee fails, this fact can be communicated immediately through callers using standard exception handling mechanisms. In distributed systems, exception handling must be built into each message protocol and it is difficult to distinguish the complete failure of a remote service from a delayed response. Finally, many service-based systems make use of asynchronous processing, where responses are handled independently from requests. This style of interaction is not well supported by modern programming language.

Service-based architectures introduce new types of defects, which must be prevented or caught through testing. Applications can hang if a deadlock occurs or if two applications have different expectations regarding the sequence of messages between them. Applications frequently must make assumptions regarding the internal state of their peers. If these assumptions are incorrect, data inconsistencies may result. Maintaining consistency is the face of failures is particularly difficult. Unfortunately, traditional approaches, such as distributed transactions, do not scale well in high throughput service-based systems [DHJ07].

Composing applications from independent services may introduce security issues. Each service interface exposes on the network a potential attack point. Even assuming all users of the system have been authorized, a number of unique issues must be addressed. Frequently, each application has its own framework for access control. There is no framework to relate access rights in one application to rights in another application. This may make it difficult to ensure that a given user has all the access rights necessary to use an application built from composed services. More seriously, if sensitive information is passed between services, nothing prevents the recipient from disclosing that data to users who do not have access in the source application.

Finally, testing and debugging service-based architectures is difficult. Due to software limitations and clock skew, it is not possible to obtain a true global perspective on the events occurring throughout a distributed system [Lam78]. Due to non-determinism in the individual services and the messaging infrastructure, the same system inputs may result in different sequences of messages. The heterogeneous nature of these systems impedes the development of debuggers which are aware of each component's internal state.

Addressing the drawbacks In this thesis, I will show how these issues can be mitigated through the use of formal methods to define lightweight analysis tools and frameworks. These can then be put into use by industrial programmers or end users, without requiring a detailed understanding of the formalisms behind them.

1.0.1 Example applications

To better understand the architecture of service-based systems, we look in more detail at a few representative applications.

Enterprise Application Integration As described in [Hig06], Washington Group International (WGI), an engineering and construction management company, automated the interactions between several of their applications using a service-based architecture. Figure 1.1 shows a simplified view of their network.

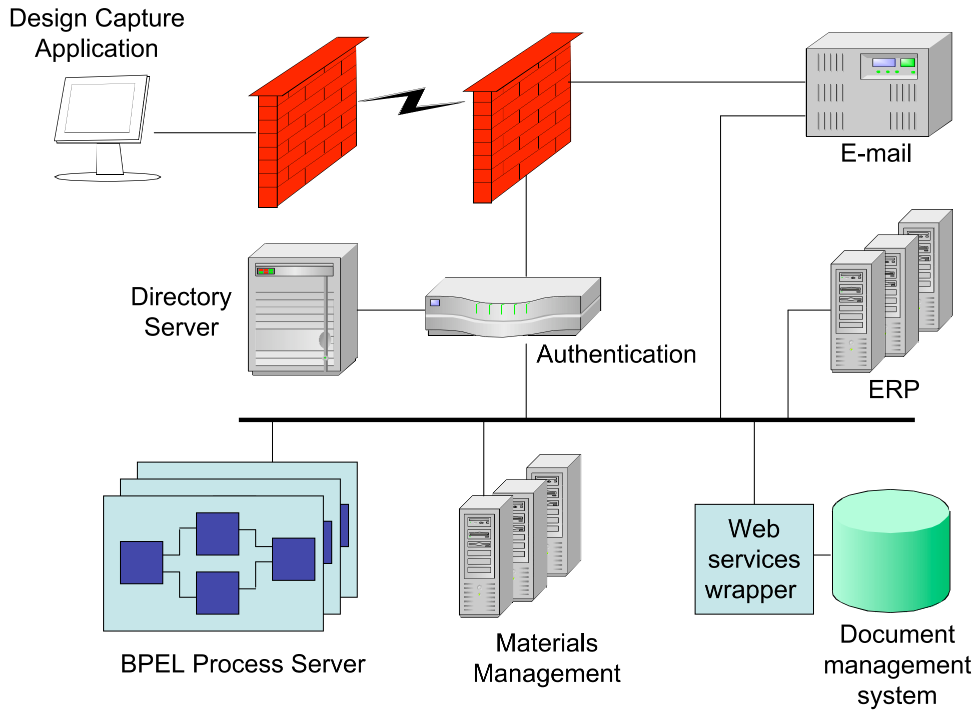


Figure 1.1: Example of enterprise applications

End users work from desktop machines, which run CAD and design capture applications locally and connect to WGI's main data center via the Internet. Authentication for their data center-based applications is centrally managed, using a *single sign-on* (SSO) architecture. Authentication requests are directed to a security appliance, which matches each query against a database of user identities stored in a directory server. Upon authentication, the security appliance issues a token that is accepted by each of the data center applications.

The applications run in WGI's data center include Enterprise Resource Planning (ERP), Materials Management from another vendor, Document Management, and an E-mail server. Requests which require the interactions of multiple applications are coordinated by *business processes* written in Business Process Execution Language for Web Services (BPEL) [BPE03]. BPEL is a *flow compo-*

sition language which provides constructs for composing services: sequential and parallel composition, branching and loops, compensation and exception handling, and asynchronous messaging.

[Hig06] describes two processes coordinated by the BPEL server. Purchase orders are created in either the ERP or Materials Management application, digitally signed by users (using E-mail to notify users that their action is needed), and then stored in the Document Management application. Engineering designs are created on user workstations, correlated with data from other applications, rendered into a PDF document, and then stored in the Document Management application. The Document Management application assigns a unique identifier to the design, which can subsequently be used by other applications to reference or retrieve the design.

These processes were previously implemented using a combination of automated and manual steps. For example, users would print out purchase orders entered into their ERP and Materials Management applications, sign the printed order, rescan the signed order, and upload the scan to the Document Management application.

Using a service-based architecture coordinated by BPEL processes has also reduced the coupling between applications. For example, the Document Management application provides only a programmatic API for other applications to call. Previously, WGI would have to write "glue" code in each application's infrastructure to call this API. Instead, they wrote a web services "wrapper" in front the Document Management application. This allows the other applications to access documents indirectly, through BPEL processes.

Although the approach used by WGI simplifies the integration of their applications, it does not address two key issues. First, even through authentication is centralized, each application still maintains its own access control infrastructure and policies. Developers of business processes must manually reconcile access control policies, which is difficult to do a priori, since user to permission mappings may be changed at any time. Alternatively, they may bypass security checks, which may lead to subversion of an access control policy and disclosure of confidential data.

Second, the loose coupling between applications makes it harder for developers to reason about the error handling scenarios for their processes. In a more tightly-coupled system, distributed transactions allow the infrastructure to handle much of the error handling and recovery. In WGI's system, each service call is an independent transaction. BPEL provides a *compensation* operator which can specify an undo function that reverses committed transactions in the event of an error. However, services requiring compensation must be identified by the developer, who must also define and implement the overall error handling strategy.

Large-scale e-commerce web applications Amazon's web applications are implemented using a decentralized service architecture, built on hundreds of individual services [DHJ07]. A page request may construct its response by sending requests to over 150 services, many with multiple dependencies.

Figure 1.2 shows a logical view of how these services are organized. Requests are first accepted into a page rendering layer, which calls aggregator services to obtain the data needed to render a web page. These aggregator services, which are usually stateless, combine the results of calls to lower-tiered services. Caching is used to reduce the number of calls made in a typical aggregation service request. The lower-tiered services provide more specialized functionality

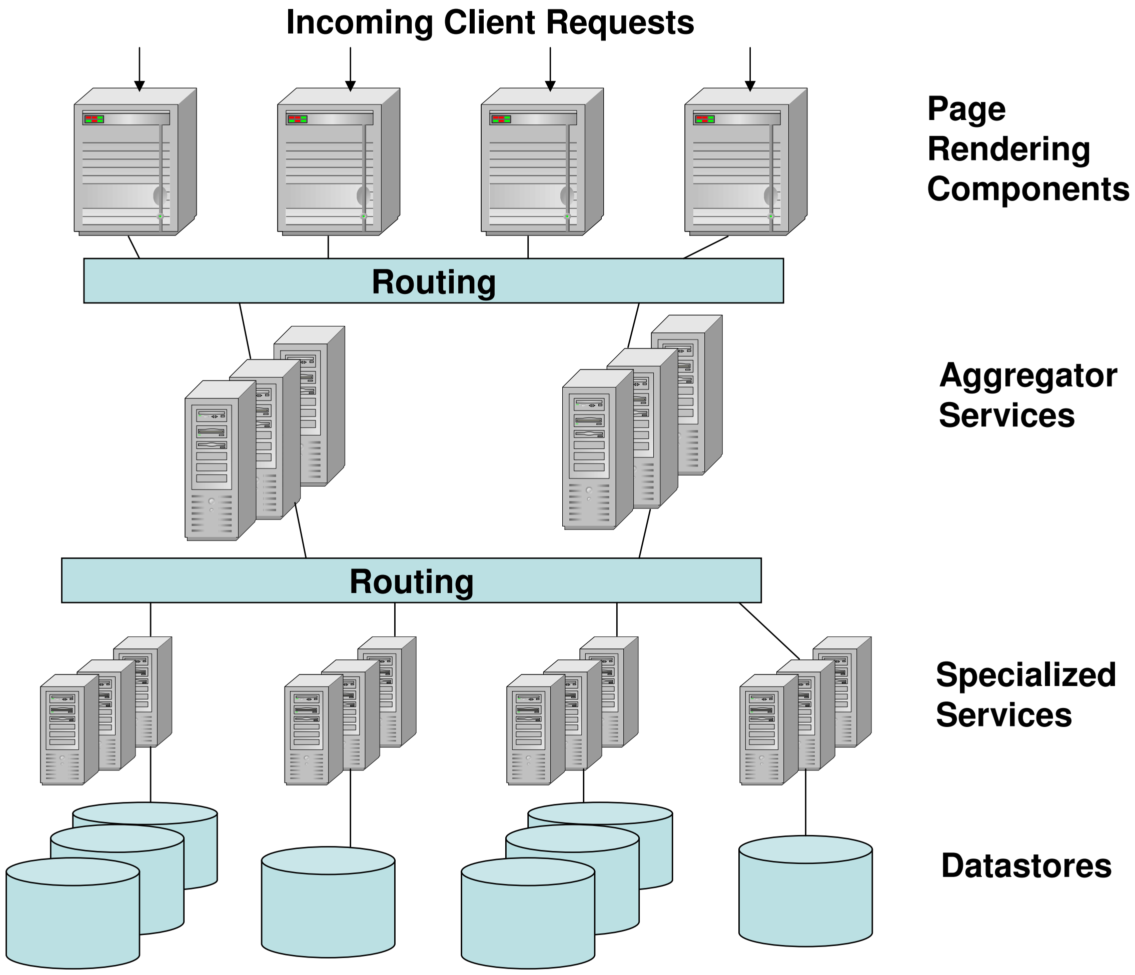


Figure 1.2: Amazon's service oriented architecture (based on a figure in [DHJ07])

and make use of a storage tier for persistent data. Several types of storage services are used by Amazon. These offer different indexing capabilities, consistency models, performance, and availability.

The storage services may in turn be service compositions. For example, *Dynamo* provides high availability and scalability through a combination of partitioning (via consistent hashing) and replication. The service is completely decentralized, with no need for manual administration to add or remove nodes.

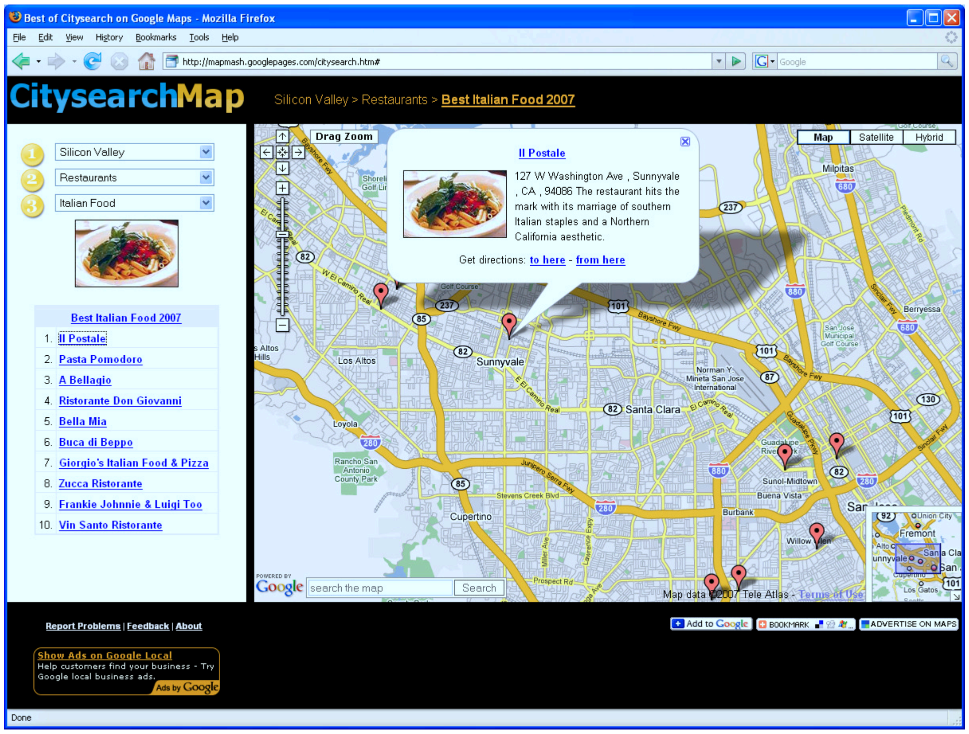
Consistency between replicas is maintained via a versioning scheme. This scheme completely sacrifices isolation and provides a weaker notion of consistency in order to ensure higher availability and scalability than typically achievable with standard transactional models.

Building applications on top of this infrastructure introduces new challenges. Due to performance issues with threading and the nature of request routing in a distributed application, responses are frequently handled independently from their associated requests. This leads to an event-driven style of programming which, while appropriate for these applications, can obscure control flow and the programmer's intent.

Second, relaxed consistency models make it harder to reason about the correctness of an application. Developers must consider what inconsistencies can occur and which are acceptable. Services like Dynamo require the user to understand the underlying implementation to answer these questions.

If the response to a request is lost, or the state of a service becomes inconsistent, application invariants may be violated (e.g. customers should be billed for what they buy) or the ongoing interactions with a customer may become deadlocked.

Mashups *Mashups* are web applications which display correlated data from multiple sources, typically other web applications. Usually, one of the applications is a map (e.g. Google Maps), and the data from the other applications has a geographical component, allowing it to be superimposed over the map. The website `http://www.programmableweb.com` provides a directory of over 2500 mashups which use over 500 different APIs or websites as their sources.


Best of Citysearch on Google Maps - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://mapmash.googlepages.com/citysearch.htm#
CitysearchMap
Silicon Valley > Restaurants > Best Italian Food 2007
Drag Zoom
Map
Satellite
Hybrid
search the map
Search
Report Problems | Feedback | About
Add to Google
BOOKMARK
ADVERTISE ON MAPS
Done

1 Silicon Valley
2 Restaurants
3 Italian Food

Best Italian Food 2007

1. Il Postale
2. Pasta Pomodoro
3. A Bellagio
4. Ristorante Don Giovanni
5. Bella Mia
6. Buca di Beppo
7. Giorgio's Italian Food & Pizza
8. Zucca Ristorante
9. Frankie Johnnie & Luigi Too
10. Vin Santo Ristorante

Il Postale

127 W Washington Ave , Sunnyvale , CA , 94086 The restaurant hits the mark with its marriage of southern Italian staples and a Northern California aesthetic.

Get directions: to here - from here

Show Ads on Google Local
Help customers find your business - Try Google local business ads.
Ads by Google

Map data ©2007 Tele Atlas - Terms of Use

Figure 1.3: Example of a mashup

Figure 1.3 shows *CitySearch Map*,[1] one such mashup. This application displays data from citysearch.com on maps provided by Google. The interface to this application requests the user to select a city, a category (e.g. restaurants or shopping), and a sub-category (e.g. cuisine). It then displays the top ten selections from Citysearch in that category, along with links to the Citysearch reviews and a map from the area showing the location of each business. The user can further interact with the application by clicking on a push-pin to get a pop-up with a picture and more details or by zooming the map in or out. The data from Citysearch is obtained through XML-formatted messages passed over HTTP. Google provides a JavaScript programming interface which allows its Google Maps functionality to be embedded in other web applications.

[1] `http://mapmash.googlepages.com/citysearch.htm`

Mashups are popular because they help users to avoid manually correlating data from multiple sources and because they are relatively easy to program using modern browser technology (HTML, JavaScript, and XML). However, composition does require programming — end users cannot create their own mashups specific to personal tasks they wish to automate. More seriously, these mashups provide no security guarantees. Users cannot be certain that the mashup will not make unauthorized updates using their credentials or pass their data to external sources. This limits the types of applications that will be built using the current technologies behind mashups.

1.1 Contributions

In this dissertation, I focus on four specific issues which impact the robustness of service-oriented architectures — asynchronous programming, consistency, access policy integration, and secure composition. Through my investigation of these issues, I will demonstrate my thesis: service-oriented architectures can be made more robust through better abstractions together with lightweight, compositional tools to automatically enforce the abstractions.

Methodology I use the following methodology to approach each issue:

1. Identify a problem to be addressed. Problems which are caused by complex, non-local interactions between independent components are of particular interest, since they cannot be easily found by testing in isolation. In this dissertation, I am focusing on domain-specific issues for service-oriented systems.

2. Define a language describing systems which may exhibit this problem, fo-

cusing only on aspects related to the problem. The language model should capture the structure of the system (syntax) as well as its runtime behavior (operational semantics). Aspects of the system not relevant to the issue under consideration should be left unconstrained (e.g. the TaskJava scheduling algorithm in Chapter 2).

3. Define an abstraction which limits the possible behaviors of systems in exchange for easier reasoning (for both humans and tools) about the issue being addressed. For example, I define in Chapter 4 the *global role schema*, an abstraction over the security policies of service oriented systems. Such abstractions may, by their nature, prevent the problem we are addressing, or may make it easier to precisely define properties of systems which do not exhibit the problem under consideration. Any properties needed to ensure the absense of the problem must also be identified.

4. Create a compositional, syntax-directed algorithm (type checker), which analyzes systems in our language, enforcing the abstraction and ensuring that any properties identified in step 3 do, in fact, hold. This algorithm should not require any fundamental changes to the underlying language. However, it may make use of additional information at interface boundaries, in the form of lightweight annotations or specifications. The algorithm should be compositional: if each individual component of the system satisfies the checks, and the composition does not violate any component's assumptions about its environment, the entire system is free from the problem in question.

5. Create tools which implement the checking algorithm for concrete systems. Due to the compositional nature of the algorithm, such tools can be used to incrementally check components of a system. Since the tools require

only minor annotations, they can be used on legacy code and by developers which might not fully understand the theory behind the model.

Now, let us look in more detail at the four issues I will investigate using this methodology.

1.1.1 Asynchronous programming

First, in Chapter 2, I consider asynchronous programming. In their simplest form, web services are synchronous requests: the requesting system waits for a response before continuing. This simplifies program design and matches the underlying HTTP protocol. However, when interactions between applications may take long periods of time (due to human interactions for approvals, for example), requests and responses are handled separately. The requester does not wait for a request to complete, but instead provides an “address” for a response as a part of the request. This avoids tying up resources (e.g. threads) on the requester and is resilient to failures that may occur before the response is sent. Highly concurrent servers may also use this approach when processing many simultaneous requests, as this consumes fewer resources than other approaches (e.g. multi-threading).

This programming style, where requests and responses are separated, is called *event-driven programming*. Unfortunately, the event-driven style severely complicates program maintenance and understanding, as it requires each logical flow of control to be fragmented across multiple independent operations (e.g. functions, procedures, methods, etc.). To reduce these challenges, I extend the Java programming language to include *tasks*, a new abstraction for lightweight concurrency. Tasks allow each logical control ow to be modularized in the traditional manner, including usage of standard control mechanisms like procedures and exceptions. At the same time, by using method annotations, task-based programs can be automatically and modularly translated into ecent event-based code, using a form of continuation passing style (CPS) translation. I have implemented support for tasks using a source-to-source translator from my extended language, TaskJava, to standard Java. The benefits of this language are illustrated through a formalization, in which I show that key safety properties are guaranteed, and a case study based on an open-source web server.

1.1.2 Consistency

In service-oriented systems, a single action on behalf of the user may require coordinating changes across several systems. Ensuring that the correct changes are made together can be challenging, especially in error scenarios. In traditional, centralized systems, these issues are mitigated through the use of transactions, which provide *atomicity*: either the task runs to completion or, if an error occurs, all partial changes are undone completely. Although distributed transactions may be used to extend this model to distributed systems, they are generally not suitable in a web services environment due to the long-running nature of many activities (distributed transactions require locking resources for the duration of the transaction) and the lack of the necessary transactional programming interfaces on many applications. As an alternative, most service-oriented systems use *compensation* to implement atomicity: changes to each system are implemented as independent transactions, but if an error occurs, any changes are undone through separate, *compensating* transactions. When all changes are undone in the event of an error, this approach is called *full cancellation*.

Although the compensation model is better than ad hoc error handling, it can still be very error prone, especially when combined with asynchronous computation. Full cancellation of partially completed work can also be too limiting, as many real-world scenarios require taking alternative paths when an error occurs (forward recovery) or leaving some state changes in place after an error.

In Chapter 3, I address the limitations of compensation through a new specification language and associated verification algorithm. Compensation is still used for error handling, but I relax the full cancellation restriction on activities. Instead, developers can specify which groups of actions must occur together. Since the set of completed actions may be different depending on the outcome of the overall activity, multiple groups of actions may be specified. It turns out that this can be compactly represented as a predicate. I call this notion of correctness *set consistency*

In some situations, the actions being coordinated may require exchanging multiple messages, as a part of a *conversation*. I extend set consistency to these conversations by modeling each conversation using an automaton and labeling states as either *committed* (a complete change has been made), *nochange* (no change has been made or the change was undone), or *inconsistent* (further messages must be exchanged to leave the conversation in either a committed or nochange state).

To demonstrate the value of consistency specifications, I have implemented BPELCheck, a consistency verifier for long running transactions written to the Business Process Execution Language (BPEL) [BPE03] specification. I have tested this on several example processes, including an industrial example from Sun Microsystems.

1.1.3 Access policy interoperability

When coordinating changes across systems, one must also address the security policies enforced by the individual services. In particular, applications in a service-oriented system are designed independently and have their own access control policies and enforcement mechanisms. When a service calls to another service, it must have access rights to the other systems. One could just give full access rights to services, but this violation of least privilege can circumvent the intent of access policies and leaves a greater “attack surface” for security exploits. In addition, when data from one system is stored in other systems, it may be possible for users to circumvent the intent of the primary system’s access control policy by reading data from secondary systems.

I address these issues in Chapter 4, where I consider the interoperability of access control policies, specifically those built using Role Based Access Control (RBAC). In RBAC, the mapping of users to permissions is abstracted using *roles*, which represent a collection of permissions needed to perform a specific job function.

I present an algorithm which infers a *global role schema*, which is a set of global roles to be used across multiple systems, along with a mapping from each global role to a set of local system-specific roles. Users are assigned by an administrator to global roles according to their job function and then are automatically assigned to all the local roles associated with their global roles. This assignment is guaranteed to be *sufficient*: service-to-service calls will not fail due to access restrictions, *non-disclosing*: data copied between applications is not disclosed to anyone who cannot see the original data, and *minimal*: no additional local roles are given to the user beyond those needed to satisfy the first two properties. My inference algorithm works by generating Boolean constraints based on the local access control policies of each application and the interactions between services. These constraints are then passed to a Boolean satisfiability solver. If a satisfying solution to the constraints can be found, it is translated into a global role schema.

I have implemented this inference algorithm in RoleMatcher, a tool which extracts RBAC metadata from applications, builds a model of the application interactions, and infers a global role schema. I show that it can quickly infer global roles for composed systems, or determine the absence of a globally consistent role schema.

1.1.4 End-user security for service composition

Finally, in Chapter 5, I examine end-user composition and the issue of trust. Mashups allow end users to correlate content from multiple services. However, as described above, mashups do not provide any guarantees about the distribution of sensitive data. This prevents applications that correlate data from services with different levels of trust. For example, consider a mashup combining a company's sales data with information from the public Internet. To be useful, the mashup must ensure that the sales data is never passed outside the company.

I address these issues in the context of mobile devices through a framework which supports secure, automatic composition. I present a new class of components, called monents (*mo*bile compo*nents*), that streamline the creation of mobile applications which combine data from multiple services. The composition framework guarantees that trusted data is not passed to untrusted services.

Monents are built by declaratively interconnecting smaller components which represent external services, UI controls, and inputs/outputs. All interactions of a monent with the external world are controlled through a security manager, which enforces user-defined security policies that restrict the monent's information flow. Security policies are not evaluated until a monent is activated, and a given monent can be run with different security policies, depending on the user's level of trust for that monent. A collection of monents can be automatically composed, using an algorithm which guarantees that the security policies of the individual monents are preserved.

I model the security and composition of monents using a novel interface formalism I call *information flow interfaces*. My formal model represents monents at a very abstract level and thus can be applied in many contexts beyond my specific implementation.

I have built a prototype implementation of the monent framework. My compiler accepts a declarative description of a monent's interconnections and produces an Adobe Flash application.

CHAPTER 2

Asynchronous programming

2.1 Overview

Most service implementations make use of an asynchronous programming style, where calls to a service are decoupled from the processing of its response. This may be done to enable the interleaved processing of requests, to avoid tying up system resources during the processing of a request, or to prevent system failures and restarts from impacting the progress of a long-lived request.

To implement asynchronous programming in most widely used programming languages, a technique called *event-driven programming* is employed. This approach relies upon a stylized programming idiom where programs use non-blocking I/O operations, and the programmer breaks the computation into fine-grained *callbacks* (or *event handlers*) that are each associated with the completion of an I/O call (or *event*). This permits the interleaving of many simultaneous logical tasks with minimal overhead, under the control of an application-level cooperative scheduler. Each callback executes some useful work and then either schedules further callbacks, contingent upon later events, or invokes a *continuation*, which resumes the control flow of its logical caller. The event-driven style has been demonstrated to achieve high throughput in server applications [PDZ99, WCB01], resource-constrained embedded devices [GLB03], and business applications [Mic].

Unfortunately, programming with events comes at a cost: event-driven programs are extremely difficult to understand and maintain. Each logical unit of work must be manually broken into multiple callbacks scattered throughout the program text. This manual code decomposition is in conflict with higher-level program structuring. For example, calls do not return directly to their callers, so it is difficult to make use of procedural abstraction as well as a structured exception mechanism.

2.1.1 Other approaches

Threads represent an alternative programming model commonly used to interleave multiple flows of control. Since each thread maintains its own call stack, standard program structuring may be naturally used, unlike in the event-driven style. However, threads have disadvantages as well, including the potential for race conditions and deadlocks, as well as high memory consumption [BCZ03]. Within the systems research community, there is currently no agreement that one approach is better than the other [PDZ99, BCB03, BCZ03, AHT02]. In addition, in some contexts, threads either cannot be used at all (such as within some operating system kernels) or can only be used in conjunction with events (such as thread-pooled servers for Java Servlets [Mic]). Thus, I believe that events are here to stay and are an important target for programming language support.

Other existing solutions attempt to preserve the asynchronous computation model, while limiting its disadvantages. This solutions include the use of first-class continuations to structure control flow (e.g. in Scheme [GKH01, Que03]), full-program transformations [MFG04, LZ04, PCM05], cooperative threading [Eng00, BCZ03], and the static analysis of event-driven programs [DGJ98, JM07]. These solutions all suffer from limitations, including expensive implementation, the need for the entire application's source, and the need to access low level machine resources not available in virtual machine based languages (e.g. Java and C#).

2.1.2 TaskJava

To address these issues, I introduce *tasks* as a new programming language construct for event-driven applications. A task, like a thread, encapsulates an independent unit of work. The logical control flow of each unit of work is preserved, and standard program structures like procedures and exceptions may be naturally used. However, unlike threads, tasks can be automatically implemented by the compiler in an event-driven style, thereby obtaining the low-overhead and high-throughput advantages of events. My compilation strategy is a restricted form of *continuation-passing style* (CPS), a well-studied compiler transformation that is popular for functional programming languages [App91]. I have instantiated this concept of tasks as a backward-compatible extension to Java called TaskJava and have implemented the TaskJava compiler in the Polyglot compiler framework [NCM03].

Tasks are a variant of *cooperative multitasking*, a form of interleaved execution where context switches only occur upon explicit yields. TaskJava provides several technical contributions over existing cooperative multitasking systems.

- First, TaskJava's modular static type system tracks the set of methods whose execution might yield, requiring each to have a new `async` modifier. Aside from serving as useful documentation for clients, these annotations tell the compiler exactly where CPS translation is required (and where it is not). In contrast, existing systems must allow for yields anywhere, which requires either low-level stack manipulation (which is not possible in virtual machine-based languages), maintaining the stack on the heap, or copying the stack onto the heap as necessary.

- Second, TaskJava is scheduler-independent: TaskJava programs can be "linked" against any scheduler that provides the semantics of a new `wait` primitive, which yields control to the scheduler. This design permits the benefits of tasks to be accrued across multiple event domains (GUI events, web server events, etc.). Prior approaches are tied to a specific scheduler and notion of events.

- Finally, TaskJava properly handles the interactions of `wait` with Java language features including checked exceptions and method overriding, and TaskJava's implementation adheres to the constraints imposed by the Java virtual machine.

TaskJava programs are guaranteed to avoid two significant classes of errors that may occur in event-driven programs, the *lost continuation* and *lost exception* problems.

The lost continuation problem occurs when a callback has an execution path in which the callback's continuation is neither invoked nor passed along to the next callback in the event chain. A lost continuation causes the intended sequential behavior of the program to be broken, often producing errors that are difficult to trace to their source. The lost exception problem occurs when an exceptional condition produced by a callback is not properly handled by the subsequent continuation, potentially causing the program to crash or continue executing in undefined ways.

2.1.3 Evalution

I evaluate TaskJava in two ways. First, I have formalized the language and its compilation strategy via CoreTaskJava (CTJ), a core language that extends Featherweight Java [IPW01]. I provide a direct operational semantics for CTJ, whereby `wait` calls block until an appropriate event is signaled, as well as a translation relation from CTJ to Featherweight Java, which formalizes the continuation-passing transformation performed by the TaskJava compiler. I have proven CTJ's type system sound, and as corollaries of this property, show that a well-typed CTJ program is guaranteed to avoid the *lost continuation* and *lost exception* problems.

Second, to evaluate TaskJava's benefits in practice, I extended Fizmez [Bon], an open source web server, to use interleaved computation. I implemented two versions: one using a manual event-driven style and the other using TaskJava. The TaskJava version maintains the same structure as the original web server, while the event-driven version requires its logic to be fragmented across many callback classes, obscuring the control flow. At the same time, the TaskJava version pays only a modest performance penalty versus the hand-coded one.

2.1.4 Chapter organization

The rest of this chapter is organized as follows. In Section 2.2, I informally present tasks and TaskJava by example and contrast with event-driven programs. In Section 2.3, I describe the CoreTaskJava formalisms. In Section 2.6, I overview the implementation of the TaskJava compiler, and in Section 2.7, I discuss a web server case study. Finally, I survey related work in Section 2.8.

2.2 Example: programming with tasks

I will highlight the features of TaskJava by example, demonstrating the use of TaskJava for managing non-blocking I/O.

2.2.0.1 Event-driven Programming

The event-driven programming style is frequently used in server programming in conjunction with *non-blocking I/O*. Non-blocking I/O libraries (such as Java's NIO package) permit input/output operations to be scheduled so that they do not block inside the operating system. Thus, independent requests can be executed in an overlapping fashion without preemptive multi-threading.

Non-blocking I/O libraries generally provide two types of calls. First, a *selection* call permits waiting for one or more channels/sockets to be ready for a new request. Examples include the Unix `select` call and the `Selector.select` method in Java's NIO package. Second, calls are provided to initiate the actual I/O operations (e.g., read and write) once the associated channel has become ready. Unlike a standard blocking read or write request, non-blocking read and write calls generally complete only the portion of a request that can be accomplished without blocking.

Selection calls are usually incorporated into a user-defined scheduler framework. Rather than calling the selection API directly, clients of the scheduler *register* to receive notification when the state of a given channel/socket changes. The scheduler then calls the selection API on behalf of all clients, notifying clients of events via callbacks. The control logic of each client is broken across a series of these callbacks and thus is interleaved by the scheduler with the callbacks of the other clients. This approach permits independent activities to cooperatively share the process's CPU and I/O resources.

2.2.0.2 Event-driven Writer

Figure 2.1 shows a simple program fragment, written in an event-driven style, which sends a buffer of data on a nonblocking channel. `Writer`'s `run` method first obtains the data to be written (not shown), which is stored in a buffer `buf`. The method then calls `Scheduler.register`, which registers a callback to be invoked upon a write-ready or error event on the channel `ch`. The `run` method returns immediately after the `register` call — execution of this logical control flow must be resumed by the scheduler.

When an event occurs on channel `ch`, the scheduler invokes the `run` method of the callback it was given (an instance of `WriteReadyCB`). This method performs a write on the channel and then checks to see if more data needs to be written. If so, the callback re-registers itself with the scheduler. Otherwise, it calls the *continuation* method `restOfRun` on the original `Writer` object, which resumes the logical control flow. If an error event is returned by the scheduler, the callback prints an error message. Since no callback is registered or continuation method invoked, the logical control flow is effectively terminated in that case.

Even this simple example illustrates the violence that the event-driven style does to a program's natural flow of control. The code in `restOfRun` logically follows the buffer write, but they must be unnaturally separated because of the intervening event registration. Similarly, performing the buffer write conceptually involves a loop that writes to the channel until the entire buffer has been written. In `WriteReadyCB.run`, this loop must be unnaturally simulated by having the callback re-register itself repeatedly.

Without care, it is easy for a programmer to introduce errors that go undetected. For example, if the call to `restOfRun` is accidentally omitted on line 30, then `Writer`'s control flow will never be resumed after the write. If the re-registration on line 28 is omitted, the write will not even be completed. These are examples of *lost continuation* problems.

2.2.1 Task-based Writer

Figure 2.2 shows a TaskJava implementation of the same program fragment. The class `WriterTask` is declared as a *task* by implementing the `Task` interface. Tasks are the unit of concurrency in TaskJava, serving a role similar to that of a thread in multi-threaded systems. Instances of a task may be created by using the `spawn` keyword, which is followed by a call to one of the task's constructors (e.g., `spawn WriterTask()`). A `spawn` causes a new instance of the task to be created and schedules the instance's `run` method for execution.

The logical control flow of our writer is now entirely encapsulated in `WriterTask`'s `run` method. The `register` call from `Writer` is replaced with a `wait` call, which conceptually blocks until one of the requested events has occurred, returning that event. In this way, explicit callback functions are not needed, so the code need not be unnaturally fragmented across multiple methods (e.g., `restOfRun`). Similarly, the logic of the buffer write can be implemented using an ordinary `do-while` loop.

The ability to use traditional program structures to express the control flow of a task avoids the lost continuation problem. The programmer need not manually ensure that the appropriate callback is registered or continuation method is invoked on each path. This work is done by the TaskJava compiler, which translates the `WriterTask` into a continuation-passing style that is very similar to the `Writer` code in Figure 2.1. In particular, `wait` calls are translated to

```java
public class Writer {
  ByteChannel ch;
  ...
  /* The main body of our task */
  public void run() {
    // get the data to write
    ByteBuffer buf = ...;
    /* wait for channel to be ready */
    Scheduler.register(ch, Event.WRITE_RDY_EVT,
                       Event.ERR_EVT,
                       new WriteReadyCB(ch, buf, this));
  }
  /* After the write has completed, we continue with what
     we were doing.  The event-driven style forces this
     in a separate method.  */
  public void restOfRun() { ...  }
  /* Callback which does the write and then registers
     itself if there still is data left */
  class WriteReadyCB implements Callback {
    ...
    public WriteReadyCB(ByteChannel ch, ByteBuffer buf,
                        WriteTask caller) {...}
    public void run(Event e) {
      switch (e.type()) {
       case Event.WRITE_RDY_EVT:
        ch.write(buf);
        if (buf.hasRemaining())
          Scheduler.register(ch, Event.WRITE_RDY_EVT,
                             Event.ERR_EVT, this);
        else caller.restOfRun();
        break;
       default:
        System.out.println(e.toString());
      } } } }
```

Figure 2.1: Implementation of an event-driven writer

`register` calls, and the portion of the `run` method after the `wait` call is placed in a separate continuation method.

TaskJava allows programmers to define their own scheduler class, their own event type and implementations, and their own type of event "tags" (e.g., `WRITE_RDY_EVT`). As long as the scheduler defines a `register` method for event registrations, TaskJava allows the scheduler to be treated as if it has a corresponding `wait` method. This approach allows existing scheduler frameworks to obtain the benefits of TaskJava without any modification. For example, the scheduler used in the manual version in Figure 2.1 may be reused in Figure 2.2. This approach also allows multiple scheduler frameworks to be used in the same program.

2.2.1.1 Asynchronous Methods

The TaskJava implementation of our writer also naturally supports procedural abstraction. For example, Figure 2.3 shows a refactoring of our task whereby the code to write the buffer is encapsulated in its own method, allowing that code to be easily used by multiple clients. Implementing this `write` method in the manual event-driven version of the code would be much more unwieldy, because event-driven programming breaks the standard call-return discipline. To return control back to caller, therefore, such a `write` method would have to take an explicit continuation argument to be called upon completion of the write.

Figure 2.3 also shows that tasks are compatible with regular Java exception handling. The `write` method throws an `IOException` when an error event is signaled, allowing its caller to handle the error as appropriate. As in Java, the TaskJava compiler ensures that all (checked) exceptions are caught. In contrast, a manual event-driven version of the `write` method would have to signal the

```
public class WriterTask implements Task {
  ByteChannel ch; ...
  /* The main body of our task */
  public void run() {
    // get the data to write
    ByteBuffer buf = ...;
    // write the buffer
    do {
      Event e =
        Scheduler.wait(ch, Event.WRITE_RDY_EVT,
                       Event.ERR_EVT);
      switch (e.type()) {
       case Event.WRITE_RDY_EVT:
        ch.write(buf);
        break;
       default:
        System.out.println(e.toString());
        return;
      }
    } while (buf.hasRemaining())
    /* the write is completed, so continue
       with the rest of the method */
    ...
  } }
```

Figure 2.2: Implementation of the writer in TaskJava

error to its caller in an *ad hoc* manner, for example by setting a flag or invoking a special `error` continuation method. This approach is tedious and loses the static assurance that all exceptions are caught, resulting in the potential for *lost exception* problems at run time.

Methods that directly or transitively invoke `wait`, like our `write` method, are called *asynchronous* methods. Such methods (other than a task's distinguished `run` method) must have the `async` modifier. To programmers, this modifier indicates that the method has the potential to block. To the TaskJava compiler, this modifier indicates that the method must be translated into continuation-

FJ$^+$	Program	P	::=	$\overline{CL}$ `return e;`
	Class List	CL	::=	`class C extends C {`$\bar{\texttt{T}}$ $\bar{\texttt{f}}$`; K `$\bar{\texttt{M}}$`}`
	Constructor	K	::=	`C(`$\bar{\texttt{T}}$ $\bar{\texttt{f}}$`) { super(`$\bar{\texttt{f}}$`); this.`$\bar{\texttt{f}}$`= `$\bar{\texttt{f}}$`;}`
	Method	M	::=	`T m(`$\bar{\texttt{T}}$ $\bar{\texttt{x}}$`) throws `$\bar{\texttt{C}}$` {return e;}`
	Type	T	::=	`C | Bag<T>`
	Base expressions	$\texttt{e}_{base}$	::=	`x | e.f | e.m(`$\bar{\texttt{e}}$`) | new C(`$\bar{\texttt{e}}$`) | (C)e | {`$\bar{\texttt{e}}$`} | throw e | try {e;} catch (C x) { e; }`
		e	::=	$\texttt{e}_{base}$
EJ		e	::=	$\texttt{e}_{base}$ `| reg(e, e)`
CTJ	Async methods	M	::=	`... | async C m(`$\bar{\texttt{C}}$ $\bar{\texttt{x}}$`) throws `$\bar{\texttt{C}}$` {return e;}`
		e	::=	$\texttt{e}_{base}$ `| spawn C(e) | wait(e)`

Table 2.1: Syntax of FJ$^+$, EJ, and CTJ

passing style.

Asynchronous methods, like regular Java methods, interact naturally with inheritance. For example, a subclass of `WriterTask` can override the `write` method to support a different or enhanced algorithm for writing a buffer. Making the same change to the `Writer` class in Figure 2.1 is less natural due to the fragmentation inherent in the event-driven style. For example, modifications to the logic for writing the buffer would require a new subclass of the `WriteReadyCB` callback class, and this modification then requires a new subclass of `Writer` whose `run` method creates the new kind of callback.

```
public class WriterTask implements Task{
  ByteChannel ch; ...
  /* The main body of our task */
  public void run() {
    // get the data to write
    ByteBuffer buf = ...;
    try {
      write(ch, buf);
    } catch (IOException e) {
       System.out.println(e.getMessage());
    }
  }
  public async void write(ByteChannel ch, ByteBuffer b)
    throws IOException {
    do {
      Event e = Scheduler.wait(ch, Event.WRITE_RDY_EVT,
                                   Event.ERR_EVT);
      switch (e.type()) {
       case Event.WRITE_RDY_EVT:
        ch.write(buf);
        break;
       default:
        throw new IOException(e.toString());
    } } while (buf.hasRemaining())
  } }
```

Figure 2.3: Use of asynchronous methods in TaskJava

2.3 Formal Model

We formalize TaskJava and prove our theorems in a core calculus extending Featherweight Java (FJ) [IPW01]. We do this in three steps: first, we define FJ$^+$, an extension to FJ with exceptions and built-in multiset data structures; second, we define EventJava (EJ), a core calculus for event-driven programs into which tasks will be compiled; and finally, we define the core features of TaskJava in CoreTaskJava (CTJ).

2.3.1 FJ$^+$

The syntactic elements of FJ$^+$ are described in Table 2.1. An FJ$^+$ program consists of a *class table* mapping class names to classes, and an initial expression. As in FJ, the notation $\bar{\mathtt{D}}$ denotes a sequence of elements from domain `D`. A class consists of a list of typed *fields*, a *constructor*, and a list of typed *methods*. The metavariable `C` ranges over class names, `f` over field names, `m` over method names, and `x` over formal parameter names. An expression is either a formal, a field access, a method call, an object allocation, a type cast, a set, the `throw` of an exception, or a `try` expression. We assume there exist built-in classes `Object` and `Throwable`. The class `Throwable` is a subclass of `Object` and both have no fields and no methods.

We define the operational semantics of FJ$^+$ as additions to the rules of FJ [1] and then EventJava and CoreTaskJava as mutually exclusive additions to these rules.

Program execution is modeled as a sequence of rewritings of the initial expression, which either terminates when a value is obtained or diverges if the rewritings never yield a value. For all three languages, programs evaluate to either non-exception values of the form $v ::= \mathtt{new}\ C() \mid \mathtt{new}\ C(\bar{\mathtt{v}}) \mid \{\bar{\mathtt{v}}\}$ or exception values of the form $\mathtt{throw}\ \mathtt{new}\ C_e(\bar{\mathtt{v}})$, where $C_e <: \mathtt{Throwable}$. In the evaluation and typing rules, we use v as shorthand for a non-exception value, $\bar{\mathtt{v}}$ for a sequence of non-exception values, and v_e for the non-exception value $\mathtt{new}\ C(\bar{\mathtt{v}})$ (used as the argument of a `throw` expression).

Figures 2.5 and 2.6 list the operational rules of FJ$^+$. We use the symbol E to represent an **evaluation context**, i.e., an expression where the next subexpres-

[1] We use the rules in chapter 19 of *Types and Programming Languages* [Pie02] rather than those of the original FJ paper [IPW01], as they provide deterministic, call-by-value evaluation.

$\boxed{\texttt{T} <: \texttt{T}'}$

$$\frac{}{\texttt{T} <: \texttt{T}} \text{ (S-Self)} \qquad \frac{\texttt{T} <: \texttt{T}' \qquad \texttt{T}' <: \texttt{T}''}{\texttt{T} <: \texttt{T}''} \text{ (S-Trans)}$$

$$\frac{CT(C) = \texttt{class}\ C\ \texttt{extends}\ D\ \{...\}}{C <: D} \text{ (S-Cls)} \qquad \frac{\texttt{T} <: \texttt{T}'}{\texttt{Bag} < \texttt{T} > \ <: \ \texttt{Bag} < \texttt{T}' >} \text{ (S-Bag)}$$

$\boxed{\texttt{T}|\bar{\tau} <: \texttt{T}'|\bar{\tau}'}$

$$\frac{\texttt{T} <: \texttt{T}' \qquad \bar{\tau} \subseteq: \bar{\tau}'}{\texttt{T}|\bar{\tau} <: \texttt{T}'|\bar{\tau}'} \text{ (S-Exc)}$$

Figure 2.4: Subtyping rules for FJ$^+$, EJ, and CTJ

sion to be evaluated (using a leftmost, call-by-value ordering) has been replaced with a placeholder []. Formally,

$$\begin{aligned} E ::= [\,] \mid E.f \mid E.m(e) \mid v.m(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \mid \texttt{new}\ C(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \\ \mid \{\bar{\texttt{v}}, E, \bar{\texttt{e}}\} \mid (C)E \mid \texttt{throw}\ E \mid \texttt{try}\ \{E;\}\ \overline{\texttt{CK}} \end{aligned}$$

We write $E[e]$ to represent the expression created by substituting the subexpression e for the placeholder in the evaluation context E. Evaluation contexts are used in the evaluation rules, the type soundness theorems, and the translation relation.

As in Featherweight Java, the computation rules for cast only permit progress when the type of the value being cast is a subtype of the target type. Otherwise, the computation becomes "stuck". The subtyping relation, defined in Figure 2.4, is extended to include bags (rule S-Bag) and exception effects (rule S-Exc). Based on the subtyping relation, we define a join operator $\sqcup$ on types, where $\texttt{T} \sqcup \texttt{T}'$ is

$\boxed{e \longrightarrow e'}$

$$\frac{fields(C) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{(\mathtt{new}\ C(\bar{\mathtt{v}})).f_i \longrightarrow v_i}\ (\text{E-1}) \qquad \frac{mbody(m, C) = (\bar{\mathtt{x}}, e_0)}{(\mathtt{new}\ C(\bar{\mathtt{v}})).m(\bar{\mathtt{v}}_e) \longrightarrow [\bar{\mathtt{v}}_e/\bar{\mathtt{x}}, \mathtt{new}\ C(\bar{\mathtt{v}})/\mathtt{this}]e_0}\ (\text{E-2})$$

$$\frac{C <: C'}{(C')(\mathtt{new}\ C(\bar{\mathtt{v}})) \longrightarrow \mathtt{new}\ C(\bar{\mathtt{v}})}\ (\text{E-3}) \qquad \frac{\forall \mathtt{T}_i \in \bar{\mathtt{T}}.\mathtt{T}_i <: \mathtt{T}}{(\mathtt{Bag} < \mathtt{T} >)\{\bar{\mathtt{v}}\} \longrightarrow \{\bar{\mathtt{v}}\}}\ (\text{E-4})$$

$$\mathtt{new}\ C(\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e\ (\text{E-5}) \qquad (\mathtt{throw}\ v_e).m(\bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e\ (\text{E-6})$$

$$v.m(\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e\ (\text{E-7}) \qquad \{\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}\} \longrightarrow \mathtt{throw}\ v_e\ (\text{E-8})$$

$$(\mathtt{throw}\ v_e).f \longrightarrow \mathtt{throw}\ v_e\ (\text{E-9}) \qquad (C)(\mathtt{throw}\ v_e) \longrightarrow \mathtt{throw}\ v_e\ (\text{E-10})$$

$$\mathtt{throw}\ \mathtt{throw}\ v_e \longrightarrow \mathtt{throw}\ v_e\ (\text{E-11}) \qquad \mathtt{try}\ \{v;\}\ \overline{\mathtt{CK}} \longrightarrow v\ (\text{E-12})$$

$$\frac{v = \mathtt{new}\ C(\bar{\mathtt{v}}) \qquad C <: C_e}{\mathtt{try}\ \{\mathtt{throw}\ v;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow [v/x]e}\ (\text{E-13})$$

$$\frac{v = \mathtt{new}\ C(\bar{\mathtt{v}}) \qquad C \not<: C_e}{\mathtt{try}\ \{\mathtt{throw}\ v;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow \mathtt{throw}\ v}\ (\text{E-14})$$

Figure 2.5: Computation rules for FJ$^+$

the least upper bound of types `T` and `T'`. It is extended to sets of types in the obvious way. We write $\sqcup\bar{\mathtt{T}}$ to denote the least upper bound of all types in the set $\bar{\mathtt{T}}$. The join operator is undefined for joins between class types and bag types.

2.3.2 EventJava

EventJava (EJ) is a core calculus that extends FJ$^+$ with support for events and event registration. Table 2.1 gives the syntax for EJ, showing the extensions from FJ$^+$. The set of EJ expressions additionally contains a built-in function `reg`, which registers a set of events and a callback with the scheduler. For use with the `reg` function, we assume the system scheduler implementation includes a class `Event`. Further, EJ provides a built-in class `Callback`:

```
class Callback extends Object {
  Object run(Object retVal) { return new Object(); }
}
```

$\boxed{e \longrightarrow e'}$

$$\frac{e_0 \longrightarrow e_0'}{e_0.f \longrightarrow e_0'.f}\ \text{(E-15)} \qquad \frac{e_0 \longrightarrow e_0'}{e_0.m(\bar{\mathtt{e}}) \longrightarrow e_0'.m(\bar{\mathtt{e}})}\ \text{(E-16)}$$

$$\frac{e_i \longrightarrow e_i'}{v_0.m(\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}}) \longrightarrow v_0.m(\bar{\mathtt{v}}, e_i', \bar{\mathtt{e}})}\ \text{(E-17)} \qquad \frac{e_i \longrightarrow e_i'}{\mathtt{new}\ C(\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}}) \longrightarrow \mathtt{new}\ C(\bar{\mathtt{v}}, e_i', \bar{\mathtt{e}})}\ \text{(E-18)}$$

$$\frac{e_0 \longrightarrow e_0'}{(T)e_0 \longrightarrow (T)e_0'}\ \text{(E-19)} \qquad \frac{e_0 \longrightarrow e_0'}{\{\bar{\mathtt{v}}, e_0, \bar{\mathtt{e}}\} \longrightarrow \{\bar{\mathtt{v}}, e_0', \bar{\mathtt{e}}\}}\ \text{(E-20)}$$

$$\frac{e_0 \longrightarrow e_0'}{\mathtt{throw}\ e_0 \longrightarrow \mathtt{throw}\ e_0'}\ \text{(E-21)} \qquad \frac{e_t \longrightarrow e_t'}{\begin{array}{c}\mathtt{try}\ \{e_t;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow \\ \mathtt{try}\ \{e_t';\}\mathtt{catch}\ (C_e\ x)\ \{e;\}\end{array}}\ \text{(E-22)}$$

Figure 2.6: Congruence rules for FJ$^+$

The type signature of `reg` is `Bag<Event>` $\times$ `Callback` $\rightarrow$ `Object`.

The operational semantics of EJ programs is given with respect to a *program state*. An EventJava program state σ_e consists of (1) an expression representing the in-progress evaluation of the currently executing callback, and (2) a bag $\mathcal{E}$ of pending event registrations of type `Bag<Event>` $\times$ `Callback`. In the operational rules, we write this state as $e|\mathcal{E}$.

After the initialization expression has been evaluated, the program enters an *event processing loop*. The event processing loop runs until the set of event registrations $\mathcal{E}$ is empty. In each iteration of the loop, one event registration (s, c) is nondeterministically removed from $\mathcal{E}$. An event η is then nondeterministically chosen from s, and the callback function $c.\mathtt{run}(\eta)$ is executed. A registration is one-time; after the selection of an event η, the entire event set s is excluded from further consideration, unless the set is explicitly re-registered with the scheduler. If an empty set of events is passed along with a callback, the callback is guaranteed to be called at some point in the future. An instance of `NullEvent` is then passed to the callback. This semantics models a single-threaded event server.

Note that the parameter of a callback's `run` method has a type of `Object`, even though it will be passed an `Event`. This slight discrepancy simplifies the translation between CTJ (which also uses callbacks for completion of asynchronous method calls) and EJ. As a result, the body of each callback class must downcast the `retVal` parameter to `Event`. Downcasts could be avoided by extending EJ and CTJ with generics, as in Featherweight Generic Java [IPW01].

We define evaluation contexts for EventJava in the same manner as FJ$^+$. The grammar for evaluation contexts has the following extensions for syntactic forms specific to EventJava:

$$E ::= \ \ldots \mid \texttt{reg}(E, e) \mid \texttt{reg}(\{\bar{\texttt{v}}\}, E)$$

Figure 2.7 lists the operational rules unique to EventJava. In these rules, s ranges over event sets and η ranges over events. We define two evaluation relations for EventJava programs. The $\longrightarrow_e$ relation extends the FJ$^+$ $\longrightarrow$ relation with congruence rules to evaluate the arguments of a `reg` call. The $\Longrightarrow_e$ relation then extends this relation to EventJava program states. Rule E_e-Con incorporates the $\longrightarrow$ relation into the context of a program state.

Each transition rule of the $\Longrightarrow_e$ relation has an associated **observable action**, denoted by a label above the transition arrow. This action represents the impact of the transition on the scheduler. A label may be either: (1) an event set, representing a registration with the scheduler, (2) a single event, representing the selection of an event by the scheduler, or (3) ϵ, representing a transition which has no impact on the scheduler's state.

$\boxed{e \longrightarrow_e e'}$

Figure 2.5, rules E-1 - E-14.
Figure 2.6, rules E-15 - E-22.

$$\frac{e_0 \longrightarrow_e e_0'}{\texttt{reg}\ e_0, e_1 \longrightarrow_e \texttt{reg}\ e_0', e_1}\ (E_e\text{-}23) \qquad \frac{e_0 \longrightarrow_e e_0'}{\texttt{reg}\ v, e_0 \longrightarrow_e \texttt{reg}\ v, e_0'}\ (E_e\text{-}24)$$

$$\frac{}{\texttt{reg throw}\ v, e \longrightarrow_e \texttt{throw}\ v}\ (E_e\text{-}25) \qquad \frac{}{\texttt{reg}\ v_0, \texttt{throw}\ v_1 \longrightarrow_e \texttt{throw}\ v_1}\ (E_e\text{-}26)$$

$\boxed{e|\mathcal{E} \overset{l}{\Longrightarrow}_e e'|\mathcal{E}'}$

$$\frac{e \longrightarrow_e e'}{e|\mathcal{E} \overset{\epsilon}{\Longrightarrow}_e e'|\mathcal{E}}\ (E_e\text{-}\textsc{Con}) \qquad \frac{}{E[\texttt{reg}\ v_0, v_1]|\mathcal{E} \overset{v_0}{\Longrightarrow}_e E[v_1]|\mathcal{E} \cup (v_0, v_1)}\ (E_e\text{-}\textsc{Reg})$$

$$\frac{(s, v_{cb}) \in \mathcal{E} \qquad \eta \in s}{v|\mathcal{E} \overset{\eta}{\Longrightarrow}_e v_{cb}.\texttt{run}(\eta)|\mathcal{E} \setminus (s, v_{cb})}\ (E_e\text{-}\textsc{Run})$$

$$\frac{(\emptyset, v_{cb}) \in \mathcal{E} \qquad \eta_0 = \texttt{new NullEvent()}}{v|\mathcal{E} \overset{\eta_0}{\Longrightarrow}_e v_{cb}.\texttt{run}(\eta_0)|\mathcal{E} \setminus (\emptyset, v_{cb})}\ (E_e\text{-}\eta_\emptyset\textsc{Run}) \qquad \frac{}{\begin{array}{c}\texttt{throw}\ v_e|\mathcal{E} \overset{\epsilon}{\Longrightarrow}_e \\ \texttt{throw}\ v_e|\emptyset\end{array}}\ (E_e\text{-}\textsc{Throw})$$

Figure 2.7: Operational Semantics of EJ

2.3.3 CoreTaskJava

The syntax of CoreTaskJava (CTJ) extends FJ$^+$ with three new constructs: the `spawn` syntax for creating tasks; a `wait` primitive, which accepts a set of events and blocks until one of them occurs; and asynchronous methods via the `async` modifier. A task in CTJ subclasses from a built-in class `Task`:

```
class Task extends Callback {}
```

A task's `run` method contains the body of the task and is called after a task is spawned. [2]

[2]In the implementation, `Task` does not subclass from `Callback` and its `run` method takes no parameters. We subclass from `Callback` here to simplify the presentation of the formalization.

Informally, tasks are modeled as concurrent threads of execution that block when waiting for an asynchronous call to complete. We define evaluation contexts for CTJ expressions in the same manner as FJ$^+$. The grammar for evaluation contexts has the following extensions for syntactic forms specific to CTJ:

$$E ::= \ldots \mid \texttt{wait}\ E \mid \texttt{spawn}\ E$$

As with EJ, the semantics is defined with respect to a *program state*. A *CTJ program state* σ_c consists of: (1) an expression representing the in-progress evaluation of the currently executing task, and (2) a set $\mathcal{B}$ of (`Bag<Event>`, $E[]$) pairs representing the evaluation contexts of blocked tasks and the events that each task is blocked on. In the operational rules, we write this state as $e|\mathcal{B}$.

After a CTJ program's initialization expression has been evaluated, the program nondeterministically selects a task from the blocked set, then nondeterministically selects an event from the task's event set, and evaluates the task until it either terminates or blocks again. Another task is then selected nondeterministically. This process repeats until the program reaches the state where the current expression is a value and the blocked set is empty. Tasks and the associated event sets are added to $\mathcal{B}$ through calls to `wait`. In addition, the `spawn` of a task is modeled by placing the task in $\mathcal{B}$ with an empty event set, ensuring that the task will eventually be scheduled.

Figure 2.8 lists the operational rules for CoreTaskJava. As with EventJava, the rules are written using two relations. The $\longrightarrow_c$ relation extends the FJ$^+$ $\longrightarrow$ relation with congruence rules to evaluate the arguments of `wait` and `spawn` calls. The $\Longrightarrow_c$ relation (defined in Figure 2.8) then extends this relation to CoreTaskJava program states. The evaluation of the $\longrightarrow$ rules in the context

$\boxed{e \longrightarrow_c e'}$

Figure 2.5, rules E-1 - E-14.
Figure 2.6, rules E-15 - E-22.

$$\frac{e_0 \longrightarrow_c e_0'}{\texttt{wait}\ e_0 \longrightarrow_c \texttt{wait}\ e_0'}\ (E_c\text{-}23) \qquad \frac{}{\texttt{wait throw}\ v \longrightarrow_c \texttt{throw}\ v}\ (E_c\text{-}24)$$

$$\frac{e_0 \longrightarrow_c e_0'}{\texttt{spawn}\ e_0 \longrightarrow_c \texttt{spawn}\ e_0'}\ (E_c\text{-}25) \qquad \frac{}{\texttt{spawn throw}\ v \longrightarrow_c \texttt{throw}\ v}\ (E_c\text{-}26)$$

$\boxed{e|\mathcal{B} \overset{l}{\Longrightarrow}_c e'|\mathcal{B}'}$

$$\frac{e \longrightarrow_c e'}{e|\mathcal{B} \overset{\epsilon}{\Longrightarrow}_c e'|\mathcal{B}}\ (E_c\text{-}\textsc{Con}) \qquad \frac{}{\texttt{throw}\ v|\mathcal{B} \overset{\epsilon}{\Longrightarrow}_c \texttt{throw}\ v|\emptyset}\ (E_c\text{-}\textsc{Throw})$$

$$\frac{w = \texttt{wait}\ \{\bar{\mathtt{v}}\}}{E[w]|\mathcal{B} \overset{\{\bar{\mathtt{v}}\}}{\Longrightarrow}_c \texttt{new Object()}|\mathcal{B} \cup (\{\bar{\mathtt{v}}\}, E[])}\ (E_c\text{-}\textsc{Wait})$$

$$\frac{(\{\bar{\mathtt{v}}\}, E[]) \in \mathcal{B} \qquad \eta \in \{\bar{\mathtt{v}}\}}{v|\mathcal{B} \overset{\eta}{\Longrightarrow}_c E[\eta]|\mathcal{B} \setminus (\{\bar{\mathtt{v}}\}, E[])}\ (E_c\text{-}\textsc{Run})$$

$$\frac{(\emptyset, E[]) \in \mathcal{B} \qquad \eta_0 = \texttt{new NullEvent()}}{v|\mathcal{B} \overset{\eta_0}{\Longrightarrow}_c E[\eta_0]|\mathcal{B} \setminus (\emptyset, E[])}\ (E_c\text{-}\eta_\emptyset\textsc{Run})$$

$$\frac{}{E[\texttt{spawn}\ C(\bar{\mathtt{v}})]|\mathcal{B} \overset{\emptyset}{\Longrightarrow}_c E[\texttt{new}\ C(\bar{\mathtt{v}})]|\mathcal{B} \cup (\emptyset, (\texttt{new}\ C(\bar{\mathtt{v}})).\texttt{run}([]))}\ (E_c\text{-}\textsc{Spn})$$

Figure 2.8: Operational Semantics of CTJ

of a program state is handled by Rule E_c-Con. Each transition of the $\Longrightarrow$ relation is labeled with an *observable action*, as defined above for EventJava.

2.3.4 Type Checking

FJ$^+$ and EJ Typing judgments for expressions in FJ$^+$ have the form $\Gamma \vdash e : T|\bar{\tau}$, where the environment Γ maps variables to types, T is a type (using the type grammar in Table 2.1), and $\bar{\tau}$ is set of exception classes (`Throwable` or a subclass of `Throwable`). For checking exceptions, we introduce a new relation $\bar{\tau} \subseteq: \bar{\tau}'$, which is true when, for each class $C \in \bar{\tau}$, there exists a class $C' \in \bar{\tau}'$ such that $C <: C'$.

Figure 2.9 lists the typing rules for FJ$^+$. These rules extend the typing rules of FJ by adding typing of bags and tracking the set of exceptions $\bar{\tau}$ which may be thrown by an expression. In particular, rule T-9 assigns an arbitrary type `T` to a throw statement, based on the statement's context.

The *override* function (rule T-11) is changed to check that the set of exceptions thrown by a method's body is a subset of those declared to be thrown by the method signature. The auxiliary function $mtype(C, m)$ returns the type signature of method m in class C. A method's type signature has the form: $\bar{\mathtt{T}}_f \rightarrow \mathtt{T}_r$ `throws` $\bar{\tau}$, where $\bar{\mathtt{T}}_f$ represents the types of the formal arguments, $\mathtt{T}_r$ represents the type of the return value, and $\bar{\tau}$ represents the exceptions declared to be thrown by the method. EventJava extends the rules of FJ$^+$ with an additional rule to assign a type to `reg` expressions (Figure 2.10).

CoreTaskJava Typing rules for CTJ are listed in Figures 2.11 and 2.12. Typing statements for expressions in CTJ have the form $\Gamma, C, M \vdash e : T|\bar{\tau}$, where C is the name of the enclosing class and M the definition of the enclosing method. The auxiliary function $isaync(M)$ returns `true` if the method definition M has an `async` modifier and `false` otherwise. Likewise, $isasync(C, m)$ returns true if the definition of method m in class C has an `async` modifier.

Rules T$_c$-3 and T$_c$-15 ensure that asynchronous calls and `wait` calls may only be made by asynchronous methods or the `run` method of a task. Rules T-11 and T-12 of FJ$^+$ have each been split into two cases, one for asynchronous methods and one for standard methods. This ensures that asynchronous methods

$\boxed{\Gamma \vdash e : T|\bar{\tau}}$

$$\frac{x : \mathtt{T} \in \Gamma}{\Gamma \vdash x : \mathtt{T}|\emptyset}\ \text{(T-1)} \qquad \frac{\Gamma \vdash e_0 : C_0|\bar{\tau} \qquad \mathit{fields}(C_0) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{\Gamma \vdash e_0.f_i : \mathtt{T}_i|\bar{\tau}}\ \text{(T-2)}$$

$$\frac{\Gamma \vdash e_0 : C_0|\bar{\tau}_0 \qquad \mathit{mtype}(m, C_0) = \bar{\mathtt{T}}_f \to \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}_m \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a \qquad \bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f}{\Gamma \vdash e_0.m(\bar{\mathtt{e}}) : \mathtt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a}\ \text{(T-3)}$$

$$\frac{\mathit{fields}(C_c) = \bar{\mathtt{T}}_c\ \bar{\mathtt{f}} \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_e|\bar{\tau} \qquad \bar{\mathtt{T}}_e <: \bar{\mathtt{T}}_c}{\Gamma \vdash \mathtt{new}\ C_c(\bar{\mathtt{e}}) : C_c|\bar{\tau}}\ \text{(T-4)} \qquad \frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T}_0 <: \mathtt{T}}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-5)}$$

$$\frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} <: \mathtt{T}_0 \qquad \mathtt{T} \neq \mathtt{T}_0}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-6)}$$

$$\frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} \not<: \mathtt{T}_0 \qquad \mathtt{T}_0 \not<: \mathtt{T} \qquad \mathit{stupid\ warning}}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-7)}$$

$$\frac{\Gamma \vdash \forall e_i \in \bar{\mathtt{e}}\ .\ e_i : \mathtt{T}_i \qquad \mathtt{T} = \sqcup \mathtt{T}_i \qquad \bar{\tau} = \cup \bar{\tau}_i}{\Gamma \vdash \{\bar{\mathtt{e}}\} : \mathtt{Bag} < \mathtt{T} > |\bar{\tau}}\ \text{(T-8)} \qquad \frac{\Gamma \vdash e_0 : C_0|\bar{\tau} \qquad C_0 <: \mathtt{Throwable}}{\Gamma \vdash \mathtt{throw}\ e_0 : \mathtt{T}|\bar{\tau} \cup C_0}\ \text{(T-9)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_t : \mathtt{T}_t|\bar{\tau}_t \\ C_e <: \mathtt{Exception} \qquad \Gamma \cup x : C_e \vdash e_c : \mathtt{T}_c|\bar{\tau}_c \qquad \bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}\end{array}}{\Gamma \vdash \mathtt{try}\ \{e_t;\}\ \mathtt{catch}\ (C_e\ x)\ \{e_c;\}\ :\ \mathtt{T}_t \sqcup \mathtt{T}_c|\bar{\tau}'_t \cup \bar{\tau}_c}\ \text{(T-10)}$$

$\boxed{\mathit{override}(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}$

$$\frac{\begin{array}{c}\mathit{mtype}(m, D) = \bar{\mathtt{T}}_d \to \mathtt{T}_{rd}\ \mathtt{throws}\ \bar{\tau}_d \Rightarrow \\ \bar{\mathtt{T}}_c = \bar{\mathtt{T}}_d\ \wedge\ \mathtt{T}_{rc} = \mathtt{T}_{rd}\ \wedge\ \bar{\tau} \subset: \bar{\tau}_d\end{array}}{\mathit{override}(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}\ \text{(T-11)}$$

$\boxed{\text{M OK in C}}$

$$\frac{\begin{array}{c}\{\bar{\mathtt{x}} : \bar{\mathtt{T}}|\emptyset, \mathtt{this} : C|\emptyset\} \vdash e_0 : \mathtt{T}_e|\bar{\tau}_e \qquad \mathtt{T}_e <: \mathtt{T}_r \\ \bar{\tau}_e \subset: \bar{\tau} \qquad CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} \qquad \mathit{override}(m, D, \bar{\mathtt{T}} \to \mathtt{T}_r, \bar{\tau})\end{array}}{\mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ \text{(T-12)}$$

$\boxed{\text{C OK}}$ $\qquad$ $\boxed{\text{P OK}}$

$$\frac{\begin{array}{c}K = C(\bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s, \bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c)\{\mathtt{super}(\bar{\mathtt{f}}_s); \mathtt{this}.\bar{\mathtt{f}}_c = \bar{\mathtt{f}}_c;\} \\ \mathit{fields}(C_s) = \bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s \qquad \overline{M}\ \text{OK in C}\end{array}}{\mathtt{class}\ C\ \mathtt{extends}\ C_s\ \{\bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c;\ K\ \overline{M}\}\ \text{OK}}\ \text{(T-13)} \qquad \frac{\vdash e : \mathtt{T}|\emptyset \qquad \overline{CL}\ \text{OK}}{\vdash^f \overline{CL}\ \mathtt{return}\ e\ \text{OK}}\ \text{(T-14)}$$

Figure 2.9: Typing rules for FJ$^+$

$\boxed{\Gamma \vdash e : T|\bar{\tau}}$

Figure 2.9, rules T-1 - T-13.

$$\frac{\Gamma \vdash e_s : \texttt{Bag<Event>}|\bar{\tau}_s \qquad \Gamma \vdash e_c : \texttt{Callback}|\bar{\tau}_c}{\Gamma \vdash \texttt{reg}\ e_s, e_c : \texttt{Event}|\bar{\tau}_s \cup \bar{\tau}_c}\ (T_e\text{-}14)$$

$\boxed{\text{P OK}}$ $\boxed{\mathcal{E}\text{ OK}}$

$$\frac{\vdash e : \texttt{T}|\emptyset \qquad \overline{CL}\text{ OK}}{\vdash^e \overline{CL}\ \texttt{return}\ e\text{ OK}}\ (T_e\text{-}15)$$

$$\frac{\begin{array}{c}\vdash \{\bar{\mathtt{v}}_e\} : \texttt{Bag} < \texttt{Event} > \\ \vdash \texttt{new}\ C(\bar{\mathtt{v}}_c) : C|\emptyset \\ C <: \texttt{Callback}\end{array}}{\vdash \{\overline{(\{\bar{\mathtt{v}}_e\}, \texttt{new}\ C(\bar{\mathtt{v}}_c))}\}\text{ OK}}\ (T_e\text{-}16)$$

Figure 2.10: Typing rules for EJ

may only override asynchronous methods and non-asynchronous methods may only override non-asynchronous methods. Rule T$_c$-12b also verifies that the `run` method of a task has not been declared `async`.

Rule T$_c$-14 ensures that sets of blocked tasks are well-formed. If $\vdash \mathcal{B}$ OK, then $\mathcal{B}$ consists of a set of pairs, where the first element of the pair is a set of events and the second element of the pair is an evaluation context $E[]$. The evaluation context must be well-typed when $[]$ is replaced with an event.

2.4 Properties of CoreTaskJava programs

We now describe the central formal properties of CTJ programs: subject reduction, progress, and soundness.

We start by defining *normal forms* for CTJ and EJ, special forms for expressions obtained by rewriting until no rewrite rule from $\longrightarrow_e$ or $\longrightarrow_c$ is possible. A CTJ expression e is in *normal form* if it matches one of the following forms: $E[\texttt{spawn}\ C(\bar{\mathtt{v}})]$, $E[\texttt{wait}\ v]$, $E[(T)v]$ where the type of non-exception value v is not a subtype of T, $\texttt{throw}\ v_e$, or a non-exception value v. Similarly, an EJ expression

$\boxed{\Gamma, C, M \vdash e : T|\bar{\tau}}$

$$\frac{x : T \in \Gamma}{\Gamma, C, M \vdash x : T|\emptyset}\ (\mathrm{T}_c\text{-}1) \qquad \frac{\Gamma, C, M \vdash e_0 : C_0|\bar{\tau} \qquad fields(C_0) = \bar{\texttt{T}}\ \bar{\texttt{f}}}{\Gamma, C, M \vdash e_0.f_i : \texttt{T}_i|\bar{\tau}}\ (\mathrm{T}_c\text{-}2)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_0 : C_0|\bar{\tau}_0 \\ mtype(m, C_0) = \bar{\texttt{T}}_f \rightarrow \texttt{T}_r\ \texttt{throws}\ \bar{\tau}_m \qquad \Gamma, C, M \vdash \bar{\texttt{e}} : \bar{\texttt{T}}_a|\bar{\tau}_a \qquad \bar{\texttt{T}}_a <: \bar{\texttt{T}}_f \\ isasync(m, C_0) \implies \\ (C <: \texttt{Task} \wedge methname(M) = \texttt{run})\ \vee\ isasync(M)\end{array}}{\Gamma, C, M \vdash e_0.m(\bar{\texttt{e}}) : \texttt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a}\ (\mathrm{T}_c\text{-}3)$$

$$\frac{fields(C_c) = \bar{\texttt{T}}_c\ \bar{\texttt{f}} \qquad \Gamma, C, M \vdash \bar{\texttt{e}} : \bar{\texttt{T}}_e|\bar{\tau} \qquad \bar{\texttt{T}}_e <: \bar{\texttt{T}}_c}{\Gamma, C, M \vdash \texttt{new}\ C_c(\bar{\texttt{e}}) : C_c|\bar{\tau}}\ (\mathrm{T}_c\text{-}4)$$

$$\frac{\Gamma, C, M \vdash e_0 : \texttt{T}_0|\bar{\tau} \qquad \texttt{T}_0 <: \texttt{T}}{\Gamma, C, M \vdash (\texttt{T})e_0 : \texttt{T}|\bar{\tau}}\ (\mathrm{T}_c\text{-}5) \frac{\Gamma, C, M \vdash e_0 : \texttt{T}_0|\bar{\tau} \qquad \texttt{T} <: \texttt{T}_0 \qquad \texttt{T} \neq \texttt{T}_0}{\Gamma, C, M \vdash (\texttt{T})e_0 : \texttt{T}|\bar{\tau}}\ (\mathrm{T}_c\text{-}6)$$

$$\frac{\Gamma, C, M \vdash e_0 : \texttt{T}_0|\bar{\tau} \qquad \texttt{T} \not<: \texttt{T}_0 \qquad \texttt{T}_0 \not<: \texttt{T} \qquad \textit{stupid warning}}{\Gamma, C, M \vdash (\texttt{T})e_0 : \texttt{T}|\bar{\tau}}\ (\mathrm{T}_c\text{-}7)$$

$$\frac{\Gamma, C, M \vdash \forall e_i \in \bar{\texttt{e}}\ .\ e_i : \texttt{T}_i \qquad \texttt{T} = \sqcup \texttt{T}_i \qquad \bar{\tau} = \cup \bar{\tau}_i}{\Gamma, C, M \vdash \{\bar{\texttt{e}}\} : \texttt{Bag} < \texttt{T} > |\bar{\tau}}\ (\mathrm{T}_c\text{-}8)$$

$$\frac{\Gamma, C, M \vdash e_0 : C_0|\bar{\tau} \qquad C_0 <: \texttt{Throwable}}{\Gamma, C, M \vdash \texttt{throw}\ e_0 : \texttt{T}|\bar{\tau} \cup C_0}\ (\mathrm{T}_c\text{-}9)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_t : \texttt{T}_t|\bar{\tau}_t \\ C_e <: \texttt{Exception} \qquad \Gamma \cup x : C_e, C, M \vdash e_c : \texttt{T}_c|\bar{\tau}_c \qquad \bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \texttt{C}_e\}\end{array}}{\Gamma, C, M \vdash \texttt{try}\ \{e_t;\}\ \texttt{catch}\ (C_e\ x)\ \{e_c;\}\ :\ \texttt{T}_t \sqcup \texttt{T}_c|\bar{\tau}'_t \cup \bar{\tau}_c}\ (\mathrm{T}_c\text{-}10)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_0 : \texttt{Bag<Event>}|\bar{\tau} \\ (C <: \texttt{Task} \wedge methname(M) = \texttt{run})\ \vee\ isasync(M)\end{array}}{\Gamma, C, M \vdash \texttt{wait}\ e_0 : \texttt{Event}|\bar{\tau}}\ (\mathrm{T}_c\text{-}15)$$

$$\frac{\Gamma, C, M \vdash e_0 : C|\bar{\tau} \qquad C <: \texttt{Task}}{\Gamma, C, M \vdash \texttt{spawn}\ e_0 : C|\bar{\tau}}\ (\mathrm{T}_c\text{-}16)$$

Figure 2.11: Expression typing rules for CTJ

$\boxed{override(m, D, \bar{\mathtt{T}}_c \rightarrow \mathtt{T}_{rc}, \bar{\tau})}$

$$\frac{\begin{array}{c} mtype(m, D) = \bar{\mathtt{T}}_d \rightarrow \mathtt{T}_{rd}\ \mathtt{throws}\ \bar{\tau}_d \Rightarrow \\ \bar{\mathtt{T}}_c = \bar{\mathtt{T}}_d\ \wedge\ \mathtt{T}_{rc} = \mathtt{T}_{rd}\ \wedge\ \bar{\tau} \subset: \bar{\tau}_d \end{array}}{override(m, D, \bar{\mathtt{T}}_c \rightarrow \mathtt{T}_{rc}, \bar{\tau})}\ (\mathrm{T}_c\text{-}11)$$

$\boxed{\text{M OK in C}}$

$$\frac{\begin{array}{c} \{\bar{\mathtt{x}} : \bar{\mathtt{T}}|\emptyset, \mathtt{this} : C|\emptyset\}, C, M \vdash e_0 : \mathtt{T}_e|\bar{\tau}_e \\ \mathtt{T}_e <: \mathtt{T}_r \qquad \bar{\tau}_e \subset: \bar{\tau} \qquad CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} \\ override(m, D, \bar{\mathtt{T}} \rightarrow \mathtt{T}_r, \bar{\tau}) \qquad \neg isasync(m, D) \end{array}}{\mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ (\mathrm{T}_c\text{-}12\mathrm{A})$$

$$\frac{\begin{array}{c} \{\bar{\mathtt{x}} : \bar{\mathtt{T}}, \mathtt{this} : C\}, C, M \vdash e_0 : \mathtt{T}_e|\bar{\tau}_e \\ \mathtt{T}_e <: \mathtt{T}_r \qquad \bar{\tau}_e \subset: \bar{\tau} \qquad CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} \\ override(m, D, \bar{\mathtt{T}} \rightarrow \mathtt{T}_r, \bar{\tau}) \qquad isasync(m, D) \qquad \neg(C <: \mathtt{Task}\ \wedge\ m = \mathtt{run}) \end{array}}{\mathtt{async}\ \mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ (\mathrm{T}_c\text{-}12\mathrm{B})$$

$\boxed{\text{C OK}}$

$$\frac{\begin{array}{c} K = C(\bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s, \bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c)\{\mathtt{super}(\bar{\mathtt{f}}_s); \mathtt{this}.\bar{\mathtt{f}}_c = \bar{\mathtt{f}}_c;\} \\ fields(C_s) = \bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s \qquad \overline{M}\ \text{OK in C} \end{array}}{\mathtt{class}\ C\ \mathtt{extends}\ C_s\ \{\bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c;\ K\ \overline{M}\}\ \text{OK}}\ (\mathrm{T}_c\text{-}13)$$

$\boxed{\text{P OK}}$

$$\frac{\vdash e : \mathtt{T}|\emptyset \qquad \overline{CL}\ \text{OK}}{\vdash^c \overline{CL}\ \mathtt{return}\ e\ \text{OK}}\ (\mathrm{T}_c\text{-}14)$$

$\boxed{\mathcal{B}\ \text{OK}}$

$$\frac{\begin{array}{c} \vdash \{\bar{\mathtt{v}}\} : \mathtt{Bag} < \mathtt{Event} > \\ \vdash \bar{E}[\mathtt{new\ Event}()] : \bar{\mathtt{T}} \end{array}}{\vdash \{\overline{(\{\bar{\mathtt{v}}\}, E[])}\}\ \text{OK}}\ (\mathrm{T}_c\text{-}17)$$

Figure 2.12: Method, class, program, and blocked task typing rules for CTJ

e is in normal form if it matches one of the following forms: $E[\texttt{reg}\, v_1, v_2]$, $E[(T)v]$ where the type of non-exception value v is not a subtype of T, $\texttt{throw}\ v_e$, or a non-exception value v.

Lemma 1 (Normal forms). *If an (CTJ or EJ) expression e is in normal form, either no reduction of the expression e by the $\Longrightarrow$ relation is possible, or the reduction step must be an observable action.*

Proof. Immediate from the structure of each normal form and the $\Longrightarrow$ relation. □

2.4.0.1 Subject Reduction

We are now ready to relate the evaluation relation to the typing rules. Subject reduction states that, if a CTJ program in a well-typed state takes an evaluation step, the resulting program state is well-typed as well. We start first with a theorem for the $\longrightarrow$ relation and then extend this to the $\Longrightarrow_c$ relation.

Theorem 1 ($\longrightarrow_c$ Subject Reduction). *If $\Gamma \vdash e : \texttt{T}_e|\bar{\tau}_e$ and $e \longrightarrow_c e'$, then $\Gamma \vdash e' : \texttt{T}_{e'}|\bar{\tau}_{e'}$ for some $\texttt{T}_{e'}|\bar{\tau}_{e'} <: \texttt{T}_e|\bar{\tau}_e$.*

The proof of this theorem is based on several technical lemmas. When the given lemma is standard for soundness proofs, we omit the proof of the lemma for brevity.

Lemma 2. *If $mtype(m, C) = \bar{\texttt{T}} \rightarrow \texttt{T}_r\ \texttt{throws}\ \bar{\tau}$, then $mtype(m, C') = \bar{\texttt{T}} \rightarrow \texttt{T}_r\ \texttt{throws}\ \bar{\tau}'$, where $\bar{\tau}' \subseteq: \bar{\tau}$, for all $C' <: C$.*

Lemma 3 (Non-exception values). *Non-exception values of the form $v ::= \texttt{new}\ C() \mid \texttt{new}\ C(\bar{\texttt{v}}) \mid \{\bar{\texttt{v}}\}$ have a type of the form $\texttt{T}|\emptyset$.*

Lemma 4 (Term substitution preserves typing)**.** *If* $\Gamma \cup \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash e_0 : \mathtt{T}_0|\bar{\tau}_0$*, and* $\Gamma \vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}'|\emptyset$ *where* $\bar{\mathtt{T}}' <: \bar{\mathtt{T}}$*, then* $\Gamma \vdash [\bar{\mathtt{v}}/\bar{\mathtt{x}}]e_0 : \mathtt{T}_0'|\bar{\tau}_0$ *for some* $\mathtt{T}_0' <: \mathtt{T}_0$*.*

Lemma 5 (Weakening)**.** *If* $\Gamma \vdash e : \mathtt{T}|\bar{\tau}$*, then* $\Gamma \cup x : \mathtt{T}' \vdash e : \mathtt{T}|\bar{\tau}$*.*

Lemma 6 (Return types)**.** *If* $mtype(m, C) = \bar{\mathtt{T}} \rightarrow \mathtt{T}_r$ $\mathtt{throws}$ $\bar{\tau}$ *and* $mbody(m, C) = (\bar{\mathtt{x}}, e)$*, then for some* C' *where* $C <: C'$*, there exists some* $\mathtt{T}_r'$ *and* $\bar{\tau}'$ *such that* $\mathtt{T}_r' <: \mathtt{T}_r$*,* $\bar{\tau} \subseteq: \bar{\tau}'$*, and* $\Gamma \cup \bar{\mathtt{x}} : \bar{\mathtt{T}} \cup \mathtt{this} : C' \vdash e : \mathtt{T}_r'|\bar{\tau}'$*.*

Lemma 7 (Subtyping of joined types)**.** $\mathtt{T} <: (\mathtt{T} \sqcup \mathtt{T}')$ *for all types* $\mathtt{T}, \mathtt{T}'$*.*

Lemma 8 (Subsets and the $\subseteq:$ relation)**.** *If* $\bar{\tau} \subseteq \bar{\tau}'$*, then* $\bar{\tau} \subseteq: \bar{\tau}'$*.*

Proof for Theorem 1. By induction on the derivation of $e \longrightarrow e'$, using a case analysis on the evaluation rules. For space reasons, the full proof is omitted here. It may be found in [FMM06].

□

We now extend subject reduction to CTJ program states.

Theorem 2 ($\Longrightarrow_c$ Subject Reduction)**.** *If* $\vdash e : \mathtt{T}_e|\bar{\tau}_e$ *and* $\mathcal{B}$ *OK and* $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$*, then* $\vdash e' : \mathtt{T}_{e'}|\bar{\tau}_{e'}$ *and* $\mathcal{B}'$ *OK.*

Proof. By using a case analysis on the evaluation rules:

- Rule E_c-Con (step via $\longrightarrow_c$ relation): By theorem 1, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$. The blocked task set remains unchanged and $\mathcal{B}$ OK is a premise of the theorem.
- Rule E_c-Throw (throw): The expression e remains unchanged and, thus is well-typed. $\mathcal{B}'$ is the empty set, which is well-typed by rule T_c-17.

- Rule E_c-Wait (wait): The value `new Object()` is well-typed by rule T-4. By the premise of the theorem, $E[$`wait` $\{\bar{\mathtt{v}}\}]$ is well-typed and, by type rule T$_c$-15, the `wait` call has type `Event`$|\emptyset$. Thus, $E[$`new Event`$()]$ will be well-typed and $\mathcal{B}'$ OK by rule T$_c$-17.

- Rules E_c-Run and E_c-$\eta_\emptyset$Run (selection of an event): By rule T$_c$-17, $E[]$ is well-typed when the placeholder is replaced with an object of type `Event`. Thus, $E[\eta]$ and $E[\eta_0]$ are well-typed. $\mathcal{B}' = \mathcal{B} \setminus (\{\bar{\mathtt{v}}\}, E[])$ and is OK by rule T$_c$-17.

- Rule E_c-Spn (spawn): By type rules T-4 and T$_c$-16, `new` $C(\bar{\mathtt{v}})$ and `spawn` $C(\bar{\mathtt{v}})$ both have type $C|\emptyset$. Thus, substituting `new` $C(\bar{\mathtt{v}})$ for `spawn` $C(\bar{\mathtt{v}})$ will result in a well-typed expression. From type rule T$_c$-16 we know that C <: `Task`. Thus, the expression $($`new` $C(\bar{\mathtt{v}})).run($`new Event`$())$ is well-typed by rule T-3 and $\mathcal{B}$' OK by T$_c$-17.

$\square$

Note that, when taking a step via the $\Longrightarrow_c$ relation, a subtype relationship no longer holds between the original expression and the new expression. If the evaluation context is of the form $E[$`wait`$\{\bar{\mathtt{v}}\}]$, the current expression may be replaced with one from the blocked set. This new expression need not have a subtype relationship with the previous one. As we shall see in theorem 5, this does not prevent us from making the usual claims about the type safety of CTJ programs.

2.4.0.2 Progress

The progress theorem states that well-typed CTJ programs cannot get "stuck", except when a bad cast occurs.

We first state a technical lemma needed for the proofs:

Lemma 9 (Canonical forms)**.** *The forms of values are related to their types as follows:*

- *If* $\vdash \nu : C|\bar{\tau}$*, and* ν *is a value, then* ν *has either the form* `new` $C(\bar{\mathtt{v}})$*, where* $\vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}_f$*, or the form* `throw new` $C(\bar{\mathtt{v}})$*, where* $C <:$ `Throwable`*.*
- *If* $\vdash \nu : \mathtt{Bag} < \mathtt{T} > |\bar{\tau}$*, and* ν *is a value, then* ν *has either the form* $\{\bar{\mathtt{v}}\}$*, where* $\vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}|\emptyset$ *and* $\mathtt{T} = \sqcup\bar{\mathtt{T}}$*, or the form* `throw new` $C(\bar{\mathtt{v}})$*, where* $C <:$ `Throwable`*.*

Theorem 3 ($\longrightarrow_c$ Progress)**.** *Suppose* $\vdash e : T|\bar{\tau}$*. Then either* e *is a normal form or there exists an expression* e' *such that* $e \longrightarrow_c e'$*.*

Proof. We prove theorem 3 by induction on the depth of the derivation of $\vdash e : T|\bar{\tau}$, using a case analysis on the last type rule for each derivation

The handling of exceptions in this proof is interesting due to the type rule for `throw` (T-9) – it assigns an arbitrary type. If the exception type $\bar{\tau}$, is non-empty, we must consider values of the form `throw` v_e whenever we consider values. Thus, expressions that may step by a computation rule may also step to a `throw` expression. This is reflected in our canonical forms lemma, which includes a `throw` expression for each value form. The full proof is omitted here (it appears in [FMM06]).

□

Theorem 4 ($\Longrightarrow_c$ Progress)**.** *Suppose* $\vdash e : T|\bar{\tau}$ *and* $\mathcal{B}$ *OK. Then one of the following must be true:*

- e *is a value and* $\mathcal{B} = \emptyset$*.*
- e *is of the form* $E[(T)v]$*, where* $E[]$ *is an evaluation context and* v *is a value whose type is not a subtype of* T*. We call this a runtime cast error.*

- *There exists an expression e' and set of blocked tasks $\mathcal{B}'$ such that $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$.*

Proof. From theorem 3, we know that either e is a normal form or there exists an expression e' such that $e \longrightarrow e'$. If $e \longrightarrow e'$, then, by evaluation rule E_c-Con, $e|\mathcal{B} \overset{\epsilon}{\Longrightarrow} e'|\mathcal{B}$. If e is a normal form, we perform a case analysis over normal forms and the contents of $\mathcal{B}$:

- If e has the form $E[\texttt{spawn}\ C(\bar{\mathtt{v}})]$ or $E[\texttt{wait}\ \{\bar{\mathtt{v}}\}]$, then e steps by rules E_c-Spn and E_c-Wait, respectively.
- If e has the form $E[(T)v]$ where v is a value whose type is not a subtype of T, then a runtime cast error has occurred, and no step may be taken.
- If e is an exception value, and $\mathcal{B}$ is not empty, then $e|\mathcal{B} \Longrightarrow e'|\mathcal{B}'$ by evaluation rule E_c-Throw.
- If e is a non-exception value and $\mathcal{B}$ is not empty, then $e|\mathcal{B} \Longrightarrow e'|\mathcal{B}'$ by either evaluation rule E_c-Run or E_c-$\eta_\emptyset$Run.
- If e is a value and $\mathcal{B} = \emptyset$, then the program has terminated and cannot step.

□

2.4.0.3 Soundness

We now combine theorems 2 and 4 to extend our guarantee to entire program executions. First, we need two lemmas regarding exceptions:

Lemma 10 ($\longrightarrow_c$ Exceptions)**.** *If* $\vdash e : \mathtt{T}|\bar{\tau}$, *and* $e \longrightarrow_c^* \texttt{throw new}\ C(\bar{\mathtt{v}})$, *then* $\exists \tau_i \in \bar{\tau} \mid C <: \tau_i$.

Proof. By induction over evaluation steps. By the inductive hypothesis, an arbitrary expression from the sequence e_i has type $\vdash e_i : T_i|\bar{\tau}_i$, where $\bar{\tau}_i \subseteq: \bar{\tau}$. If $e_i \longrightarrow e_{i+1}$, then, by theorem 2, $\vdash e_{i+1} : T_{i+1}|\bar{\tau}_{i+1}$, where $\bar{\tau}_{i+1} \subseteq: \bar{\tau}_i$.

From type rule T-9, $\vdash$ `throw new` $C(\bar{\mathtt{v}}) : \mathtt{T}|C$. If $e \longrightarrow^*$ `throw new` $C(\bar{\mathtt{v}})$, then $C \subseteq: \bar{\tau}$. □

Lemma 11 ($\Longrightarrow_c$ Exceptions)**.** *If* $\vdash e : \mathtt{T}|\emptyset$, *and* $e|\emptyset \Longrightarrow_c^* e'|\mathcal{B}'$, *then* $\vdash e' : T'|\emptyset$.

Proof. From lemma 10, we know that evaluation of e through the $\longrightarrow_c$ relation cannot yield an exception value (there is no class C such that $C \subseteq: \emptyset$).

During the program's evaluation, new expressions may be created through application of evaluation rule E_c-Spn (spawn) followed by evaluation rule E_c-$\eta_\emptyset$Run (execution of a null event), but these expressions will be of the form (`new` $C(\bar{\mathtt{v}})$)`.run(`η_0`)`, where C <: `Task`. Since the `run` method of C does not declare any thrown exceptions, then, by type rule T-11, neither does the `run` method of class C. By lemma 10, we know that such expressions will not evaluate to an uncaught exception. □

Now, we can state the main result for this section: a well-typed CTJ program either terminates to a non-exception value, diverges, or stops due to a runtime cast error.

Theorem 5 (Soundness)**.** *If* $P_c = \overline{CL}$ `return` e *is a CTJ program and* $\vdash^c P_c$ *OK, then one of the following must be true:*

- $e|\emptyset \Longrightarrow_c^* v|\emptyset$, *where* v *is a non-exception value.*
- $e|\emptyset \uparrow$
- $e|\emptyset \Longrightarrow_c^* E[(T)v]|\mathcal{B}$, *where the type of* v *is not a subtype of* T.

Proof. By induction over evaluation steps. The initial state of P is $e|\emptyset$. By theorem 4, either e is a value, e is a runtime cast error, or there exists an $e'|\mathcal{B}'$ such that $e|\emptyset \Longrightarrow_c e'|\mathcal{B}'$. By theorem 2, e' is well-typed and $\mathcal{B}'$ OK.

Next, we look at an arbitrary state $e_i|\mathcal{B}_i$ such that $e|\emptyset \Longrightarrow_c^* e_i|\mathcal{B}_i$. By the inductive hypothesis, $e_i|\mathcal{B}_i$ is well-typed. By theorem 4, either e_i is a value, e_i is a runtime cast error, or there exists an $e_{i+1}|\mathcal{B}_{i+1}$ such that $e_i|\mathcal{B}_i \Longrightarrow_c e_{i+1}|\mathcal{B}_{i+1}$. By theorem 2, e_{i+1} is well-typed and $\mathcal{B}_{i+1}$ OK.

Thus, the program will either step forever, terminate, or become stuck due to a runtime cast error. If the program terminates, by lemma 11, it cannot terminate due to an uncaught exception. □

2.4.0.4 No lost exceptions

Note that a CTJ program never evaluates to an uncaught exception. The type system statically ensures that both the initialization expression and the `run` methods of any spawned tasks catch all exceptions thrown during evaluation. We state this more formally as follows:

Corollary 1 (No lost Exceptions)**.** *If* $\vdash e : \mathtt{T}|\emptyset$*, and* $e|\emptyset \Longrightarrow_c^* e'|\mathcal{B}'$*, then* $\vdash e' : T'|\emptyset$*.*

2.4.0.5 No lost continuations

As discussed in the overview, when one writes event-driven programs in a continuation passing style, it is possible to drop a continuation and thus never pass the result of an asynchronous operation to its logical caller. This problem is easily avoided in CTJ programs by using asynchronous methods and `wait` calls instead of continuations. Such calls are evaluated in the same manner as standard method calls. More specifically, the language semantics ensure that, if program execution reaches an asynchronous method call or `wait` call, either evaluation of the calling expression is eventually resumed (with the results of the call), execution stops due to a runtime cast error, or the program diverges. More formally, we can state:

Corollary 2 (No lost continuations)**.** *For any program state $E[e_0]|\mathcal{B}$, where e_0 is an asynchronous method call or a* `wait` *call, either:*

- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E[v]|\mathcal{B}'$*, where v is a value,*
- $E[e_0]|\mathcal{B} \uparrow$*, or*
- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E'[(T)v]|\mathcal{B}'$*, where the type of v is not a subtype of T.*

We first prove for the more general case where e_0 is an arbitrary expression. We do this using two lemmas:

Lemma 12 ($\longrightarrow_c$ Evaluation to normal forms)**.** *An expression e either evaluates to a normal form or diverges.*

Proof Idea This can be proven by induction over evaluation steps using theorem 3 (at each step, either an evaluation rule can be applied or a normal form has been reached).

Lemma 13 ($\Longrightarrow_c$ Evaluation to normal forms)**.** *For any program state $E[e_0]|\mathcal{B}$, either:*

- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E[v]|\mathcal{B}'$*, where v is a value,*
- $E[e_0]|\mathcal{B} \uparrow$*, or*
- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E'[(T)v]|\mathcal{B}'$*, where the type of v is not a subtype of T.*

Proof. By induction over evaluation steps. By lemma 12, either $e_0 \longrightarrow_c^* e_0'$, where e_0' is a normal form, or evaluation diverges. We look at each possible outcome:

- If the evaluation diverges, the second case of the lemma is satisfied.
- If the normal form is a runtime cast error, the third case of the lemma is satisfied.
- If the normal form is a `spawn` call, evaluation rule E_c-Spn replaces the call with an expression of the form `new` $C(\bar{\mathtt{v}})$. Then, either the entire expression is a value, satisfying the first case of the lemma, or by the inductive hypothesis, further evaluation either diverges, reaches a value, or reaches a runtime cast error.
- If the normal form is a `wait` call, evaluation rule E_c-Wait can be applied to add the current evaluation context $E_w[]$ and the set of waited-for events s_w to the blocked set $\mathcal{B}_w$, resulting in a new program state `new Object`$|\mathcal{B} \cup (s_w, E_w[])$. The only evaluation rules that may be applied to this new state are E_c-Run and E_c-$\eta_\emptyset$Run, which select a blocked evaluation context for execution. If $\mathcal{B} = \emptyset$, then $(s_w, E_w[])$ must be selected. Otherwise, induction over evaluation steps and theorem 4 can be used to show that either $(s_w, E_w[])$ is selected for evaluation through rule E_c-Run, evaluation diverges, or a runtime cast error occurs. The second two cases satisfy this lemma.

  If $(s_w, E_w[])$ is selected by rule E_c-Run, then an event η is selected from s_w and $E_w[\eta]$ is evaluated. Either the entire expression is a value, satisfying the first case of the lemma, or, by the inductive hypothesis, further evaluation either diverges, reaches a value, or reaches a runtime cast error.

□

Proof of corollary 2. Immediate from lemma 13. □

2.5 Translating CoreTaskJava to EventJava

A CTJ program is translated to EJ by rewriting tasks and asynchronous methods to use a continuation-passing style. We describe this translation using a set of syntax transformation rules. Informally, these rules:

- Change each asynchronous method to forward its result to a continuation object passed in as an input parameter, rather than returning the result to its caller.
- Methods containing asynchronous method calls are split at the first asynchronous call. The evaluation context surrounding the call is moved to the `run` method of a new continuation class. An instance of this class is added to the parameter list of the call. The continuation class itself may need to be split as well, if the new `run` method contains asynchronous calls.
- Methods containing `wait` calls are split in the same manner. The evaluation context surrounding the call is moved to the `run` method of a new continuation class. The `wait` call is replaced by a `reg` call, with the continuation class passed as the callback parameter. As above, the continuation class itself may need to be split as well, if the new `run` method contains asynchronous calls.
- If the original body of an asynchronous method may throw exceptions to its caller, the method is changed to catch these exceptions and pass them to a special `error` method on the continuation object. This `error` method contains the same body as the `run` method. However, it replaces the use of

`retVal` (the result of a call) with a `throw` of the exception.

2.5.0.6 Translation rules

For the formalization of the translation, we consider a dynamic translation strategy where a CoreTaskJava program is translated as it executes. The original CTJ program is run until it reaches an asynchronous call or normal form. Then, the expression in the current evaluation context is translated to EventJava. If the expression is an asynchronous method or `wait` call, the class table is augmented with a new callback class which contains the evaluation context as its body. In either case, evaluation then continues until another asynchronous method or normal form is reached, upon which another translation is performed. We will show that the executions of the original TaskJava program and the dynamically translated EventJava program are observationally equivalent.

Definition 1 ($\hookrightarrow$ relation)**.** *We use the symbol $\hookrightarrow$ to represent the evaluation relation created by this interleaving of execution and translation.*

We use this approach to simplify the state equivalence relation and to avoid creating extra classes to store partial results. Of course, the TaskJava compiler implementation performs a static translation. There are also slight differences in the translation strategy due to limitations in the core calculus (e.g., lack of a `switch` statement).

We assume that the translation of a CTJ program occurs after a typechecking pass, so that all expressions are elaborated with their types. We only show type elaborations where they are pertinent to the rules. In particular, they are used for distinguishing asynchronous from standard methods and for properly casting the result of an asynchronous call.

$\boxed{e_c \| \overline{CL} \rightsquigarrow e_e \| \overline{CL}'}$

$$\dfrac{\begin{array}{c} e_c = E[(v_0 : C_{rcv}).m(\bar{\mathtt{v}}_a)] \\ \bar{\tau} \neq \emptyset \qquad CL_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{async}\ T_r\ m(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_b;\} \in \bar{\mathtt{M}} \\ e'_b = \mathtt{try}\ \{\mathtt{cb.run}(e_b);\}\ \mathtt{catch}\ (\mathtt{Exception}\ e)\ \{\mathtt{cb.error}(e);\} \\ \bar{\mathtt{M}}' = \bar{\mathtt{M}} \cup \mathtt{Object}\ m'(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}}, \mathtt{Callback\ cb})\ \{\mathtt{return}\ e'_b;\} \\ CL'_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}'\} \qquad \mathit{fresh}\ C_{cb} \end{array}}{e_c \| \overline{CL} \cup CL_{rcv} \rightsquigarrow v_0.m'(\bar{\mathtt{v}}_a, \mathtt{new}\ C_{cb}()) \| \overline{CL} \cup CL_{rcv} \cup \mathit{Callback}_{exc}(C_{cb}, E[], T_r)} \ (\text{TR-AC1})$$

$$\dfrac{\begin{array}{c} e_c = E[(v_0 : C_{rcv}).m(\bar{\mathtt{v}}_a) \qquad CL_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{async}\ T_r\ m(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}})\ \{\mathtt{return}\ e_b;\} \in \bar{\mathtt{M}} \\ \bar{\mathtt{M}}' = \bar{\mathtt{M}} \cup \mathtt{Object}\ m'(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}}, \mathtt{Callback\ cb})\ \{\mathtt{return\ cb.run}(e_b);\} \\ CL'_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}'\} \qquad \mathit{fresh}\ C_{cb} \end{array}}{e_c \| \overline{CL} \cup CL_{rcv} \rightsquigarrow v_0.m'(\bar{\mathtt{v}}_a, \mathtt{new}\ C_{cb}()) \| \overline{CL} \cup CL_{rcv} \cup \mathit{Callback}_{noexc}(C_{cb}, E[], T_r)} \ (\text{TR-AC2})$$

$$\dfrac{e_c = E[\mathtt{wait}\ \{\bar{\mathtt{v}}\}] \qquad \mathit{fresh}\ C_{cb} \qquad e_{cb} = \mathtt{new}\ C_{cb}()}{e_c \| \overline{CL} \rightsquigarrow \mathtt{reg}\ \{\bar{\mathtt{v}}\}, e_{cb} \| \overline{CL} \cup \mathit{Callback}_{noexc}(C_{cb}, E[], \mathtt{Event})} \ (\text{TR-W}\textsc{t})$$

$$\dfrac{}{\begin{array}{c} E[\mathtt{spawn}\ C(\bar{\mathtt{v}})] \| \overline{CL} \rightsquigarrow \\ E[\mathtt{reg}\ \emptyset, \mathtt{new}\ C(\bar{\mathtt{v}})] \| \overline{CL} \end{array}} \ (\text{TR-S}\textsc{p}) \qquad \dfrac{}{\begin{array}{c} v \| \overline{CL} \rightsquigarrow \\ v \| \overline{CL} \end{array}} \ (\text{TR-V}\textsc{al})$$

$$\dfrac{}{\begin{array}{c} \mathtt{throw\ new}\ C_e(\bar{\mathtt{v}}) \| \overline{CL} \rightsquigarrow \\ \mathtt{throw\ new}\ C_e(\bar{\mathtt{v}}) \| \overline{CL} \end{array}} \ (\text{TR-T}\textsc{h}) \qquad \dfrac{\vdash v : \mathtt{T}' \qquad \mathtt{T}' \not<: \mathtt{T}}{E[(\mathtt{T})v] \| \overline{CL} \rightsquigarrow E[(\mathtt{T})v] \| \overline{CL}} \ (\text{TR-C}\textsc{st})$$

Figure 2.13: Translation rules for CTJ

The translation relation $\rightsquigarrow$ is defined in Figure 2.13. Rules TR-AC1 and TR-AC2 perform a dynamic translation of asynchronous method calls. They rewrite the asynchronous method about to be called, adding a continuation parameter to which the result of the call is forwarded. In addition, if the original call may throw exceptions, these are caught in the rewritten method and passed to the continuation's `error` method. Lastly, the evaluation context of the call is moved to a newly created callback class, C_{cb}, based on the code templates listed in Figure 2.14.

Rule TR-Wt translates `wait` calls to `reg` calls, moving the evaluation context to a new continuation class, which is then passed as a callback to `reg`. Rule

```
Callback_exc(C_cb, E[], T) ≡
  class C_cb extends Callback {
    Object run(Object retVal) {
      return E[(T)retVal];
    }
    Object error(Exception exc) {
      return E[throw exc];
    }
  }
```

```
Callback_noexc(C_cb, E[], T) ≡
  class C_cb extends Callback {
    Object run(Object retVal) {
      return E[(T)retVal];
    }
  }
```

Figure 2.14: Code templates used by the CTJ to EJ transformation

TR-Sp translates a `spawn` call to a `reg` call with an empty event set. The last three rules handle normal forms that do not need to be translated.

2.5.0.7 Soundness of Translation

We now prove observational equivalence between a CTJ program and the EJ program obtained by evaluating the CTJ program under the $\hookrightarrow$ relation. As consequence of this equivalence, we prove the lost continuation property.

Mapping Relation First, we must establish a relationship between CTJ and EJ program states.

Definition 2. *We write $e_0 \longleftrightarrow e_1$ for CTJ expression e_0 and EJ expression e_1 if there exists an e_0' and e_1', such that e_0' and e_1' are in normal form, $e_0 \longrightarrow_c^* e_0'$, $e_1 \longrightarrow_e^* e_1'$, and one of the following is true:*

- $e_0' = e_1'$
- $e_0' = E_0'[(T)v]$ *and* $e_1' = E_1'[(T)v]$, *where the type of v is not a subtype of T.*
- $e_0' = E_0'[\texttt{wait}\ s]$ *and* $e_1' = \texttt{reg}\ s,\ \texttt{new}\ CB()$, *where, for all $\eta \in s$, $E_0'[\eta] \longleftrightarrow (\texttt{new}\ CB()).\texttt{run}(\eta)$.*

- *$e'_0 = E'_0[\texttt{spawn } C(\bar{\texttt{v}})]$ and $e'_1 = E'_1[\texttt{reg } \emptyset, \texttt{new } C(\bar{\texttt{v}})]$, where, $E'_0[\texttt{new } C(\bar{\texttt{v}})] \longleftrightarrow E'_1[\texttt{new } C(\bar{\texttt{v}})]$.*

Definition 3. *We define a relation $\Longleftrightarrow$ between CoreTaskJava states and EventJava states:*

$$e_{c0}|(s_1, E_{c1}[])...(s_n, E_{cn}[]) \Longleftrightarrow e_{e0}|(s_1, \texttt{new } CB_1())...(s_n, \texttt{new } CB_n())$$

where:

- *$e_{c0} \rightsquigarrow e_{e0}$ and*
- *For all η in s_i, $E_{ci}[\eta] \longleftrightarrow (\texttt{new } CB_i()).\texttt{run}(\eta)$.*

Lemma 14 (Evaluation to normal form)**.** *If $e_c \rightsquigarrow e_e$, then $e_c \longleftrightarrow e_e$.*

Proof (outline). We prove this lemma by structural induction on the forms of e_c where a translation rule may be applied. For asynchronous method call rules TR-AC1 and TR-AC2, evaluation leads to either another asynchronous method call (for which we use the inductive hypothesis) or $e_e \longrightarrow^* e'_e$, where e'_e is a normal form. For the second case, we show that $e_c \longrightarrow^* e'_c$, where e'_c is also in normal form, and one of the following is true:

- $e'_c = e'_e$ (both evaluate to a value)
- $e'_c = E'_c[(T)v]$ and $e'_e = E'_e[(T)v]$, where the type of v is not a subtype of T (both encounter a bad cast)
- $e'_c = E'_c[\texttt{wait } s]$, $e'_e = \texttt{reg } s, \texttt{new } CB()$, where, for all $\eta \in s$, $E'_c[\eta] \longleftrightarrow (\texttt{new } CB()).\texttt{run}(\eta)$.
- $e'_c = E'_c[\texttt{spawn } C(\bar{\texttt{v}})]$ and $e'_e = E'_e[\texttt{reg } \emptyset, \texttt{new } C(\bar{\texttt{v}})]$, where, $E'_c[\texttt{new } C(\bar{\texttt{v}})] \longleftrightarrow E'_e[\texttt{new } C(\bar{\texttt{v}})]$.

For rule TR-Wt, both expressions are already in normal form. Thus, zero $\longrightarrow$ steps are required to reach normal form, $e_c = E[\texttt{wait}\ s]$, and $e_e = \texttt{reg}\ s, \texttt{new}\ CB()$. We then show that, for all $\eta \in s$, $E[\eta] \longleftrightarrow (\texttt{new}\ CB()).\texttt{run}(\eta)$.

For rule TR-Sp, both expressions are already in normal form, Thus zero $\longrightarrow$ steps are required to reach normal form, $e_c = E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, and $e_e = E[\texttt{reg}\ \emptyset, \texttt{new}\ C(\bar{\texttt{v}})]$. Upon completion of the `spawn` or `reg` call, the placeholder will be replaced with the null event η_0, leading to the same resulting expression in both cases.

For the remaining rules, $e_c = e_e$, and the theorem is trivially true. □

Lemma 15 (Observable steps). *If e_c and e_e are in normal form, and $e_c|\mathcal{B} \Longleftrightarrow e_e|\mathcal{E}$, then either:*

- $e_c|\mathcal{B} \overset{l}{\Longrightarrow} e'_c|\mathcal{B}'$ *and* $e_e|\mathcal{E} \overset{l}{\Longrightarrow} e'_c|\mathcal{E}'$*, where* $e'_c|\mathcal{B}' \Longleftrightarrow e'_e|\mathcal{E}'$*.*
- *Both e_c and e_e are of the form $E[(T)v]$, where the type of v is not a subtype of T.*
- *Both e_c and e_e are values and $\mathcal{B}$ and $\mathcal{E}$ are empty.*

Proof Idea Case analysis on normal forms. Only the forms $E[\texttt{wait}\ s]$ and $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$ are non-trivial. In each case, there is a corresponding $\Longrightarrow$ evaluation step for the translated expression which registers the same event set along with a callback for which the $\longleftrightarrow$ relation holds.

2.5.0.8 Bisumulation

To relate executions of a CTJ program through the $\Longrightarrow_c$ and the $\hookrightarrow$ relations, we must precisely define equivalence. We use *stutter bisimulation*, which permits each relation to take an arbitrary number of non-observable steps (through the $\longrightarrow$ relation) before taking a matching pair of observable steps which interact with the scheduler. This is necessary because the translated program may need to take additional steps to reach the same execution state as the original program.

Definition 4. *A relation r between program states is a* ***stutter bisimulation*** *relation if $\sigma_1\ r\ \sigma_2$ implies:*

1. *For all $\sigma_1 \overset{\epsilon}{\Longrightarrow}^* \sigma_1' \overset{o}{\Longrightarrow} \sigma_1''$, there exists a σ_2', σ_2'' such that $\sigma_1''\ r\ \sigma_2''$ and $\sigma_2 \overset{\epsilon}{\Longrightarrow}^* \sigma_2' \overset{o}{\Longrightarrow} \sigma_2''$, where o is either an `In` or `Out` label.*
2. *For all $\sigma_2 \overset{\epsilon}{\Longrightarrow}^* \sigma_2' \overset{o}{\Longrightarrow} \sigma_2''$, there exists a σ_1', σ_1'' such that $\sigma_1''\ r\ \sigma_2''$ and $\sigma_1 \overset{\epsilon}{\Longrightarrow}^* \sigma_1' \overset{o}{\Longrightarrow} \sigma_1''$, where o is either an `In` or `Out` label.*

Theorem 6 (Bisimulation)**.** *The relation $\Longleftrightarrow$ is a stutter bisimulation relation.*

Proof. Consider an arbitrary pair of program states $e_{ci}|\mathcal{B}_i$ and $e_{ei}|\mathcal{E}_i$ from the original and translated programs where $e_{ci} \rightsquigarrow e_{ei}$. Suppose that the program states are related by the $\Longleftrightarrow$ relation. Then, by lemma 14, if e_{ci} steps to a normal form, e_{ei} steps to a corresponding normal form. Once at a normal form, if the original program takes an observable step, the translated program can take a step with the same observation, by lemma 15. The resulting program states satisfy the $\Longleftrightarrow$ relation.

We have shown the proof for only one direction – if the CTJ program takes a sequence of steps, the EJ program can take a corresponding sequence of steps. To show the other direction, we use lemmas 16 and 17 below, which state that at most one transition rule applies for each expression. Thus, once we have identified a corresponding sequence of evaluation steps between the two executions, we know that no other sequences are possible from the same initial states (assuming the same choice of selected events). $\square$

Lemma 16 (Deterministic execution of CTJ programs)**.** *If* $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$ *and* $e|\mathcal{B} \Longrightarrow_c e''|\mathcal{B}''$*, then* $e' = e''$ *and* $\mathcal{B}' = \mathcal{B}''$*.*

Proof Idea From theorem 4, we know that, if $e|\mathcal{B}$ is well-typed, then, either the program execution halts (due to normal termination or a runtime cast error), or a step can be taking via the $\Longrightarrow_c$ relation. A case analysis for each rule of the $\Longrightarrow_c$ relation shows that, if e is well-typed, then no other rule may be applied to e.

Lemma 17 (Deterministic execution of EJ programs)**.** *If* $e|\mathcal{E} \Longrightarrow_e e'|\mathcal{E}'$ *and* $e|\mathcal{E} \Longrightarrow_e e''|\mathcal{E}''$*, then* $e' = e''$ *and* $\mathcal{E}' = \mathcal{E}''$*.*

Proof Idea Same approach as used for lemma 16.

2.5.0.9 No lost continuations

We can now state the lost continuation property for translated CTJ programs. Informally, in the dynamic translation of a CTJ program, if a callback is passed to an asynchronous method call or `reg` call, either the program diverges, gets stuck due to a runtime cast error, or the callback is eventually called.

Theorem 7 (No lost continuations)**.** *Consider a CoreTaskJava program* $P_c = \overline{CL}_c$ `return` e_0 *such that* $\vdash^c P_c$ *OK. If* $e_0|\emptyset \hookrightarrow^* e'|\mathcal{E}'$*, where* e' *has either the form* $E[(\texttt{new } C_r(\bar{\texttt{v}})).m(\bar{\texttt{v}}, \texttt{ new } C_{cb}(\bar{\texttt{v}}_{cb}))]$ *or the form* $E[\texttt{reg}(\{\bar{\texttt{v}}\}, \texttt{ new } C_{cb}(\bar{\texttt{v}}_{cb}))]$ *where* $C_{cb} <: \texttt{Callback}$*, then either* $e'|\mathcal{E}'$ *diverges or* $e'|\mathcal{E}' \hookrightarrow^* e''|\mathcal{E}''$*, where* e'' *has either the form* $E[(\texttt{new } C_{cb}(\bar{\texttt{v}}_{cb})).\texttt{run}(v)]$ *or the form* $E[(T)v_{err}]$*, where the type of* v_{err} *is not a subtype of* T*.*

Proof. We use theorem 6 to construct a proof by contradiction. Consider an arbitrary CTJ program fragment of the form $E[e_0]$, where e_0 is an asynchronous call or wait call, and $E[]$ contains an observable action based on the result of this call. By corollary 2, the subexpression e_0 either evaluates to a value, diverges, or reaches a runtime cast error. By translation rules TR-AC1, TR-AC2, or TR-Wt, the call will be translated to one of the EventJava forms listed in the theorem above.[3] The evaluation context containing the observable action will be moved to a callback in the translated program. If this callback is never called (violating theorem 7), the observable action in $E[]$ will not occur, violating theorem 6. $\square$

2.6 Implementation

2.6.1 Compiling TaskJava Programs to Java

The TaskJava compiler implements a source-to-source translation of TaskJava programs to (event-driven) Java programs. Invocations of `wait` and `async` methods are referred to collectively as *asynchronous calls*. Note that this translation is only needed for methods containing asynchronous calls — all other methods are left unchanged.

In this section, I provide a high level overview of the compiler's implementation, using a series of small examples.

CPS transformation of Tasks. The compiler uses continuation-passing style to break up the `run` methods of tasks into a part that is executed up to an asynchronous call and a continuation. Rather than implement the continuation as a separate class, I keep the continuation within the original method. The body of a task's `run` is now enclosed within a `switch` statement, with a case for the initial code leading up to the first asynchronous call and a case for each

[3]We assume that the `Callback` base class is not available to CTJ programmers. Thus, subclasses of `Callback` appearing in a translated program must have been generated from one of these three translation rules.

continuation. Thus, the `switch` statement acts as a kind of structured `goto`. I use this approach instead of separate methods to avoid building up the call stack when a loop's body contains an asynchronous call, since Java does not optimize tail-recursive calls.

Task state. Any state which must be kept across asynchronous calls (e.g., the next step of the switch statement) is stored in a new `_state` field of the task. An inner class is defined to include these new fields.

If local variables are declared in a block that becomes broken across continuations, they must be declared in a scope accessible to both the original code and the continuation. Currently, we solve this problem by changing all local variables to be fields of the `_state` object.

Calls to `spawn`. The spawning of a new task is implemented by creating a new task object and then registering this object with the scheduler, which will then call the task's `run` method. An empty event set is provided to the scheduler, which indicates that the task should be run unconditionally.

Calls to asynchronous methods. When an asynchronous method is called, a callback object is created and passed to the callee. The `run` method of this callback should be invoked upon completion of the callee method. The caller returns immediately upon return from the callee, to be resumed later by the callback. For example, consider the following asynchronous call which returns a concatenated string:

```
...
x = concat(‘‘abc’’, ‘‘xyz’’);
```

This would be translated to:

```
case 1:
    ...
    concat(``abc'', ``xyz'', new run_callback(this, 2));
    return;
  case 2:
    x = (String)this._state._retVal;
```

Here, `concat` is passed a third parameter, a new callback object. The callback is initialized with a reference to the calling task (`this`) and the `switch` step to resume upon completion of the call. The actual assignment of `x` now occurs in the following `switch` step.

Callback classes are created by the compiler. To resume a task, the callback simply assigns to two compiler-generated fields in the task and re-invokes the task's `run` method. The first compiler-generated field, `_state._step`, indicates the `switch` case to resume (2 in our example). The second field, `_state._retVal`, contains the result of the asynchronous call (the concatenated string, in our example).

I introduce temporary variables in situations where breaking up an expression at asynchronous calls becomes difficult. For example, a nested asynchronous call, such as in `concat(concat(x, y), z)`, is first assigned to a temporary variable, which is passed to the outer call. Temporaries are also used when an asynchronous call occurs inside an `if` or loop condition.

Calls to `wait`. Calls to `wait` may be translated in a similar manner to asynchronous method calls, replacing the `wait` call itself with a scheduler event registration. In my implementation, I take a slightly different approach, described in section 2.6.2.

Asynchronous methods. The signature of an asynchronous method is changed to include an additional callback parameter. This callback is called upon completion of the method. Any return value is passed to the callback, instead of being returned to the asynchronous method's caller.

The bodies of `async` methods are translated in a similar manner to tasks. However, since simultaneous calls of a given method are possible, the `_state` object is passed as a parameter to the method, rather than added as a field to the containing class. To achieve this, the main body of the method is moved to a separate (private) continuation method. The original (externally callable) method just constructs a state object, stores the method arguments in this state, and then calls the continuation method. As with tasks, asynchronous methods return immediately after calling an asynchronous method or `register`, and are resumed through a callback.

Loops. If an asynchronous call occurs within a loop, the explicit loop statement is removed and replaced with a "branch and goto" style of control flow, simulated using steps of the `switch` statement. The entire `switch` statement is then placed within a `while` loop.

For example, consider the following call to `concat`:

```
String s = ‘‘’’; int i = 0;
while (i<5) {
  s = concat(s, ’a’);  i = i + 1;
}
...
```

This would be translated as follows:

```
while (true) {
  switch (_state._step) {
  case 1:
    _state.s = ‘‘’’; _state.i = 0;
  case 2:
    if (!(_state.i < 5)) { _state._step = 4; break; }
    concat(_state.s, ’a’, new run_callback(this, 3));
    return;
  case 3:
    _state.s = (String)_state._retVal;
    _state.i = _state.i + 1;
    _state._step = 2; break;
  case 4:
    ...
```

In the first case, we see the translated initialization assignments. The local variables have been made into fields of the the task's `_state` member. We fall through to the second case, which implements the "top" of the original `while` loop. If the original loop condition is false, we simulate a `goto` to step 4 by setting the step variable to 4 and breaking out to the enclosing `while` loop. Otherwise, we call `concat`, passing a new callback object, and then return. Upon completion of `concat`, the callback will set the step to 3 and invoke the task's `run` method. This gets us back to case 3 of our `switch` statement. At the end of this case, we simulate a `goto` back to the top of the loop by setting the step variable to 2 and breaking out of the enclosing `switch`.

Exceptions. Due to the CPS translation, asynchronous methods cannot simply throw exceptions to their callers. Instead, exceptions are passed from callee to caller via a separate `error` method on the callback. The body of an asynchronous method which may throw exceptions is enclosed in a `try..catch` block. If an exception is thrown, the `error` method of the callback is called (instead of the normal control flow's `run` method), with the exception passed as a parameter.

The callback's `error` method assigns its exception to a compiler-generated `_error` field of the `_state` object and then resumes the associated caller. When an asynchronous call may have thrown an exception, the continuation code of the task or asynchronous method then checks whether the `_error` field has been set. If so, it re-throws the exception. If the asynchronous call was enclosed in a `try..catch` block, the `try..catch` is duplicated across each continuation.

Consider the following example:

```
try {
  x = concat(‘‘abc’’, ‘‘xyz’’);
} catch (IOException e) {
  System.out.println(‘‘error!’’);
}
...
```

This would be translated as:

```
case 1:
  concat(‘‘abc’’, ‘‘xyz’’, new run_callback(this, 2));
  return;
case 2:
  try {
    if (_state._error!=null) throw _state._error;
    x = (String)_state._retVal;
  } catch (IOException e) {
    System.out.println(‘‘error!’’);
  }
  ...
```

We initiate the `concat` asynchronous call as before. However, upon resumption of the caller, we check the `_error` field to see if an exception occurred. If so, we re-throw the exception. The continuation block is enclosed in a `try` statement. Thus, if the callee throws an `IOException`, the appropriate `catch` block is invoked.

2.6.2 The scheduler

An important design goal for TaskJava is to avoid making the language dependent on a specific scheduler implementation and its definition of events. One approach (used in the examples of section 2.2) is to specify one or more schedulers to the compiler, perhaps as a command-line option. The compiler then replaces `wait` calls with event registrations for this scheduler.

I chose a more flexible approach in my implementation. I do not include a `wait` call at all, but instead provide a second type of asynchronous method — `asyncdirect`. From the caller's perspective, an `asyncdirect` method looks like an asynchronous method with an implicit (rather than explicit) callback. However, the declaration of an `asyncdirect` method must contain an explicit callback. No translation of the code in the method's body is performed — it is the method's responsibility to call the callback upon completion. Typically, an `asyncdirect` method registers an event, stores a mapping between the event and the callback, and then returns. Upon completion of the event, the mapping is retrieved and the callback invoked.

This approach easily permits more than one scheduler to be used within the same program. Also, existing scheduler implementations can be easily wrapped with `asyncdirect` methods and used by TaskJava.

For our experiments, we implemented a single-threaded scheduler on top of

Java's nonblocking I/O package (`java.nio`). Clients may register a callback to be associated with events on a given channel (socket). The scheduler then registers interest in the requested events with the Java `nio` layer and stores an association between the events and callbacks in a map. The scheduler's main loop blocks in the `nio` layer, waiting for an event to occur. Upon waking up, the scheduler iterates through the returned events and calls each associated client callback.

I have also implemented a thread-pooled scheduler which can concurrently process events. Event registrations are transparently mapped to threads by hashing on the associated channel. This scheduler provides the same API as our single-threaded scheduler, permitting applications which do not share data across tasks to take advantage of thread-pooling without any code changes.

2.7 Case Study

Fizmez. As a proof-of-concept for TaskJava, I modified an existing program to use interleaved computation. I chose *Fizmez* [Bon], a simple, open source web server, which originally processed one client request at a time. I first extended the server to interleave request processing by spawning a new task for each accepted client connection. To provide a basis for comparison, I also implemented an event-driven version of Fizmez.

Task version. In this version, each iteration of the server's main loop accepts a socket and spawns a new `WsRequest` task. This task reads HTTP requests from the new socket, retrieves the requested file and writes the contents of the file to the socket.

The original Fizmez server used standard blocking sockets provided by the `java.io` package. To port Fizmez to TaskJava, I needed to convert the server to use our event scheduler. I used TaskJava's asynchronous methods to build an abstraction on top of our scheduler with an API that mirrors that of the `java.io` package. This approach allowed me to convert I/O calls to TaskJava simply by changing class names in field and method argument declarations.

Overall, I was able to maintain the same organization of the web server's code as was used in the original implementation. The main change I made was to refactor the request-processing code out of the main web server class and into a new class. This change was necessary since requests are now processed concurrently, so each request must maintain its own state.

Explicit event version. The event-driven implementation required major changes to the original Fizmez code. The web server no longer has an explicit main loop. Instead, a callback re-registers itself with the scheduler to process the next connection request. More seriously, the processing of each client request, which is implemented in a single method in the original and TaskJava implementations, is split across six callback classes and a shared state class in the explicit event implementation.

2.7.1 Performance Experiments.

I compared the performance of the TaskJava and explicit event-driven web server implementations using a multi-threaded driver program that submits 25 requests per thread for a 100 kilobyte file (stored in the web server's cache). Latency is measured as the average time per request and throughput as the total number of requests divided by the total test time (not including client thread initialization).

The performance tests were run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. Table 2.2 shows the experimental results.

Client	Latency(ms)		Throughput(req/sec)	
threads	Event	Task	Event	Task
1	33.0	31.1	30.2	32.1
25	76.8	79.2	322.1	306.3
50	112.4	120.0	443.4	413.5
100	187.6	197.0	351.0	262.2
200	317.3	345.8	403.5	225.8
300	455.8	462.4	324.2	328.6
400	601.4	695.9	216.0	212.0

Table 2.2: Web server performance test results

The columns labeled "Event" and "Task" represent results for the event-driven server and the TaskJava server, respectively.

The overhead that TaskJava contributes to latency is within 10%, except at 400 client threads, where it reaches 16%. The throughput penalty for TaskJava is low up through 50 threads, but then becomes more significant, reaching 44% at 200 threads. Above 200 threads, the total throughput of both implementations drops, and the overhead becomes insignificant.

These results are not surprising, as I have not yet made any efforts to optimize the continuation-passing code generated by our compiler. There are two main differences between the TaskJava compiler-generated code and the hand-optimized event code. First, compared to the event version, each TaskJava asynchronous call involves one extra method call, extra assignments (for the `_step`, `_retVal`, and `_error` fields), and an extra `switch` statement. Second, the event-driven server pre-allocates and reuses callbacks. For example, in the event-driven implementation, I associate reused callbacks with each connection, as I know that, by design, there will be only one read or write request pending on a given connection at a time. In contrast, the TaskJava compiler currently allocates a new callback for each asynchronous call.

I am investigating approaches to reduce this overhead for a future version of the TaskJava compiler. To reduce the cost of the `switch` statement and extra assignments, I can embed the continuation code directly in a callback, except when it occurs within a loop. Alternatively, I may achieve more flexibility in structuring control flow by compiling directly to JVM bytecode. Without an interprocedural analysis, one cannot remove all extra callback allocations. However, I can allocate a single callback per enclosing method rather than per called method. This will eliminate the majority of runtime allocations that occur in the web server.

For many applications, the readability and reliability benefits of TaskJava outweigh the downside of the performance cost. Over time, this group of applications should grow larger, as I reduce the penalty by optimizing the code generated from our compiler.

2.8 Related Work

Event-driven programming is pervasive in many applications, including servers [PDZ99, WCB01], GUIs, and sensor networks applications [GLB03, HSW00]. In [AHT02], event-based and thread-based styles are broken into two distinct differences: manual vs. automatic stack management and manual vs. automatic task management. Threads provide automatic stack and task management, while events provide manual stack and task management. By this classification, TaskJava provides manual task management and automatic stack management. A hybrid cooperative/preemptive approach to task management is also possible in TaskJava by using a thread-pooled scheduler. Asynchronous methods in TaskJava make explicit when a method may yield control, addressing the key disadvantage of automatic stack management cited by [AHT02].

Cooperative multitasking. The overview compared TaskJava's approach with the concept of cooperative multitasking. Many implementations exist for cooperative multitasking in C and C++. (State Threads [SA] and GNU Pth [Eng00], for example). In fact, [Eng00] lists twenty such implementations. Aside from the differences discussed in the introduction, context switching in these systems is typically implemented through C or assembly-level stack manipulation. Stack manipulation is not possible for virtual machine-based languages, like Java, so TaskJava uses the CPS approach instead. While this approach is more complicated, it can be advantageous. In particular, the stack-based approach requires a contiguous stack space to be allocated per thread, which may result in a significant overhead when many tasks are created.

The C library and source-to-source compiler Capriccio [BCZ03] provides cooperative threading, implemented using stack manipulation. It avoids the memory consumption problems common to most cooperative and operating system thread implementations by using a whole-program analysis and dynamic checks to reduce the stack memory consumed by each thread. This downside of this approach is the loss of modular compilation. Capriccio also suffers from the other weaknesses of cooperative threading — difficulty implementing on top of a VM architecture and lack of scheduler flexibility.

Functional Programming Languages. The functional programming community has explored the use of continuations to preserve control flow in the context of concurrent programming. For example, [GKH01] and [Que03] describe the use of Scheme's first-class continuations to avoid the inversion of control in web programs. Concurrent ML [Rep91] builds pre-emptive threads on top of continuations. Concurrent ML also adds first-class events to the SML language, including a `choose` primitive, which can be used to build constructs equivalent to TaskJava's `wait`.

TaskJava's asynchronous methods can be viewed as a limited form of continuation. Although asynchronous methods do not support some programming styles possible with continuations, providing a more limited construct enables the TaskJava compiler to statically and modularly determine which calls may be saved and later resumed. This limits the performance penalty for supporting continuations (such as storing call state on the heap) to those calls which actually use this construct.

The functional programming community has also worked to extend the Continuation Passing Style transformation to better serve the needs of concurrent programs. *Trampolined style* [GFW99], a programming style and transformation, permits the interleaving of tail recursive functions. In [MFG04], web program inversion of control issues are addressed via a sequence of code transformations, without requiring language support for continuations. However, neither approach provides a limited, modular translation which can coexist with existing codebases. In addition, both papers describe translations in the context of late-bound, functional languages, as opposed to a statically-typed, object-oriented, non-tail-recursive language like Java.

In [PCM05], a transformation for Scheme programs is described, which permits the implementation of first class continuations on top of a non-cooperating virtual machine. The transformation installs an exception handler around the body of each function. When the current continuation is to be captured, a special exception is thrown. The handler for each function saves the current function's state to a continuation. This approach avoids changing function signatures, permitting some interoperability between translated and non-translated code. However, if a non-translated function appears on the stack when a continuation is captured, a runtime error is thrown. By using method annotations to direct the translation, TaskJava avoids this issue while still permitting interoperability between translated and non-translated code.

Languages and Tools for Embedded Systems. nesC [GLB03] is a language for embedded systems with direct language support for writing in a continuation passing style. As such, it suffers from the lost continuation problem — there is no guarantee that a completion event will actually be called. This approach was chosen by the designers of nesC because it can be implemented with a fixed-size stack and without any dynamic memory allocation.

A source-to-source translator for Java Card applications is descibed in [LZ04]. Via a whole-program translation, it converts code interacting with a host computer to a single large state machine. Like TaskJava, it must break methods up at blocking calls (limited to the Java Card communication API) and must handle classes and exceptions. However, the translator does not support concurrent tasks or recursive method calls. In addition, rather than use method annotations, method bodies are split at each method call, regardless of whether they contain blocking calls. These limitations, appropriate to an embedded environment, significantly simplify the translation algorithm. Tasks in TaskJava are more general and thus useful in a wider range of applications.

Simplifying event systems through meta-programming. The Tame framework [KrK06] implements a limited form of CPS transformation through C++ templates and macros. The goal of Tame, like TaskJava, is to reuse existing event infrastructure without obscuring the program's control flow. Continuations are passed explicitly between functions. However, the code for these continuations is generated automatically and the calling function rewritten into case blocks of a switch statement, similar to the transformation performed by the TaskJava compiler. Thus, Tame programs can have the benefits of events without the software engineering challenges of an explicit continuation passing style.

By using templates and macros, Tame can be delivered as a library, rather than requiring a new compiler front-end. However, this approach does have disadvantages: the syntax of asynchronous calls is more limited, exceptions are not supported, template error messages can be cryptic, and the implementation only works against a specific event scheduler. Tame favors flexibility and explicit continuation management over safety. As such, it does not prevent either the lost continuation or the lost exception problems.

Static analysis of event-driven programs. Techniques for analyzing and verifying event-driven systems has been an active research direction (e.g., [DGJ98, GKK03]). Hybrid approaches are also possible. For example, [CK05] implements a combination of library design with debugging and program understanding tools. TaskJava has the potential to greatly aid such techniques, by making the dependencies among callbacks and the event flows in the system syntactically apparent.

2.9 Recap

In this chapter, we have investigated the challenges of asynchronous programming. For many reasons, it can be advantageous to break a computation into an initialization call and a completion callback. Unfortunately, partitioning a program's control flow in this manner may increase the likely-hood of errors, in particular the *lost continuation* and *lost exception* problems. To address this, I defined the TaskJava programming model, which introduces three new abstractions: *tasks*, *asynchronous methods*, and the *event scheduler*. By using these abstractions, one can create a program containing asynchronous calls without disrupting the natural control flow of its source text. The TaskJava compiler performs a modular translation of the task-based source to standard Java code written in an event-driven style. This is achieved through the introduction of a few simple annotations, which do not impose a significant burden on the programmer and which permit the reuse of existing Java libraries.

CHAPTER 3

Consistency

3.1 Overview

When an application changes the state of other systems through service calls, it should ensure that it always leaves those systems in a *consistent* state, both internally and with respect to each other. Obviously, the meaning of consistency is specific to a given system. Here, we consider the consistency of *business processes*. Business processes are repeatable sequences of activities that span a business's functional organizations [Bet07]. Such processes have a definite beginning and end. In addition, they generally involve multiple applications and are *long-running*, as they require human intervention for approvals and other actions. In the context of business processes, two important aspects of consistency are:

1. individual systems should not make incorrect assumptions about the states of their peers (e.g. assuming that an action was successful when it may have failed), and

2. persistent changes made by the individual systems may be correlated, possibly depending on the overall result of the process (e.g. two actions should either both succeed or both be undone in the event of an error).

Let us now look at approaches to ensure or verify these two properties.

3.1.1 Mechanisms for ensuring consistency

In some situations, (distributed) transactions may be used to ensure that either all or none of a set of changes occur. However, traditional transactions, which require the holding of resources such as locks on each participating system, are not appropriate for long-running activities. In addition, web services rarely expose distributed transaction interfaces, due to organizational boundaries, protocol limitations, and application limitations.

As an alternative to distributed transactions, *compensation* may be used to simulate atomicity in long-running activities. Each service call becomes an independent, atomic transaction, which commits immediately upon completion. Each action of a service has an associated compensation action which reverts the effects of the original action. If an error occurs midway through a sequence of service calls, completed service calls can be undone by calling the compensation action for each call. *Flow composition languages* support the development of long running transactions by tracking at runtime a stack of compensation operations to be invoked in the event of an error. Examples include BPEL (*B*usiness *P*rocess *E*xecution *L*anguage for Web Services) [BPE03] and BPML (*B*usiness *P*rocess *M*arkup *L*anguage) [BPM].

As a running example, we will use an order management process that executes in the back-end of an e-commerce website. This process might include actions to make a credit reservation against a user's account, gather the order items from the warehouse, ship the order, and charge the user's account for the order. Each of these actions involves an interaction with an external system (e.g. the customer's bank or the warehouse management system) and is executed as a separate transaction. If the order cannot be completed for some reason (e.g. some of the items are not available), we should not charge the customer. In addition, we need some way to release the credit reservation. To do so, we introduce a `release_credit` action which acts as compensation for the `reserve_credit` action.

3.1.2 Existing notions of consistency

Cancellation semantics The compensation of all successfully invoked actions in the event of an error is termed *cancellation semantics* [BHF04]. This notion of correctness is easy for developers to understand. In addition, this property can be checked with little additional input from the developer — only the relationship between forward and compensation service calls must be specified. Unfortunately, cancellation semantics are not sufficient to describe the consistency requirements of many real-world transactions, where some actions need not be undone, some actions cannot be undone, and other actions have alternatives for forward recovery.

Whether our order process example has cancellation semantics depends on the details of each action (e.g. Does it fail atomically? Can it be compensated?) and on the overall control flow of the process. Here is one set of conditions which, if all true, will ensure cancellation semantics:

- If the processing of the order fails, `release_credit` is always called to undo the credit reservation.
- Gathering and sending the order are implemented as a single, atomic action. If this part of the process is instead multiple steps, then each sub-step will need compensation.
- Billing the customer never fails (since the necessary credit has already been reserved). If billing can fail, then all the previous steps will need compen-

sating actions.

Checking for cancellation semantics can be done efficiently [HA00] and catches important classes of errors (e.g. missing compensations). However, there are many important properties of our order process that are not ensured by checking for cancellation semantics. For example, one might want to specify that sending the order and billing the customer always occur together. Also, if the credit check fails, we might want to try an alternative account provided by the customer. In such a situation, we allow actions to fail and use forward recovery instead of compensation.

Temporal logics Alternatively, one may use temporal logics to specify the consistency requirements of a process. However, such specification languages can be difficult for developers to understand and specifications for long running transactions can be tedious to write, given the number of possible error scenarios for a given transaction. Finally, checking temporal properties of compensating programs is undecidable [EM07], and checking even finite systems is expensive (e.g. PSPACE-complete for LTL [SC85]).

Here is a possible LTL (Linear Temporal Logic) specification for our order process example:

$$\begin{aligned} G(\texttt{reserve\_credit} \rightarrow & (F(\texttt{send\_order}) \wedge F(\texttt{charge\_account}) \wedge G(\neg\texttt{release\_credit})) \vee \\ & (F(\texttt{release\_credit}) \wedge G(\neg\texttt{send\_order} \wedge \neg\texttt{charge\_account}))) \vee \\ G(\neg\texttt{reserve\_credit}) \rightarrow & (G(\neg\texttt{send\_order}) \wedge G(\neg\texttt{charge\_account}) \wedge G(\neg\texttt{release\_credit})) \end{aligned}$$

where the atomic propositions `reserve_credit`, `release_credit` `send_order`, and `charge_account` are true only at the point in time immediately after the corresponding action completes successfully. This specification says that:

1. If `reserve_credit` is successful, then either:
    - `send_order` is eventually successful, `charge_account` is eventually successful, and `release_credit` is never run or fails, or
    - `send_order` and `charge_account` are not run or fail, and `release_credit` is successful.
2. If `reserve_credit` is never successful, then each of `charge_account`, `release_credit`, and `send_order` are either never called or unsuccessful.

This specification captures much more about the desired behavior of our process than cancellation semantics. However, it is perhaps more complex than necessary, for two reasons:

- Temporal logic operators encode statements about the relative order of actions. However, we are really only interested in which actions have or have not occurred when the process terminates. If multiple orderings are possible, then all relevant orderings must be encoded in the specification.
- A compensated action has the same permanent effect as an action which was never run. However, we must account for both possibilities in our specification.

3.1.3 Set Consistency

To address these limitations, I propose a simple notion of correctness for long running transactions, called *set consistency.* A set consistency specification captures the set of services which made observable changes, when viewed at the *end* of a process's execution. Such a specification can be compactly represented using propositional logic, and may be verified by tracking, for all traces, the set of actions which completed successfully and were not undone through compensation. Set consistency specifications can represent cancellation semantics, as well as both stronger and weaker restrictions on processes. For example, they can specify behaviors which relax the self-cancellation requirement.

Set consistency can capture many requirements currently modeled using temporal logic. In most situations, a set consistency specification will be more compact (and, I believe, easier to understand) than the corresponding temporal logic specification. This is because: 1) it treats non-invocations, atomic failure, and compensated invocations as equivalent, avoiding the need to specify each separately, and 2) explicit order dependencies do not need to be specified. Moreover, this restriction in input reduces the complexity of when verifying finite-state systems (co-NP complete rather than PSPACE-complete).

Here is a possible set consistency specification for our order process example:

$$\begin{array}{l}(\texttt{reserve\_credit} \wedge \texttt{send\_order} \wedge \texttt{charge\_account}) \vee \\ (\neg\texttt{reserve\_credit} \wedge \neg\texttt{send\_order} \wedge \neg\texttt{charge\_account})\end{array}$$

This specification states that either `reserve_credit`, `send_order`, and `charge_account` all complete successfully, or each of the three either fails, is never run, or is compensated. For this process, we would also need to specify that `release_credit` is a compensation action for `reserve_credit`. Clearly, this specification captures our requirements better than cancellation semantics, while being more concise than the temporal logic specification.

In general, set consistency makes it easy to describe required correlations between the results of actions, the second property we proposed at the beginning of this section. However, set consistency specifications cannot explicitly represent the assumed states of each peer, the first property that I proposed. For some situations this is not important (e.g. when each action is an atomic transaction), but for others it would be helpful to explicitly represent the message protocols between a process and its peers. I address this through an extension to set consistency that I call *conversational consistency*.

3.1.4 Conversational Consistency

The individual actions of a process may, in some situations, be part of a message exchange with another long-running process. We can model these collaborators explicitly as *peer processes*, grouping together the actions which relate to each peer. The overall sequence of interactions between the two peers is called a *conversation*. For example, our order process might have two peers: the *bank*, which is called by the `reserve_credit`, `charge_account`, and `release_credit` actions, and the *warehouse*, which is called by the `send_order` action. Each peer maintains its own internal state and interacts independently with our order process.

Given this view of the problem, we can break the order process's specification into two parts:

1. When our order process completes, each conversation must be left in a *consistent* state. By consistent, I mean that neither participant is left waiting for a message from the other. These consistent states usually correspond to either a successful transaction (e.g. when the bank is called with the sequence `reserve_credit` and `charge_account`) or a transaction that failed and was cleaned up (e.g. when the bank is called with the sequence `reserve_credit` and `release_credit`).

2. We also need to correlate the final states of the conversations. In our or-

> der example, we need to ensure that either both the bank and warehouse conversations end in a successful state (the customer was changed and the product was shipped) or both conversations end in a "not run" or compensated state (the customer was not charged and the product was not shipped).

More formally, *conversational consistency* is specified in two levels. First, each pair-wise conversation is modeled using a finite automaton, called a *conversation automaton*. The state of an atomaton changes whenever a message is exchanged between the two peers. As a first condition of conversational consistency, we require that all runs of a process leave all conversation automata in final (consistent) states.

Second, a process's conversations are related using a consistency predicate, much like the predicates used in set consistency specifications. However, in conversational consistency, the predicate is written over the final states of conversation automata rather than over individual actions. This approach can permit the compact specification of processes which exchange multiple messages with their peers. Note that it also explicitly address the two properties we wish to check for in processes (consistent state assumptions and correlations between actions).

In this chapter, I develop the theory for both notions of consistency (set and conversational), and in Section 3.6, discuss considerations for choosing between the two approaches. Note that any set consistency specification can be mechanically mapped to an equivalent conversational consistency specification by treating each action as a separate peer.

3.1.5 Chapter organization

In Section 3.2, I first illustrate set and conversational consistency through two extended examples. I provide semantics for a core process calculus in Section 3.3. This core language composes atomic actions using sequential and parallel composition, choice, compensation, and exception handling constructs, and is an abstraction of common flow composition languages such as BPEL. Then, in Section 3.4, I define the *set consistency verification problem*, which checks that all feasible executions of a process are contained in the set of executions defined by the set consistency specification. I show that this problem is co-NP complete and present an algorithm for verifying set consistency by constructing a predicate representing all feasible process executions. This reduces the verification problem to propositional validity which can be checked using an off-the-shelf SAT solver. Next, in Section 3.5, I formally define the conversational consistency problem and present a verification algorithm for conversational consistency. As with set consistency, conversational consistency verification is reduced to propositional validity.

In Section 3.6, I describe my experiences in using these consistency models to find problems in real world processes. My collaborators and I have implemented the algorithms for set and conversational consistency verification, including a front-end for the BPEL language for the conversational consistency verifier. The resulting tool, called BPELCheck, is available as a plug-in for the NetBeans Integrated Development Environment.

I then apply the consistency verifiers to several case studies, including industrial examples and processes from the BPEL specification. The verification problems resulting from my case studies can each be discharged within a second, showing that the formalism provides an intuitive yet expressive formalism for real-world properties which is also tractable for real-world applications.

Finally, in Section 3.8, I place this work in the context of prior research.

3.2 Examples

3.2.1 Set Consistency

I now informally describe *set consistency* using an example inspired by a bug actually seen in a production system. I previously worked for an enterprise applications vendor. Once, when visiting a customer (a large bank), I reviewed a set of the customer's business processes, which integrated a mainframe-based financial application with a CRM (Customer Relationship Management) application. When an account was created or changed in the financial application, a message containing the account was sent to the CRM system via a transactional queue. The CRM system took the message from the queue and updated its version of the account accordingly. When examining the business process run by the CRM system, the author found a consistency bug, which inspired the work in this paper. Transactional queues provide lossless asynchronous message passing by implementing two distributed transactions — one with the sender and another with the receiver. This ensures that messages are not lost and that duplicate messages are not sent.

Upon taking a message from the queue, the CRM system executed a business process which transformed the message to its internal account representation, performed other book-keeping, and saved the account. If the account was saved successfully, the CRM system should commit its transaction with the queue, to avoid duplicate messages. If the account was not saved, the CRM system should abort its transaction with the queue, to avoid losing the update message. Upon the abort of a message "take", the message was automatically put back on the queue, unless a retry threshold was exceeded. In this case, the message went to a special “dead-letter” queue, to be investigated by an administrator. One can model the interactions of the CRM system with the queue using the following process:[1]

$$AcctRecv = \mathbf{TakeMsg}; (body \rhd (\mathbf{LeaveOnQ}; \mathsf{throw})); \mathbf{CommitQ}$$

Here, **bold fonts** represent atomic actions (the implementation of *body* is not shown). The “;” operator composes two actions or processes sequentially: it runs the first process, and if it is successful, runs the second. The “$\rhd$” operator catches exceptions: it runs the first process, and, only if it fails, runs the second. throw is a special built-in action which throws an exception. The **TakeMsg** process takes a message off the queue. The subprocess *body* handles each account. If *body* fails, then the message transaction is aborted (by calling **LeaveOnQ**), putting the message back on the queue. Otherwise, **CommitQ** commits the message transaction, permanently removing it from the queue. Ignoring all the implementation details of *body*, we want to ensure that, if the action **SaveAcct** is called within *body*, then **CommitQ** is called, and if **SaveAcct** is not called, then **LeaveOnQ** is called.

Deep within the *body* subprocess, I found the equivalent of the following code:

$$\mathbf{SaveAcct} \rhd \mathbf{LogErr}$$

where **SaveAcct** performs the actual account write. Someone had added an exception handler which, in the event of an error in saving, logged a debug message. Unfortunately, they left this code in their production system. The exception handler “swallows” the error status — **CommitQ** will always be called, even if the

[1]The process calculus used here is an extension of CCS (Calculus of Communicating Systems) and is described formally in Section 3.3.

save fails, violating our correctness requirement. If, due to a data validation error or a transient system problem, the CRM system rejects an account, bank tellers will not be able to find the customer.

Traces Executing a process gives rise to a trace. A *trace* is the sequence of successfully executed actions invoked by the process, along with the final status of the run (either ✓ or ×). For example, if the **SaveAcct** action fails, we get the following trace:

$$T = \textbf{TakeMsg}, \textbf{Preprocess}, \textbf{LogErr}, \textbf{CommitQ}\langle \checkmark \rangle$$

The **Preprocess** action represents the preprocessing which occurs in *body* before **SaveAcct** is called. The error from **SaveAcct** is not propagated to the outer process, and thus the queue transaction is committed. Note that the failed invocation of **SaveAcct** does not appear between **Preprocess** and **LogErr**. Failed invocations are left out as they have no permanent, observable effect on the system.

Set Consistency Note that the bug in *AcctRecv* can be caught if we know that there exists a feasible trace in which **CommitQ** is executed and **SaveAcct** is either not called or is invoked, but failed. That is, we can abstract away the relative order of individual actions, and only consider the *set* of actions that executed successfully. Accordingly, an *execution* is defined as the set of actions which appear in a trace. The execution E associated with the above trace is:

$$E = \{\textbf{TakeMsg}, \textbf{Preprocess}, \textbf{LogErr}, \textbf{CommitQ}\}$$

A *set consistency specification* defines a set of "good" executions. For example, correct executions of our account process either:

1. include both **SaveAcct** and **CommitQ**, or

2. include **LeaveOnQ**, but not **SaveAcct**.

Set consistency specifications are written in a predicate notation, where the literal a means that action a is included in the execution, and $\neg a$ means that action a is not included in the execution and literals are combined using the boolean $\wedge$ (and), $\vee$ (or), and $\neg$ (not) operators. A *set consistency predicate* is interpreted over the actions which appear in the process. It represents a set consistency specification which contains exactly those executions that satisfy the predicate. To represent the above two conditions, we can define the specification predicate φ_{q1} as follows:

$$\varphi_{q1} = (\mathbf{SaveAcct} \wedge \mathbf{CommitQ}) \ \vee (\neg \mathbf{SaveAcct} \wedge \mathbf{LeaveOnQ})$$

Any actions not included in the predicate are left unconstrained. Other consistency predicates can be specified for our process. For example, we might want to ensure that **LeaveOnQ** and **CommitQ** are never called in the same run:

$$\varphi_{q2} = (\mathbf{LeaveOnQ} \wedge \neg \mathbf{CommitQ}) \ \vee (\neg \mathbf{LeaveOnQ} \wedge \mathbf{CommitQ})$$

Both requirements can be checked simultaneously by taking the conjunction of the two predicates: $\varphi_{q3} = \varphi_{q1} \wedge \varphi_{q2}$.

3.2.1.1 Verification

The *set consistency verification problem* takes as input a process P and a consistency predicate φ and asks if all feasible executions of P satisfy φ. If so, the process *satisfies* the specification. Clearly, the process *AcctRecv* does not satisfy the specification φ_{q1} —a counterexample is the execution E above, which clearly does not satisfy φ_{q1}. However, the process does meet the specification φ_{q2}.

One can check a specification by constructing a predicate ϕ_p that represents all the feasible executions for the process P. Then, P satisfies the specification φ

if ϕ_p implies φ, which is a propositional satisfiability check. If *AcctRecv* is verified using this approach, using specification φ_{q1} or φ_{q3}, the execution E above can be identified as a counterexample.

Note that one can fix the problem by either removing the exception handler in *body* or re-throwing the exception after calling **LogErr**. With either of these fixes, the process passes verification.

3.2.1.2 Other Features

In practice, business process flows contain, in addition to sequencing and exception handling, parallel composition (where processes execute in parallel) and *compensations* (processes that get executed to undo the effect of atomic actions should a subsequent process fail). My process language (and verifier) allows one to write both parallel composition and compensation actions.

As an example, consider an alternative version of *body* which saves the account in two steps. First, it writes a header using the action **SaveHdr** and then writes contact information using the action **AddContact**. Both of these actions can fail. If **SaveHdr** succeeds, but **AddContact** fails, we undo the account change through the compensating action **DelHdr**, which never fails. In my process language, one specifies a compensation action using the "÷" operator. Our new process is written as:

$$\begin{aligned} AcctRecv_2 &= \textbf{TakeMsg}; (body_2 \rhd (\textbf{LeaveOnQ}; \mathsf{throw})); \textbf{CommitQ} \\ body_2 &= (\textbf{SaveHdr} \div \textbf{DelHdr}); \textbf{AddContact} \end{aligned}$$

A set consistency requirement for $AcctRecv_2$ states that if both the account header and contact actions are successful, then **CommitQ** should be run. Otherwise, **LeaveOnQ** should be run. That is,

$$\begin{gathered}(\mathbf{SaveHdr} \wedge \mathbf{AddContact} \wedge \mathbf{CommitQ}) \vee \\ (\neg\mathbf{SaveHdr} \wedge \neg\mathbf{AddContact} \wedge \mathbf{LeaveOnQ})\end{gathered}$$

The negated action $\neg$**SaveHdr** captures the scenarios where either **SaveHdr** is not run, or it is run but subsequently compensated by **DelHdr**. The effects of compensations are captured using a programmer-defined *normalization set* $\mathcal{C} = \{(\mathbf{SaveHdr}, \mathbf{DelHdr})\}$ that encodes that the effect of running **SaveHdr** can be undone by running its compensation **DelHdr**. Given the predicate and the normalization set, our verifier automatically expands the predicate to

$$\begin{aligned}&(\mathbf{SaveHdr} \wedge \neg\mathbf{DelHdr} \wedge \mathbf{AddContact} \wedge \mathbf{CommitQ}) \vee \\ &((\neg\mathbf{SaveHdr} \vee (\mathbf{SaveHdr} \wedge \mathbf{DelHdr})) \wedge \\ &\quad \neg\mathbf{AddContact} \wedge \mathbf{LeaveOnQ})\end{aligned}$$

This expanded specification predicate is then used by the verification algorithm, which shows that process $AcctRecv_2$ satisfies this set consistency specification.

3.2.2 Conversational Consistency

I now illustrate how to extend consistency to conversations, using a simple e-commerce example, inspired by the one in [BFH03]. As shown in Figure 3.1, this example involves the interactions of four business processes: a store process, which is attempting to fulfill an order, a bank process, which manages the funding for an order, and two warehouse processes, which manage inventory. To process an order, the store first authorizes with the bank the total value of the order. If this fails, the processing of the order stops. If the authorization is successful, the store checks whether the two warehouses have the necessary products in stock. If not, the order processing stops. The inventory checks actually reserve

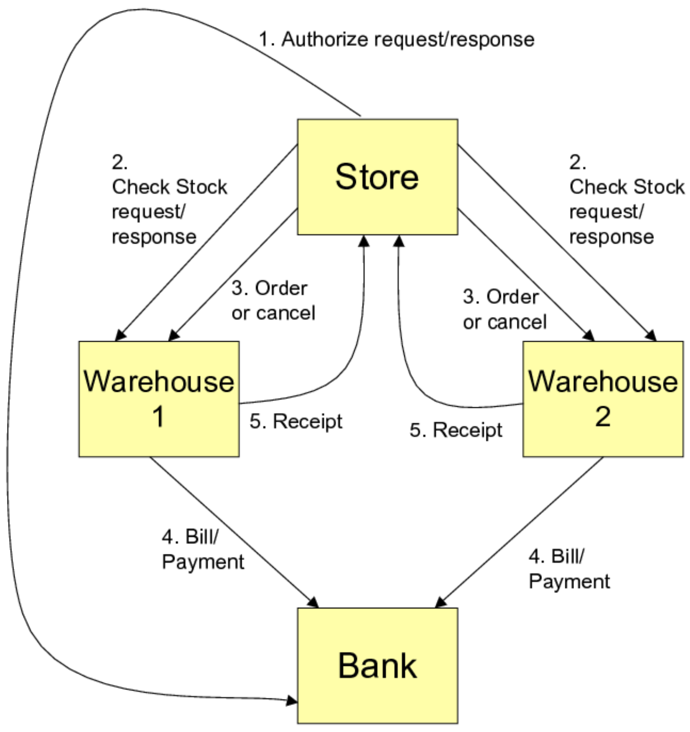


Figure 3.1: Peers interactions for e-commerce example.

the products, so if the order fails at this point, any reserved inventory must be released. If both checks succeed, the store confirms the orders with the two warehouses. The warehouses then ship the goods and send a bill to the bank. The bank responds to the bill messages with payments to the warehouses. Finally, upon receiving payment, the warehouses each send a receipt to the store.

Conversation automata We will now focus on this scenario from the store's perspective. The store interacts with three *peer processes*: the bank and the two warehouses. We can represent the *conversation* (exchange of messages) with

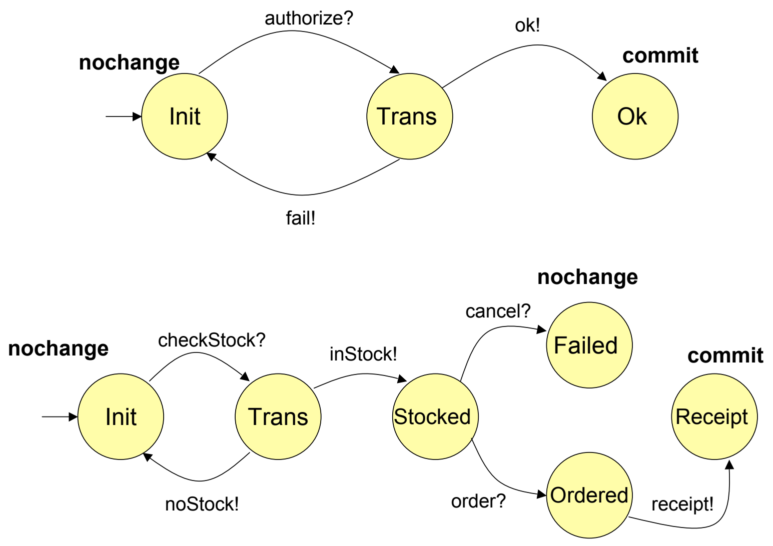


Figure 3.2: Automata for store-bank conversation (top) and store-warehouse conversation (bottom).

each peer using an automaton. The top automaton of Figure 3.2 represents the conversation with the bank. It has three states: `Init`, representing the initial state of the conversation, `Trans` representing the conversation after an `authorize` request has been sent to the bank, but before the bank has responded, and `Ok`, representing a successful authorization. Transitions between states occur when a message is sent or received between the process and the peer represented by the automata. We write `authorize?` over the transition from `Init` to `Trans`, indicating that the peer receives an `authorize` message sent by the process. Likewise, `ok!` and `fail!` over transitions indicate when the peer sends `ok` and `fail` responses, respectively, to the process.

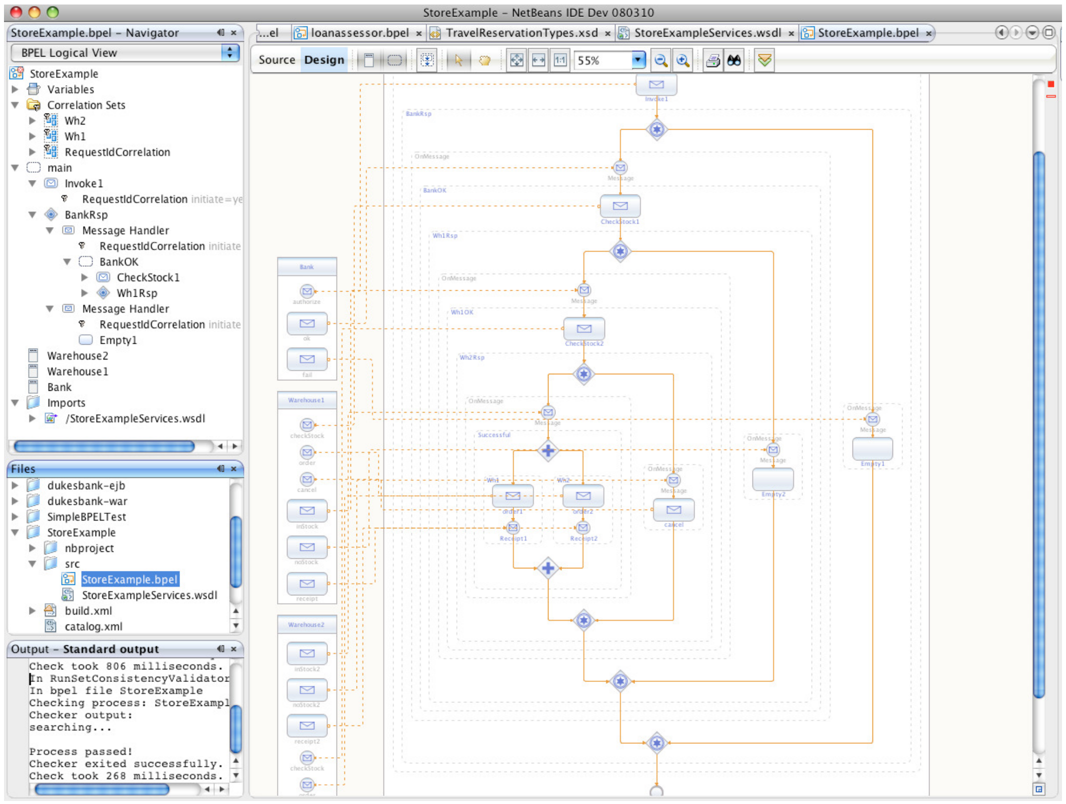
We label the `Init` state with `nochange` and the `Ok` state with `commit`. This labeling reflects the notion that a successful authorization changes the state of the bank, but a failed authorization does not. The `Trans` state is unlabeled, as it is an *inconsistent* state. The conversation between the store and the bank should not end in this state.

The bottom automaton of Figure 3.2 represents the conversation between the store and one of the warehouses (both have the same conversation protocol). This automaton is more complex, as it must encode a two-phase commit and then account for the receipt message. The store initiates the conversation by sending a `checkStock` message. If the product is available, the warehouse responds with a `inStock` message. If the product is unavailable, the warehouse responds with a `noStock` message, which terminates the conversation. In the successful case, the store must then respond by either committing the transaction with a `order` message or aborting it with a `cancel` message. The cancel is sent when the other warehouse was unable to fulfill its part of the order. Finally, a successful conversation will end with a `receipt` message from the warehouse. The `Init` and `Failed` states are both labeled as `nochange`, while the `Receipt` state is labeled as `commit`. All the other states are inconsistent.

Consistency predicates In addition to guaranteeing that the conversations with the bank and the warehouses are always left in a consistent state, the designer of the store process would like to ensure certain relationships between these conversations. For the store, we wish to guarantee a correlation between the bank and warehouse conversations. Specifically, either

1. the bank authorization is successful and both warehouse transactions occur, or
2. the bank authorization fails or is undone and neither warehouse transaction occurs.

We can specify these requirements using a *conversational consistency predicate*, a predicate over the nochange/commit labels of the conversation automata.



```
Check took 806 milliseconds.
In RunSetConsistencyValidator
In bpel file StoreExample
Checking process: StoreExampl
Checker output:
searching...

Process passed!
Checker exited successfully.
Check took 268 milliseconds.
```

Figure 3.3: BPEL implementation of store process in NetBeans

$\mathsf{comm}(A)$ is `true` when the process terminates with peer A's conversation in a state labeled `commit`. $\mathsf{nochange}(A)$ is `true` when the process terminates with peer A's conversation in a state labeled `nochange`. A consistency predicate ψ, built from these atomic predicates over the peer automata, is *satisfied* when ψ evaluates to `true` for all executions of the process.

We can write our store process invariants as a consistency predicate:

$$(\mathsf{nochange}(\mathbf{Bank}) \wedge \mathsf{nochange}(\mathbf{Warehouse1}) \wedge \mathsf{nochange}(\mathbf{Warehouse2})) \vee$$
$$(\mathsf{comm}(\mathbf{Bank}) \wedge \mathsf{comm}(\mathbf{Warehouse1}) \wedge \mathsf{comm}(\mathbf{Warehouse2}))$$

Store process implementation Figure 3.3 shows an implementation of the store process in the NetBeans BPEL designer. Each peer process is represented by a *partner link*, appearing to the left of the flowchart. Thus, our approach of defining conversations between pairs of peers is directly reflected in the BPEL representation of the process. I extend the core language to include the peer associated with each action: `msg?p` and `msg!p` mean that the process receives message `msg` from peer `p` and sends message `msg` to peer `p`, respectively. The store process is written in the core language as:

```
authorize!Bank;
  ((ok?Bank; (checkStock!Warehouse1;
    ((inStock?Warehouse1; (checkStock!Warehouse2;
      ((inStock2?Warehouse2;
        ((order!Warehouse1; receipt?Warehouse1) ||
         (order!Warehouse2; receipt?Warehouse2))) □
      (noStock?Warehouse2; cancel!Warehouse1)))) □
    (noStock?Warehouse1; skip)))) □
  (fail?Bank; skip))
```

The store process first sends an `authorize` message to the bank. Then, it waits for either an `ok` or `fail` message. If the authorization is successful, the store checks the stock of the two warehouses sequentially. If the first is successful but the second fails, the store must cancel the first warehouse's stock reservation. If both are successful, the store submits the two order confirmation messages and waits for receipts in parallel. When we run this process through BPELCheck, we find that it is indeed *conversationally consistent*: the process always terminates with all peer conversations in a consistent state and with the consistency predicate

`Atomic Action`	`A`	`::=`	skip | throw	built-ins
			| $\mathtt{A}_i \in \mathcal{A}$	defined actions
`Process`	`P`	`::=`	`A`	atomic actions
			| `P; P`	sequence
			| $\mathtt{P} \parallel \mathtt{P}$	parallel
			| $\mathtt{P} \ \square \ \mathtt{P}$	choice
			| $\mathtt{P} \div \mathtt{P}$	compensation
			| $\mathtt{P} \rhd \mathtt{P}$	exception handler

Figure 3.4: Syntax of process calculus

satisfied.

Bugs in our process can cause this verification to fail. For example, if the developer forgets to cancel the first warehouse's stock reservation when the second reservation fails, the first warehouse's conversation will be left in an inconsistent state, and BPELCheck will report an error. Processes which leave all conversations in consistent states but violate the consistency predicate will also fail. For example, a process which runs both warehouse conversations in parallel avoids leaving them in inconsistent states but violates the requirement that the two conversations must succeed or fail together. BPELCheck will report an error for this process as well.

3.3 Process Calculus

I use a simple process calculus, based on CCS (Calculus of Communicating Systems) [Mil80], to describe long-running transactions. I include extensions for compensation and exceptions, which appear in web service orchestration languages like BPEL. The syntax and semantics of my notation is similar to other formal models for such languages [BF00, BF04, BMM05].

3.3.1 Syntax

Figure 3.4 defines the syntax for our language. Processes are constructed from *atomic actions*, using a set of composition operations. Atomic actions are indivisible operations which either succeed completely or fail and undo all state changes. There are two built-in actions: skip, which always succeeds and does nothing, and throw, which throws an exception. We use $\mathcal{A}$ for the set of atomic actions defined by the environment and Σ for the set of all atomic actions: $\Sigma \equiv \mathcal{A} \cup \{\mathsf{throw}, \mathsf{skip}\}$.

A process is either an atomic action or a composition of atomic actions using one of five composition operators. The sequence operator ";" runs the first process followed by the second. If the first fails, the second is not run. The parallel operator "$\|$" runs two processes in parallel. The choice operator "$\Box$" non-deterministically selects one of two processes and runs it. The compensation operator "$\div$" runs the left process. If it completes successfully, the right process is installed as a compensation to run if the parent process terminates with an error. The exception handler "$\rhd$" runs the left process. If that process terminates with an error, the error is ignored and the right process is run. If the left process terminates successfully, the right process is ignored. Our core language does not include iteration operators —we will add iteration to our language in Section 3.4.2.

In our examples, we also use *named subprocesses* — sub-processes which are defined once and then appear as atomic actions in the overall process — as syntactic sugar to enhance readability. Named subprocesses are not true functions — they are simply inlined into their parent. Thus, recursive calls are not permitted.

We define $|P|$, the size of process P, by induction: the size $|A|$ of an atomic action in Σ is 1, and the size $|P_1 \otimes P_2|$ for any composition operation $\otimes$ applied to P_1 and P_2 is $|P_1| + |P_2| + 1$.

`Process form`	Π
$\Gamma \vdash A : \{\checkmark, \times\}$	$\{(A\langle\checkmark\rangle, \mathsf{skip}), (\langle\times\rangle, \mathsf{skip})\}$
$\Gamma \vdash A : \{\checkmark\}$	$\{(A\langle\checkmark\rangle, \mathsf{skip})\}$
$\Gamma \vdash A : \{\times\}$	$\{(\langle\times\rangle, \mathsf{skip})\}$
`P; Q`	$\{(pq\langle s\rangle, Q'; P') \mid (p\langle\checkmark\rangle, P') \in \Pi(P), (q\langle s\rangle, Q') \in \Pi(Q)\} \cup$ $\{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\}$
`P` $\parallel$ `Q`	$\{(r\langle s\&t\rangle, P' \parallel Q') \mid r \in p \bowtie q, (p\langle s\rangle, P') \in \Pi(P), (q\langle t\rangle, Q') \in \Pi(Q)\}$ $\cup \ \{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\} \ \cup \ \{(q\langle\times\rangle, Q') \mid (q\langle\times\rangle, Q') \in \Pi(Q)\}$
`P` $\Box$ `Q`	$\{(p\langle s\rangle, P') \mid (p\langle s\rangle, P') \in \Pi(P)\} \ \cup \ \{(q\langle t\rangle, Q') \mid (q\langle t\rangle, Q') \in \Pi(Q)\}$
`P` $\div$ `Q`	$\{(p\langle\checkmark\rangle, Q) \mid (p\langle\checkmark\rangle, P') \in \Pi(P)\} \ \cup \ \{(p\langle\times\rangle, P') \mid (p\langle\times\rangle, P') \in \Pi(P)\}$
`P` $\triangleright$ `Q`	$\{(p\langle\checkmark\rangle, P') \mid (p\langle\checkmark\rangle, P') \in \Pi(P)\} \cup$ $\{(pp'q\langle t\rangle, Q') \mid pp'\langle\times\rangle \in [\![P]\!], (q\langle t\rangle, Q') \in \Pi(Q)\}$

Figure 3.5: Trace semantics for core process language

3.3.2 Trace Semantics

We now define a trace-based semantics for processes. A *run* is a (possibly empty) sequence of atomic actions from Σ. A *trace* is a run followed by either $\langle\checkmark\rangle$ or $\langle\times\rangle$, representing successful and failed executions, respectively. For example, if action A_1 is run successfully and then action A_2 fails, the corresponding trace would be $A_1\langle\times\rangle$. In the following, we let the variable A range over atomic actions; the variables P, P', Q, Q', and R range over processes; the variables p, q, and r range over runs, and the variables s and t range over $\checkmark$ and $\times$.

We define an operator "&" which combines two process status symbols ($\checkmark$, $\times$). Given $s\&t$, if both s and t are $\checkmark$, then $s\&t = \checkmark$. Otherwise, $s\&t = \times$. For the parallel composition rules, we introduce an operator $\bowtie$: $\mathcal{R} \times \mathcal{R} \to 2^{\mathcal{R}}$, where, given two runs p and q, $p \bowtie q$ produces the set of all interleavings of p and q.

We use the symbol Γ to represent an *action type environment*, which maps each action to a set of possible results $2^{\{\checkmark,\times\}} = \{\{\checkmark\},\{\times\},\{\checkmark,\times\}\}$ from running the associated action. This allows us to distinguish actions which may fail from actions which never fail. The result set $\{\times\}$ is for the special action throw, which unconditionally throws an error.

For each process form P, we define using mutual induction two semantic functions $\Pi(P)$ and $[\![P]\!]$. The rules for $\Pi : \mathcal{P} \to 2^{(\mathcal{T},\mathcal{P})}$, found in Figure 3.5, define a set of pairs. Each pair $(p\langle s\rangle, Q) \in \Pi(P)$ consists of a trace $p\langle s\rangle$, representing a possible execution of the process P, and a process Q, representing a *compensation process* associated with the trace that will get run on a subsequent failure.

The function $[\![P]\!] : \mathcal{P} \to 2^{\mathcal{T}}$, maps a process P to a set of feasible traces of P. To compute the actual traces possible for a top-level process, this function must consider the successful and failed traces independently. Compensation processes are dropped from successful traces. For a failed trace $p\langle\times\rangle$, one computes all possible compensation traces for the associated compensation process P' and appends these to p. $[\![P]\!]$ is defined as follows:

$$\begin{aligned}[\![P]\!] = \{(p\langle\checkmark\rangle|(p\langle\checkmark\rangle, P') \in \Pi(P)\} \cup \\ \{pp'\langle\times\rangle|(p\langle\times\rangle, P') \in \Pi(P), p'\langle s\rangle \in [\![P']\!]\}\end{aligned}$$

We now describe the rules in Figure 3.5 in more detail.

Atomic Actions For atomic actions, we enumerate the possible results for each action. Individual atomic actions use skip as a compensation process, since no compensation is provided by default. Compensation for an atomic action must be explicitly defined using the $\div$ operator.

Sequential Composition For a sequential composition $P; Q$, we consider two cases. If P succeeds, we have a successful trace $p\langle\checkmark\rangle$ for P, after which we concatenate a trace q from Q. The status for the overall trace is then $\langle s\rangle$, the status from the trace q. The compensation process (if Q fails) invokes the compensation for Q first, followed by the compensation for P. On the other hand, if P fails, we have a failed trace $p\langle\times\rangle$ for P. The process Q is not run at all.

Parallel Composition For parallel composition, we first consider the case where both sub-processes run. If both are successful, the entire process is successful. If one fails, the process throws an error. We simulate the parallel semantics by generating a possible trace for all interleavings of the two subprocesses. The compensation for the two sub-processes is also run in parallel. Note that, if a sub-process fails, the other sub-process runs to completion, unless it also encounters an error. However, if the second sub-process has not started yet, and the first fails, an implementation can avoid running the second at all. This is handled by the last two sets in the union.

Choice Composition The traces for $P \,\Box\, Q$ are simply the union of the traces for P and Q.

Compensation The compensation operator $P \div Q$ runs P and then sets up process Q as compensation. If P is successful, Q overrides any previous compensation for P — e.g., P' if $\Pi(P) = (p\langle\checkmark\rangle, P')$. If P fails, then the original compensation process returned by $\Pi(P)$ is instead returned. In this case, the process Q is never run.

Exception Handler The rule for the exception handling operator has two scenarios. Given $P \rhd Q$, if P is successful, Q is ignored. If P fails, the compensation for P is run and then process Q is run. To express this, we use $[\![P]\!]$ to obtain a full trace for P, including compensation. This makes the $[\![\cdot]\!]$ and Π functions mutually recursive.

Example 1. *To demonstrate these rules, we compute the set of feasible traces for the SimpleOrder process defined as:*

$$\begin{aligned} \mathit{SimpleOrder} =& \mathit{Billing}; \mathbf{ProcessOrder} \\ \mathit{Billing} =& \mathbf{Charge} \div \mathbf{Credit} \end{aligned}$$

This process first calls the Billing sub-process. This sub-process invokes the **Charge** *action to bills the customer's action. If this fails, the process terminates. If the charge succeeds,* **Credit** *is registered as a compensation action. Next,* **ProcessOrder** *is run to handle the actual order. If* **ProcessOrder** *is successful, the process terminates successfully. Otherwise, the* **Charge** *compensation is run and the process terminates with an error.*

We assume that the **Charge** *and* **ProcessOrder** *atomic actions both can fail, but the* **Credit** *compensation never fails. Thus, we obtain the following definition for* Γ*:*

$$\langle \mathbf{Charge} \mapsto \{\checkmark\ \times\},\ \mathbf{Credit} \mapsto \{\checkmark\}, \mathbf{ProcessOrder} \mapsto \{\checkmark, \times\}\rangle$$

If we apply the rules for atomic actions to each action in this process, we get the *following values for* Π*:*

$$\begin{aligned}\Pi(\mathbf{Charge}) =& \{(\mathbf{Charge}\langle\checkmark\rangle, \mathsf{skip}), (\langle\times\rangle, \mathsf{skip})\}\\ \Pi(\mathbf{Credit}) =& \{(\mathbf{Credit}\langle\checkmark\rangle, \mathsf{skip})\}\\ \Pi(\mathbf{ProcessOrder}) =& \{(\mathbf{ProcessOrder}\langle\checkmark\rangle, \mathsf{skip}), (\langle\times\rangle, \mathsf{skip})\}\end{aligned}$$

To compute the feasible traces for the Billing subprocess, we apply the rule for compensation, with $P = \mathbf{Charge}$ *and* $Q = \mathbf{Credit}$*, obtaining the following:*

$$\Pi(\mathit{Billing}) = \{(\mathbf{Charge}\langle\checkmark\rangle, \mathbf{Credit}), (\langle\times\rangle, \mathsf{skip})\}$$

To compute the feasible traces for SimpleOrder we use the sequential composition rule, using $P = \mathit{Billing}$ *and* $Q = \mathbf{ProcessOrder}$*:*

$$\begin{aligned}\Pi(\mathit{SimpleOrder}) =& \{(\mathbf{Charge}\ \mathbf{ProcessOrder}\langle\checkmark\rangle, \mathsf{skip}; \mathbf{Credit}),\\ & (\mathbf{Charge}\langle\times\rangle, \mathbf{Credit}),\ (\langle\times\rangle, \mathsf{skip})\}\end{aligned}$$

Finally, we compute the feasible traces $[\![\mathit{SimpleOrder}]\!]$ *by dropping compensation for successful traces and computing the compensation traces for failed traces:*

$$\{\mathbf{Charge}\ \mathbf{ProcessOrder}\langle\checkmark\rangle, \mathbf{Charge}\ \mathbf{Credit}\langle\times\rangle, \mathsf{skip}\langle\times\rangle\}$$

□

3.3.3 Trace composition

We end this section by looking at how the composition of processes affects the composition of traces. Given a process R and trace $T_r = r_f r_c\langle s\rangle$, where $T_r \in [\![R]\!]$, $(r_f\langle s\rangle, R') \in \Pi(R)$, and $r_c\langle s'\rangle \in [\![R']\!]$, we call the trace $r_f\langle s\rangle$ the *forward sub-trace* of T_r and $r_c\langle s'\rangle$ the *compensation sub-trace* of T_r. We write $r_f r_c\langle s\rangle$ for a trace r with forward sub-trace r_f and compensation sub-trace r_c and status $\langle s\rangle$. Note that the overall status of the trace is always equal to the status of the forward sub-trace.

Let $R = P \otimes Q$, where $\otimes$ is one of the composition operators. For each feasible trace T_r of R, either P is not called (e.g., if $\otimes = \Box$ and Q is chosen), or there is a trace T_p of P such that the forward subtrace of P occurs within the forward subtrace of T_r and the compensation subtrace of T_p appears in the compensation subtrace of T_r.

Theorem 8. *Let $R = P \otimes Q$, $p_f p_c\langle t\rangle \in [\![P]\!]$, and $r_f r_c\langle s\rangle \in [\![R]\!]$ such that $r_f = r_{f0} p_f r_{f_1}$ (the run p_f appears within r_f), and $r_c = r_{c0} p_c r_{c_1}$ (the run p_c appears within r_c). The forward sub-trace $r_{f_1}\langle s\rangle$ of R following the call to P depends only on $\langle t\rangle$ (the forward status of P), not on the individual actions in p_f. Likewise, the compensation sub-trace $r_{c_1}\langle s'\rangle$ of R following the call to P' depends only on the compensation status of P', not on the individual compensation actions in p_c.*

Proof (outline). The proof is by induction on the derivation of $[\![R]\!]$, using a case analysis on the syntactic forms of each subprocess. For each composition operator, inspection of the corresponding rules of Figure 3.5 show that, in each case, there is no dependency on the individual actions called by the subprocesses and the status of the parent process depends only upon the status of the subprocesses. $\Box$

3.4 Set Consistency

We can now formalize our notion of set consistency with respect to the process calculus.

Executions For a trace $p\langle s\rangle \in [\![P]\!]$, we define the *execution* for $p\langle s\rangle$ as the set $\pi_p \subseteq \Sigma$ of atomic actions that appear in p, that is, the execution $e(p) = \{a \in \Sigma \mid \exists p_1, p_2 . p\langle s\rangle \equiv p_1 a p_2\langle s\rangle\}$. For a process P, let $\mathsf{execs}(P) \subseteq 2^{\Sigma}$ represent the set of all executions of P, defined by $\mathsf{execs}(P) = \{e(p) \mid p\langle s\rangle \in [\![P]\!]\}$.

Example 2. *If we drop calls to* skip*, the set of feasible executions* $\mathsf{execs}(\mathit{SimpleOrder})$ *for* $\mathit{SimpleOrder}$ *is*

$$\{\{\mathbf{Charge}, \mathbf{ProcessOrder}\}, \{\mathbf{Charge}, \mathbf{Credit}\}, \emptyset\}$$

□

Set Consistency Specifications A *set consistency specification* $\mathcal{S} \subseteq 2^{\Sigma}$ is a set of action sets representing the permissible executions for a given process. A process P is *set consistent* with respect to a set consistency specification $\mathcal{S}$ if all executions of P fall within the set consistency specification: $\mathsf{execs}(P) \subseteq \mathcal{S}$.

Set consistency specifications are semantic objects. We use a boolean predicate-based syntax for describing sets of executions. Given a set of atomic actions Σ, a *set consistency predicate* is an expression built from combining atomic predicates Σ with the logical operators $\neg$ (not), $\wedge$ (and), and $\vee$ (or). The size $|\varphi|$ of predicate φ is the defined by induction: $|a| = 1$ for a literal, and $|\varphi_1 \otimes \varphi_2| = |\varphi_1| + |\varphi_2| + 1$ for a logical operator $\otimes$. To evaluate a predicate φ on an execution $e \in 2^{\Sigma}$, we assign the value `true` to all atomic actions that occur in e and `false` to all the atomic actions that do not occur in e. If the predicate φ evaluates to `true` with these assignments, we say the execution e *satisfies* the specification φ, and write $e \models \varphi$. The set consistency specification $\mathcal{S}$ defined by the set consistency predicate φ is the set of satisfying assignments of φ:

$$\mathsf{spec}(\varphi) = \{e \in 2^{\Sigma} \mid e \models \varphi\}$$

Normalization When defining set consistency specifications, we wish to treat the compensated execution of an action as equivalent to an execution where the action (and its compensation) never occurs. If A° is a compensation action for A, then the term $\neg A$ in the specification predicate should yield two executions in the final, expanded specification: one with neither A nor A°, and one with both A and A°. We call a specification that has been adjusted in this manner a *normalized* specification. Normalization is performed with respect to a programmer-specified *normalization set* $\mathcal{C} \subseteq \{(a, a^{\circ}) | a, a^{\circ} \in \Sigma\}$ of atomic action pairs, where the second action in each pair is the compensation action for the first action. Given the consistency specification predicate φ and a normalization set $\mathcal{C}$, we apply the function $\mathsf{spec\_norm}(\varphi, \mathcal{C})$. This function uses DeMorgan's laws and the normalization set to convert the predicate to a form where (1) the negation operator only appears in front of literals, and (2) given a pair (a, a') from $\mathcal{C}$, each occurrence of $\neg a$ is replaced with $(\neg a \wedge \neg a') \vee (a \wedge a')$ and each occurrence of a is replaced with $a \wedge \neg a'$.

For successful executions of an action, the normalized specification asserts that the compensation action was not run. For failed executions, spec_norm changes the specification to treat equivalently the following three scenarios:

1. Actions a and a' are never run.
2. Action a is executed but fails, undoing any partial state changes. Action a' is never run.
3. Action a is executed successfully, but compensation a' is later run to undo

the effects of a.

Example 3. *We consider a specification for the SimpleOrder process. We assume the action typing Γ_{so} of Example 1 (in which the* **Credit** *action never fails) and the normalization set $\mathcal{C}_{so} = \{(\mathbf{Charge}, \mathbf{Credit})\}$.*

We wish to have cancellation semantics, where either both **Charge** *and* **ProcessOrder** *complete successfully, or, in a failed execution, any completed actions are undone. Given the normalization set, we do not need to distinguish between failure cases. Thus, we can write a set consistency predicate φ_{so} for SimpleOrder as:*

$$(\mathbf{Charge} \wedge \mathbf{ProcessOrder}) \ \vee (\neg\mathbf{Charge} \wedge \neg\mathbf{ProcessOrder})$$

We now expand φ_{so} to get $\mathsf{spec\_norm}(\varphi_{so}, \mathcal{C}_{so})$. It is already in the form we need for substitution (negation only of literals). We replace all occurrences of $\neg\mathbf{Charge}$ with $(\neg\mathbf{Charge} \wedge \neg\mathbf{Credit}) \vee (\mathbf{Credit} \wedge \mathbf{Charge})$ and all occurrences of **Charge** *with $(\mathbf{Charge} \wedge \neg\mathbf{Credit})$:*

$$\begin{aligned}&(\mathbf{Charge} \wedge \neg\mathbf{Credit} \wedge \mathbf{ProcessOrder}) \ \vee \\ &\big((\neg\mathbf{Charge} \wedge \neg\mathbf{Credit}) \vee (\mathbf{Charge} \wedge \mathbf{Credit})\big) \wedge \neg\mathbf{ProcessOrder}\end{aligned}$$

□

Set consistency verification For a process P, a normalization set $\mathcal{C}$, and a set consistency predicate φ, the set consistency *verification problem* is to check if all the executions of P w.r.t. $\mathcal{C}$ satisfy φ, that is, if $\mathsf{execs}(P) \subseteq \mathsf{spec}(\mathsf{spec\_norm}(\varphi, \mathcal{C}))$.

Theorem 9. *The set consistency verification problem is co-NP complete.*

Proof (outline). Verification is in co-NP, as finding a counterexample is in NP. To find a counterexample, we nondeterministically enumerate feasible executions of a process using the trace semantics. At each possible decision point (e.g. success or failure of actions and the choice operator), we guess an outcome. We track only the set of actions which were called successfully, and, upon termination, check whether this execution is a satisfying assignment of the specification predicate. An execution is polynomial in the size of the process, since it includes each action at most once. Checking whether the execution satisfies the specification predicate φ is polynomial with respect to $|\varphi|$.

To show that verification is co-NP hard, we reduce checking for tautology to set consistency verification. To determine whether a predicate ϕ, consisting of literals from the set Σ_ϕ, is a tautology, we first construct a process P whose feasible executions are the powerset of Σ_ϕ. One such process is $A_1 \parallel A_2 ... \parallel A_n$ where $A_1, ...A_n \in \Sigma_\phi$ and $\forall i \,.\, \Gamma \vdash A_i : \{\checkmark, \times\}$. Now, we interpret ϕ as a set consistency predicate: If process P satisfies the specification ϕ, then ϕ is a tautology. $\square$

3.4.1 Predicate-based verification

We now give an algorithm for verifying set consistency. We define a syntax-directed analysis which takes a process as input and constructs a predicate ϕ whose satisfying assignments precisely represent the feasible execution set. The predicate, ϕ is composed from atomic predicates Σ using logical operators $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. For the moment, we assume that a given action A is referenced only once in a process. Later, we will extend our approach to remove this restriction. A predicate ϕ over atomic predicates in Σ defines a set of executions $\mathcal{E}(\phi) = \{e \in 2^\Sigma \mid e \models \phi\}$.

Execution Predicate Construction We create the predicate ϕ by recursively applying the rules of Figure 3.6, based on the form of each sub-process. These rules define seven mutually-recursive functions: $\phi_{\checkmark 0}$, $\phi_{\checkmark\checkmark}$, $\phi_{\checkmark\times}$, $\phi_{\times 0}$, $\phi_{\times\checkmark}$, $\phi_{\times\times}$, and ϕ_{00}, all with the signature $\mathcal{P} \times \mathcal{G} \rightarrow \Phi$, where $\mathcal{P}$ is the set of all processes, $\mathcal{G}$ is the set of all action type environments, and Φ the set of all predicates. The two subscripts of each function's name represent the forward and compensation results of running the process, respectively, where $\checkmark$ is a successful execution, $\times$ is a failed execution, and 0 means that the process was not run. For example, $\phi_{\checkmark\times}(P, \Gamma)$ returns a predicate representing all executions of process P where the forward process completes successfully and the compensation process (whose execution must be initiated by the failure of a containing process) fails, given a type environment Γ. As a shorthand, we use terms of the form $P_{\checkmark\times}$ to represent the function $\phi_{\checkmark\times}(P, \Gamma)$. We also leave out the conjunction symbol ("$\wedge$") when it is obvious from the context (e.g. $P_{\checkmark 0} Q_{\checkmark 0}$ for $P_{\checkmark 0} \wedge Q_{\checkmark 0}$).

Given these functions, we compute the predicate ϕ, representing the possible executions of a process, using the function $\mathsf{pred} : \mathcal{P} \times \mathcal{G} \rightarrow \Phi$, defined as:

$$\mathsf{pred}(P, \Gamma) \equiv \phi_{\checkmark 0}(P, \Gamma) \ \vee \ \phi_{\times\checkmark}(P, \Gamma) \ \vee \ \phi_{\times\times}(P, \Gamma)$$

Example 4. *We illustrate the predicate generation algorithm by constructing a predicate for the SimpleOrder process. We start from the execution predicate definition:*

$$\mathsf{pred}(SimpleOrder, \Gamma_{so}) = \phi_{\checkmark 0} \ \vee \ \phi_{\times\checkmark} \ \vee \phi_{\times\times}$$

and iteratively apply the appropriate sub-predicates from Figure 3.6. We show the steps of the computation in Figure 3.7. We obtain step (2) from step (1) in Figure 3.7 by substituting each term with the sub-predicates from the sequence *rule, since the top-level process is a sequence. Next, from step (2) to step (3), we use the atomic action rules to simplify the predicates for* **ProcessOrder***. Notice that the fourth conjunction in (3) can be dropped since it conjoins* `false`*. Now, we expand the predicates for Billing using rules for the compensation operator to get step (4). Finally, we expand the* **Charge** *and* **Credit** *predicates again using rules for the atomic actions to get (5). Simplifying, we get (6), the execution predicate for SimpleOrder.*

Note that the three conjunctions in the final predicate correspond to the three possible executions of SimpleOrder:

1. *The actions* **Charge** *and* **ProcessOrder** *are successful, and the compensation* **Credit** *is not run.*
2. *The action* **Charge** *completes successfully, but* **ProcessOrder** *fails. Then,* **Credit** *is run to compensate for* **Charge***.*
3. *The action* **Charge** *fails, and the actions* **ProcessOrder** *and* **Credit** *are never run.*

□

Memoization As usual, one can memoize each computation by giving names to sub-predicates. While generating the execution predicate, we name each non-atomic sub-predicate, using the names as literals instead of expanding the sub-predicates of ϕ. The resulting execution predicate is then conjoined with a definition $(n \leftrightarrow \phi_n)$ for each name n and its definition ϕ_n.

Using memoization, the execution predicate of our previous example becomes:

$$
\begin{aligned}
&(\mathit{SimpleOrder}_{\checkmark 0} \vee \mathit{SimpleOrder}_{\times\checkmark}) \\
&\wedge\ (\mathit{SimpleOrder}_{\checkmark 0}\ \leftrightarrow\ \mathit{Billing}_{\checkmark 0} \wedge \mathbf{ProcessOrder}) \\
&\wedge\ (\mathit{Billing}_{\checkmark 0}\ \leftrightarrow\ \mathbf{Charge} \wedge \neg\mathbf{Credit}) \\
&\wedge\ (\mathit{SimpleOrder}_{\times\checkmark}\ \leftrightarrow \\
&\qquad (\mathit{Billing}_{\checkmark\checkmark} \wedge \neg\mathbf{ProcessOrder} \vee \\
&\qquad\ \mathit{Billing}_{\times\checkmark} \wedge \neg\mathbf{ProcessOrder})) \\
&\wedge (\mathit{Billing}_{\checkmark\checkmark}\ \leftrightarrow\ \mathbf{Charge} \wedge \mathbf{Credit}) \\
&\wedge (\mathit{Billing}_{\times\checkmark}\ \leftrightarrow\ \neg\mathbf{Charge} \wedge \neg\mathbf{Credit})
\end{aligned}
$$

Although the memoized execution predicate for this particular example is larger than the non-memoized predicate, in general, the worst-case size of the memoized predicate is polynomial in the size of a process, whereas the size of a non-memoized predicate can be exponential.

Theorem 10. *For any process P and action type environment Γ,*

1. *The execution set obtained from P's execution predicate is equal to the set of all P executions: $\mathcal{E}(\mathsf{pred}(P,\Gamma)) = \mathsf{execs}(P,\Gamma)$.*
2. *$|\mathsf{pred}(P,\Gamma)|$ is polynomial in $|P|$.*

Proof (outline). We prove the first assertion by induction on the derivation of $\mathsf{pred}(P)$, first considering direct generation of the execution predicate, without memoization. The base cases are each of the atomic action rules. For each rule, we show that the satisfying assignments to the predicates correspond to the feasible executions. The inductive cases are the compositional operator rules. We can simplify the proof for these cases using theorem 8, which shows that, when composing subprocesses, one can compose the traces, only considering the status (success or failure) of the sub-traces. For each operator, we show that, assuming that the component predicates (e.g. $P_{\checkmark\checkmark}$, etc.) are correct, then the predicate constructed for the operator has satisfying assignments corresponding to the executions obtained from the associated trace semantics rule.

Memoization. To prove that our memoization algorithm is correct, we show a more general property for an arbitrary predicate ϕ containing a sub-predicate φ. An assignment $\mathbb{A}$ of values to the variables of ϕ is a satisfying assignment for ϕ, if and only if it is satisfying assignment for $(\phi[C/\varphi]) \wedge (C \leftrightarrow \varphi)$, where C is a literal not appearing in ϕ. To see this, we use several boolean algebra identities. We convert $\phi[C/\varphi]$ to disjunctive normal form, yielding ϕ_{dnf}.

We assume without proving here that replacing each occurrence of C with φ in ϕ_{dnf} would yield a predicate equivalent to ϕ. Given this, we only need to show that conjoining ϕ_{dnf} with $C \leftrightarrow \varphi$ is equivalent to performing this substitution. We conjoin each term t_i of ϕ_{dnf} with $C \wedge \varphi \vee \neg C \wedge \neg\varphi$, creating terms t_i^C and $t_i^{\neg C}$. If C and not $\neg C$ appear in t_i, then $t_i^C = t_i \wedge \varphi$ and $t_i^{\neg C}$ is unsatisfiable. When considering satisfiability, we can ignore the term C in t_i^C, since it is a free variable in ϕ_{dnf}. The case where t_i contains $\neg C$ but not C is similar. If t_i contains C and $\neg C$, then t_i and $t_i[\varphi/C]$ are unsatisfiable. Thus, $\phi_{dnf} \wedge (C \leftrightarrow \varphi)$ is equivalent to $\phi_{dnf}[\varphi/C]$.

Execution predicate size. To show that the execution predicate's size is polynomial with respect to the original process, we use induction on the structure of processes. The base causes are atomic actions which all have predicates of one term. For the inductive case, we consider an arbitrary subprocess $P \otimes Q$ where the predicates $\phi_{\checkmark 0}(P), ...\phi_{00}(P)$ are all polynomial in the size of P and the predicates $\phi_{\checkmark 0}(Q), ...\phi_{00}(Q)$ are all polynomial in the size of Q. Using memoization, we need only reference each of these predicates once. The predicates

$\phi_{\checkmark 0}(P \otimes Q), ... \phi_{00}(P \otimes Q)$ are thus independent in size from the sizes of P and Q. The largest possible size is, in fact, 67 for $\phi_{\times\times}(P \parallel Q)$. $\square$

3.4.1.1 Checking Consistency

We check a specification by checking if the execution predicate $\mathsf{pred}(P, \Gamma)$ implies the normalized specification predicate $\mathsf{spec\_norm}(\varphi, \mathcal{C})$. If the implication is valid, then all executions satisfy the specification and the solution to the consistency verification problem is "yes." Otherwise, there is some execution that does not satisfy the consistency specification. Therefore, to check a process for consistency, we can build the execution and normalized specification predicates and check the implication by a Boolean satisfiability query.

Theorem 11. *For any process P and specification predicate φ, with action type environment Γ and normalization set $\mathcal{C}$,*

1. $\mathsf{execs}(P, \Gamma) \subseteq \mathsf{spec}(\mathsf{spec\_norm}(\varphi, \mathcal{C}))$ *iff* $\mathsf{pred}(P, \Gamma) \rightarrow \mathsf{spec\_norm}(\varphi, \mathcal{C})$ *is valid.*
2. *The consistency verification problem can be solved in time exponential in some polynomial function of $|P|$, $|\Gamma|$, $|\varphi|$, and $|\mathcal{C}|$.*

Proof (outline). This follows from the previous theorems. If the execution predicate is a sound and complete representation of a process's executions, then the process satisfies a specification only if the execution predicate implies the specification.

The verification predicate can be generated in polynomial time with respect to the algorithm's inputs: only one pass must be made through the process and the size of the generated predicate is polynomial with respect to $|P|$, $|\Gamma|$, $|\varphi|$, and $|\mathcal{C}|$. From theorem 10 we know that the execution predicate is polynomial in size with respect to P. $|\Gamma|$ and $|\mathcal{C}|$ are not larger than the number of unique actions in P and $|\phi|$ is independent from Γ. Finally, $\mathsf{spec\_norm}$ causes at most a polynomial expansion of the specification predicate. $\square$

Example 5. *Given the previously computed values of* $\mathsf{pred}(SimpleOrder, \Gamma_{so})$ *and* $\mathsf{spec\_norm}(\varphi_{so}, \mathcal{C}_{so})$*, the verification problem for process SimpleOrder may be reduced to checking the validity of the following predicate:*

$$\begin{aligned}
&\big(\mathbf{Charge} \wedge \neg\mathbf{Credit} \wedge \mathbf{ProcessOrder} \vee \\
&\ \mathbf{Charge} \wedge \mathbf{Credit} \wedge \neg\mathbf{ProcessOrder} \vee \\
&\ \neg\mathbf{Charge} \wedge \neg\mathbf{Credit} \wedge \neg\mathbf{ProcessOrder}\big) \ \rightarrow \\
&\ \big((\mathbf{Charge} \wedge \mathbf{ProcessOrder}) \vee \\
&\ \big((\neg\mathbf{Charge} \wedge \neg\mathbf{Credit}) \vee (\mathbf{Charge} \wedge \mathbf{Credit})\big) \\
&\quad \wedge \neg\mathbf{ProcessOrder}\big)
\end{aligned}$$

$\square$

Multiple Calls to an Action So far, we have assumed that each action in $\mathcal{A}$ is called at most once in a given process. If an action may be called multiple times by a process, we do not distinguish the individual calls. Given an action A, which appears more than once in the text of the process P, the specification predicate A is `true` if A is called at least once, and `false` if A is never called. This follows directly from our definition of a process execution, which is a *set* of called actions, rather than a *bag*. These semantics are useful for situations where an action may be called in one of several situations, and we wish to verify that, given some common condition, the action is called. For example, in $OrderProcess$ of Section 3.7.1.1, the action `MarkFailed` must be called if the order fails, either due to a failed credit check or to a failed fulfillment subprocess.

If an action is called more than once, $\mathsf{pred}(P, \Gamma)$ may produce an unsatisfiable predicate. For example, $\phi_{\checkmark 0}(A \,\square\, A) = (A \wedge \neg A) \;\vee\; (\neg A \wedge A)$, which is clearly unsatisfiable. We solve this by applying the function $\mathsf{trans\_calls} : \mathcal{P} \rightarrow \mathcal{P} \times \Phi$, which takes as input a process P and returns a translated process P' along with a predicate ϕ_{tc}. Given a set of actions $\{A^1, ...A^n\} \subseteq \mathcal{A}$, which occur more than once in P, each occurrence of these actions the translated process P' is given a unique integer subscript. For example, given the process $P = (A; B) \,\square\, (A; B)$, $\mathsf{trans\_calls}(P)$ will return the process $P' = (A_1; B_1) \,\square\, (A_2; B_2)$. The predicate $\phi_{tc} = \phi^1_{tc} \wedge ...\phi^n_{tc}$, where ϕ^i_{tc} uses the boolean $\leftrightarrow$ operator to associate the atomic predicate A^i with the disjunction of the predicates for each subscript. For example, ϕ_{tc} for $P = (A; B) \,\square\, (A; B)$ will be $(A \leftrightarrow (A_1 \vee A_2)) \wedge (B \leftrightarrow (B_1 \vee B_2))$. We now re-define our predicate function pred, to combine $\mathsf{trans\_calls}$ with $\mathsf{pred}_{\mathsf{mem}}$, our original predicate generation function:

$$\mathsf{pred}(P, \Gamma) = \texttt{let}\ (P', \phi_{tc}) = \mathsf{trans\_calls}(P)\ \texttt{in}$$

$$\mathsf{pred}_{\mathsf{mem}}(P', \Gamma)\ \wedge \phi_{tc}$$

Lemma 18. *For any process P, which may contain multiple calls to the same action, and atomic action typing Γ, the execution set obtained from applying* pred *to P and Γ is equal to the set of all P executions: $\mathcal{E}(\mathsf{pred}(P, \Gamma)) = \mathsf{execs}(P, \Gamma)$.*

Proof (outline). We show this through boolean algebra identities. Assume that process P contains an action A which is called n times. Thus, $\mathsf{trans\_calls}(P) = (P', A \leftrightarrow (A_1 \vee A_2... \vee A_n))$ and $\phi_{p'} = \mathsf{pred}_{mem}(P')$, where P' has actions $A_1, A_2, ...A_n$ substituted for each occurrence of A. We convert $\phi_{p'}$ to disjunctive normal form, yielding $\phi^{dnf}_{p'}$. We state, without proving here, that each term of $\phi^{dnf}_{p'}$ will contain all actions of P', either directly or in negated form. We conjoin each term t_i of $\phi_{p'}^{dnf}$ with $A \leftrightarrow (A_1 \vee ... A_n)$. This yields the terms $t_i^{A_1} = t_i \wedge A \wedge A_1$, $t_i^{A_2} = t_i \wedge A \wedge A_2$, ... $t_i^{A_n} = t_i \wedge A \wedge A_n$, and $t_i^{\neg A} = t_i \wedge \neg A \wedge \neg A_1 ... \wedge \neg A_n$. The terms $t_i^{A_j}$ are satisfiable if t_i satisfiable, A is assigned `true`, and t_i contains A_j as a conjunct (and thus not $\neg A_j$). The term $t_i^{\neg A}$ is satisfiable if t_i is satisfiable, A is assigned `false`, and t_i contains $\neg A_1, ... \neg A_n$ as conjuncts. If we view the terms t_i as possible executions for P, this matches our intuition: if any of the subscripted actions A_j are included in the execution, then we include A. $\square$

Named subprocesses As mentioned in Section 3.3, named subprocesses are simply expanded at each call point. This is done before running trans_calls, since expansion may introduce actions which are called more than once.

Counterexample traces As a consequence of Theorem 11, if a process does not satisfy a specification, we can obtain a model for the execution predicate which violates the specification predicate. This model corresponds to a counterexample *execution*. Using the process definition and a model for the memoized execution predicate, we can also obtain a counterexample *trace* — an ordered list of actions called by the process, annotated with the success/failure status of each action.

This is accomplished by walking the process definition, simulating its execution per the trace semantics of Section 3.3. When a choice process is encountered, the model is checked to see which subprocess of the choice has a true value for its ϕ_{00} predicate, indicating that it was not run. The traversal continues with the other subprocess, ignoring the not-chosen subprocess. When an atomic action is reached, the model is checked to see if that action completed successfully. If so, it is added to the trace as a successful action. Otherwise, it is added to the trace as a failed action, and result propagated up through its parent processes.

The model is ambiguous — both non-execution and failure of atomic actions are represented by mappings to `false`. We resolve this ambiguity by ensuring that the traversal only reaches an atomic action if it was attempted. For example, if the first action of a sequence fails, then the rest of the sequence is not traversed. There are a few cases where either failure or non-execution will cause the same future behavior of a process (e.g., the first action of the overall process). For these cases, we assume failure rather than non-execution, unless a given action cannot fail. The computation of a counterexample trace is polynomial with respect to the process size — each subprocess and action is traversed only once.

3.4.2 Iteration

We now extend our core process language to support iteration by introducing two new process composition operators, "$**$" and "$*|$", with the following syntax:

```
Process  P  ::=  ...
               |  **P   sequential iteration
               |  *|P   parallel iteration
```

The sequential iteration operator runs a process one or more times, one copy at a time. The parallel iteration operator runs one or more copies of a process in parallel. For both operations, the number of copies is not known a priori — this is determined at runtime.

We define the trace semantics of these operators by taking the fixpoint of the following two equations:

$$\Pi(**P) = \Pi(P) \ \cup \ \Pi(P;(**P))$$

$$\Pi(*|P) = \Pi(P) \ \cup \ \Pi(P \parallel (*|P))$$

The sequential iteration operator generates a sequence of traces from $\Pi(P)$, where all but the last trace must correspond to successful executions. As with pairwise sequential composition, the compensation process for sequential iteration runs the compensation processes for each action sequentially, in reverse order from the forward execution. Parallel set iteration interleaves traces of P arbitrarily. If any of these traces fails, the parent has a failed status. As with parallel composition, processes may not be interrupted in the event of a failure, but may be skipped if they have not started. Compensation is run in parallel as well.

We wish to generalize set consistency predicates to processes with iteration. We encounter two problems with iteration. First, the set of potential traces for a given process now becomes infinite. Second, consider two atomic actions A and B, which occur within an iterated subprocess. The specification $A \wedge B$ will be true for all traces where both A and B are called at least once, even if A and B are never called in the same iteration. This is usually too weak. In an executable process, an iterated subprocess would likely be parameterized by some value (dropped when abstracting to our core language), which changes each iteration. Thus, one is more likely interested in checking if, whenever A is called, B is also called within the same iteration. We will see an example of such a process in Section 3.7.1.1.

Quantified specification predicates We now generalize the semantics of specification predicates to permit specifying that all iterations of an subprocess obey a constraint. As a running example, we use the following process:

$$P = (*\,*\,(A; B)); (*|(C \,\square\, D))$$

First, we augment process traces to uniquely identify each call to an action within iterative subprocesses. We number the iterations of each iterative subprocess, starting at 1. We add this iteration number as a subscript to the actions called within each iteration. For example, given an run of process P which executes the $A; B$ iteration twice and runs two copies of C and one copy of D for the parallel iteration, we obtain the trace: $A_1\ B_1\ A_2\ B_2\ C_1\ C_2\ D_3$. For parallel iteration, since subprocesses may be interleaved, the mapping of subprocess executions to iterations is arbitrary. If an iterative subprocess is nested within another iterative subprocess, we subscript each inner subprocess's actions with a sequence of iteration numbers, starting with the outermost subprocess. We extend the trace semantics function $[\![\cdot]\!]$ to produce traces of this form for iterative subprocesses.

The execution for an iterative trace is simply the set of actions called, propagating the iteration subscripts. Thus, the actions of each iteration are distinguished. For example, execution for the above trace of process P would be: $\{A_1, A_2, B_1, B_2, C_1, C_2, D_3\}$. We extend the execution function $\mathsf{execs} : \mathcal{P} \rightarrow 2^{\mathcal{A}}$ to produce executions of this form.

Next, we extend the specification semantics function $\mathsf{spec} : \Phi \times \mathcal{P} \rightarrow \mathcal{E}$ to permit specification predicates over iterative executions. As discussed above, we wish to consider whether a property holds over all iterations. Therefore, we forbid the use of specific iteration subscripts in the specification predicate. Instead, we assign a unique index variable to each iterative subprocess and universally quantify the specification predicate over all index variables. For example, to specify that, for each sequential iteration in process P, the execution of B implies the execution of A, we write $\forall_i(\neg B_i \vee A_i)$. We write $\varphi^{\forall}$ to represent a quantified specification predicate. Predicates outside any iteration are not subscripted.

Thus, given the process $P' = (**(X; Y)); Z$, the predicate $\forall_i(\neg Z \vee (X_i \wedge Y_i))$, means "in all executions of the process where Z is called, X and Y are called together for all iterations."

We can now define the semantics of an execution predicate, given a universally quantified specification predicate $\varphi^{\forall}$ and a process P:

$$\mathsf{spec}^{\forall}(\varphi^{\forall}, P) = \{e \in \mathsf{execs}(P) \mid e \models \varphi^{\forall}\}$$

Verification Clearly, if a quantified specification predicate is satisfied by all possible iterations of a process, then any single iteration will satisfy the predicate as well. More importantly, the reverse is also true. To show this, we first define a function $\mathsf{erase} : \mathcal{P} \to \mathcal{P}$, which removes the iteration operators from a process, replacing them with the underlying subprocess:

$$\mathsf{erase}(P) = \begin{cases} \mathsf{erase}(P') & \text{if } P = **(P') \\ \mathsf{erase}(P') & \text{if } P = *|(P') \\ \mathsf{erase}(Q) \otimes \mathsf{erase}(R) & \text{if } P = Q \otimes R \\ A & \text{if } P = A \end{cases}$$

Given a specification predicate $\varphi^{\forall}$, we write φ^{ϵ} for the predicate obtained by removing the universal quantifier and any iteration subscripts on literals.

Theorem 12. *Given a process P, a normalization set $\mathcal{C}$, and quantified specification predicate $\varphi^{\forall}$, iff $\mathsf{erase}(P)$ is correct with respect to φ^{ϵ}, then P is correct with respect to $\varphi^{\forall}$.*

Proof (outline). Assume that process P contains an iterative subprocess $Q = **(R)$ or $Q = *|(R)$.

The function $\mathsf{select\_iters} : 2^{\mathcal{A}} \to \mathcal{E}$ takes an execution E of P and returns the set of executions $E^1, E^2, ...E^n$ where each E^i substitutes the execution E_{q_i} of Q iteration i for the execution E_q of all Q iterations. We can lift $\mathsf{select\_iters}$ to sets of executions by applying it to each execution in the input set and accumulating all resulting executions in a single set.

We may evaluate a feasible execution set $\mathsf{execs}(P)$ of process P against the quantified execution predicate $\varphi^{\forall}$ by evaluating each execution in $\mathsf{select\_iters}(\mathsf{execs}(P))$ against φ^{ϵ}. If all these executions satisfy φ^{ϵ}, then P satisfies $\varphi^{\forall}$.

Lemma 19.

$$\forall_{e \in \mathsf{select\_iters}(\mathsf{execs}(P))} \, . \, (e \models \varphi^{\epsilon}) \rightarrow \forall_{e' \in \mathsf{execs}(P)} \, . \, (e' \models \varphi^{\forall})$$

From the trace definitions for iteration, we can see that the possible traces for an single iteration of Q are the same as if the subprocess R is run standalone. In other words, iterations are "stateless." Thus, the possible executions and forward/compensation status pairs for any one iteration of Q are the same as those for R. From theorem 8, we know that the trace of the process calling Q depends only on the status of Q, not the individual actions. Thus, the set of feasible executions we get when replacing Q with R are the same as when we select each iteration from the executions of P. More formally:

Lemma 20. $\mathsf{execs}(\mathsf{erase}(P)) = \mathsf{select\_iters}(\mathsf{execs}(P))$.

Theorem 12 follows from lemmas 19 and 20. □

Thus, we can check an iterative process by simply erasing the iterative operators, and checking against the associated unquantified specification.

`Predicate`	$\Gamma \vdash A : \{\checkmark\}$	$\Gamma \vdash A : \{\checkmark, \times\}$	$\Gamma \vdash A : \{\times\}$
$\phi_{\checkmark 0}$	A	A	`false`
$\phi_{\checkmark\checkmark}$	A	A	`false`
$\phi_{\checkmark\times}$	`false`	`false`	`false`
$\phi_{\times 0}$	`false`	$\neg A$	$\neg A$
$\phi_{\times\checkmark}$	`false`	$\neg A$	$\neg A$
$\phi_{\times\times}$	`false`	`false`	`false`
ϕ_{00}	$\neg A$	$\neg A$	$\neg A$

`Predicate`	`P;Q`	`P`$\|$`Q`	`P`$\Box$`Q`
$\phi_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{00}Q_{\checkmark 0}\vee\ P_{\checkmark 0}Q_{00}$
$\phi_{\checkmark\checkmark}$	$P_{\checkmark\checkmark}Q_{\checkmark\checkmark}$	$P_{\checkmark\checkmark}Q_{\checkmark\checkmark}$	$P_{00}Q_{\checkmark\checkmark}\vee\ P_{\checkmark\checkmark}Q_{00}$
$\phi_{\checkmark\times}$	$P_{\checkmark 0}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{\checkmark 0}$	$P_{\checkmark 0}Q_{\checkmark\times}\vee\ P_{\checkmark\checkmark}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{\checkmark 0}\vee\ P_{\checkmark\times}Q_{\checkmark\checkmark}\vee\ P_{\checkmark\times}Q_{\checkmark\times}$	$P_{00}Q_{\checkmark\times}\vee\ P_{\checkmark\times}Q_{00}$
$\phi_{\times 0}$	$P_{\checkmark 0}Q_{\times 0}\vee\ P_{\times 0}Q_{00}$	$P_{00}Q_{\times 0}\vee\ P_{\checkmark 0}Q_{\times 0}\vee\ P_{\times 0}Q_{00}\vee\ P_{\times 0}Q_{\checkmark 0}\vee\ P_{\times 0}Q_{\times 0}$	$P_{00}Q_{\times 0}\vee\ P_{\times 0}Q_{00}$
$\phi_{\times\checkmark}$	$P_{\checkmark\checkmark}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}$	$P_{00}Q_{\times\checkmark}\vee\ P_{\checkmark\checkmark}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark\checkmark}\vee\ P_{\times\checkmark}Q_{\times\checkmark}$	$P_{00}Q_{\times\checkmark}\vee\ P_{\times\checkmark}Q_{00}$
$\phi_{\times\times}$	$P_{\checkmark 0}Q_{\times\times}\vee\ P_{\checkmark\times}Q_{\times\checkmark}\ \vee\ P_{\times\times}Q_{00}$	$P_{00}Q_{\times\times}\vee\ P_{\checkmark 0}Q_{\times\times}\vee\ P_{\checkmark\checkmark}Q_{\times\times}\vee\ P_{\checkmark\times}Q_{\times 0}\vee\ P_{\checkmark\times}Q_{\times\checkmark}\vee\ P_{\checkmark\times}Q_{\times\times}\vee\ P_{\times 0}Q_{\checkmark\times}\vee\ P_{\times 0}Q_{\times\times}\vee\ P_{\times\checkmark}Q_{\checkmark\times}\vee\ P_{\times\checkmark}Q_{\times\times}\vee\ P_{\times\times}Q_{00}\vee\ P_{\times\times}Q_{\checkmark 0}\vee\ P_{\times\times}Q_{\checkmark\checkmark}\vee\ P_{\times\times}Q_{\checkmark\times}\vee\ P_{\times\times}Q_{\times 0}\vee\ P_{\times\times}Q_{\times\checkmark}\vee\ P_{\times\times}Q_{\times\times}$	$P_{00}Q_{\times\times}\vee\ P_{\times\times}Q_{00}$
ϕ_{00}	$P_{00}Q_{00}$	$P_{00}Q_{00}$	$P_{00}Q_{00}$

`Predicate`	`P`$\div$`Q`	`P`$\rhd$`Q`	
$\phi_{\checkmark 0}$	$P_{\checkmark 0}Q_{00}$	$P_{\checkmark 0}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark 0}$	
$\phi_{\checkmark\checkmark}$	$P_{\checkmark 0}Q_{\checkmark 0}$	$P_{\checkmark\checkmark}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark\checkmark}$	
$\phi_{\checkmark\times}$	$P_{\checkmark 0}Q_{\times\checkmark}\vee\ P_{\checkmark 0}Q_{\times\times}$	$P_{\checkmark\times}Q_{00}\vee\ P_{\times\checkmark}Q_{\checkmark\times}$	
$\phi_{\times 0}$	$P_{\times 0}Q_{00}$	$P_{\times\checkmark}Q_{\times 0}$	
$\phi_{\times\checkmark}$	$P_{\times\checkmark}Q_{00}$	$P_{\times\checkmark}Q_{\times\checkmark}$	
$\phi_{\times\times}$	$P_{\times\times}Q_{00}$	$P_{\times\checkmark}Q_{\times\times}\vee\ P_{\times\times}Q_{00}$	
ϕ_{00}	$P_{00}Q_{00}$	$P_{00}Q_{00}$	

Figure 3.6: Inference rules for computing ϕ

$$\begin{aligned}
\mathsf{pred}(\mathit{SimpleOrder}, \Gamma_{so}) &= \phi_{\checkmark 0} \lor \phi_{\times\checkmark} \lor \phi_{\times\times} \quad (3.1)\\
&= \mathit{Billing}_{\checkmark 0}\mathbf{ProcessOrder}_{\checkmark 0} \lor \mathit{Billing}_{\checkmark\checkmark}\mathbf{ProcessOrder}_{\times\checkmark} \lor \\
&\quad \mathit{Billing}_{\times\checkmark}\mathbf{ProcessOrder}_{00} \lor \mathit{Billing}_{\checkmark 0}\mathbf{ProcessOrder}_{\times\times} \lor \\
&\quad \mathit{Billing}_{\checkmark\times}\mathbf{ProcessOrder}_{\times\checkmark} \lor \mathit{Billing}_{\times\times}\mathbf{ProcessOrder}_{00} \quad (3.2)\\
&= \mathit{Billing}_{\checkmark 0} \land \mathbf{ProcessOrder} \lor \mathit{Billing}_{\checkmark\checkmark} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \mathit{Billing}_{\times\checkmark} \land \neg\mathbf{ProcessOrder} \lor \mathit{Billing}_{\checkmark 0} \land \texttt{false} \lor \\
&\quad \mathit{Billing}_{\checkmark\times} \land \neg\mathbf{ProcessOrder} \lor \mathit{Billing}_{\times\times} \land \neg\mathbf{ProcessOrder} \quad (3.3)\\
&= \mathbf{Charge}_{\checkmark 0} \land \mathbf{Credit}_{00} \land \mathbf{ProcessOrder} \lor \\
&\quad \mathbf{Charge}_{\checkmark 0} \land \mathbf{Credit}_{\checkmark 0} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \mathbf{Charge}_{\times\checkmark} \land \mathbf{Credit}_{00} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad (\mathbf{Charge}_{\checkmark 0}\mathbf{Credit}_{\times\checkmark} \lor \mathbf{Charge}_{\checkmark 0}\mathbf{Credit}_{\times\times}) \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \mathbf{Charge}_{\times\times} \land \mathbf{Credit}_{00} \land \neg\mathbf{ProcessOrder} \quad (3.4)\\
&= \mathbf{Charge} \land \neg\mathbf{Credit} \land \mathbf{ProcessOrder} \lor \\
&\quad \mathbf{Charge} \land \mathbf{Credit} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \neg\mathbf{Charge} \land \neg\mathbf{Credit} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad (\mathbf{Charge} \land \texttt{false} \lor \mathbf{Charge} \land \texttt{false}) \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \texttt{false} \land \neg\mathbf{Credit} \land \neg\mathbf{ProcessOrder} \quad (3.5)\\
&= \mathbf{Charge} \land \neg\mathbf{Credit} \land \mathbf{ProcessOrder} \lor \\
&\quad \mathbf{Charge} \land \mathbf{Credit} \land \neg\mathbf{ProcessOrder} \lor \\
&\quad \neg\mathbf{Charge} \land \neg\mathbf{Credit} \land \neg\mathbf{ProcessOrder} \quad (3.6)
\end{aligned}$$

Figure 3.7: Computing $\mathsf{pred}(\mathit{SimpleOrder}, \Gamma_{so})$ for Example 4

3.5 Conversational Consistency

We now look at how to extend our formalization to model conversational consistency.

3.5.1 Web Services

Processes We describe processes using the same core language as in Section 3.3, with two changes:

- Given a set $\mathsf{Actions}$ of atomic messages and a set Peers of peers, atomic actions are of the form:

$$\begin{array}{lll} \texttt{A} & ::= & \\ & \mid & a?i \quad a \in \mathsf{Actions}, i \in \mathsf{Peers} \\ & \mid & a!i \quad a \in \mathsf{Actions}, i \in \mathsf{Peers} \end{array}$$

- To simplify the presentation, we leave out the compensation operator. A sketch of how to extend the algorithm to handle compensation is given in Section 3.6.2.1.

Such processes can express many concrete BPEL implementations; we extend the core language to additional features in the implementation.

Atomic messages are indivisible send or receive operations to peers. For an action $a \in \mathsf{Actions}$ and a peer $i \in \mathsf{Peers}$, we write $a?i$ to denote a message a received from peer i, and write $a!i$ to denote a message a sent to peer i. We write $\bowtie$ for ? or !, and $\overline{\bowtie}$ for the opposite of $\bowtie$, i.e., $\overline{\bowtie} =?$ if $\bowtie=!$ and $\overline{\bowtie} =!$ if $\bowtie=?$. We also write just a when the peer and whether it is a send or a receive is not relevant.

Conversation Automata We also need to model the conversations with each peer. A *conversation automaton* is a tuple $(Q, \Sigma, \delta, q^0, F^1, F^2)$ where Q is a finite set of states, $\Sigma = \mathsf{Actions} \times \{?, !\}$ is the alphabet, where $\mathsf{Actions}$ is the set of atomic messages and ? and ! denote message receive and message send respectively, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q^0 \in Q$ is an initial state, and $F^1 \subseteq Q$ and $F^2 \subseteq Q$ are disjoint subsets of Q called the *nochange* and *committed* states respectively. We call $F^1 \cup F^2$ the set of *consistent* states, and $Q \setminus (F^1 \cup F^2)$ the set of *inconsistent* states. We write $q \xrightarrow{a?} q'$ (respectively, $q \xrightarrow{a!} q'$) for $(q, (a, ?), q') \in \delta$ (respectively, $(q, (a, !), q') \in \delta$).

We assume that a conversation automaton is *complete*, that is, for every state $q \in Q$ and every action $a \bowtie \in \Sigma$, there is some state q' with $q \xrightarrow{a\bowtie} q'$. This can be ensured in the standard way by adding "dead" states to the automaton.

Web Services We bring together a process definition and the associated conversation automata in a *web service*. Formally, a web service $WS = (\mathsf{Actions}, P, \langle C_1, \ldots, C_k \rangle)$ consists of a process P and an indexed set of conversation automata $\langle C_1, \ldots, C_k \rangle$, one for each peer, where $\Sigma_i = \mathsf{Actions} \times \{?, !\}$ for each C_i, and each atomic action $a \bowtie i$ in P satisfies $i \in \{1, \ldots, k\}$.

Semantics We assume the same trace semantics for processes as in Section 3.3. These semantics must be extended to track the state changes of conversation automata. To do this, we use a small-step operational semantics, given in Figure 3.8. These semantics assume the structural congruence rules given in Figure 3.9. Henceforth, we identify structurally congruent processes.

The state of a web service consists of a process P and the states $s \in Q_1 \times \ldots \times Q_k$ of the conversation automata of the peers. We write s_i to denote the ith component of the state s, and write $s[i \mapsto q]$ for the state whose ith component is q and all other components are the same as s.

The (Msg) rule deals with a message send or receive to the ith peer. This changes the state of the ith conversation automaton but leaves all other states unchanged. The (Throw) rule replaces an exception with its handler, and the (Normal) rule drops an exception handler for a process if the process terminates normally (via skip).

The operational semantics defines a transition relation $\rightarrow$ on the states of a web service, which synchronizes the atomic actions (message sends and receives) of the process with the states of the peer automata. Let $\rightarrow^*$ denote the reflexive transitive closure of $\rightarrow$.

A *(complete) run* of the web service $WS = (P, \langle C_1, \ldots, C_k \rangle)$ is a sequence $\langle P_0, s^0 \rangle$, …, $\langle P_n, s^n \rangle$ such that (1) $P_0 \equiv P$ and for each $i \in \{1, \ldots, k\}$, we have $s_i^0 = q_i^0$, i.e., each conversation automaton starts in its initial state; (2) for each $i \in \{0, \ldots, n-1\}$, we have $\langle P_i, s_i \rangle \rightarrow \langle P_{i+1}, s_{i+1} \rangle$, and (3) there is no $\langle P, s \rangle$ such that $\langle P_n, s_n \rangle \rightarrow \langle P, s \rangle$.

The process at the last state of a complete run is always either skip or throw. In case it is skip, we say the run terminated normally. In case it is throw, we say the run terminated abnormally. The run is *consistent* if, for each $i \in \{1, \ldots, k\}$, we have $s_i^n \in F_i^1 \cup F_i^2$, i.e., each conversation automaton is in a consistent state at the end of the run.

3.5.2 Consistency for Web Services

Let $WS = (P, \langle C_1, \ldots, C_k \rangle)$ be a web service. Let $AP = \{\mathsf{nochange}(i), \mathsf{comm}(i) \mid i \in \{1, \ldots, k\}\}$ be a set of atomic propositions. Intuitively, the proposition $\mathsf{nochange}(i)$ is true if the conversation automaton C_i of peer i is in a `nochange` state, and the proposition $\mathsf{comm}(i)$ is true if the conversation automaton C_i of peer i is in a `committed` state.

$$\dfrac{\delta_i(s_i, a\overline{\bowtie}, q) \qquad s' = s[i \mapsto q]}{\langle a \bowtie i, s\rangle \rightarrow \langle \mathsf{skip}, s'\rangle}\ (\textsc{Msg}) \qquad \dfrac{\langle P, s\rangle \rightarrow \langle P', s'\rangle}{\langle P; Q, s\rangle \rightarrow \langle P'; Q, s'\rangle}\ (\textsc{Seq})$$

$$\dfrac{\langle P, s\rangle \rightarrow \langle P', s'\rangle}{\langle P \,\Box\, Q, s\rangle \rightarrow \langle P', s'\rangle}\ (\textsc{Choice}) \qquad \dfrac{\langle P, s\rangle \rightarrow \langle P', s'\rangle}{\langle P \parallel Q, s\rangle \rightarrow \langle P' \parallel Q, s'\rangle}\ (\textsc{Par})$$

$$\dfrac{}{\langle \mathsf{throw} \rhd Q, s\rangle \rightarrow \langle Q, s\rangle}\ (\textsc{Throw}) \qquad \dfrac{}{\langle \mathsf{skip} \rhd Q, s\rangle \rightarrow \langle \mathsf{skip}, s\rangle}\ (\textsc{Normal})$$

$$\dfrac{\langle P, s\rangle \rightarrow \langle P', s'\rangle}{\langle P \rhd Q, s\rangle \rightarrow \langle P' \rhd Q, s'\rangle}\ (\textsc{Except})$$

$$\dfrac{P \equiv P' \qquad \langle P', s\rangle \rightarrow \langle Q', s'\rangle \qquad Q' \equiv Q}{\langle P, s\rangle \rightarrow \langle Q, s'\rangle}\ (\textsc{Cong})$$

Figure 3.8: Small-step operational semantics for core process language

$$\begin{array}{ccc}
P \;\equiv\; \mathsf{skip}; P & \qquad & P \;\equiv\; P; \mathsf{skip} \\
\mathsf{throw}; P \;\equiv\; \mathsf{throw} & & \mathsf{throw} \parallel P \;\equiv\; \mathsf{throw} \\
(P_0; P_1); P_2 \;\equiv\; P_0; (P_1; P_2) & & P_0 \,\Box\, P_1 \;\equiv\; P_1 \,\Box\, P_0 \\
P \;\equiv\; P \parallel \mathsf{skip} & & P_0 \parallel P_1 \;\equiv\; P_1 \parallel P_0
\end{array}$$

$$P_0 \parallel (P_1 \parallel P_2) \;\equiv\; (P_0 \parallel P_1) \parallel P_2$$

Figure 3.9: Structural congruence axioms

A conversational consistency predicate is a Boolean formula over AP. Let $s \in Q_1 \times \ldots \times Q_k$ be a state and ψ a conversational consistency predicate. We write $s \models \psi$ if the Boolean formula ψ evaluates to true when each predicate $\mathsf{nochange}(i)$ is replaced by true if $s_i \in F_i^1$ (i.e., C_i is in a `nochange` state) and by false otherwise, and each predicate $\mathsf{comm}(i)$ is replaced by true if $s_i \in F_i^2$ (i.e., C_i is in a committed state) and by false otherwise,

A web service *WS* is *consistent* with respect to a conversational consistency predicate ψ if every complete run of *WS* is consistent and for every complete run with last state $\langle \mathsf{skip}, s \rangle$ we have that $s \models \psi$. The conversation consistency verification problem takes as input a web service *WS* and a conversational consistency predicate ψ and returns "yes" if *WS* is consistent with respect to ψ and "no" otherwise.

Theorem 13. *The conversation consistency verification problem is co-NP complete.*

In case a web service is inconsistent, there is a violating execution that is polynomial in the size of the service. This shows membership in co-NP. The problem is co-NP hard by a reduction from Boolean validity, similar to the reduction for set consistency in Theorem 9.

3.5.3 Consistency Verification

We now give an algorithm for the conversation consistency verification problem that reduces the problem to satisfiability checking. Given a web service $WS = (P, \langle C_1, \ldots, C_k \rangle)$ and a consistency specification ψ, we construct a formula φ such that φ is satisfiable iff *WS* is not consistent w.r.t. ψ. We build φ by induction on the structure of the process, in a way similar to bounded model checking. The only technical difficulty is to ensure φ is polynomial in the size of the input; clearly, a naive encoding of the complete runs of the web service yields an exponential formula.

We start with some definitions. Given a process P, define the function depth by induction as follows:

$$\begin{array}{ll}
\mathsf{depth}(\mathsf{skip}) = 1 \quad \mathsf{depth}(\mathsf{throw}) = 1 & \\
\mathsf{depth}(a?i) = 1 \quad \mathsf{depth}(a!i) = 1 & \\
\mathsf{depth}(P_1 \otimes P_2) = \mathsf{depth}(P_1) + \mathsf{depth}(P_2) & \otimes \in \{;, \|, \rhd\} \\
\mathsf{depth}(P_1 \,\Box\, P_2) = \max(\mathsf{depth}(P_1), \mathsf{depth}(P_2)) &
\end{array}$$

For a process P, the value $\mathsf{depth}(P)$ gives the maximum number of message exchanges on any run of the process P.

A process P can give rise to up to $\mathsf{depth}(P)$ transitions. We call each evaluation step of P a *step* in its run. We introduce symbolic variables that represent the state and transitions of the web service for each step. A *state variable* is a k-tuple taking values in the domain $Q_1 \times \ldots \times Q_k$. In the following, we use (superscripted) variables s^j to stand for states. Each s^j is a k-tuple taking values over $Q_1 \times \ldots \times Q_k$, and the element s^j_i takes values over the set of states Q_i of the conversation automaton C_i and represents the state of C_i in the state s^j. A *transition predicate* is a formula over two state variables s and s'. A transition predicate gives a relation between the old state s and a new state s' that encodes one or more steps of a process execution.

For each conversation automaton C_i, and each action $a \bowtie \in \Sigma$, we define a transition predicate, written $\Delta_i(a \bowtie)(s, s')$, that encodes the transition relation of C_i for symbolic states s and s'. The symbolic transition relation $\Delta_i(a \bowtie)(s, s')$ is a formula with free variables s and s' that encodes the transition relation in the obvious way:

$$\bigvee_{(q, a\bowtie, q') \in \delta_i} s = q \wedge s' = q'$$

Constructing the Consistency Predicate. For simplicity of exposition, we first give the construction of the consistency predicate for processes without exception handling, i.e., without throw and $\cdot \rhd \cdot$. We construct φ in two steps. First, given a process P, we construct by induction a sequence of $\mathsf{depth}(P)$ transition predicates that represent the transitions in possible executions of P. Second, we tie together the transitions, constrain the executions to start in the initial state, and check whether at the end, the state is consistent according to the consistency specification.

We introduce some notation. Let P be a process, let $p = \mathsf{depth}(P)$. Let $(s^1, t^1), \ldots, (s^p, t^p)$ be pairs of state variables. We shall construct a formula $\mathsf{transitions}[P](s^1, t^1, \ldots, s^p, t^p)$ that encodes the sequence of transitions in an execution of P. This formula can have additional free variables. We construct $\mathsf{transitions}[P](s^1, t^1, \ldots, s^p, t^p)$ by induction on the structure of P as follows.

If $P \equiv \mathsf{skip}$, then $\mathsf{transitions}[P](s^1, t^1)$ is $t^1 = s^1$ (that is, the state does not change on executing a skip).

If $P \equiv a \bowtie i$, the formula $\mathsf{transitions}[a \bowtie i](s^1, t^1)$ is

$$\bigwedge_{j \neq i} t^1_j = s^1_j \wedge \Delta_i(a\overline{\bowtie})(s^1_i, t^1_i)$$

Intuitively, this specifies that the ith part of the state changes according to the step of the automaton, while every other part of the state (i.e., the states of the other conversation automata) remain the same.

Let $P \equiv P_1; P_2$, and $p_1 = \mathsf{depth}(P_1)$ and $p_2 = \mathsf{depth}(P_2)$.

We construct recursively the formulas $\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1})$ and $\mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2})$, where the free variables in the two formulas are disjoint. Let $x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2}$ be a fresh sequence of state variables that we shall use to encode the transitions of P. Then $\mathsf{transitions}[P](x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2})$ is given by

$$\begin{gathered}\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1}) \wedge \\ \mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2}) \wedge \\ \textstyle\bigwedge_{j=1}^{p_1}(x^j = s^j \wedge y^j = t^j) \wedge \bigwedge_{j=1}^{p_2}(x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j)\end{gathered}$$

Intuitively, the transitions for a sequential composition of processes consist of the transitions of the first process followed by the transitions of the second process.

Let $P \equiv P_1 \square P_2$, and $p_1 = \mathsf{depth}(P_1)$ and $p_2 = \mathsf{depth}(P_2)$. Without loss of generality, assume $p_1 \geq p_2$. We construct recursively the formulas $\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1})$ and $\mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2})$, where the free variables in the two formulas are disjoint. Let $x^1, y^1, \ldots, x^{p_1}, y^{p_1}$ be a set of fresh state variables. Then, $\mathsf{transitions}[P](x^1, y^1, \ldots, x^{p_1}, y^{p_1})$ is given by

$$\begin{gathered}\left(\begin{gathered}\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1}) \wedge \\ \textstyle\bigwedge_{j=1}^{p_1}(x^j = s^j \wedge y^j = t^j)\end{gathered}\right) \\ \vee \\ \left(\begin{gathered}\mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2}) \wedge \\ \textstyle\bigwedge_{j=1}^{p_2}(x^j = u^j \wedge y^j = v^j) \wedge \bigwedge_{j=p_2+1}^{p_1-p_2} y^j = x^j\end{gathered}\right)\end{gathered}$$

Intuitively, the transitions of a choice are either the transitions of the first subprocess or the transitions of the second (hence the disjunction), and we add enough "skip" transitions to ensure that each side of the disjunction has the same number of transitions.

If $P \equiv P_1 \parallel P_2$, and $p_1 = \mathsf{depth}(P_1)$ and $p_2 = \mathsf{depth}(P_2)$. We recursively construct formulas $\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1})$ and $\mathsf{transitions}[P_2](u^1, v^1 \ldots, u^{p_2}, v^{p_2})$. We construct the formula $\mathsf{transitions}[P](x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2})$ as follows.

For $i \in \{1, \ldots, p_1 + 1\}$ and $j \in \{1, \ldots, p_2 + 1\}$, let $\alpha^{i,j}$ be fresh Boolean variables. For each $i \in \{1, \ldots, p_1\}$ and $j \in \{1, \ldots, p_2\}$, we introduce the following constraints on $\alpha^{i,j}$:

$$\alpha^{ij} \leftrightarrow \begin{pmatrix} (x^{i+j-1} = s^i \wedge y^{i+j-1} = t^i \wedge \alpha^{i+1,j}) \\ \vee \\ (x^{i+j-1} = u^j \wedge y^{i+j-1} = v^j \wedge \alpha^{i,j+1}) \end{pmatrix}$$

For $j \in \{1, \ldots, p_2\}$, we introduce the constraint

$$\alpha^{p_1+1,j} \leftrightarrow (x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j \wedge \alpha^{p_1+1,j+1})$$

For $i \in \{1, \ldots, p_1\}$, we introduce the constraint

$$\alpha^{i,p_2+1} \leftrightarrow (x^{p_2+i} = s^i \wedge y^{p_2+i} = t^i \wedge \alpha^{i+1,p_2+1}$$

Finally, we set

$$\alpha^{p_1+1,p_2+1} = true$$

Then, $\mathsf{transitions}[P](x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2})$ is the conjunction of the constraints above for all i, j, together with the constraint $\alpha^{1,1}$.

Intuitively, the construction above takes two sequences of transitions, and encodes all possible interleavings of these sequences. A naive encoding of all interleavings leads to an exponential formula. Therefore, we use dynamic programming to memoize sub-executions. The variable $\alpha^{i,j}$ indicates we have executed the first $i-1$ transitions from the first sequence and the first $j-1$ transitions from the second sequence. Then the constraint on $\alpha^{i,j}$ states that we can either execute the ith transition of the first sequence and go to state $\alpha^{i+1,j}$, or execute the jth transition of the second and go to state $\alpha^{i,j+1}$. Conjoining all these constrains encodes all possible interleavings.

Finally, we construct φ as follows. Let P be a process and $\mathsf{depth}(P) = p$. Let $\mathit{Init}(s)$ be the predicate

$$\bigwedge_{i=1}^{k} s_i = q_i^0$$

that states that each automaton is in its initial state, and $\mathit{Consistent}(s)$ be the predicate

$$\bigwedge_{i=1}^{k} s_i \in F_i^1 \cup F_i^2$$

stating that each automaton is in a consistent state (note that we can expand each set-inclusion into a finite disjunction over states).

Given the formula $\mathsf{transitions}[P](s^1, t^1, \ldots, s^p, t^p)$, we construct φ by "stitching together" the transitions, conjoining the initial and consistency conditions:

$$\begin{gathered}\mathit{Init}(s^1)\wedge \\ \mathsf{transitions}[P](s^1, t^1, \ldots, s^p, t^p) \wedge \textstyle\bigwedge_{i=1}^{p-1} t^i = s^{i+1} \\ \wedge(\neg \mathit{Consistent}(t^p) \vee \neg\psi(s^p))\end{gathered}$$

Consistency Predicate with Exception Handling. We now extend the above construction with throw and exception handling. In this case, we keep an additional Boolean variable in the state that indicates whether an error has occurred. Initially, this variable is set to 0. The variable is set to 1 by a throw

statement, and reset to 0 by the corresponding exception handler. Additionally, we modify the transitions function $\mathsf{transitions}$ to maintain the state once an error has occurred in order to simulate the propagation of exceptions.

Formally, a *state variable* is now a $(k+1)$-tuple, where the 0th element is a Boolean variable indicating if an error has occurred and the 1st to the kth elements are states of the k peer automata as before. We extend the constraints to additionally track the error status. First, $\mathsf{transitions}[\mathsf{throw}](s,t)$ below sets the error status:

$$t_0 = 1 \wedge \bigwedge_{j=1}^{k} t_j = s_j$$

Second, let P be the process $P_1 \rhd P_2$ with $\mathsf{depth}(P_1) = p_1$ and $\mathsf{depth}(P_2) = p_2$. Let $x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2}$ be new state variables. Then $\mathsf{transitions}[P](x^1, y^1, \ldots, x^{p_1+p_2}, y^{p_1+p_2})$ is given by

$$\begin{gathered}
\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1}) \wedge \textstyle\bigwedge_{j=1}^{p_1}(x^j = s^j \wedge y^j = t^j) \wedge \\
\mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2}) \wedge \\
(y_0^{p_1} = 1 \rightarrow \begin{pmatrix} x_0^{p_1+1} = 0 \wedge \bigwedge_{i=1}^{k} x_i^{p_1+1} = u_i^1 \wedge \\ \bigwedge_{j=2}^{p_2}(x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j) \end{pmatrix}) \\
\wedge \\
(y_0^{p_1} = 0 \rightarrow \textstyle\bigwedge_{j=1}^{p_2}(y^{p_1+j} = x^{p_1+j}))
\end{gathered}$$

That is, the first p_1 steps of $P_1 \rhd P_2$ coincides with the steps of P_1 (line 1), and the next p_2 steps are the steps of P_2 if the error bit is set to 1 at the end of executing P_1 (line 3), in which case the error bit is reset to 0, or identity steps (line 4) if the error bit is not set.

The constructions for sequential and parallel composition and choice are modified to stop executing once an error has been thrown. For example, the constraints for sequential composition are

$$\begin{gathered}
\mathsf{transitions}[P_1](s^1, t^1, \ldots, s^{p_1}, t^{p_1}) \wedge \\
\mathsf{transitions}[P_2](u^1, v^1, \ldots, u^{p_2}, v^{p_2}) \wedge \\
\textstyle\bigwedge_{j=1}^{p_1}(x^j = s^j \wedge y^j = t^j) \wedge \\
(y_0^{p_1} = 0 \rightarrow \textstyle\bigwedge_{j=1}^{p_2}(x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j) \wedge \\
(y_0^{p_1} = 1 \rightarrow \textstyle\bigwedge_{j=1}^{p_2}(x^{p_1+j} = y^{p_1+j}))
\end{gathered}$$

We omit the similar modifications for $\Box$ and $\|$.

Finally, the initial condition $\mathit{Init}(s)$ has the additional conjunct $s_0 = 0$ and the predicate $\mathit{Consistent}(s)$ has the additional conjunct $s_0 = 0$.

Soundness and Completeness. The following theorem states that our construction is sound and complete. It is easily proved by induction on the run of the web service.

Theorem 14. *For any web service* $\mathit{WS} = (P, \langle C_1, \ldots, C_k \rangle)$ *and consistency specification* ψ*, we have* φ *is satisfiable iff* WS *does not satisfy* ψ*. Further,* φ *is polynomial in the size of* WS*.*

Further, while we have used equality in our description of φ, since each variable varies over a finite domain, we can compile φ into a purely propositional formula.

Corollary 3. *The consistency verification problem is in co-NP.*

Relationship to Assume-Guarantee. A *concrete web service* is a collection $W = \langle P_1, P_2, \ldots, P_k \rangle$ of processes, whose semantics is given by the obvious modifications of the rules in Figure 3.8 (see e.g., [CRR02]). *Stuck-freedom* [RR02] formalizes the notion that a concrete web service does not deadlock waiting for messages that are never sent or send messages that are never received.

An assume-guarantee principle would state that the local consistency checks using our algorithm would imply that the concrete web service is stuck-free. Assume-guarantee principles make strong non-blocking assumptions about processes. Since our processes may not be non-blocking, we do not have an assume-guarantee principle. Consider the three processes: process P_1 is $a?2; a!3$, process P_2 is $a?3; a!1$, and process P_3 is $a?1; a!2$. These processes are in deadlock. However, consider a conversation automaton $A = (\{q_1, q_2\}, \{a?, a!\}, \delta, q_1, \{q_1, q_2\}, \emptyset)$ with $\delta(q_1, a!, q_2)$ and $\delta(q_2, a?, q_1)$. It is clear that (P_i, A) is consistent for each $i \in \{1, 2, 3\}$. Thus, our local consistency checks may not imply stuck-freedom.

Process	Spec	Proc size	Spec size	Pred size	Time (ms)
AccountReceive	φ_{q1}	13	8	124	70
OrderProcess	φ_{o1}	26	21	247	112
OrderProcess	φ_{o2}	26	20	236	130
BrokenOrder	φ_{o2}	22	20	201	90
Travel	φ_{t1}	13	24	181	90
Travel	φ_{t2}	13	39	210	86

Table 3.1: Experimental Results for set consistency verifier. "Spec" is the consistency specification, "Spec size" is the size of the specification, "Pred size" is the size of the execution predicate.

3.6 Experiences

To demonstrate the viability of this approach, we have implemented verifiers for set and conversational consistency, based on the core process language of Section 3.3. I then developed BPELCheck, a front-end to the conversational consistency verifier that translates from the BPEL process language to our core language. I have modeled several case studies, in either the core process language (for the set consistency verifier), or BPEL (for conversational consistency) and verified them using these tools.

3.6.1 Set Consistency Verifier

I implemented the set consistency verification algorithm of Section 3.4 in Objective Caml. To determine the validity of verification predicates, I use the MiniSat satisfiability solver [ES03]. If an error is found, the solver returns a satisfying assignment to the verification predicate. In the even that an error is found, a counter-example analysis then maps the boolean variable assignment back to steps in the process and generates a trace of a violating execution.

3.6.2 Conversational Consistency Verifier

BPELCheck, which verifies BPEL processes against conversational consistency specifications, is implemented as a plug-in for the Sun NetBeans Enterprise Service Pack. The implementation has three parts: a Java front-end converts the NetBeans BPEL internal representation into a process in our core language and also reads in the conversation automata for the peers, an OCaml library compiles the consistency verification problem into a satisfiability instance, and finally, the decision procedure Yices [DM06] is used to check for conformance, and in case conformance fails, to produce a counterexample trace.

3.6.2.1 Extensions to the Core Language

In addition to our core language, BPEL has a set of other source-level constructs such as compensations, synchronizing flows, and (serial and parallel) loops. I now show how we can extend the consistency verification algorithm to handle these constructs. I sketch the intuition, but do not give the formal details.

Variables BPEL allows processes to have local variables. Our core language can be augmented with state variables and *assignment* and *assume* constructs. An assignment stores a computed value in a variable, and an assume statement continues execution iff the condition in the assume is true. The operational semantics of processes is extended in the standard way [Mit96] to carry a *variable store* mapping variables to values. Assignments update the store, and assume conditions continue execution if the condition evaluates to true under the current store. Similarly, we can augment the construction of the consistency predicate to track variable values in the state variables, and model their updates in the transitions. We assume a finite-width implementation of integer data, which allows all constraints on integer variables to be reduced to Boolean reasoning.

Links BPEL allows processes executing in parallel to synchronize through *links* between concurrently executing flows. A link between process P_1 and process P_2 means that P_2 can start executing only after P_1 has finished. We model links using Boolean variables for each link that are initially $false$, then set to $true$ when the source activity terminates. We guard each target process with the requirement that the link variable is $true$.

Loops BPEL processes can also have sequential or parallel loops. While in general the presence of loops makes the theoretical complexity of the problem go up from co-NP complete to PSPACE-complete, we have found that in practice most loops can be unrolled a fixed finite number of times. For example, once the peers are fixed, all loops over sets of peers can be unrolled. We plan to add support for loops to our tool in the near future, using simple syntactic heuristics to unroll loops a minimal number of times.

Compensations BPEL allows processes to declare *compensation handlers* that get installed when a process completes successfully and get run when a downstream process fails. Our core language can be extended with compensations by adding the syntax $P_1 \div P_2$ to denote a process P_1 compensated by P_2. The semantics is modified by carrying a *compensation stack*: the semantics of $P_1 \div P_2$ runs P_1, and if it terminates normally, pushes the process P_2 on the compensation stack. If a process P terminates abnormally, the processes on the compensation stack are executed in last-in first-out order [BF00, BF04, BMM05].

We can extend our construction of the consistency predicate by tracking compensations. With each process, we now associate both a depth (the number of steps in a normal evaluation) and a *compensation depth*, the number of compensation steps if the process terminates abnormally. The sum of the depth and the compensation depth is polynomial in the size of the process, thus the consistency predicate remains polynomial in the size of the process. The construction of the consistency predicate becomes more complicated as there is a forward process that encodes the transitions of the forward (normal) execution as well as constructs the compensation stack, and a compensation execution that executes the compensation actions on abnormal termination. We have not yet implemented compensations in our tool.

3.6.2.2 Optimizations

We also implement several optimizations of the basic algorithm from Section 3.5. The main optimization is a partial-order reduction that only considers a particular interleaving for the parallel composition of processes. For a process P, let $\mathsf{Peers}(P)$ be the set of peers which exchange messages with P. Formally, $\mathsf{Peers}(P)$ is defined by induction:

$$\mathsf{Peers}(\mathsf{skip}) = \emptyset \quad \mathsf{Peers}(\mathsf{throw}) = \emptyset \quad \mathsf{Peers}(m \bowtie i) = \{i\}$$
$$\mathsf{Peers}(P_1 \otimes P_2) = \mathsf{Peers}(P_1) \cup \mathsf{Peers}(P_2) \quad \otimes \in \{\Box, ;, \|, \rhd\}$$

Define two processes P_1 and P_2 to be *independent* if (1) neither process has a throw subprocess that escapes the scope of the process, and (2) $\mathsf{Peers}(P_1) \cap \mathsf{Peers}(P_2) = \emptyset$. Independent processes talk to disjoint peers, so if they run in parallel, the state of the conversation automata is the same for any interleaving. Thus, for independent processes, instead of constructing the interleaved formula for parallel composition, we construct the sequential composition of the two processes. This reduces the number of case splits in the satisfiability solver. In our expe-

$$\begin{aligned}
OrderProcess =& \mathbf{SaveOrder}; CreditCheck;\\
& \mathbf{SplitOrder}; FulfillOrder; CompleteOrder\\
CreditCheck =& (\mathbf{ReserveCredit} \rhd (\mathbf{Failed}; \mathsf{throw}))\\
& \div OrderFailed\\
OrderFailed =& \mathbf{RestoreCredit}; \mathbf{Failed}\\
FulfillOrder =& * \mid (ProcessPO_i)\\
ProcessPO_i =& (\mathbf{FulfillPO}_i \rhd (\mathbf{MarkPOFailed}_i; \mathsf{throw}))\\
& \div \mathbf{CancelPO_i}\\
CompleteOrder =& \mathbf{BillCustomer}; \mathbf{Complete}
\end{aligned}$$

Figure 3.10: Order management process

rience, most parallel composition of processes in BPEL web services satisfy the independence property.

3.7 Case Studies

Next, we examine the case studies I performed using the two verifiers.

3.7.1 Set Consistency Case Studies

3.7.1.1 Order Process

Butler *et al* [BHF04] presents a simple order fulfillment transaction which has cancellation semantics, and thus compensation is sufficient for expressing the required error handling. Figure 3.10 shows a more complex order fulfillment transaction, inspired by the example application in [SBM04], which does not have cancellation semantics. The process *OrderProcess* receives a customer's order and makes a reservation against the customer's credit (**ReserveCredit**). If the customer does not have enough credit, the order is marked *failed* (**Failed**) and processing stopped. Otherwise, if the credit check was successful, the subprocess *OrderFailed* is installed as compensation for the credit check. Then, the order is broken down into sub-orders by supplier, and these sub-orders are submitted to their respective suppliers in parallel (subprocess $FulfillOrder$). If all sub-orders complete successfully, the subprocess *CompleteOrder* finalizes the credit transaction with a call to **BillCustomer** and marks the order as *complete* (**Complete**).

Failure semantics There are two types of errors that may occur in the order transaction. If the *ReserveCredit* call fails (e.g., due to insufficient credit), the order is marked as failed and execution is terminated before submitting any purchase orders. Alternatively, one or more purchase orders may be rejected by the associated suppliers. If any orders fail, the credit reservation is undone and the order marked as failed. Note that neither error scenario causes the entire transaction to be undone. This is consistent with real world business applications, where many transactions have some notion of partial success and records of even failed transactions are retained. We assume the normalization set

$$\{(\mathbf{ReserveCredit}, \mathbf{RestoreCredit}), (\mathbf{FulfillPO}, \mathbf{CancelPO})\}$$

and that the actions **SaveOrder**, **RestoreCredit**, **CancelPO**, **Failed**, and **Complete** never fail.

The order should always be saved. If the order process is successful, the customer should be billed, all the purchase orders fulfilled, and the order marked complete. If the order process fails, the order should be marked as failed, the customer should not be billed, and no purchase orders fulfilled. These requirements are written as the following set consistency predicate φ_{o1}:

$$\begin{aligned}&\textbf{SaveOrder} \wedge \\ &\big((\textbf{BillCustomer} \wedge \textbf{FulfillPO} \wedge \textbf{Complete} \wedge \neg\textbf{Failed}) \vee \\ &(\neg\textbf{BillCustomer} \wedge \neg\textbf{FulfillPO} \wedge \neg\textbf{Complete} \wedge \textbf{Failed})\big)\end{aligned}$$

When checked with the verifier, $OrderProcess$ does indeed satisfy this specification.

Next, consider an alternative, orthogonal specification. Assume that the **ReserveCredit**, **RestoreCredit**, and **BillCustomer** actions all belong to an external credit service. We wish to ensure that our process always leaves the service in a consistent state: if **ReserveCredit** succeeds, then either **RestoreCredit** or **BillCustomer** (but not both) must eventually be called. Also, if **ReserveCredit** fails, neither should be called. We model these requirements with the predicate φ_{o2}:

$$\begin{aligned}&(\neg\textbf{ReserveCredit} \rightarrow (\neg\textbf{RestoreCredit} \wedge \neg\textbf{BillCustomer})) \wedge \\ &(\textbf{ReserveCredit} \rightarrow (\textbf{RestoreCredit} \oplus \textbf{BillCustomer}))\end{aligned}$$

where $\rightarrow$ and $\oplus$ are syntactic sugar for logical implication and logical exclusive-or, respectively. Since we are referencing **RestoreCredit** directly in our specification, we remove the pair $(\textbf{ReserveCredit}, \textbf{RestoreCredit})$ from our compensation set. Our verifier finds that $OrderProcess$ satisfies this specification.

Finally, we consider the process $BrokenOrder$, a variation of $OrderProcess$ where the $OrderFailed$ compensation is left out of the $CreditCheck$ subprocess:

$$CreditCheck = \textbf{ReserveCredit} \rhd (\textbf{Failed}; \mathsf{throw})$$

When checking this process, the verifier finds that the φ_{o2} specification is not satisfied and returns the following counter-example execution:

$$\{\textbf{SaveOrder}, \textbf{ReserveCredit}, \textbf{SplitOrder}, \textbf{Failed}\}$$

This execution corresponds to a trace where the process runs successfully until it reaches **FulfillPO**, which fails. The exception handling for **FulfillPO** runs **Failed**, but **RestoreCredit** is never run to undo the effects of **ReserveCredit**.

3.7.1.2 Travel Agency

Many real world applications involve *mixed transactions*. A mixed transaction combines both compensatable and non-compensatable subtransactions [ELL90]. Frequently, these processes involve a *pivot action* [SAB02], which cannot be compensated or retried. To obtain cancellation semantics, actions committing before the pivot must support compensation (backward recovery) and actions committing after the pivot must either never fail or support retry (forward recovery).

Set consistency specifications can capture these requirements and our verifier can check these properties. To illustrate this, I use a travel agency example from [HA00]. It can be modeled in the core language as follows:

$$\begin{aligned} Travel = ((&\textbf{BookFlight} \div \textbf{CancelFlight};\\ &(\textbf{RentCar} \div \textbf{CancelCar})) \rhd \textbf{ReserveTrain});\\ &(\textbf{ReserveHotel1} \rhd \textbf{ReserveHotel2}) \end{aligned}$$

In this transaction, a travel agent wishes to book transportation and a hotel room for a customer. The customer prefers to travel by plane and rental car. These reservations can be canceled. If a flight or rental car is not available, then the agent will book a train ticket to the destination. Once made, the train reservation cannot be canceled. There are two hotel choices at the destination. The first hotel may be full and a reservation may fail, but the second hotel reservation is always successful. We can model the failure and compensation properties of these services as follows:

$$\begin{aligned}\Gamma_t = \langle & \mathbf{BookFlight} \mapsto \{\checkmark, \times\}, \mathbf{CancelFlight} \mapsto \{\checkmark\}, \\ & \mathbf{RentCar} \mapsto \{\checkmark, \times\}, \mathbf{CancelCar} \mapsto \{\checkmark\}, \\ & \mathbf{ReserveTrain} \mapsto \{\checkmark, \times\}, \\ & \mathbf{ReserveHotel1} \mapsto \{\checkmark, \times\}, \mathbf{ReserveHotel2} \mapsto \{\checkmark\}\rangle \\ \mathcal{C} = \{&(\mathbf{BookFlight}, \mathbf{CancelFlight}), (\mathbf{RentCar}, \mathbf{CancelCar})\}\end{aligned}$$

The **ReserveTrain** action is a pivot action, as it has no compensation or exception handler. From inspection, we see that the requirements for cancellation semantics are met:

- If **ReserveTrain** is called, then the actions **BookFlight** and **RentCar** have been compensated by **CancelFlight** and **CancelCar**, respectively.
- If **ReserveHotel1** fails, we recover forward by calling the alternate action **ReserveHotel2**, which cannot fail.

We can check this with the verifier using the following specification predicate:

$$\begin{aligned}\varphi_{t1} = (((&\mathbf{BookFlight} \wedge \mathbf{RentCar}) \vee \mathbf{ReserveTrain}) \wedge \\ &(\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})) \vee \\ \neg(&\mathbf{BookFlight} \vee \mathbf{RentCar} \vee \mathbf{ReserveTrain} \vee \\ &\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})\end{aligned}$$

The process does indeed satisfy this specification. We can use consistency specifications to check stronger properties as well. For example, we can alter the specification predicate to check that the process does not book both the flight/car and the train and that it only books one hotel:

$$\begin{aligned}\varphi_{t2} = (((&\mathbf{BookFlight} \wedge \mathbf{RentCar}) \oplus \mathbf{ReserveTrain})\wedge \\ &(\mathbf{ReserveHotel1} \oplus \mathbf{ReserveHotel2})) \vee \\ &\neg(\mathbf{BookFlight} \vee \mathbf{RentCar} \vee \mathbf{ReserveTrain}\vee \\ &\mathbf{ReserveHotel1} \vee \mathbf{ReserveHotel2})\end{aligned}$$

3.7.1.3 Performance Results

Table 3.1 shows the results of running this verifier on the example of Section 3.2.1 and the two case studies above. The run times are for a 1.6Ghz Pentium M laptop with 512 MB of memory, running Windows XP. The runs all complete in less than 130 milliseconds. The verification predicates are approximately 10 times larger than the original processes. Since process languages are intended for composing lower-level services, the size of real-world processes are usually quite small (in my industrial experience, not more than an order of magnitude larger than these examples). This is well within the capabilities of current SAT solvers.

3.7.2 Conversational Consistency Case Studies

To evaluate BPELCHECK, we ran it on several example BPEL processes. *Store* implements the store process example of Figure 3.3. We also implemented the two error scenarios described at the end of Section 3.2.2.

Travel is the same travel agency example described above in Section 3.7.1.2.

Process	Result	Proc size	States	Spec size	Time (ms)
Store 1	pass	31	18	14	408
Store 2	fail	31	18	14	339
Store 3	fail	31	18	14	384
Travel	pass	38	12	22	375
Auction	pass	15	9	11	245
ValidateOrder	fail	51	17	1	448

Table 3.2: Experimental Results for BPELCheck. "Result" is the result returned by BPELCheck, "States" is the total number of states across all peer automata, and "Spec size" is the size of the consistency predicate.

In this version, calls to make reservations for transportation and a hotel become message exchanges with three peer processes. The consistency predicate for the conversational version of this process checks that either all conversations were left in a `nochange` state or the hotel reservation succeeded along with one of the two transport reservations.

Auction is from the BPEL specification [BPE03]. The process waits for two incoming messages — one from a buyer and one from a seller. It then makes an asynchronous request to a registration service and waits for a response. Upon receiving a response, it sends replies to the buyer and seller services. Each interaction in this process is a simple request-reply pair. Our specification checks that every request has a matching reply.

ValidateOrder is an industrial example provided by Sun Microsystems. It accepts an order, performs several validations on the order, updates external systems, and sends a reply. If an error occurs, a message is sent to a JMS queue. Using BPELCheck, we found an error in this process. In BPEL, each *request* activity should have a matching *reply* activity, enabling the process to implement a synchronous web service call. However, the *ValidateOrder* process does not send a reply in the event that an exception occurs.

3.7.2.1 Performance Results

Table 3.2 lists the results of running these examples through BPELCheck. In each case, we obtained the expected result. We measure the size of a process as the number of atomic actions plus the number of composition operators, when translated to our core language. We compute the size of consistency predicates by summing the number of atomic predicates and boolean connectives. These experiments were run on a 2.16 Ghz Intel Core 2 Duo laptop with 2 GB of memory using MacOS 10.5. The running times were all less than a second, validating that this approach works well in practice. In general, the running times were roughly linear with respect to input size.

3.7.3 Comparing set and conversational consistency

A comparison of the set and conversational consistency case studies demonstrates the trade-offs between these two approaches. For processes that make independent synchronous calls or where there is a one-to-one mapping between forward and compensation actions (e.g. the CRM, travel, and order examples of Sections 3.2.1 and 3.7.1), set consistency is more appropriate. A conversational consistency specification for such processes will be larger but not any more precise. On the other hand, it is awkward to describe more complex compensation protocols (e.g. the two-phase commit used in warehouse example of Section 3.2.2) using set consistency. In addition, set consistency cannot capture the implied relationship between asynchronous request and reply messages in BPEL processes. For example, in the $ValidateOrder$ example of Section 3.7.2, the entire process appears as a synchronous call to the peer which initiates the process via a request. The relationship between the receive call which spawns the process its associated response can be succinctly captured using a conversation automaton. Finally, if conversation automata are included in a web service's definition (e.g. as done with the Web Services Conversation Language [BBB02]), then these automata can be reused by all processes calling a web service, reducing the effective size of a process's specification.

Since any set consistency specification can be encoded in a conversational consistency specification (by treating each action as a separate peer), it makes sense to build BPELCheck on conversational consistency. As a future enhancement, we plan to add better support for creating default automata, based on the relationships between activities encoded in BPEL process definitions. This will allow users to avoid the need for conversation specifications except where relevant to the properties they want to guarantee.

3.8 Related Work

Flow Composition Languages. Many formalizations of flow composition languages that support composition and compensation have been proposed in the literature [BF00, BF04, BMM05]. These formalisms such as StAC [BF00], the saga calculus [BMM05], and Compensating CSP [BHF04] formalize process orchestration using atomic actions and composition operations similar to ours. They differ mostly in the features supported (e.g., whether recursion is allowed, whether parallel processes can be synchronized, or whether there are explicit commit mechanisms), in assumptions on atomic actions (whether or not they always succeed), and in the style of the semantics (trace based or operational).

I chose a core language that includes only the features relevant to my exposition. However, I borrowed extensively from the above languages and believe that similar results hold in the other settings. Like the Saga Calculus [BMM05], I assume that atomic actions can succeed or fail, as this more closely matches the semantics of industrial languages such as BPEL [BPE03]. I support all the composition operators of the Saga Calculus and Compensating CSP [BHF04], except that I automatically apply compensation in the event of an error, rather than requiring a transaction block. Our sequential and parallel iterations are inspired by StAC's *generalized parallel* operator. However, my core language does not support interruptible recovery of parallel processes or recursion.

Notions of Correctness Sagas [GS87] model a long-running, two-level transaction, where the first level consists of atomic actions and the second level provides atomicity, but not isolation, through compensation. The usual notion of correctness is *cancellation semantics* [BHF04]. One can ensure that a process is self-canceling by restricting processes to have a compensation action for each forward action where compensations cannot fail and are independent of any other compensations running in a parallel branch [BHF04]. Although order-independence between compensations is a realistic restriction, requiring a compensation for each action seems limiting. Verification becomes more involved when this restriction is relaxed. For example, [HA00] describes an algorithm which checks that an OPERA workflow, potentially containing non-compensatable actions and exception handling, satisfies cancellation in $O(n^2)$ time. In [YCC06], cancellation is checked on collections of interacting processes by creating an atomicity-equivalent abstraction of each process and checking the product of the abstractions.

Set consistency specifications can capture cancellation semantics. In addition, such specifications can model scenarios where self-cancellation is not desired (e.g., the order case study of Section 3.7.1.1) and can capture stronger requirements than cancellation (e.g., mutually exclusive actions in the travel agency case study of Section 3.7.1.2).

Other specification approaches have been suggested for composing web services, independent of compensation and transactional issues. For example, [BCH05] proposes *consistency interfaces*, which define, for a method m of the service and result o of calling that method, the methods called by m and their associated results which lead the result o. The specification language for method calls includes union and intersection, thus providing similar capabilities as a set consistency specification. Consistency interfaces do not treat non-execution of an action the same as atomic failure, and there is no compensation. This precludes the use of negation in specifications and the interpretation of satisfying assignments as executions of the process. Our algorithm can be applied to check processes against consistency interfaces.

Finally, temporal logic specifications, frequently used by model checking tools, can also be used for compensating processes. While temporal logic is a more powerful specification language, set consistency can already model many properties of interest, and provides a more compact representation.

The problem of checking recursive processes against regular sets of traces is undecidable [EM07]. Note that any program with compensation and potentially infinite traces (e.g., due to loops or recursion) can have an infinite state space, even when no program variables are modeled. Thus, model checkers usually bound recursion depth and loop iterations (e.g., XTL for StAC [JLB03]). This is less of an issue for set consistency verification, since the number of loop iterations can be abstracted without sacrificing soundness. In addition, as discussed in the introduction, the complexity bound for the verification of finite systems is lower for set consistency (co-NP complete vs. PSPACE complete).

Global Process Interactions Specifications may also focus on the messages exchanged between processes. Such specifications can be global, tracking messages from the perspective of an omniscient viewer of the entire composition, or local, specifying only the requirements for a single process. *Conversation specifications* [BFH03, FBS04a, FBS05] are an important example of the former case: such specifications consist of an automata representing the sequence of message sends across all the processes in a composition. Each peer in a composition can be checked to see whether it conforms to its role in the global conversation. In addition, temporal logic specifications can be checked against the conversation.

Web Services Analysis Tool (WSAT) [FBS04b] checks BPEL processes against conversation specifications and was the initial inspiration for our work. WSAT compiles processes into automata, which are then model checked using SPIN. Our approach is local: we avoid building the product space using conversation automata. This is also a practical requirement as the source code for all peer processes may not be available. Finally, instead of an enumerative model checker, our analysis reduces the reachability analysis to SAT solving. In addition to conformance checks, WSAT can perform checks for *synchronizability* (the conversation holds for asynchronous as well as synchronous messaging) and *realizability* (a conversation which fully satisfies the specification can actually be implemented. The BPEL standard uses WSDL to describe the interfaces of each service, which assumes a synchronous transport (HTTP). However, vendor-specific extensions permit the use of asynchronous transport layers, such as JMS. Thus, this is a relevant issue for our tool. We hope to define compositional algorithms for determining synchronizability and realizability which can be used by BPELCheck.

Message Sequence Charts, although less expressive than automata, can also be used to specify global process interactions. In [FUM03], process interactions are specified using Message Sequence Charts and individual processes specified using BPEL. Both definitions are translated to automata (the BPEL translation obviously over-approximating the actual process's behavior) and checked against each other.

Global specifications make it easier to check for non-local properties such as deadlock and dependencies across multiple processes (e.g. process A sends a request to process B, which forwards it to C, which sends a response back to A). However, such specifications limit the ability to abstract the details of a given process, particularly the interactions a service has with peers to complete a request. For example, in the store scenario of Section 3.2.2, each warehouse may need to coordinate with other services in order to ensure the delivery of a requested product. However, this may not be of interest to the store and bank, and ideally should be left out when verifying their conversations. In addition, many services may be implemented locally and not have external dependencies. In this situation, a global specification is overkill and reduces the re-usability of a service specification.

Local Process Interactions Message exchanges can also be specified with respect to a single process. On the practical side, Web Services Conversation Language (WSCL) [BBB02] describes web services as a collection of *conversations*, where a conversation is an automata over *interactions*. Interactions represent message exchanges with a peer and can be either *one-way*, *send-receive*, or *receive-send*. This language is very similar to our conservation automata, and we could conceivably use WSCL instead of our chosen automata format (with the addition of nochange/commit labels for the states).

On the more theoretical side, the interactions of a web service can be specified using *session types* [THK94, HVK98], which provide a core language for describing a sequence of interactions over a channel and a type system for checking compatibility between the two sides of a conversation. The interaction language includes the passing of values and channel names, which are not currently modeled by our conversation automata. Compositional proof principles have been studied for message passing programs modeled in the π-calculus and interactions coded in CCS [CRR02, RR02]. They introduce a conformance notion that ensures services are not stuck.

Web service contracts [CCL06, CGP08] specify the allowable iterations of a web service using a subset of CSS (Calculus of Communicating Systems). These contracts have a flexible subtyping relation for determining when a service satisfying a given contract can be used where a different contract is specified.

3.9 Recap

In this chapter, we looked into specifications for the consistency of web services, particularly those implemented using flow composition languages. Reasoning about the changes made by a long-running transaction can be difficult, particularly when considering asynchronous calls, non-deterministic interactions with remote services, parallelism, and exceptions. I have introduced two new abstractions to address these issues: *executions* which are sets of actions that, at the end of a transaction, have completed successfully and were not undone through compensation, and *conversations*, which represent the exchange of messages between two peer services. From these abstractions, I obtained two specification languages for web transactions: *set consistency* and *conversational consistency*. Such specifications can capture key requirements for a service's implementation while being more compact (and probably easier to understand) than similar requirements expressed in temporal logics. I then created verification algorithms for each approach, which reduce the verification problem to Boolean satisfiability. Finally, I implemented BPELCheck, a tool which verifies conversational consistency specifications for BPEL processes.

CHAPTER 4

Access policy interoperability

4.1 Overview

Each application in a service-based architecture may implement its own security framework. This usually includes *authentication*, where a user's identity claim is verified and *authorization*, which determines whether a user can call an individual service. Applications may access a common service to perform authentication (*single sign-on*) or use *identity federation* to correlate users across systems.

Relating authorization rules across systems, which I will call the *access policy integration problem*, is more difficult. In particular, two problems may occur:

- In the processing of a request, a service may call other services. Even though the user had access rights to the original service, they might not have access to the called services, leading to *indirect authorization errors*. Such errors are difficult to prevent through testing, since access rights are usually assigned uniquely to each user.
- When confidential data is passed between services, the receiver could potentially disclose that data to users and services which do not have the necessary access rights in the source system. This may violate the intent of the source system's security policy.

In addressing these issues, I will focus on systems that use *Role Based Ac-*

cess Control (RBAC) [FK92], is a frequently-used access control mechanism for enterprise systems [FK92]. In RBAC, *roles* represent functions within a given organization; authorizations for resource access are granted to roles rather than to individual users. Authorizations granted to a role are strictly related to the data needed by a user in order to exercise the functions of the role. Users "play" the roles, acquiring the privileges associated with the roles, and administrators grant or revoke role memberships. RBAC simplifies access control administration: if a user moves to a new function within the organization, the appropriate role memberships are granted or revoked, rather than access permissions on resources. In other words, RBAC is an abstraction over the underlying permission system.

While RBAC provides an elegant and scalable access control mechanism for a single system, organizations frequently deploy many interacting applications, each with its own RBAC policy, thus introducing the access policy integration problem.

4.1.1 Existing solutions

Current industrial practice is to independently manage the access control policies of each application. Indirect authorization errors may be resolved only when encountered in production, and perhaps only for those users with enough visibility to demand the attention of the IT department. Attempts to pro-actively address such issues may lead to violations of least privilege, when administrators attempt to fix issues by indiscriminately granting access rights. Without any systematic analysis of the information flows between an organization's systems, it is unlikely that inter-application disclosure issues are addressed at all.

Recently, tools have emerged to support a global role schema which is centrally managed and enforced across an organization.[1] However, it can be difficult, or impossible, to centralize role management due to technical issues (legacy applications may not be amenable to external access controls) and organizational issues (each application may be locally administered by different groups). The enforcement of a centralized policy does not preclude the use of the original application-level access enforcement mechanisms (in fact, it may be impossible to remove an application's original access infrastructure). Thus, local policies must be kept synchronized with the global policy, to avoid indirect authorization errors. Finally, external enforcement approaches are necessarily implemented at a coarse-grained level (e.g. for entire web pages) and might not capture all the places where an application discloses sensitive data (e.g., data included in other pages or disclosed through integrations).

4.1.2 Global Schema Inference

To address these issues, I have developed an algorithm that computes, if possible, a global RBAC policy from the RBAC policies of the different applications. I call this process *global schema inference.* A *global schema* is a system-wide notion of roles, and a mapping from local roles to (possibly multiple) global roles, such that, with these global assignments of roles to users, there are no indirect authorization errors (the global roles are *sufficient*) or information leaks (the global roles are *non-disclosing*).

Global schemas are created and used as follows. Most applications manage RBAC in a metadata-driven manner, and one can use automated tools to extract this metadata and construct *role interfaces* for each application. Information flow

[1]Products in this space include Aveksa Role Management, Bridgestream SmartRoles (acquired by Oracle in September 2007), Securent Entitlement Management (acquired by Cisco in November 2007), and Vaau RBACx (acquired by Sun in November 2007).

data can be obtained through static analysis techniques [SM03] or by manual annotation, as schema inference requires only coarse-grained inter-service flow. A constraint-based algorithm is then used to infer, given the local role assignments for a set of components, whether there exists a sufficient and non-disclosing global schema that is consistent with the local roles in each component. This schema is then used by administrators as a guide in assigning roles to end users. For example, one might assign sets of global roles to users, based on their job function and required access rights. An automated tool would then grant the user the associated local roles for each application. The resulting assignment thus guarantees sufficiency and does not permit the user to access beyond the rights implied by their global roles.

4.1.3 Chapter organization

In Section 4.2, I illustrate the access policy integration problem and global schema inference through an example. In Sections 4.3 and 4.4, I formalize the notion of role interfaces and present an algorithm for inferring sufficient global role schemas. This algorithm is shown to be NP-complete and works by reducing schema inference to Boolean satisfiability. Next, in Section 4.5, I augment role interfaces to model information flow between services. I extend the global schema inference algorithm to produce global role schemas that are sufficient as well as non-disclosing. Section 4.6 describes the implementation of a tool incorporating the schema inference algorithm and its use on several case studies. Finally, related research work is evaluated in Section 4.7.

Application	Service	Roles
Clinical Management	Scheduling	C:Receptionist, C:Nurse, C:Doctor
		C:Doctor
	Vitals	C:Nurse, C:Doctor
	CareOrders	C:Doctor
Laboratory	TestOrders	L:Clinician, L:Billing
	TestResults	L:Clinician
Patient Records	PatientHistory	P:Clinician

Figure 4.1: Services used in our examples.

4.2 Example

I demonstrate these techniques on a hypothetical healthcare information system at a medical clinic. The clinic has three applications:

- Clinical management: this application manages the scheduling of patients and captures the actions performed by doctors and nurses.
- Laboratory information system: this application tracks the tests to be performed and their results.
- Patient records: this application maintains historical data about each patient's health.

Each system provides one or more web services, which expose a set of callable methods and encapsulates access to the underlying data and application functionality. These services are protected by Role-Based Access Control (RBAC). A set of roles is associated with each service and with each user. To access a

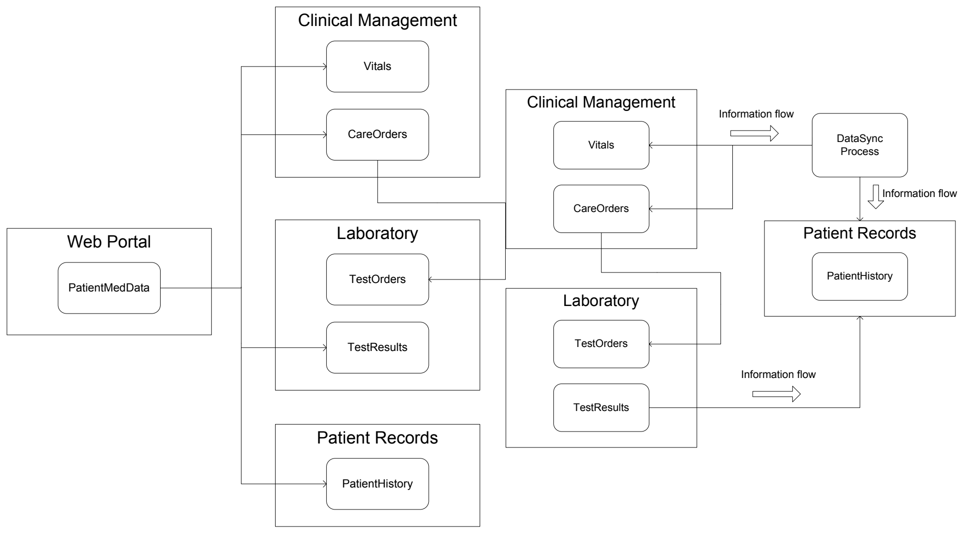


Figure 4.2: (a) Services of the web portal application. Boxes denote applications with their own notions of roles. Inner boxes show services available for each application. Directed arrows show other services that are called by a service. (b) Data synchronization between applications. Wide arrows denote direction of information flow.

service, the set of active roles for the current user must include at least one of the roles required by the service.

Unfortunately, each application has its own RBAC schema, so integrating these systems requires reconciling multiple schemas. Figure 4.1 lists the services provided by each application and the roles required for accessing these services. We prefix application-local role names with a unique letter for each application (C, L, or P). We call the roles defined within a given application *local roles*.

4.2.1 Web Portal Application

Suppose the clinic adds a new web portal which provides convenient web access across the other three applications (Figure 4.2(a)). The application does not store any confidential data locally. Instead, when the user requests a page, the portal makes service calls to the other applications using the requesting user's login.

Consider the PatientMedData service of the portal which permits doctors and nurses to see the relevant medical information for a patient. Figure 4.2(a) shows the services called by PatientMedData: the Vitals and CareOrders services of Clinical Management, the TestResults service of Laboratory, and the PatientHistory service of Patient Records. The CareOrders service, in turn, calls the TestOrders service to retrieve details of tests that have been ordered. This service is accessible to users with either the W : Doctor or W : Nurse roles.

Global Roles Since each application has its own notion of roles, it is difficult to determine whether a given user will have all the access permissions needed to retrieve the data for each page. This is the *global role compatibility* problem: does there exist some global set of roles that represent sets of local roles from the different applications that can be used to maintain consistent user to role mappings? The global role compatibility problem takes as input the applications and their role requirements, and produces, if possible, a set of global roles and a mapping from each global role to a set of local roles such that the following constraints are satisfied:

1. [**Separation**] No two local roles from the same application should be mapped to the same global role. This ensures that the semantics of authorization within an application is not changed by the global map. Otherwise, administrators will be unable to independently assign the two roles to users. Also, this restriction prevents degenerate solutions, such as assigning all local roles from a given application to the same group.

2. [**Sufficiency**] For each service of the web portal application, services that it (transitively) calls must be accessible by all of the global roles required

by the web portal service. Thus, if a user has any one of the required roles for the PatientMedData service, they will have a required role for each called service. This ensures no indirect authorization failures.

3. [**Ascriptions**] Administrators may optionally specify sets of local roles that must map to the same global role. This permits the representation of semantic constraints specific to an application domain.

A global role mapping is the *minimal* mapping which satisfies the constraints: if there are no calls combining two roles on separate systems, then the roles should be mapped to two separate global roles. For example, the C : Receptionist and L : Billing roles are unrelated to each other or any other roles through calls. Thus, they should be mapped to unique global roles. Minimality ensures a form of least privilege — a global role mapping should not give users any more access rights than strictly necessary to accomplish their objectives.

Note that a given local role can map to more than one global role. This may occur when one system has a less precise security model than the others. In our example, the Laboratory system has only a clinician role for both doctors and nurses, while the Clinical Management application distinguishes between doctors and nurses. By requirement 1 above, we need to maintain two separate global roles for doctors and nurses. If the Vitals service of Clinical Management is used in conjunction with the TestResults service of Laboratory, we need to map Clinician to both of these global roles.

Global Schema Inference We solve the global schema inference problem by formulating it as a Boolean constraint satisfaction problem. Notice that requirement 1 constrains W : Doctor and W : Nurse to be in different global roles (Constraint (1)), and similarly C : Doctor and C : Nurse must be in different global

Global Role	Local Roles
G:Doctor	W:Doctor, C:Doctor, L:Clinician, P:Clinician
G:Nurse	W:Nurse, C:Nurse, L:Clinician, P:Clinician

(a)

Global Role	Local Roles
G:Doctor	C:Doctor, P:Clinician, L:Clinician
G:Nurse	C:Nurse, L:Clinician

(b)

Figure 4.3: Role Mappings: (a) Separation, sufficiency, and ascription constraints (b) With information flow

roles (Constraint (2)). Since PatientMedData invokes CareOrders, the set of global roles for PatientMedData must be included in the set of global roles for CareOrders (Constraint (3)). To reflect the idea that doctor and nurse roles should be common across applications, we add an ascription to force the roles W : Doctor and C : Doctor to map to the same global role (Constraint (4)) and, similarly, an ascription to force the roles W : Nurse and C : Nurse to the same global role (Constraint (5)).

Surprisingly, it is not possible to find a mapping which satisfies all these constraints. From Constraints (1) and (2), the Doctor and Nurse roles within each service must be mapped to different global roles. Constraint (3) forces C : Doctor to map to both W : Doctor and W : Nurse. However, this violates the ascription constraints, as W : Nurse should be mapped to role C : Nurse. Thus, our initial portal design does not admit global role schemas. With a tool to infer global role schemas, we can catch such problems at design time rather when users are assigned to roles and attempt to use the system (as common with ad hoc approaches to access control integration).

Now consider an alternative design of the web portal where we split the PatientMedData service into two services: PatientMedDataD, which contains all the data from the original PatientMedData, but is only accessible to the W:Doctor role, and PatientMedDataN, which does not include data from the CareOrders service, and is accessible to the W : Nurse role. In this case, we obtain the (global) role mappings from Figure 4.3(a) that maintains consistent global authorization across applications.

4.2.2 Roles and Information Flow

We now extend the role mapping algorithm in the presence of information flow. If one application keeps a copy of protected data from another application, it must control access to this copied data. Otherwise, a user that does not have access to the original application may be able to retrieve the same data through the target application. In our example, the Patient Records application maintains an archive of data from the Clinical Management and Laboratory applications. Thus, Patient Records must deny access to any users which do not have access to both the Clinical Management and Laboratory applications.

As shown in Figure 4.2(b), the Patient Records application is populated with data in the following manner. The DataSync process periodically calls the Vitals and CareOrders services to retrieve patient clinical data older than a certain age, saves this data to the PatientHistory service, and then deletes the original copies from Clinical Management. This DataSync process runs as a super user on both systems, and thus does not have any issues with accessing the appropriate services. The TestResults service periodically connects directly to PatientHistory as a super user and saves any new test results since the last update. Finally, as with the web portal example, the CareOrders service makes a call to the TestOrders, but only relays the resulting data to its caller, without saving it locally. When computing the global roles for this scenario, we enforce the same properties as listed before. In addition, we want to ensure that the users which can access the target service of a data synchronization are always a subset of the users which can access the source service. To implement this, we use sets of roles as a proxy for sets of users and thus add a new requirement:

4. [**Information Flow**] Whenever data flows from one service to another, the target service's global roles must be a subset of the source service's roles.

This ensures that the target service provides at least the same level of access control for the data as its originating service. Note that information may flow in the opposite direction as a call (e.g., the calls to Vitals and CareOrders by DataSync).

For the data synchronization scenario, we obtain the following additional constraints for the doctor and nurse related roles from information flow considerations:

1. From the information flow from Vitals to PatientHistory, the global roles for P : Clinician must be a subset of the global roles for the set { C : Nurse,C : Doctor }.
2. From the information flow from CareOrders to PatientHistory, the global roles for P : Clinician must be a subset of the global roles for C : Doctor.
3. From the information flow from TestResults to PatientHistory, the global roles for P : Clinician must be a subset of the global roles for L : Clinician.

If we solve these constraints to find a set of global roles, we obtain the mapping from Figure 4.3(b). Note that this mapping shuts nurses out from accessing the PatientHistory service (by excluding role P : Clinician) — it exposes data from CareOrders, which is only visible to doctors. A naive mapping of roles that allows both doctors and nurses to access PatientHistory would subvert the access controls applied to CareOrders. This could be a serious issue, violating privacy regulations (such as HIPAA) and opening the system to abuse.

4.3 Semantics of Roles

I now define an interface describing an application's services, the roles which may access each service, and the possible outbound calls made by each service. I then describe the runtime behavior of these interfaces using a small-step operational semantics. This allows us to prove properties relating the static structure and dynamic behavior of such systems.

4.3.1 Services and their Semantics

Let Names be a set of *web service names* and Users a set of *users*. A *web application* $A = (\mathsf{Roles}, \mathsf{Services}, \mathsf{Perm})$ consists of a set of *roles* Roles, a set of *services* $\mathsf{Services}$, and a *user permission mapping* $\mathsf{Perm} : \mathsf{Users} \to 2^{\mathsf{Roles}}$ from the (global) set of users to subsets of roles in Roles. A *(web) service* $S = (n, R, C, M)$ in $\mathsf{Services}$ consists of a name $n \in \mathsf{Names}$, a subset of roles $R \subseteq \mathsf{Roles}$ denoting the required permissions to call n, a set of called service names $C \subseteq \mathsf{Names}$, and a mapping $M : C \to 2^R$ from the services in C to subsets of R. We write $A.\mathsf{Roles}$, $A.\mathsf{Services}$, and $A.\mathsf{Perm}$ to refer to the roles, services, and user maps of A, respectively, and for a service S, we write $S.n$, $S.R$, $S.C$, and $S.M$ to reference the components of a service. We assume that there is exactly one service with name n, and we write $\mathsf{Svc}.n$ for that service. With abuse of notation, we identify a service S with its name $S.n$, and say, e.g., that a service n is in an application A.

The required roles are disjunctive — one of the roles must be satisfied to call the service. The mapping M represents a more precise subset of the roles known to be active when calling a service.

A *system* Sys consists of a set of applications, where we assume that the services and roles of the applications are pairwise disjoint. With abuse of notation, we speak of a service or role in a system for a service or role in an application in the system. Thus, we speak of a service $S \in \mathsf{Sys}$ if there is an application $A \in \mathsf{Sys}$ such that $S \in A.\mathsf{Services}$. We write $\mathsf{AllRoles} = \cup_{A \in \mathsf{Sys}} A.\mathsf{Roles}$ for the set of all (local) roles in system Sys.

We say that a system Sys is *well-formed* if, (a) [service names are unique] for each $n \in \mathsf{Names}$, there is at most one service S in Sys with $S.n = n$, (b) [all called services exist] for each application $A \in \mathsf{Sys}$, each service $S \in A.\mathsf{Services}$, and each called service name $n \in S.C$, there exists an application $A' \in \mathsf{Sys}$ and a service $S' \in A'.S$ such that $S'.n = n$. We say that a system has *non-redundant* roles if no two roles are assigned to the same subset of the services, formally, if there does not exist an application A and roles $r_1, r_2 \in A.\mathsf{Roles}$ such that for all services $S \in A.\mathcal{S}$, we have $r_1 \in S.R$ iff $r_2 \in S.R$. Well-formedness and non-redundancy are syntactic checks, and henceforth we assume all systems have both properties.

4.3.2 Operational Semantics

A system represents the composition of several web service applications, each with its own notion of roles and web services. Users can invoke a service in the system. The roles determine if the user has sufficient permissions to use the service as well as services transitively invoked by the called service. That is, before executing a user-invoked service request, every service S first checks if the initiator of the request (the user, or the service invoking the call) has appropriate permissions (roles) —determined by the roles $S.R$— to execute the request. If the initiator has permissions to call the service, it is executed to completion (this might involve calls to other services), otherwise there is an authentication failure. We formalize the runtime behavior of service invocations using *service call expressions* and their operational semantics.

$\boxed{e \longrightarrow e'}$

$$\frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Call}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u)} \text{ (E-CallEval)}$$

$$\frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) = \emptyset}{\mathsf{Call}(S.n, u) \longrightarrow \mathsf{AuthFail}} \text{ (E-CallFail)}$$

$$\frac{S \in A_i \qquad c \in S.C \qquad (M[c] \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Call}(c, u)} \text{ (E-EvalCall)}$$

$$\frac{S.C \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u); \mathsf{Eval}(S.n, u)} \text{ (E-EvalSeq)}$$

$$\frac{}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Done}} \text{ (E-EvalDone)} \qquad \frac{e_1 \longrightarrow e_1'}{e_1; e_2 \longrightarrow e_1'; e_2} \text{ (E-SeqRed)}$$

$$\frac{}{\mathsf{Done}; e_2 \longrightarrow e_2} \text{ (E-SeqDone)}$$

Figure 4.4: Operational semantics for service calls

Service call expressions are generated by the following grammar:

$$e ::= \mathsf{Call}(n, u) \mid \mathsf{Eval}(n, u) \mid \mathsf{Done} \mid \mathsf{AuthFail} \mid e; e$$

A Call expression represents a service call before being checked for permissions; an Eval expression represents the evaluation of a service, the symbol Done represents the successful completion of a service call; the symbol AuthFail represents early termination of a service call due to an authentication failure. The sequential composition of two expressions is represented by $e_1; e_2$. The variable n ranges over service names and u ranges over users. For clarity, we omit other control structures from this core language, they do not introduce additional conceptual difficulties.

Figure 4.4 defines the small step operational semantics of service calls, formalized by a binary relation $\longrightarrow$ on service call expressions.

Evaluation starts with a single Call expression. Rules E-CallEval and E-

CallFail represent the checking of the user's roles against those required for the service. If the check is successful, the call steps to the evaluation of the service. Otherwise, evaluation stops with the $\mathsf{AuthFail}$ symbol. An Eval expression may step to a service call (rule E-EvalCall), successful termination of the subcomputation (rule E-EvalDone), or a sequence of two Eval expressions (rule E-EvalSeq). Duplication of Eval expressions captures the nondeterministic nature of services: a service can call any service in its call set zero or more times. In addition, a service call is not guaranteed to terminate. The E-SeqRed and E-SeqDone rules permit reduction of expression sequences by evaluating the first expression in the sequence.

We write $\longrightarrow^*$ for the reflexive transitive closure of the $\longrightarrow$ relation. A *direct service invocation* is an expression of the form $\mathsf{Call}(n, u)$ representing the direct call of a service by a user.

Proposition 1 (Evaluation)**.** *Let* Sys *be a well-formed system,* $n \in \mathsf{Names}$ *a service in* Sys*, and* $u \in \mathsf{Users}$ *a user. Then, the evaluation via* $\longrightarrow^*$ *of the call* $\mathsf{Call}(n, u)$ *either diverges or eventually terminates with* Done *or* $\mathsf{AuthFail}$*.*

4.3.3 Accessibility and Sufficiency

Let Sys be a system, $n \in \mathsf{Names}$ a service in Sys, and $u \in \mathsf{Users}$ a user. The call $\mathsf{Call}(n, u)$ is *accepted* by Sys if there is a service $S \in \mathsf{Sys}$ with $S.n = n$ and $S.R \cap A_i.\mathsf{Perm}.u \neq \emptyset$. Otherwise $\mathsf{Call}(n, u)$ is *rejected*. Intuitively, a call to service n by user u is accepted if the user has at least one of the roles required to execute the service named n. In this case, the call evaluates in one step (by rule E-CallEval) to $\mathsf{Eval}(n, u)$.

For a system Sys, the function $\mathsf{rolesOf} : \mathsf{Users} \rightarrow 2^{\mathsf{AllRoles}}$ maps each user u to the set of roles available to u, that is, $\mathsf{rolesOf}.u = \cup_{A \in \mathsf{Sys}} A.\mathsf{Perm}.u$. A set of services $\mathsf{Services}$ is *accessible* to a set of roles R if for each service $S \in \mathsf{Services}$, there exists an $r \in R$ such that $r \in S.R$. Similarly, a set of services $\mathsf{Services}$ is *accessible* to a user $u \in \mathsf{Users}$ if $\mathsf{Services}$ is accessible to $\mathsf{rolesOf}.u$. In this case, all calls by u to any service in $\mathsf{Services}$ will be accepted; however, transitive calls made by these services may cause authentication failures.

We wish to ensure that if a call is accepted by a system, no further authorization errors can occur. This is provided by the stronger notion of *sufficiency*. We say that a set of roles R is *sufficient* if, for every user u with $\mathsf{rolesOf}.u = R$ and service S accessible to R, there is no trajectory $\mathsf{Call}(S.n, u) \longrightarrow^* \mathsf{AuthFail}$. A system is sufficient if for all users $u \in \mathsf{Users}$, we have that $\mathsf{rolesOf}.u$ is sufficient.

4.3.4 Role Compatibility

For systems with a single application, sufficiency can be checked by a dataflow analysis [PFF07]. In general though, each application in a system comes with its own notion of local roles, and a call in application A_1 to service S in application A_2 only provides information about roles in A_1 held at the call point, not the roles in application A_2. Thus, in order to check sufficiency, we must somehow "convert" the local roles in each application to a global set of roles. We introduce *global role schemas* to do this.

Global Role Schema Let Sys be a system. A *global role schema* $\mathsf{Grs} = (\mathcal{R}, G)$ consists of a set $\mathcal{R}$ of *global role names* and a mapping $G : \mathcal{R} \mapsto 2^{\mathsf{AllRoles}}$ that maps global role names to sets of local roles in the system Sys. This schema guides role assignments for individual users: if a user is assigned to a global role $g \in \mathcal{R}$, then that user must also be assigned all local roles in the set $G.g$.

We can also take an existing set of user assignments and see whether it corresponds to our global role schema. We say that the assignment of roles $\{A.\mathsf{Perm} \mid A \in \mathsf{Sys}\}$ *conforms* to a global role schema Grs if there exists a user to global role assignment $\mathsf{Perm} : \mathsf{Users} \to 2^{\mathcal{R}}$ such that for all users u

$$\bigcup_{g \in \mathsf{Perm}.u} G.g = \mathsf{rolesOf}.u$$

That is, there is a mapping of users to global roles such that the set of local roles designated by the global role schema to each user u is exactly the set of local roles $\mathsf{rolesOf}.u$ assigned to the user.

Sufficiency A global role schema Grs is *fully sufficient* if, for each global role $g \in \mathcal{R}$, the set of local roles $G.g$ is sufficient. Given a user-role assignment that conforms to a fully sufficient role schema, any service call by an arbitrary user will either be rejected or execute without authentication failure.

Separation A global role schema $\mathsf{Grs} = (\mathcal{R}, G)$ has *role separation* if no two roles from the same application map to the same global role, that is, for all $g \in \mathcal{R}$ and $A \in \mathsf{Sys}$, we have $|G.g \cap A.\mathsf{Roles}| \leq 1$.

Role separation ensures that the roles of each application can be assigned to users independently. If multiple roles of an application appear in the same global role, these roles are effectively combined, potentially violating the intent of the original roles (e.g., allowing users access to data they should not see).

Minimality Minimality encodes the requirement that a global role schema should not grant access to more services than necessary to ensure sufficiency. A set of local roles R is *minimal* if it is sufficient and there exists an $l \in R$ such that any subset of R containing l is not sufficient. We extend minimality to global role schemas as follows: a global role schema is *minimal* if there exists an injective mapping $\mu : \mathcal{R} \rightarrow \mathsf{AllRoles}$ from global roles to local roles $\mathsf{AllRoles}$ such that (a) for all $g \in \mathcal{R}$ we have $\mu.g \in G.g$, and (b) any subset of $G.g$ containing $\mu.g$ is not sufficient. These conditions ensure that each global role g has unique local role which requires the local role set $G.g$ for sufficiency. Note that there may be more than one minimal global role schema.

4.3.5 Global Schema Inference

The *global schema inference problem* (GSI) takes as an input a system Sys and asks if there is a minimal global schema Grs which has separation and is fully sufficient.

Theorem 15. *GSI is NP-complete.*

Proof (outline). Given a global schema and a witness for minimality, one can check the properties in polynomial time. The hardness is by a reduction from one-in-three 3SAT. One-in-three 3SAT is a variant of 3SAT which determines whether, for a list of three-literal causes, there exists an assignment to the referenced boolean variables such that every clause contains exactly one true literal.

Given an instance of one-in-three 3SAT with N clauses, we first create an application A_L which defines a special local role L and contains a single service S_L that is accessible to role L. For each distinct boolean variable v, we create an application A_v with local roles v^+ and v^-, corresponding to the literals v and $\neg v$, respectively. Each of these applications contains three services: S_v^+, accessible to role v^+, S_v^-, accessible to role v^-, and S_v, accessible to both roles. Service S_v is called by service S_L. These calls represent the constraint that each variable is either true or false.

For each clause i, we also define an application A_i with three local roles and four services. The local roles are created using the following naming convention:

- If the clause i contains the positive literal v, we define a local role v_i^+.
- If the clause i contains the negative literal $\neg v$, we define a local role v_i^-.

The first service is named S_i, is accessible to all three local roles, and is called by the service S_L. These calls represent the constraint that only one literal is true in each clause. The other three services correspond to the local roles as follows:

- If the local role v_i^+ is defined, then a service $S_{v_i}^+$ is created and protected by this local role.
- If the local role v_i^- is defined, then a service $S_{v_i}^-$ is created and protected by this local role.

Finally, each service S_v^+ calls any services $S_{v_i}^+$ defined above and each service S_v^- calls any services $S_{v_i}^-$ above.

If we solve the global role schema for a group containing global role L, we obtain a set of local roles that includes one of v^+ or v^- for each boolean variable v in the original one-in-three 3SAT problem. If we assign `true` to those variables for which role v^+ is in the group and `false` to those variables for which role v^- is in the group, we obtain a solution to the one-in-three 3SAT problem. If no global role schema is found, then no solution exists to the satisfiability problem either. $\square$

4.4 Constraint-Based Inference

We solve the global schema inference problem through Boolean constraint solving. First, notice that, due to our minimality requirement, the number of global roles is at most the total number of roles in $\mathsf{AllRoles}$. Although each local role can be in one or more global roles, each global role must be sufficient for at least one local role. If the number of global roles is larger than the number of local roles $\mathsf{AllRoles}$, then global roles can be eliminated while still ensuring a sufficient global role for each local role.

We generate a set of global roles that include a local role in the following way. Fix a global role g. For each local role $r \in \mathsf{AllRoles}$, we define an atomic predicate r^g which is `true` if the role r is included in the global role g and `false` otherwise. The predicates r^g satisfy the following constraints.

1. [**Separation Constraints**] No two local roles from the same application should be mapped to the same global role. That is, for each $A \in \mathsf{Sys}$, at most one local role $r \in A.\mathsf{Roles}$ can be in g. Thus, for each application $A \in \mathsf{Sys}$, we have (considering each r^g to be a 0-1 variable) $\sum_{r \in A.\mathsf{Roles}} r^g \leq 1$, or equivalently,
$$\bigwedge_{A \in \mathsf{Sys}} \bigwedge_{r_1, r_2 \in A.\mathsf{Roles}, r_1 \neq r_2} (\neg r_1^g \vee \neg r_2^g)$$

2. [**Sufficiency Constraints**] The sufficiency constraints dictate that for each service S and each service $c \in S.C$ called from S, if one of the roles in $S.M.c$ is mapped to the global roles g, then one of the roles in $\mathsf{Svc}.c.R$ must also be mapped to g. That is,
$$\bigwedge_{A \in \mathsf{Sys}} \bigwedge_{S \in A.\mathsf{Services}} (\bigvee_{r \in S.M.c} r^g) \rightarrow (\bigvee_{\hat{r} \in \mathsf{Svc}.c.R} \hat{r}^g) \tag{4.1}$$

Algorithm 1 solve_full and minimize

```
function solve_full
input System Sys
$\mathcal{R} \leftarrow \{r \mid r \in A, A \in \mathsf{Sys}\}$
$\phi_{\mathsf{Sys}} \leftarrow \mathsf{constraint\_pred}(\mathsf{Sys})$
$G \leftarrow \texttt{Map.empty}$
while $\mathcal{R} \neq \emptyset$ do
   $r \leftarrow \mathsf{choose}(\mathcal{R})$ {Pick a role to solve}
   $g \leftarrow \mathsf{makename}(r)$ {Name the group for $r$}
   $\mathcal{R}_g \leftarrow \mathsf{SAT}(\phi_{\mathsf{Sys}} \wedge r^g)$
   if $\mathcal{R}_g \neq \emptyset$ then
      $G.g \leftarrow \mathsf{minimize}(\mathsf{Sys}, \mathcal{R}_g, \{r\})$
      $\mathcal{R} \leftarrow \mathcal{R} \setminus G.g$
   else
      return no_solution
   end if
end while
return $G$
```

```
function minimize
inputs System Sys, Role group $\mathcal{R}_g$,
   Required Roles $\mathcal{R}_{req}$
$\mathcal{R}_{min} \leftarrow \mathcal{R}_{req}$
$\mathcal{R}_{new} \leftarrow \mathcal{R}_{req}$
$\mathcal{S}_{min} \leftarrow \mathsf{accessible}(\mathcal{R}_g)$
repeat
   $\mathcal{S}_{new} = \mathsf{callable}(\mathcal{R}_{new}) \setminus \mathcal{S}_{min}$
   $\mathcal{R}_{new} = \mathsf{roles\_for}(\mathcal{S}_{new}, \mathcal{R}_g)$
   $\mathcal{R}_{min} = \mathcal{R}_{min} \cup \mathcal{R}_{new}$
   $\mathcal{S}_{min} = \mathcal{S}_{min} \cup \mathcal{S}_{new}$
until $\mathcal{R}_{new} = \emptyset$
return $\mathcal{R}_{min}$
```

Let ϕ_{Sys} be the conjunction of the constraints from Equation 1 and Equation 4.1. Clearly, ϕ_{Sys} is polynomial in the size of Sys. A satisfying assignment for ϕ is a function mapping each r^g to $true$ or $false$ such that ϕ evaluates to $true$.

Theorem 16. *Let ρ be a satisfying assignment to ϕ_{Sys}. Then the set of roles $\{r \mid \rho.r^g = true\}$ is a global role which is fully sufficient and has role separation.*

Given the constraints, we can find a global group containing local role r by conjoining ϕ_{Sys} with r^g. To construct a global role schema G, we iterate through the set of local roles AllRoles, finding a global role group for each local role. This is done in Algorithm solve_full.

The function $\mathsf{SAT}(\phi)$ returns a set of local roles which are assigned `true` in a satisfying assignment for the constraint ϕ. The resulting set of roles is then passed to minimize (described below), which removes any local roles not required for full sufficiency, while keeping r in the group. The roles from the resulting minimized group are then removed from the workset $\mathcal{R}$, and another role is selected for solving. When $\mathcal{R}$ is empty, all the necessary global groups has been created.

If there is no satisfying assignment to $\phi_{\mathsf{Sys}} \wedge r^g$, where r is one of the local roles in $\mathcal{R}$, then $\mathsf{SAT}(\phi_{\mathsf{Sys}} \wedge r^g)$ will return $\emptyset$ and $\mathsf{solve\_full}$ will stop with `no_solution`.

The $\mathsf{minimize}$ function is called with the role group $\mathcal{R}_g$, computed from the boolean constraints in $\mathsf{solve\_full}$, and $\mathcal{R}_{req}$, a subset of $\mathcal{R}_g$ roles which must be present in the final minimized group $\mathcal{R}_{min}$. Three sets are maintained: $\mathcal{R}_{min}$ is the minimized group, initialized to $\mathcal{R}_{req}$, $\mathcal{R}_{new}$ contains the roles added by the previous iteration, initialized to $\mathcal{R}_g$, and $\mathcal{S}_{min}$ contains the services accessible, given the set of roles $\mathcal{R}_{min}$. For each iteration, the services directly callable from $\mathcal{S}_{new}$ (the services added the previous iteration) are added to $\mathcal{S}_{min}$. Then, any roles needed to makes these callable services accessible are added to $\mathcal{R}_{min}$. These roles are selected from the role group $\mathcal{R}_g$ by taking the intersection between $\mathcal{R}_g$ and each service's role set. At the fixpoint, $\mathcal{R}_g$ is both fully sufficient and minimal.

Theorem 17. *If $\mathsf{solve\_full}$ returns a global role schema G for Sys, then G has role separation, is fully sufficient, is minimal with respect to the role signatures of Sys, and each local role appears in at least one global role. If $\mathsf{solve\_full}$ terminates with `no_solution` for Sys, then no such global role schema exists for Sys.*

Solving ascribed roles The administrator can specify a subset R' of local roles such that there must be a global role g with $R' \subseteq G.g$. The above algorithm does not address these *role ascriptions*. To extend $\mathsf{solve\_full}$ for ascribed roles, we define the following constraints.

- [**Ascription Constraints**] For each ascription $\{r_1, \ldots, r_k\}$, we have $r_1^g \leftrightarrow r_2^g \leftrightarrow \ldots \leftrightarrow r_k^g$.

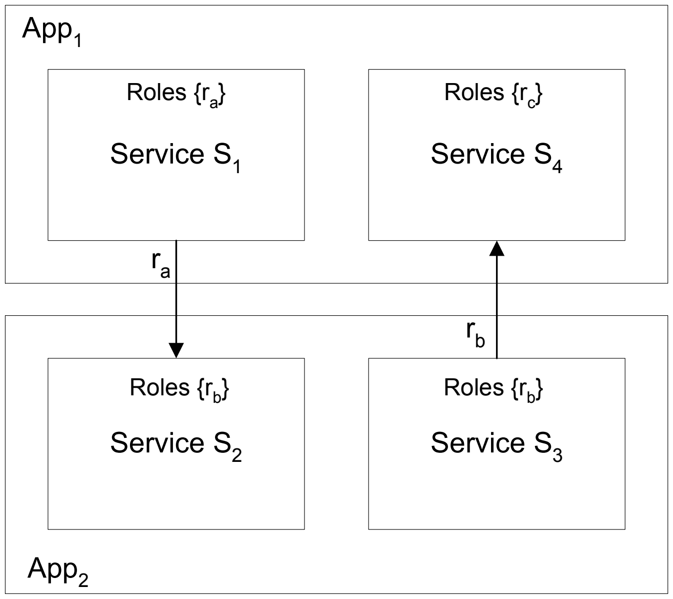


Figure 4.5: Example requiring relaxed sufficiency

We first solve for each of the ascribed roles, conjoining the associated ascription constraint with ϕ_{Sys}. minimize is called for ascribed groups with $\mathcal{R}_{req} = \{r_1, \ldots, r_k\}$, keeping the ascribed roles in the minimized group. After solutions are found for each ascribed group, we then solve for the remaining roles without any ascription constraints.

Note that we permit ascribed groups to be extended as needed to achieve sufficiency. Due to minimize, we will not unnecessarily add roles. If all local roles are ascribed, then the problem is reduced to *global schema checking*, rather than *global schema inference*.

4.4.1 Relaxing Full Sufficiency

Requiring that $G.g$ is sufficient for each global role g may be too strict for some situations. For example, consider the system of Figure 4.5, which has two applications, app_1 and app_2. app_1 has services S_1 and S_4, S_1 requires the role r_a, and S_4 the role r_c. app_2 has services S_2 and S_3, both require the role r_b. S_1 calls S_2, and S_3 calls S_4. There is no fully sufficient global role schema with separation for this system. Since S_1 calls S_2, a global role containing role r_a must also contain role r_b. However, if a user has role r_b, then service S_3 is accessible. Since S_3 calls S_4, which requires role r_c, the global role must also contain r_c. But this violates separation.

In many situations, we only require a relaxed version of sufficiency. For example, in a "bottom-up" approach to role assignment, the systems are administered based on local roles and global roles are used to ensure interoperability. Each local role has an associated global role containing any remote roles needed to avoid indirect authorization errors. When a user needs access to a system, the administrator picks a local role for that system and then assigns to the user all the roles included in the associated global group. We model this scenario using *subset sufficiency*.

We say that global role g is *subset sufficient* for local role set R_s if $R_s \subseteq G.g$ and R_s is sufficient. In this case, users requiring local roles in R_s can be assigned group g. With this role assignment, all direct calls to services accessible to R_s will not have authorization errors. Of course, any direct call to a service not accessible to $\mathcal{R}_s$ (but accessible to $G.g$) is not guaranteed to execute without authentication failure. Thus, instead of looking for fully sufficient solutions, we can look for global role schema such that for each user u, there is some set of global roles that is subset sufficient for $\mathsf{rolesOf}.u$. Using subset sufficiency, and assuming the role sets $\{r_a\}$, $\{r_b\}$, and $\{r_c\}$ for users, the above system has the solution: $G_1 = \{r_a, r_b\}$ (subset sufficient for $\{r_a\}$), $G_2 = \{r_b, r_c\}$ (subset sufficient for $\{r_b\}$), and $G_3 = \{r_c\}$ (subset sufficient for $\{r_c\}$).

We adjust our definition of minimality to account for subset sufficiency. A set of local roles R is *subset minimal* if it is subset sufficient for a $R_s \subseteq R$ and any subset $R' \subset R$ where $R_s \subseteq R'$ is not subset sufficient for R_s. We extend this definition to global role schemas as follows: a global role schema is *subset minimal* if there exists an injective mapping $\mu : \mathcal{R} \rightarrow \mathsf{AllRoles}$ from global roles to local roles $\mathsf{AllRoles}$ such that (a) for all $g \in \mathcal{R}$ we have $\mu.g \in G.g$, and (b) any R' such that $\mu.g \in R'$ and $R' \subset G.g$ is not subset sufficient for $\{\mu.g\}$.

To infer a global role schema that only has subset sufficiency, we must adjust the boolean constraint ϕ_{Sys}. For each service S in the system, we introduce an atomic predicate S^g which is true if S is transitively callable from a service accessible from a group's required roles. The sufficiency constraints are modified in the following way:

2' [**Subset Sufficiency Constraints**] For each called service c in the signature of a service s, we add:

$$(S^g \wedge \bigvee_{r \in S.M.c} r^g) \rightarrow c$$

Additionally, to ensure that callable services are accessible, for each service S, we add $S^g \rightarrow \bigvee_{r \in S.R} r^g$.

To find a group for a specific role r, we call SAT with $\phi_{\mathsf{Sys}} \wedge r \wedge \bigwedge_{S | r \in S.R} S^g$. We then minimize the result, using a modified version of $\mathsf{minimize}$, which only adds those services callable from the services added in the previous iteration, rather than all services accessible to the newly added roles.

$$\boxed{e \longrightarrow e'}$$

$$\frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Call}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u)} \text{ (E-CallEval)} \qquad \frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) = \emptyset}{\mathsf{Call}(S.n, u) \longrightarrow \mathsf{AuthFail}} \text{ (E-CallFail)}$$

$$\frac{(c, n) \notin S.I \qquad (n, c) \notin S.I \qquad S \in A_i \qquad c \in S.C \qquad S.M[c] \cap A_i.\mathsf{Perm}.u \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Call}(c, u)} \text{ (E-EvalCall)}$$

$$\frac{(S.n, c) \in S.I \qquad S \in A_i \qquad c \in S.C \qquad S.M[c] \cap A_i.\mathsf{Perm}.u \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Send}(c, u, \mathsf{Data}\ S.n)} \text{ (E-EvalSend)}$$

$$\frac{\begin{array}{c} S \in A_i \\ n_1 \in S.C \qquad M[n_1] \cap A_i.\mathsf{Perm}.u \neq \emptyset \qquad n_2 \in S.C \qquad M[n_2] \cap A_i.\mathsf{Perm}.u \neq \emptyset \qquad (n_1, n_2) \in S.I \end{array}}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Send}(n_2, u, \mathsf{Request}(n_1, u))} \text{ (E-EvalMove)}$$

$$\frac{(c, S.n) \in S.I \qquad S \in A_i \qquad c \in S.C \qquad S.M[c] \cap A_i.\mathsf{Perm}.u \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Save}(S.n, \mathsf{Request}(c, u))} \text{ (E-EvalSave)}$$

$$\frac{S.C \neq \emptyset}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u); \mathsf{Eval}(S.n, u)} \text{ (E-EvalSeq)} \qquad \frac{}{\mathsf{Eval}(S.n, u) \longrightarrow \mathsf{Done}} \text{ (E-EvalDone)}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1; e_2 \longrightarrow e_1'; e_2} \text{ (E-SeqRed)} \qquad \frac{}{\mathsf{Done}; e_2 \longrightarrow e_2} \text{ (E-SeqDone)}$$

Figure 4.6: Semantics for service calls with information flow (part 1)

4.5 Services with Information Flow

We now extend our results to services and systems where we model flow of sensitive data between applications. We must now ensure that information that can only be accessed under some role constraints is not "disclosed" to applications that do not hold the required roles.

To model information flow, we extend the services to include a directed *information flow graph*. Thus, a service is now a 5-tuple (n, R, C, M, I), where (n, R, C, M) are as before, and $I \subseteq (C \cup \{n, \mathsf{Caller}_{in}\}) \times (C \cup \{n, \mathsf{Caller}_{out}\})$ is a set of pairs of service names (or the special symbols Caller_{in} and Caller_{out} denoting the entry and exit points respectively of a caller of the service). A pair $(n_1, n_2) \in I$ represents an information flow from n_1 to n_2, which may occur when the service is called, and self-links of the form (c, c) are not permitted. The information flow graph models two forms of information flow: *synchronization*, where data from one service is saved in another service, and *disclosure*, where

$$\boxed{e \longrightarrow e'}$$

$$\frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Send}(S.n, u, \mathsf{Data}\ n_s) \longrightarrow \mathsf{Recv}(S.n, u, n_s)}\ (\text{E-}\textsc{SendRecv}) \qquad \frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) = \emptyset}{\mathsf{Send}(S.n, u, \mathsf{Data}\ n_s) \longrightarrow \mathsf{AuthFail}}\ (\text{E-}\textsc{SendFail})$$

$$\frac{}{\mathsf{Send}(S.n, u, \mathsf{NoData}) \longrightarrow \mathsf{Done}}\ (\text{E-}\textsc{SendNoData}) \qquad \frac{e \longrightarrow e'}{\mathsf{Send}(S.n, u, e) \longrightarrow \mathsf{Send}(S.n, u, e')}\ (\text{E-}\textsc{SendRed})$$

$$\frac{(\mathsf{Caller}_{in}, c) \in S.I \qquad c \in S.C}{\mathsf{Recv}(S.n, u, n_s) \longrightarrow \mathsf{Send}(c, u, \mathsf{Data}\ n_s)}\ (\text{E-}\textsc{RecvSend})$$

$$\frac{(\mathsf{Caller}_{in}, S.n) \in S.I}{\mathsf{Recv}(S.n, u, n_s) \longrightarrow \mathsf{Save}(S.n, \mathsf{Data}\ n_s)}\ (\text{E-}\textsc{RecvSave}) \qquad \frac{}{\mathsf{Recv}(S.n, u, n_s) \longrightarrow \mathsf{Eval}(S.n, u)}\ (\text{E-}\textsc{RecvEval})$$

$$\frac{}{\mathsf{Recv}(S.n, u, n_s) \longrightarrow \mathsf{Recv}(S.n, u, n_s); \mathsf{Recv}(S.n, u, n_s)}\ (\text{E-}\textsc{RecvSeq})$$

$$\frac{(S.n, \mathsf{Caller}_{out}) \in S.I \qquad S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Request}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u); \mathsf{Reply}(S.n, \mathsf{Data}\ S.n)}\ (\text{E-}\textsc{ReqData})$$

$$\frac{S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) = \emptyset}{\mathsf{Request}(S.n, u) \longrightarrow \mathsf{AuthFail}}\ (\text{E-}\textsc{ReqFail})$$

$$\frac{(n, \mathsf{Caller}_{out}) \in S.I \qquad S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Request}(S.n, u) \longrightarrow \mathsf{Eval}(S.n, u); \mathsf{Reply}(S.n, \mathsf{Request}(n, u))}\ (\text{E-}\textsc{ReqReq})$$

$$\frac{\nexists n \in \{S.n\} \cup S.C\ .\ (n, \mathsf{Caller}_{out}) \in S.I \qquad S \in A_i \qquad (S.R \cap A_i.\mathsf{Perm}.u) \neq \emptyset}{\mathsf{Request}(S.n, u) \longrightarrow \mathsf{NoData}}\ (\text{E-}\textsc{ReqNoData})$$

$$\frac{e \to e'}{\mathsf{Save}(n_d, e) \longrightarrow \mathsf{Save}(n_d, e')}\ (\text{E-}\textsc{SaveRed}) \qquad \frac{}{\mathsf{Save}(n_d, \mathsf{Data}\ n_s) \xrightarrow{n_s \bowtie n_d} \mathsf{Done}}\ (\text{E-}\textsc{SaveDone})$$

$$\frac{}{\mathsf{Save}(n_d, \mathsf{NoData}) \longrightarrow \mathsf{Done}}\ (\text{E-}\textsc{SaveNoData}) \qquad \frac{}{\mathsf{Reply}(n_d, \mathsf{Data}\ n_s) \xrightarrow{n_s \rhd n_d} \mathsf{Data}\ n_s}\ (\text{E-}\textsc{ReplyDisc})$$

$$\frac{}{\mathsf{Reply}(n_d, \mathsf{NoData}) \longrightarrow \mathsf{NoData}}\ (\text{E-}\textsc{ReplyNoData}) \qquad \frac{e \longrightarrow e'}{\mathsf{Reply}(n_d, e) \longrightarrow \mathsf{Reply}(n_d, e')}\ (\text{E-}\textsc{ReplyRed})$$

Figure 4.7: Semantics for service calls with information flow (part 2)

data from a service is made available to callers of a (potentially different) service. Given a service S and callee $c \in S.C$, the callee-to-self link $(c, S.n)$, represents a synchronization of data from service c to S. The callee-to-caller link $(c, \mathsf{Caller}_{out})$ represents a disclosure of data from c by S.

Synchronization and disclosure are distinguished from benign *non-disclosing transfers*, where a service moves data between two other services without saving or disclosing it. Callee-to-callee and caller-to-callee links, where there is no additional link from the source to the current service or its caller, are all non-disclosing. For example, if the information flow graph for service S contains $(c_1, c_2) \in S.I$, where $c_1, c_2 \in S.C$, and there are no links from c_1 or c_2 to $S.n$ in

$S.I$, then S facilitates an information flow from c_1 to c_2 without disclosure.

4.5.1 Operational semantics

To extend the dynamic semantics of services to include information flow effects, we extend the grammar of expressions as follows:

$$\begin{aligned} e ::= &\ \ldots \mid \mathsf{Send}(n, u, e) \mid \mathsf{Recv}(n_d, u, n_s) \mid \mathsf{Request}(n, u) \\ &\mid \mathsf{Reply}(n, e) \mid \mathsf{Save}(n_d, n_s) \mid \mathsf{Data}\ n \mid \mathsf{NoData} \end{aligned}$$

where n, n_s, and n_d range over service names and u ranges over users. The source of an information flow is represented by an expression of the form $\mathsf{Data}\ n$ where n is the name of the originating service. This expression may be passed between services until a call to Save is made, creating a synchronization, or a call to Reply is made, creating a disclosure.

Figures 4.6 and 4.7 list the inference rules which define our operational semantics with information flow. Each step of the $\longrightarrow$ relation may now include an optional information flow effect, which is written above the arrow, if present. We write $e \xrightarrow{n_s \bowtie n_d} e'$ to indicate that the expression e steps to expression e' and as a side-effect, data from service n_s is saved in service n_d. We write $e \xrightarrow{n_s \rhd n_d} e'$ to indicate that expression e steps to expression e' and service n_d discloses information from service n_s.

A service invocation may take the form of an Call, Send, or $\mathsf{Request}$ expression. As before, Call does not assume any information flow between the caller and callee. A Send expression represents the flow of information from the caller to the callee. The source of the flow is represented by the third parameter of the Send, which is an expression that should eventually step to a Data expression (by rule E-SendRed). For direct user invocations of Send, we use the current service's name as the data source (e.g. Send(n, u, n)). A Request expression represents an information flow from the callee to the caller. This expression should eventually step to a Data expression. In the event that the callee does not have a corresponding information flow to its caller, a special NoData expression is returned instead (rule E-ReqNoData).

Invocations of Call, Send, and Request all require an authorization check before evaluation of the service is performed. If these checks fail, evaluation stops with AuthFail (rules E-CallFail, E-SendFail, and E-ReqFail).

If authorization is successful, Call steps to Eval (rule E-CallEval). An Eval expression may step immediately to Done (rule E-EvalDone), duplicate itself (rule E-EvalSeq), or call other services (rules E-EvalCall, E-EvalSend, E-EvalMove, and E-EvalSave). The form of service invocation depends on the information flow graph and whether a given call is permitted for the user (based on the callee map M). If more than one invocation is possible, then one is chosen nondeterministically.

A Send expression steps to Recv upon successful authorization (rule E-SendRecv). A Recv expression may step immediately to Done (rule E-RecvDone), duplicate itself (rule E-RecvSeq), or step to Eval (rule E-RecvEval). If the information graph contains a caller-to-callee link, then the Recv may step to a Save of the incoming data (rule E-RecvSave). If the information graph contains a caller-to-callee link, then the Recv may step to a Send of the incoming data to the associated callee (rule E-RecvSend).

If a Send occurs within a service invocation, the associated data source expression is first reduced (rule E-SendRed). If this steps to a NoData expression, the Send steps directly to Done without invoking the target service (rule

E-SendNoData).

A Request expression, upon successful authorization, steps to an Eval followed by a Reply (rules E-ReqData and E-ReqReq), assuming the service has an information flow link terminating at the caller. If no such link is present, the Request steps to NoData (rule E-ReqNoData).

Information flow effects are represented using the Save and Reply expressions. First, the data source parameter must be reduced to a Data expression by rules E-SaveRed and E-ReplyRed. Then, Save reduces to Done and Reply reduces to Data, emitting an information flow effect — either a synchronization from the data source to the current service (Save, via rule E-SaveDone) or a disclosure of the data source by the current service (Reply, via rule E-ReplyDisc). If the data source expression reduces to NoData, then the enclosing Save or Reply reduces with no information flow effect (rules E-SaveNoData and E-ReplyNoData).

We write $\longrightarrow^*$ to represent the transitive closure of the $\longrightarrow$ relation. A *direct service invocation* is an expression of the forms $\mathsf{Call}(n, u)$, $\mathsf{Send}(n, u, n)$, or $\mathsf{Request}(N, u)$ representing the direct call of a service by a user.

Proposition 2 (Evaluation). *Let* Sys *be a well-formed system, n a service in* Sys*, and $u \in$* Users *a user. The evaluation via $\longrightarrow^*$ of a direct service invocation of the forms* $\mathsf{Call}(n, u)$ *or* $\mathsf{Send}(n, u, \mathsf{Data}\ n)$ *either diverges or eventually terminates with* Done *or* AuthFail*. The evaluation via $\longrightarrow^*$ of a direct service invocation* $\mathsf{Request}(n, u)$ *either diverges or terminates with* Done*,* AuthFail*,* Data n*, or* NoData*.*

4.5.2 Sufficiency

We now extend our definition of sufficiency to include Send and $\mathsf{Request}$ service invocations. We say that a direct service invocation $\mathsf{Call}(n, u)$, $\mathsf{Send}(n, u, \mathsf{Data}\ n)$, or $\mathsf{Request}(n, u)$ is *accepted* if $\mathsf{Svc}.n \in A_i$ for some $A_i \in \mathsf{Sys}$ and $\mathsf{Svc}.n.R \cap A_i.\mathsf{Perm}.u \neq \emptyset$, i.e., if it does not evaluate in one step to $\mathsf{AuthFail}$.

We say that a set of roles R is *sufficient* for Sys if, for every user u with $\mathsf{rolesOf}.u = R$, any direct service invocation in Sys by user u that is accepted does not evaluate, via $\longrightarrow$, to $\mathsf{AuthFail}$.

4.5.3 Non-disclosing Global Schema

Informally, we say that a global role schema Grs is *non-disclosing* for a conforming user assignment, if it does not permit the disclosure to a user $u \in \mathsf{Users}$ of data originating at a service S for which the user does not have access. This is the requirement that a user cannot subvert access control rules by exploiting information flow between services. To state this precisely with respect to our operational semantics, we define the following predicates (where $n, n' \in \mathsf{Names}$ are service names and $u \in \mathsf{Users}$ is a user). The predicate $\mathsf{Disclose}(n, n')$ is true if there exists a direct service invocation which, when evaluated via $\longrightarrow^*$, emits the information flow disclosure $n \rhd n'$. The predicate $\mathsf{Sync}(n, n', u)$ is true if there exists a direct service invocation which, when evaluated via $\longrightarrow^*$, emits the information flow synchronization $n \bowtie n'$. Finally, Flow is the reflexive and transitive closure of the union of $\mathsf{Disclose}$ and Sync: $\mathsf{Flow} = (\mathsf{Disclose} \cup \mathsf{Sync})^*$. If $\mathsf{Flow}(n, n')$, then there exists a sequence of direct service invocations which will result in the disclosure of data from n at service n'.

A global role schema Grs is *non-disclosing* if there does not exist services S

and S', a global role g, and a user u with $\mathsf{rolesOf}.u = G.g$ such that (a) role g does not have access to service S: $G.g \cap S.R = \emptyset$, (b) role g has access to service S': $G.g \cap S'.R \neq \emptyset$, and (c) $\mathsf{Flow}(S.n, S'.n)$ is true.

A *global information flow (GIF)* graph I_g for a system Sys is a directed graph constructed from the local information flow graph of each service. For each service S, the GIF graph has the set of nodes $S.C \cup \{S.n, S.\mathsf{Caller}_{in}, S.\mathsf{Caller}_{out}\}$, consisting of a node for each service called by S, a node $S.n$ for the service S itself. The set of all nodes in I_g is the disjoint union of the set of nodes for each service. To distinguish a node v from service S, we write $S.v$. For a service S, we create an edge $(S.v_1, S.v_2)$ if $(v_1, v_2) \in S.I$. For different services S and S' (with names n and n'), we create the following additional edges:

- S sends to S': there is a link $(S.n, S'.\mathsf{Caller}_{in})$ if $(n, n') \in S.I$ (service S sends data to S') and $(\mathsf{Caller}_{in}, v') \in S'.I$ for some v'.
- S' requests from S: there is a link $(S.\mathsf{Caller}_{out}, S'.v)$ if $(n, v) \in S'.I$ and there is a v' with $(v', \mathsf{Caller}_{out}) \in S.I$.

We can now define a static version of (dynamic) information flow, based on the global information flow graph: $\mathsf{StatFlow}(S.v, S'.v')$ is true if there exists a path in I_g from $S.v$ to $S'.\mathsf{Caller}_{out}$. This function is an over-approximation of Flow: $\mathsf{Flow}(n, n')$ implies $\mathsf{StatFlow}(n, n')$, but it is possible to have a path in the global information flow graph that is not feasible due to authorization errors. However, there is no loss in precision if the role schema is sufficient.

Theorem 18. *[Disclosure] Given a sufficient global schema* Grs *for system* Sys*, for any two services* S *and* S' *in* Sys*,* $\mathsf{Flow}(S.n, S'.n)$ *iff* $\mathsf{StatFlow}(S.n, S'.n)$*.*

To compute global role schemas that are non-disclosing, we conjoin additional constraints with ϕ_{Sys} to ensure only permitted information flow. For each pair of services S, S' such that $\mathsf{StatFlow}(S.n, S'.n)$ is true, we add the constraint:

$$(\bigvee_{r \in S.R} r^g) \rightarrow (\bigvee_{\hat{r} \in S'.R} \hat{r}^g).$$

We now modify the function constraint_pred, called from solve_full in algorithm 1, to include these extra information flow constraints. The resulting version of solve_full will infer non-disclosing global role schemas.

Theorem 19. *If the modified solve_full returns a global role schema* Grs *for* Sys*, then* Grs *has role separation, is non-disclosing, is fully sufficient, and is minimal with respect to the role signatures of* Sys*. If solve_full terminates with* `no_solution` *for* Sys*, then no such global role schema exists for* Sys*.*

System	Num svcs	Num calls	Num grps	Max pred	Time (ms)
portal1	8	5	N/A	96	4
portal2	9	8	5	92	5
data_sync	8	5	6	69	5
it_mgt	7	6	5	94	5

Table 4.1: Performance results for RoleMatcher

4.6 Experiences

To evaluate our approach, I have implemented RoleMatcher, a tool to infer global role schemas, and applied it to two case studies, one which models an industrial problem and a second which involves extracting role metadata from an electronic medical records application. RoleMatcher takes as input a textual representation of the `Sys` definition described in Section 4.3. It produces a global role schema via Algorithm 1, using the MiniSat [ES03] satisfiability solver to resolve the boolean constraints. Both fully sufficient and subset sufficient solutions may be obtained.

Table 4.1 summarizes the results of running our tool on several small examples, using a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. The “Num svcs” and “Num calls” columns represent the total number of services in the system description and the total number of service calls, respectively. “Num grps” lists the number of groups in the inferred schema (or N/A if no solution was possible). “Max pred” is the size of the largest predicate passed to the solver and “Time” the elapsed time in milliseconds. `portal1` and `portal2` correspond to the clinic web portal of Figure 4.2(a), `data_sync` corresponds to the data synchronization example of Figure 4.2(b), and `it_mgt` represents the case study described below. Since the problem is NP-complete, the use of an exponential procedure is inevitable. Even though the algorithm involves SAT solving,

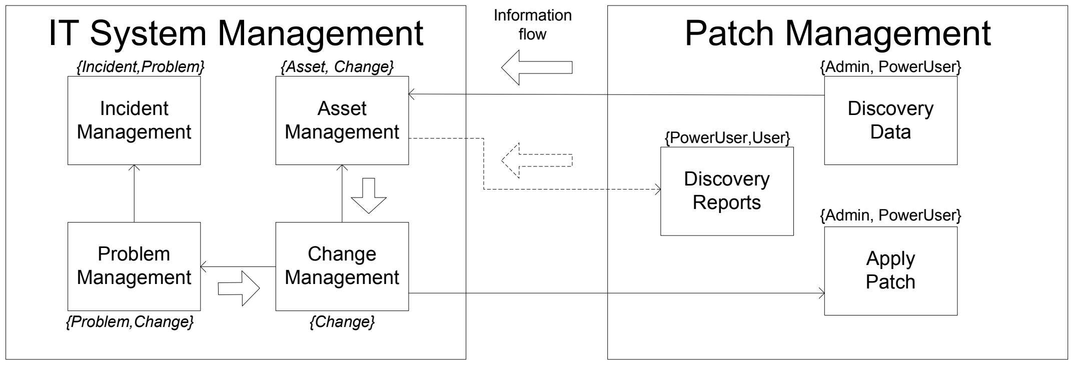


Figure 4.8: A role integration scenario

this has not been a bottleneck. This is because the constraints are a combination of 2-literal clauses (for separation) and Horn clauses (for sufficiency), and Boolean constraint propagation and unit resolution heuristics in a modern SAT solver are particularly tuned for these types of clauses.

4.6.1 Case Study: IT Management Applications

In my first case study, I considered a real-world scenario described to me by an industrial colleague. An IT management applications company had several independently-developed products from companies it had acquired. The company wished to integrate these applications (including their security models) in order for customers to use them as an end-to-end solution to their IT management needs.

Figure 4.8 shows a (simplified) view of how two such applications might interact. The *IT System Management* (ITSM) application includes modules for incident, problem, and change management, as well as a database to track a company's hardware and software assets. The *Patch Management* application gathers an inventory of the patches currently installed on the company's computers and manages the application of new patches. The integrations between these systems are straightforward: the patch inventory data should be included in the ITSM asset database (the arrow from *Discovery Data* to *Asset Management*) and the application of patches should be controlled via the ITSM change management module. (the arrow from *Change Management* to *Apply Patch*).

Both systems use role-based access control, but with very different role models. The Patch Management application has a very simple model with three fixed roles: *User*, *PowerUser*, and *Admin*. The ITSM application allows system administrators at the customer to define their own roles and mappings to data/service access permissions. The roles shown in Figure 4.8 for the ITSM application are only representative of a typical customer configuration. Thus, it is not feasible for the application vendor to ship a fixed role mapping. A better solution is to build a tool to extract role interfaces from the ITSM system's role metadata and infer a global role schema as a part of application deployment and configuration.

The role schema for a simple system definition, like the one in Figure 4.8, can be computed by hand. A quick inspection shows that the *Asset* and *Change* ITSM roles should be mapped to either the *Admin* or *PowerUser* patch management roles. However, a real system is more complicated. Customers can add new integration points between the two applications (e.g. the dotted line from *Asset Management* to *Discovery Reports*). Also, the asset database is likely to contain data from several discovery applications. Thus, I believe that such customers would benefit from the use of a global role inference tool.

4.6.2 Case Study: OpenMRS

In my second case study, I examined the issues involved in applying global schema inference to an open source application. OpenMRS [OPE] is a open source electronic medical records system created to support medical providers in the developing world. Implemented in Java, it provides a metadata-driven, role-based access control model, which is configurable through administration screens.

OpenMRS RBAC implementation The runtime configuration of OpenMRS access control policies is achieved by adding a layer of indirection, called *Privileges*, between roles and the objects to be protected by the access control infrastructure. To protect an object, checks for privileges must be inserted by the programmer into the relevant method calls (this is done using an Aspect-Oriented Programming framework). By convention, privileges are created for each combination of object type (e.g. patient, observation, order, etc.) and access operation (add, view, edit, and delete). Example privileges include *Add Patients*, *View Encounters*, and *Edit Orders*. The application database is then seeded with the list of privileges created by the Java programmers. From an administration screen, one can then create new roles and assign privileges to roles. Roles are arranged in a hierarchy (changeable by the administrator), with child roles inheriting all the privileges of their parents. Finally, when a new user is created, they are assigned a set of roles. This grants the user any privileges associated with their roles.

By default, OpenMRS comes configured with the following roles:

- *Anonymous* — this role represents an unauthenticated user and only has one privilege, to access the navigation and login screen.
- *Authenticated* — this role represents a user which has authenticated. However, it is not, by default, configured with any privileges.
- *Provider* — a provider has read-only access to the patient dashboard, encounters (a record of a patient visit), observations (notes from the doctor, test results), and orders (directions from the doctor).

- *System Developer* — developers have full access to the entire system. This role bypasses the usual privilege checks.

Note that, by default, OpenMRS does not contain a role to create and edit patient data. It is expected that each installation will define a customized role schema to implement access control policies for users who can create and edit patient data.[2]

Experiment configuration OpenMRS can receive patients, encounters, observations, and orders via HL7 messages [HL7], a standard format for medical record data. However, it does not come with any pre-configured integrations. To evaluate RoleMatcher with OpenMRS, I considered a simple system involving two instances of OpenMRS: *Main hospital*, a master system, and *Satellite clinic*, a slave system. Patient data originates at the main hospital and is sent to the satellite clinic via HL7. When patients are seen at the satellite clinic, new encounter, observation and order records are created, which are then uploaded to the main hospital system.

To support this scenario, I configured additional roles for both systems. For the main hospital, I created three new roles: *Doctor*, *Patient Admissions*, and *System Administrator*. The *Doctor* role has read/write privileges on patients, encounters, observations, and orders. The *Patient Admissions* role can read/write patients and encounters, but not observations or orders. The *System Administrator* role has all privileges and is for system administrators. For the satellite clinic, I created two new roles: *Provider2* and *all*. The *Provider2* role inherits from the default *Provider* role and adds the privileges to create and edit encounter, observation, and order records. Users with the *Provider2* role cannot create or edit

[2]New records can also be entered via offline forms (created using Microsoft's InfoPath), which are uploaded to OpenMRS as XML data.

patient records, since those are not uploaded back to the main hospital system. The *all* role is for system administrators and has all privileges.

Extracting a system description Programmatically extracting RBAC metadata from OpenMRS is straightforward. This is done by reading the metadata directly from the OpenMRS database. Mapping roles to our model requires that the role hierarchy be flattened — this is done by added to each role all the permissions of its (transitive) parent roles. Unfortunately, determining the services and information flow for OpenMRS is not as straightforward. Services and the interactions between systems are not described via metadata. To get a complete system definition for our analysis, one would have to build a static analysis of the OpenMRS source code. Although this type of static analysis is well-understood [BC85, PFF07], it is 1) not simple to implement, particularly since any analysis must take into account the transformations made by the Aspect-Oriented Programming framework used by OpenMRS, and 2) not currently the focus of our project. Instead, we looked at an alternative means of obtaining the data we need for our system representation.

High level model Rather than a full static analysis, we defined an abstract model of the application, capturing its key entities (patient, observation, order, etc.) as services and access methods (view, add, edit, deleted) as operations on these services. Each operation is associated with a specific permission (e.g. *view* on the *patients* entity is associated with the *View Patients* permission). Information flow is fixed, based on the type of access represented by each operation. For example the *view* operation on the *patient* service causes a flow from the service to its caller. The roles for each operation are determined by mapping the permission assigned to the operation to a set of roles, based on the role-to-permission mapping obtained from the application database.

Since no metadata is available to describe intra-application calls, this must be manually provided by an administrator. Descriptions of intra-application calls can be stored in a new table added to the application database and then extracted by the same tool which obtains the role metadata. This table stores an association between each local service and any remote services it calls, along with the information flow between these services.

Role inferencer changes This new model required a change to the design of RoleMatcher. In the model presented in Sections 4.3 – 4.5, the entry points of an application are *services*, and information is passed between these entry points only through explicit intra-service calls. This approach is insufficient to capture our desired model of OpenMRS: we want several entry points (e.g. view, edit, etc.) to share the same *self* node of the information flow graph, representing the service's persistent state. To accomplish this, I extended RoleMatcher to support *operations*. A service contains one or more operations, which are the entry points of the service. In the information flow graph, each operation has its own Caller_{in} and Caller_{out} nodes, but the operations of a service all share a common *self* node.

Example 6. *Figure 4.9 shows the information flow graph for a typical service S. This service supports four operations:* create*,* edit*,* view*, and* delete*. There are edges from the* Caller_{in} *nodes of the* create *and* edit *operations, representing an information flow from the caller to the service's persistent state. There is an edge from the self node to the* Caller_{out} *node of the* view *operation, representing an information flow out of the service. There is no information flow associated with the* delete *operation.* □

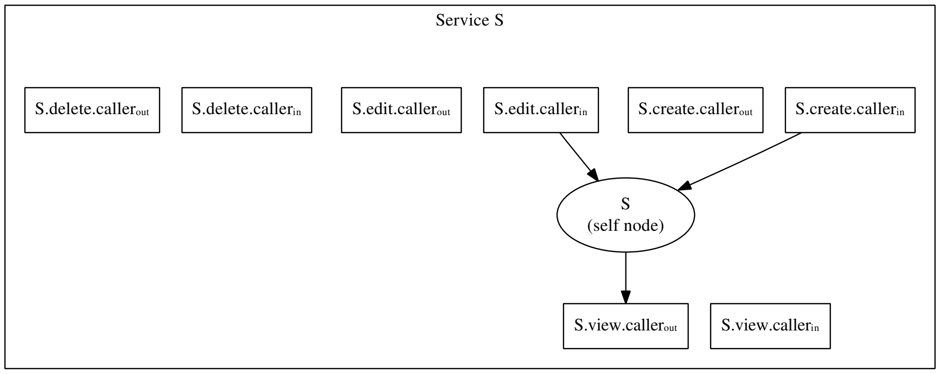


Figure 4.9: Information flow graph for a service with operations

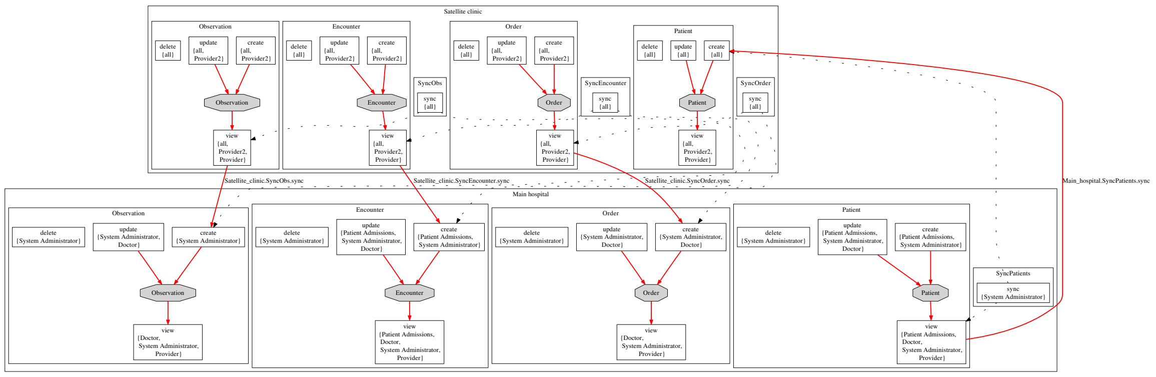

Satellite clinic
Observation
Encounter
Order
Patient
SyncObs
SyncEncounter
SyncOrder
Satellite_clinic.SyncObs.sync
Satellite_clinic.SyncEncounter.sync
Satellite_clinic.SyncOrder.sync
Main_hospital.SyncPatients.sync
Main hospital
SyncPatients

Figure 4.10: Graph of full OpenMRS case study

The introduction of operations also changes information flow constraints. In the extended model, if an operation discloses data from another service, each of its roles must be mapped to a role which discloses data directly at the originating service.

Results The extraction of RBAC metadata from each instance of OpenMRS takes less than a second. Figure 4.10 shows a program-generated visualization of the extracted model. Unfortunately, it is too large to fit on a single page without the details being unreadable. Figure 4.11 shows just the subgraph involving the

Global group name	**Local role members**
Doctor	Main hospital.Doctor Satellite clinic.Provider
Patient Admissions	Main hospital.Patient Admissions Satellite clinic.Provider
Provider	Main hospital.Provider Satellite clinic.Provider
System Administrator	Main hospital.System Administrator Satellite clinic.all
Satellite Provider	Main hospital.Provider Satellite clinic.Provider2
Hospital Anonymous	Main hospital.Anonymous
Clinic Anonymous	Satellite clinic.Anonymous
Hospital Authenticated	Main hospital.Authenticated
Clinic Authenticated	Satellite clinic.Authenticated

Table 4.2: Role mappings for OpenMRS case study

Patient services on the two applications. In these graphs, the outermost boxes represent the applications *Main hospital* and *Satellite clinic*. The boxes directly inside the outermost boxes represent the services (e.g. *Patient*, *Encounter*, etc.). Inside services, boxes represent operations and gray octagons represent the service's *self* node. Operation nodes are labeled with the operation's name and the set of roles which can access the operation. Arrows with dotted lines represent calls made by the source operation to the target operation. Solid arrows (red, if you are viewing a color version of this thesis) represent information flow from the arrow's source to its destination. In cases where the operation that causes the information flow is not obvious, information flow arrows are labeled with their associated operations.

From the patient subgraph of Figure 4.11, we can see the following:

1. The *SyncPatients* service, which copies patient data from the *Main hospital* application to the *Satellite clinic*, is accessible only to the *System*

Administrator role. To copy the data it calls the *create* operation of the *Satellite clinic*'s *Patient* service. This service is accessible only to the *all* role. Thus, to ensure sufficiency, any global group containing the *System Administrator* role must also contain the *all* role.

2. The *Patient* service of *Satellite clinic* discloses data from the *Patient* service of *Main hospital*. Each user which can access *Satellite clinic* patients should have similar access rights in the *Main hospital* application. Thus, each role associated with the *view* operation of the *Satellite clinic*'s *Patient* service (*all*, *Provider*, and *Provider2*) must be associated with a role which can access the *view* operation of the *Main hospital*'s *Patient* service (*System Administrator*, *Patient Admissions*, *Provider*, or *Doctor*).

In Table 4.2, we see the schema inferred for this case study by RoleMatcher. In order to satisfy sufficiency for the *SyncPatients* service, the *System Administrator* and *all* roles are placed in the same global group. In order to satisfy information flow constraints, the roles which can access patient data (the *Patient*, *Encounter*, *Observation*, and *Order* services), have been paired: each local role has been placed in a group also containing a role from the other system. Finally, the roles which are not involved in intra-application calls (*Anonymous* and *Authenticated*) are left in singleton groups, thus following the principle of least privilege.

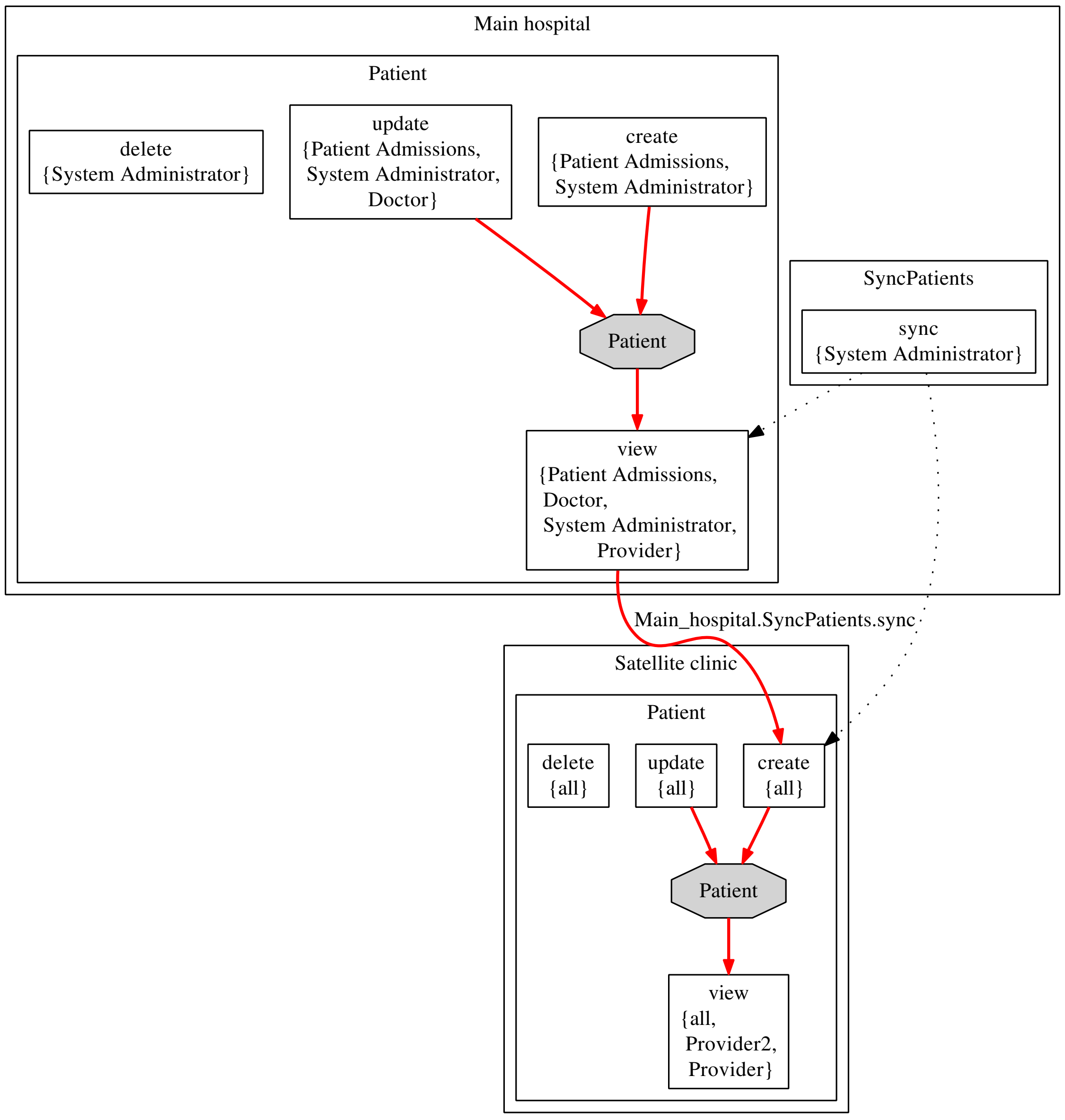


Figure 4.11: Patient subgraph of OpenMRS case study

4.7 Related work

4.7.1 Access control for web services

The eXtensible Access Control Markup Language (XACML) [ext05] defines an access control policy language for web services which is flexible enough to express many access control models, including RBAC [And04]. However, XACML policy definition and enforcement are not tied to the underlying access policies of individual services. Thus, while clearly a useful tool for defining security policies, XACML does not address the basic issues addressed by global role schema inference. XACML policies could be generated from a global role schema. This would enable centralized enforcement while avoiding the problems associated with two independent policy layers.

As an alternative, [SIM07] proposes an access policy language which can reference the past history of service invocations, using pure-past linear temporal logic (PPLTL). Like XACML, it uses a centralized approach to policy specification and enforcement. In order to reason about access policies across systems, the administrator must provide a role mapping. Thus, it can be used as a layer on top of global schema inference.

Other approaches to ensuring access control constraints are possible, e.g., in situations where multiple providers for a service are available, a broker can dynamically select among the available providers to satisfy security requirements [CFH06].

Other standards address orthogonal issues to access control: Security Assertions Markup Language (SAML) provides a framework for querying authentication and authorization statements across security domains, WS-Security defines how encryption and digital signatures may be applied to web service messages, and WS-Policy establishes a format for services to advertise their security requirements.

4.7.2 Theory of access control policy interoperation

The interoperability between ACL (Access Control List) based security models was first addressed from a theory perspective in [GQ96]. The desired interoperation between systems is specified as a set of accessibility links between principals of the individual systems and a set of restrictions between principals. It is shown that finding a maximal subset of the accessibility links, such that the security constraints of the individual systems are not violated, is NP-Complete. A similar result for the interoperation of partial order based security models is shown in [BSS96].

The Ismene policy language [MP02] uses a predicate-based notation for specifying general security policies. When a collection of entities wish to interact in a distributed system, their individual policies are *reconciled* to create a *policy instance* which governs the current session. Reconciliation involving finding a satisfying assignment to the conjunction of the individual policies. The access control policies implied by role interfaces could be represented in Ismene. However, the reconciliation of Ismene policies occurs at runtime, and the transitive nature of role composition makes runtime evaluation an impractical approach.

A related problem is *decentralized trust management* [BFL96], where credentials from independent systems are combined according to a well-defined policy. Trust management has been implemented using a role-based model in the policy language RT [LMW02, LWM03].

4.7.3 Relation to standard RBAC models

Three RBAC models, which form the basis of American National Standard 359-2004 for RBAC, are defined in [SCF96]. RBAC_0 models users, roles, permissions and sessions. The model described in this paper captures all of RBAC_0, except for sessions. A *session* is a map from a user to a subset of their roles and captures that idea that user need not activate all of their roles for a given task. Since the roles of a session are at the user's discretion, they are not relevant to the computation of global role schemas. However, if a session does not activate all of a users roles, indirect authorization errors can still occur. Thus, the role schema may be a useful guide to the user on which roles need to be activated to accomplish a given task.

RBAC_1 models a hierarchy of roles, where granting a role gives a user all the permissions associated with any role dominated by the granted role. While we do not currently include role hierarchies, it is straightforward to represent such a hierarchy in our boolean constraints and to extend the inference algorithm to find the *lowest* role in a hierarchy which satisfies a given constraint, keeping with the principle of least privilege. This is left as future work.

RBAC_2 extends RBAC_0 by adding constraints on the sets of roles assigned to users. These constraints include *mutual exclusion* constraints (e.g., a user may not be assigned roles A and B together), *prerequisite roles* (e.g., to have role B, a user must also be assigned role A), and *cardinality constraints* (e.g., a user may be assigned at most one role). Clearly, a global role schema should not violate any role assignment constraints. Our separation constraints are just a form of mutual exclusion constraint. It is trivial to extend our model to represent arbitrary boolean role constraints, including mutually exclusive and prerequisite roles. Cardinality constraints are less critical to global role schemas, as they mainly concern user-role and role-permission associations, rather than the relations between roles.

4.7.4 Role mapping

A role-based access control policy may be seen as an abstraction of an underlying ACL security policy. *Role mapping* [VAG07, ZRE07] attempts to find this abstraction automatically by finding a minimal set of roles for a single system which captures the underlying relationship between users and the resources they may access. In [VAG07], it is shown that this problem is NP-Complete. Algorithms for both full and approximate solutions are provided. Alternatively, roles may be constructed using data mining techniques which attempt to infer the functional responsibilities of users, based on patterns in a organization's access rights [KSS03].

Role mapping may be extended to address the interoperability of RBAC systems. In this context, it is assumed that an inter-system call will include the set of underlying permissions required on the target system. *Inter-domain role mapping* (IDRM) attempts to find the minimal set of target system roles which satisfy the requested permissions [PJ05, DJ06, CC07]. A static version of IDRM where roles are directly mapped between systems is presented in [BGJ05]. Links between roles are determined by grouping similar objects across systems (e.g., accounts, insurance claims, etc.) and then linking their underlying permissions (e.g., access to accounts on system A implies access to accounts on system B). Mappings between roles attempt to satisfy as many of these links as possible while avoiding the subversion of the individual systems' access policies. This problem is formulated as a system of integer programming constraints.

When inferring a global schema, we use each service interface's set of called services as a proxy for required access permissions. This is appropriate and necessary for systems whose internals are encapsulated by services. Our approach globally optimizes the role mappings to minimize the number of mappings needed while preserving interoperation.

4.7.5 Static analysis of RBAC systems

The Enterprise Security Policy Evaluator [PFF07] implements a static analysis for roles within a single Java Enterprise Edition application. This analysis checks for three types of errors: indirect authorization errors, redundant role definitions, and the subversion of an RBAC policy by exploiting unchecked intra-component calls. Our work can be viewed as extending these static checks across systems.

4.7.6 Information flow

In [ML97], a static program analysis is described which computes information flow in a modular fashion. All variables, arguments, and procedure return values are labeled with a lattice element. Each lattice element represents the set of owners for data flowing into a variable and the set of readers to which the variable may eventually flow. Distributed communication is modeled using *channels*, which are also labeled with their information flow properties. This work has inspired the use of decentralized information flow security in programming languages [ML00], operating systems [KYB07], and web service compositions [ONS07]. Rather than use information flow as *the* access control mechanism, we use information flow to inform a standard access control policy, leveraging the use of existing RBAC infrastructure. Information flow constraints can be viewed as an instantiation of the standard lattice model [Den76] by defining a lattice whose elements consist of sets of global roles, where the *top* element is the set of all global roles and the *bottom* element is the empty set. Each service is assigned a lattice element corresponding to the set of global roles by which it is accessible. If a service B discloses data from a service A, it must have an element equal to the lattice element computed for A or an element lower in the lattice. Information flow has also been studied in the context of RBAC. [Osb02] computes the information flow for a single system's RBAC policy due to two causes: 1) the ability to pass data between two objects protected by the same role, and 2) the ability to pass data between objects protected by different roles, when those roles may be simultaneously activated.

4.8 Recap

In this chapter, we looked at issues that occur when attempting the integration of disparate access control policies. Three specific issues were identified: indirect authorization errors, unintended disclosures due to data copying, and violations of the least privilege principle. I defined a new abstraction, the *global role schema* to capture relationships between local roles on different applications. By generating constraints based on the access control policies of each application and the calls between web services, a global role schema can be inferred which prevents the three problems I identified. This schema can then be used as a guide for administrators to systematically assign local roles to users. As shown in the OpenMRS case study, developing an adapter to automatically extract role interfaces from an application is straightforward. This work can then be amortized over all deployments of that application.

CHAPTER 5

Secure composition

5.1 Overview

Modern computing tasks of end-users involve the frequent use of customized web services and device applications, interconnected with each other through the sharing of computation results and user preferences. In practice, users have to customize each of these lifestyle software applications by hand, and enable sharing of computation results through some variation of the cut-and-paste operation. The problem is exacerbated on small form factor devices like mobile phones, where such frequent user manipulation is costly. For example, consider the number of steps required to plan a movie outing, perhaps involving the use of separate websites to find which movies are playing, read movie reviews, purchase tickets, and notify one's friends.

Software to address these issues has been slow in arriving, due to three problems. First, the specific tasks and objectives vary greatly from user to user. One-size-fits-all applications do not address user needs, and thus it is important for mobile applications to be configurable and composable. Second, security issues cause vendors to limit the access applications have to their platforms. In addition, software developers are frequently unclear about what security guarantees they should provide for lifestyle software and therefore employ ad hoc solutions. Third, mobile software platforms (e.g. Java Mobile Edition or BREW [Bre]) provide only low-level programming interfaces, which discourage casual programmers, and offer no clear notion of software composition. Easier-to-use, web-based technologies can be re-purposed for mobile applications, but such technologies do not provide access to device features, largely due to security issues.

I believe that this situation can be improved by providing a higher level programming model based on components that 1) have a simple, parameter-based mechanism for composition (permitting reuse and semi-automatic composition), and 2) externalize their information flow properties (permitting flexible security policies).

5.1.1 Existing approaches

Two recent categories of applications demonstrate how these three qualities can be used to improve user experience. *Mashups* combine multiple web applications into a single UI, correlating content from the constituent applications. For example, `packagemapper.com` superimposes the route taken by a user's package on a Google Map. This combines Google Maps with package tracking services from FedEx and UPS, along with a Yahoo service which converts addresses to latitude/longitude coordinates.

Widgets or *gadgets* [App, Goo, Wid] are small mini-applications that can be embedded into a larger web page or downloaded to a mobile phone. These applications usually provide a simple interface to a subset of an external website (e.g. BBC News, Wikipedia, Facebook, etc.). They can be customized by the user through a page of settings or parameters. In addition, most widget framework providers offer a website to facilitate the creation, sharing, and adapting of widgets.

Unfortunately, both mashups and widgets suffer from security issues which limit the types of applications deployed using these new approaches. To be useful, mashups and widgets must have the user's access rights to the underlying applications they rely on for content. However, the user has no control over what the mashup/widget does with those rights. In addition, these applications are built on top of existing web platforms, with limited device access and, in general, little support for application-level composition.

5.1.2 Monents

I propose to address these issues through a new class of application components, called *monents* (*mo*bile compo*nents*). Like widgets, monents provide simple user interfaces to external services. They can be customized through *settings* — persistent parameters adjustable by the end user. Like mashups, monents can correlate data from multiple websites/services. Unlike mashups, this capability is achieved without custom programming: monents provide a simple, automatic composition paradigm where data is exchanged through shared settings. Only the desired set of monents to be composed must be provided by the user — the rest is done by automatically matching inputs and outputs by name/type.

A monent is comprised of three elements: customizable wrappers around existing services, categorized settings for communicating data to other monents and to the environment in a controllable way, and a UI layer for customizable user interaction. The connections between these components are specified declaratively, and "glue" code is automatically generated to provide event-driven propagation of data changes.

Monents interact with the external (potentially unsafe) world through their settings and services, and all such interaction is controlled by a *security manager*. The security manager decides whether to activate the monent by evaluating the information flow between the monent's inputs and outputs, using a *security policy* that categorizes data sources and sinks according to a user-defined labeling.

Monents promote software reuse in several ways. First, the simple composition model encourages the building of applications from smaller components. Second, the external services of a monent are not hard-coded, and can be dynamically changed by the user. Finally, a monent can be used in more or less restrictive environments, depending on a user's level of trust in the services interacting with the monent.

5.1.3 Information flow interfaces

I model the composition and security of monents through *information flow interfaces*. This formalism describes components as a collection of inputs, outputs, and an information flow relation. My model is not specific to monents, but can be applied to any component system where inter-component interactions can be controlled and the information flow within a component externalized. This first class, declarative treatment of information flow is novel for component models.

Using information flow interfaces, one can represent both the internal connections within a single monent, as well as monent composition. I use an optimistic approach to composition: each input and output is matched independently, and, if a composition does not provide an value for an input, the environment is assumed to provide the value. Of course, this value may be later provided through a subsequent composition. Security policies are evaluated with respect to a monent's runtime environment. Thus, the same component may satisfy the security policy of a permissive environment but not of a more restrictive environment. Execution in a restrictive environment is analogous to sandboxing, but our model permits fine-grained control over permitted interactions.

My composition algorithm respects the intent of security policies. In particular, I demonstrate formally that the security policy of a single component has the same effect when the component is used in a composition, and rights are not granted or lost through composition.

5.1.4 Chapter organization

In Section 5.2, I informally present monents, monent composition, and my approach to security. Then, Sections 5.4 and 5.5 formally define information flow interfaces, describe the properties I wish to guarantee with respect to this model, and show how monents can be represented as information flow components. In Section 5.6, I describe a prototype implementation of the monent framework which includes a compiler, a monent composer, runtime infrastructure, and a server-side monent sharing application. My compiler accepts a declarative description of a monent's services, settings, and UI components. From these, it generates an Adobe Flash$^{\text{TM}}$ application. The composer accepts the source descriptions of two or more monents and creates a combined monent. Finally, I compare this work to existing research and frameworks in Section 5.7.

5.2 Monents

5.2.1 Motivating example

To motivate the architecture for monents, consider a scenario where a user wants to schedule a movie outing via her mobile phone. To do so, she might look up which movies are playing, read movie reviews for a subset, pick a movie and showtime, buy the tickets, add an entry for the movie to her calendar, and email the details to her friends. Each of those six sub-tasks might involve several

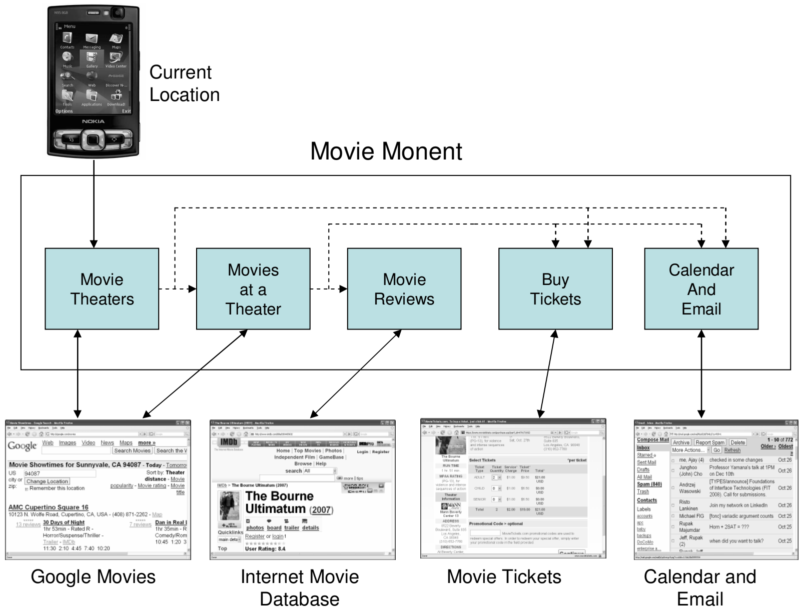


Figure 5.1: The movie monent

interactions (clicks, entries) at a website or application. She might very well lose patience and give up on the exercise altogether.

To address this problem, a monent should provide a task-focused UI which avoids unnecessary levels of navigation. Context should be used to further economize the user interactions. For example, once our user selects a theater and movie at one website, she should not have to re-select the theater and movie at each subsequent site. Our user has some security expectations for this monent: she must be confident that these sites are not malicious and that the monent will not pass her credit card number to the other websites. In addition, she must be confident that the monent will not spam everyone in her address book.

Figure 5.1 shows how this monent might work. The monent is composed of five logical sub-applications, which may be monents themselves. Each of these sub-applications represents a step involved in completing the movie outing task and corresponds to an interaction with a website or service. Due to limits on the mobile device display size, only one step is presented to the user at a time. Navigation between sub-applications is accomplished using a tab metaphor: each sub-application has a tab at the top of the display and selecting an application's tab causes it to be displayed.

Solid arrows in figure 5.1 represent the links between the logical sub-applications and external services. Dashed arrows represent the passing of context between sub-applications. The user-selected movie theater is passed to the movie selection, ticket purchase, and calendar/email sub-applications. The selected movie and show time is passed to the movie review, ticket purchase, and calendar/email sub-applications.

To see how this movie monent can be built, we first look in more detail at the three elements of monents: services, settings, and UI controls.

5.2.2 Elements of a monent

Services A *service* represents a callable entity which provides content for a monent. Services may include external websites, web services, or device features (hardware or applications) that have been wrapped in a service interface. A monent may contain an arbitrary number of service clients/wrappers. In our movie example, each external website will be represented as a service, along with the user's email application. Like websites, a service is identified through its URL. We divide this URL into two portions: a *location*, which is evaluated by the security policy, followed by a *query*, which contains data specific to a particular request. This division is specific to each website and is decided by the service developer based on the URL structure of the website and the granularity of security policies needed for a given site. The result of a service call is structured content, such as XML, HTML, or JSON. We will refer to a monent's service clients/wrappers simply as “services”.

Settings *Settings* are key/value pairs used to facilitate communication within and between monents. In our movie example, we will use settings for the current location (e.g. a zip code or latitude/longitude), the selected theater/movie/showtime, and the user’s credit card information. Each setting is either an *input*, an *output*, or *internal* to the monent – the categorization corresponds to how its values can be set and shared. Input settings can obtain their values from direct user input, other monents, or the *environment*. The environment is a collection of settings representing information available from the local device (e.g. location or personal information) or enclosing application context (e.g. bookmarks or email recipients).

Output and internal settings obtain their values from services or UI controls within the monent. The values of output settings are made available to other monents.

UI Controls The user interface controls provided for monent developers are designed to allow the quick implementation of simple front-ends to external content. The supported controls include labels, buttons, text input boxes, data grids (which display record-like content from a service), and tabs. Other controls can be easily accommodated.

5.2.3 Connecting monent elements

The connections between the various elements of a monent are specified declaratively. Settings can obtain their values from the data in UI controls or by extracting a value from the result of a service call. Likewise, UI controls can obtain their

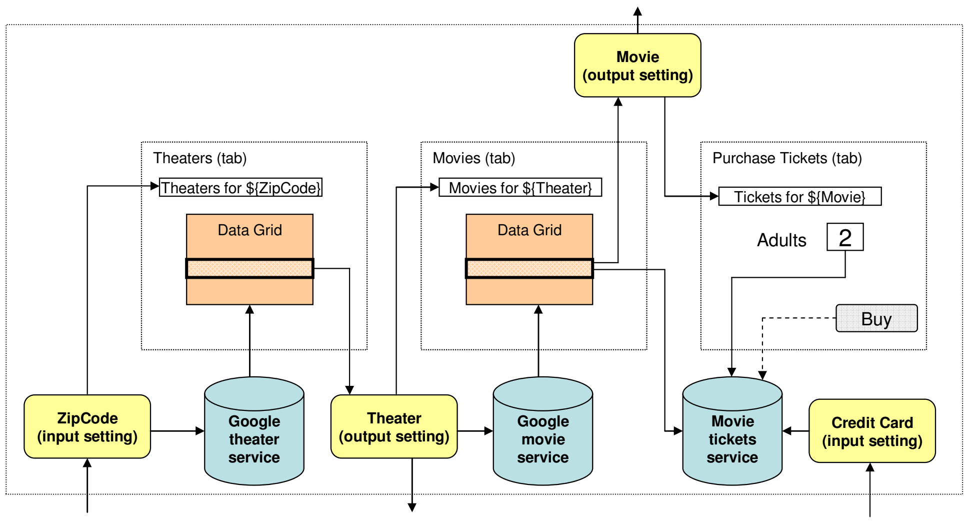


Figure 5.2: Connections between elements of the movie monent

values from settings, services, and other UI controls. For example, if the user's zip code is an input setting, then a UI label might include the value of this setting as follows: `Current movies for zip code ${ZipCode}`. The `${ZipCode}` string is replaced with the value of the ZipCode setting.

The location and query of a monent's service can be parameterized by settings or values from UI controls, using a similar template notation. For example, the service which calls Google Movies to obtain the listings for a specific zip code might use a location of `google.com/movies` (for Google Movies, this part of the URL remains constant and is evaluated as a data sink in the security policy) and a query of `?near=${ZipCode}` (this part of the URL changes with each request and is not considered in the security evaluation). When making the request, the service first replaces the `${ZipCode}` string with the value of the ZipCode setting and then concatenates the location and query to obtain a URL. Thus, given a zip code of 94304, the monent would send a request to `google.com/movies?near=94304`.

Example 7. *Figure 5.2 shows element interconnections for a subset of our movie monent. We use arrows to show the flow of data between elements. The* ZipCode *setting is used by the* Google Theater *service and by a UI label on the* Theaters *tab. The results of the* Google Theater *service are then displayed by a data grid on the same tab. Each record displays the name and address of a particular theater. The user-selected theater's name is stored in the* Theater *output setting. This setting is used by the* Google Movie *service to build a second query, which retrieves movies playing at the specified theater. In addition, the* Theater *setting is included in a UI label on the* Movies *tab. The results returned by the* Google Movie *service are displayed using a second data grid. The name of the selected movie is output via the* Movie *output setting.*

The details of the selected movie are passed to the Movie Tickets *service, along with the quantity to be purchased, which is input from the* Purchase tickets *tab. The credit card number for the purchase is passed from an input setting to the service. The* Movie Tickets *service is not actually called until the user presses the "Buy" button.*

□

5.2.3.1 Responding to changes

If the user happens to change the ZipCode input setting or the selected theater, we would like the change to be reflected in our monent as well. The monent framework assumes that any setting, service result, or user-selection may change. To address this, it tracks the connection dependencies between monent elements and automatically generates the event-driven code to refresh dependent elements when a change occurs. For, example, if the user selects a different theater, the THEATER setting is changed and any previously selected movie is invalidated.

In some situations (e.g. making a purchase), we do not wish to initiate a service request without explicit input from the user. To address this situation, a UI button control may be provided as the only connection to a service. If a button is connected to the service, the call is not initiated until the button is pressed and the service has valid inputs.

5.2.4 Monent composition

Rather than build a single monent which interacts with multiple services/websites, we can build several smaller monents and then *compose* them into the monent we need. For example, each tab of the movie monent of Figure 5.1 could be implemented as a separate monent. If monents are small and general enough, they can be reused in new contexts. Here are some examples of reuse involving the components of our movie monent:

- Perhaps we have another website we prefer to use to select movies. If we build monents to select movies for that site, we can use them with our existing movie review and ticket purchase monents.
- The calendar monent could be used with any other monents providing a date, time, and location.
- A map and directions monent could be composed with the movie monent to provide directions to the theater.

Implicit linking Composition should be easy to understand: we would like end users to compose their own monents. In addition, if composition is to occur on mobile devices, it must require only simple UI actions. Ideally, users only need to select the set of monents they wish to compose.

To achieve these goals, we implicitly link the input and output settings of monents, matching by name and datatype. For example, if one monent provides a THEATER output setting of type `string` and another monent provides a THEATER input setting of type `string`, these settings are connected in the composite monent. In addition, THEATER is no longer an input in the composite monent, as it obtains its value internally. In the event that two monents have incompatible settings (e.g. two outputs of the same name or an input/output pair with the same name but different data types), the appropriate elements are kept separate and renamed to avoid a conflict.

The individual UI controls and services of the source monents are not linked directly. Since monent user interfaces are organized by tabbed navigation, user interfaces can be combined by simply taking the union of the individual tab sets.

This implicit approach to composition requires some planning and forethought in the design of monent "libraries" in order to ensure reusability. We lose some flexibility to gain a simplified user interface for composition, which is important for small form-factor devices. Future work will look at ways to better address name mismatches and conflicts in composition, perhaps through renaming or scoping operations (cf. Section 6.4).

Static versus dynamic composition A set of monents can be composed statically, at compile time, or dynamically, at runtime. We have chosen to implement static composition, as it does not require a dynamic linking mechanism or runtime loading of code. Dynamic composition of monents should certainly be possible as well.

5.2.5 Security model

Monents interact with the external world through settings and services. Monents cannot deny service through the UI or by consuming too much CPU: they are composed from trusted components which are only invoked in response to a user action. Thus, the primary security issues of monents are their interactions with the outside world. To address such issues, we implement a security model with three components: an *information flow analysis*, a *security policy*, and the *security manager*.

5.2.6 Information flow

At compile time, an information flow analysis is performed, and the results summarized in a *flow relation*, which is compiled into the monent. This flow relation associates each input with the possible outputs to which incoming data may flow. The inputs of a monent include its input settings and the responses from service calls. The outputs of a monent are its output settings and service requests.

5.2.7 Security policies

A security policy independently assigns *tags* to the inputs and outputs of a monent. These tags categorize data according to its sensitivity and intended recipients, and should have an intuitive meaning to users (e.g. categories like `public` and `financial`). A partial ordering is defined between tags, based on relative sensitivity and size of a category. For example, we write `financial` $\sqsubseteq$ `public` to say that the tag `financial` appears before `public` in our ordering, and represents more sensitive data. Any data that can comes from a source labeled `public` can be passed to a destination labeled `financial`, but the converse is not true.

Special tags `nobody` and `everyone` are defined to represent the empty set of categories and the set of all categories, respectively. In addition, we allow users to associate a collection of tags, not just a single tag, with an input or output.

To specify a security policy, users can manually apply tags to services and settings. This gives them the most control over the operation of their monents. Alternatively, *reputation services* can return tags for services based on white- or black- listing. One might imagine other approaches to defining security policies. For example, services might return their own tags, cryptographically signed by a trusted third-party authority, or a rule based policy language could be used to determine tags, based on a combination of the other approaches.

Note that users do not have to understand the mathematics behind our security model to define and use security policies. The tag-based model should be no more difficult for users to understand than, say, the hierarchical expense categories of Intuit's *Quicken*® home accounting application.

5.2.8 Security manager

The security manager is a run-time infrastructure component which evaluates the security policy with respect to the monent's flow relation. If (according to the flow relation) the monent never permits data to flow beyond its intended scope (as defined by the security policy), the security manager activates the monent. Otherwise, it leaves the monent disabled and pops up an error message to the user.

By separating the tagging of a monent from the compile-time information flow analysis, monents can be reused in different security contexts. This allows users to personalize their security policies. For example, a mobile service provider might define a default security policy which is not very permissive. Users can then adjust the policy to suit their needs, based on their understanding of the security framework and their trust of different websites and people.

In addition, a security policy can be reused across different monents. Thus, a user can have a global policy for all monents running on their device.

5.3 Security and composition

One should not be able to circumvent a monent's security policy by composing it with other monents. Since the monents of a composition can only communicate through shared settings, this cannot happen. Given a collection of monents which satisfy a security policy, the composition of these monents will also satisfy the policy. In addition, the composition of monent that violates a policy with other monents will always violate the original policy. These properties are presented more formally in section 5.5.3.

5.4 Information Flow Interfaces

To describe monents and their properties precisely, I define a formal model called *information flow interfaces.*

An *information flow interface* $\mathcal{C}$ is a triple $(\mathbb{I}, \mathbb{O}, F)$, where $\mathbb{I}$ is a set of input identifiers, $\mathbb{O}$ is a set of output identifiers, and F is a relation over input-output pairs. $F(i, o)$ is true if there exists an information flow from input i to output o. We write $\mathcal{C}.\mathbb{I}$, $\mathcal{C}.\mathbb{O}$, and $\mathcal{C}.F$ to refer to the input, output and flow relation components of interface $\mathcal{C}$.

Identifiers are opaque in the abstract model, but may be compared using the equality relation $=$. Each identifier in $\mathbb{I} \cup \mathbb{O}$ must be unique.

5.4.0.1 Dynamic behavior

The components described by information flow interfaces are opaque: we cannot see into their implementation. The inputs and outputs of a component may be viewed as named channels, in the sense of the π-calculus [Mil99]. A component may receive messages on any of its inputs, store them, and forward them to any output channel, subject to the restrictions imposed by the information flow relation F. In other words, a message originating from input i can be sent to any output o for which $F(i, o)$ is true. Components may also drop messages or create new messages, which can be sent on any output. We can implement this model in the π-calculus by using disjoint namespaces for variable names and channel names. As a result, the set of channels is static and the information flow relation can be computed via a syntactic analysis.

Note that, in a real system, messages may also be transformed and combined. However, it is not necessary to model changes to messages to capture the information flow properties of a component. For example, if messages from inputs i_1 and i_2 are combined and sent to output o_3, then we model this as two paths in the flow relation: $F(i_1, o_3) = F(i_2, o_3) = \texttt{true}$.

5.4.1 Composition

The function $\mathsf{compose}(\mathbb{C}, \mathbb{O}_h)$ *composes* a set of information flow interfaces $\mathbb{C}$, given a set of hidden outputs $\mathbb{O}_h \subseteq \bigcup(\mathcal{C}_i.\mathbb{O} | \mathcal{C}_i \in \mathbb{C})$. We say that a composition is *well-formed* if each output identifier in the composition is unique. The result of a composition will be a new information flow interface $\mathcal{C}_c$, which describes the information flow for the interconnected components.

Topology graph We connect the inputs and outputs of component interfaces by matching on identifiers. More precisely, the semantics of a composition are defined via a *component topology graph*, which describes the interconnections of the combined interfaces. The component topology graph $G_c = (\mathbb{V}_c, \mathbb{E}_c)$ contains a vertex $v \in \mathbb{V}_c$ for each interface $c \in \mathbb{C}$ and a set of labeled, directed edges $\mathbb{E}_c$. We write $\mathbb{C}(v)$ as a shorthand for the interface $\mathcal{C} \in \mathbb{C}$ associated with vertex $v \in \mathbb{V}_c$. An edge $e \in \mathbb{E}_c$ from vertex v_1 to vertex v_2 and labeled with l is created if there exists an $o_1 \in \mathbb{C}(v_1).\mathbb{O}$ and an $i_2 \in \mathbb{C}(v_2).\mathbb{I}$ where $o_1 = i_2 = l$.

The dynamic behavior of a composition is defined by considering each edge labeled l to be a named channel l. We use a broadcast semantics — if there are multiple outgoing edges from an interface's vertex with the same label, then each message is sent on all the edges.

We say that there is an *information flow path* in G_c from input $i \in \mathbb{I}$ to output $o \in \mathbb{O}$ if there exists a sequence of identifiers $l_1 ... l_n$ such that:

1. $l_1 = i$ and there is a vertex $v \in G_c$ such that $i \in \mathbb{C}(v).\mathbb{I}$.

2. $l_n = o$ and there is a vertex $v \in G_c$ such that $o \in \mathbb{C}(v).\mathbb{O}$.

3. For each consecutive pair (l_i, l_{i+1}) in the identifier sequence, there is a vertex $v \in G_c$ such that $l_i \in \mathbb{C}(v).\mathbb{I}$, $l_{i+1} \in \mathbb{C}(v).\mathbb{O}$, and $\mathbb{C}(v).flow(l_i, l_{i+1}) = \texttt{true}$.

Proposition 3. *Given a composition* $\mathsf{compose}(\mathbb{C}, \mathbb{O}_h)$ *and its topology graph* G_c*, the composition permits an information flow from an input* i *to an output* o *if and only if there exists an information flow path from* i *to* o *in* G_c.

Information flow graph From a component topology graph G_c, we can construct an *information flow graph* $G_f = (\mathbb{V}_f, \mathbb{E}_f)$, representing the information flow between component inputs and outputs. First, we create a vertex for each input and output of each interface. We refer to these vertices as $\mathcal{C}.n$, where $\mathcal{C}$ is the associated interface and n the name of the input or output. We create edges for both intra- and inter-component information flow. For each component $\mathcal{C}$, we create an edge from $\mathcal{C}.i$ to $\mathcal{C}.o$ if $\mathcal{C}.F(i, o)$ is true. For each edge in the component graph labeled l from v_{c_1} to v_{c_2}, we create an edge in the information flow graph from $\mathcal{C}(v_{c_1}).l$ to $\mathcal{C}(v_{c_2}).l$. Finally, we take the transitive closure of the edge relation: if there is an edge from $\mathcal{C}_1.n_1$ to $\mathcal{C}_2.n_2$ and an edge from $\mathcal{C}_2.n_2$ to $\mathcal{C}_3.n_3$, we add an edge from $\mathcal{C}_1.n_1$ to $\mathcal{C}_3.n_3$, if one does not already exist.

Proposition 4. *Given a composition* $\mathsf{compose}(\mathbb{C}, \mathbb{O}_h)$ *with a topology graph* G_c *and an information flow graph* G_f*, there exists an edge from* $\mathcal{C}.i$ *to* $\mathcal{C}'.o$ *in* G_f *if and only if there is an information flow path from* i *to* o *in* G_c*.*

We can now define a new information flow interface $\mathcal{C}_c = (\mathbb{I}_c, \mathbb{O}_c, F_c)$ resulting from composition $\mathsf{compose}(\mathbb{C}, \mathbb{O}_h)$ as follows:

- $\mathbb{I}_c$ is the union of the inputs of the constituent interfaces, with any inputs removed that are also referenced as outputs in the constituent interfaces: $\bigcup(i | i \in \mathcal{C}.\mathbb{I} \ \wedge \ \mathcal{C} \in \mathbb{C}) \setminus \bigcup(o | o \in \mathcal{C}.\mathbb{O} \ \wedge \ \mathcal{C} \in \mathbb{C})$
- $\mathbb{O}_c$ is the union of the outputs of the constituent interfaces, with any outputs removed that are in the hidden set $\mathbb{O}_h$: $\bigcup(o | o \in \mathcal{C}.\mathbb{O} \ \wedge \ \mathcal{C} \in \mathbb{C}) \setminus \mathbb{O}_h$
- The relation F_c, defined over $\mathbb{I}_c \times \mathbb{O}_c$, is true for (i, o) if, in the information flow graph G_f for our composition, there exists an edge from $\mathcal{C}.i$ to $\mathcal{C}'.o$ for some pair of interfaces $\mathcal{C}, \mathcal{C}'$ in the composition.

Proposition 5 (Information flow)**.** *Given a composition* $(\mathbb{I}_c, \mathbb{O}_c, F_c) = \mathsf{compose}(\mathbb{C}, \mathbb{O}_h)$*, the relation* $F_c(i, o)$ *is true if and only if a message arriving at input* i *may flow to output* o*, given the assumptions of the individual flow relations.*

In other words, the composite flow relation F_c describes the possible information flows of the overall system.

Proposition 6. *Given a well-formed set of interfaces $\mathbb{C}$, composition of these interfaces is associative and commutative, providing that no outputs are hidden until the last composition of any sequence.*

As a consequence of Proposition 5, composing information flow interfaces in any order will result in the same final component. However, this is not always true if an output is hidden, as subsequent compositions will leave an input of the same name unconnected.

5.4.2 Security

We wish to ensure that components do not permit sensitive data to reach unintended recipients. We can use an interface's flow relation to track the potential flow of data. However, we need a means of deciding whether a given flow should be allowed. To do so, we model the sensitivity of data using a partially ordered set $PO = (\Theta, \top, \bot, \sqsubseteq)$, where Θ represents the elements of our partial order, $\top$ represents the last (least sensitive) element of our order, $\bot$ the first (most sensitive) element of our order, and $\sqsubseteq$ is an ordering relation. If $\theta_1 \sqsubseteq \theta_2$ is true for some $\theta_1, \theta_2 \in \Theta$, then θ_1 is equivalent to or more sensitive than θ_2. The exact meaning of sensitivity is defined by each instantiation of our framework.

To specify the security requirements for an information flow interface, one provides a *security policy* $\mathbb{P} = (PO, \mathbb{T})$, where PO is a partial order and $\mathbb{T} : \mathbb{I} \cup \mathbb{O} \rightarrow \Theta$ is a mapping from an interface's input and output identifiers to elements of the partial order. To evaluate the security policy, one uses the interface's information flow relation F. Data should never flow from a more sensitive input to a less sensitive output. We state this more formally as follows:

Definition 5 (Soundness)**.** *We say that an information flow interface* $(\mathbb{I}, \mathbb{O}, F)$ *is* sound *with respect to a security policy* $(PO, \mathbb{T})$ *if, whenever* $F(i, o)$ *is* ***`true`****,* *then* $\mathbb{T}(o) \sqsubseteq \mathbb{T}(i)$*.*

5.4.2.1 Security and composition

Composing information flow interfaces should not change their security properties. We can explore this by considering an interface set $\mathbb{C}$ and a security policy $\mathbb{P}$. We assume that the policy's mapping $\mathbb{T}$ is total with respect to the inputs and outputs of the components. If not, we can *extend* a policy by assigning $\perp$ to every unmapped input and $\top$ to every unmapped output.

Proposition 7. *If a set of information flow interfaces* $\mathbb{C}$ *are individually sound with respect to a security policy* $\mathbb{P}$*, then the composition of these interfaces will also be sound.*

We prove this by contradiction. For the composition to be unsound when the individual interfaces are sound, there must be an identifier pair (l_1, l_n) such that $F(l_1, l_n)$ is true in the composition and $\mathbb{T}(l_1) \sqsubseteq \mathbb{T}(l_n) =$ `false` but $F(l_1, l_n)$ is false for each individual interface. This requires a new information flow path in the graph starting at l_1 and ending at l_n. An arbitrary sequence of two identifiers on the path (l_i, l_{i_1}) must have $F(l_i, l_{i+1}) =$ `true` for some component in the composition. However, if that interface is sound, then we know that $\mathbb{T}(l_i) \sqsubseteq \mathbb{T}(l_{i_1}) =$ `true`. By induction, this must also be true for the endpoints: $\mathbb{T}(l_1) \sqsubseteq \mathbb{T}(l_{i+1}) =$ `true`. This contradicts our original assumption that $\mathbb{T}(l_1) \sqsubseteq \mathbb{T}(l_{i+1}) =$ `false`.

Proposition 8. *Given a set of information flow interfaces* $\mathbb{C}$*, which are in-*

dividually sound with respect to a security policy $\mathbb{P}$*, and a second set* $\mathbb{C}'$ *of interfaces which are not individually sound with respect to* $\mathbb{P}$*, the composition* $\mathsf{compose}(\mathbb{C} \cup \mathbb{C}', \emptyset)$ *is not sound with respect to* $\mathbb{P}$*.*

If an interface $\mathcal{C}_i$ is unsound, then there exists an input i and output o such that $F_i(i, o) = \texttt{true}$ and $\mathbb{T}(i) \sqsubseteq \mathbb{T}(o) = \texttt{false}$. If no outputs are hidden in the composition, then $F(i, o) = \texttt{true}$ for the composition's flow relation as well.

Note that we can make a composition sound by hiding the offending outputs. This suggests three approaches for handling a security policy violation:

1. we can disallow the entire composition,
2. we can exclude any components which individually fail the soundness check, or
3. we can hide any offending outputs and allow the composition (potentially with reduced functionality).

5.5 Representing Monents

In this section, we look at how to represent monents as information flow interfaces. By doing this, we can obtain all the guarantees provided by Propositions 5 through 8 of Section 5.4.

5.5.1 Monent elements

To represent an individual monent, we define an information flow interface for each monent element. As shown in Figure 5.3, we can create information flow interfaces for settings (both input and output), services, and UI controls (label, text input, and data grid). In the graphical depiction, the arrows into

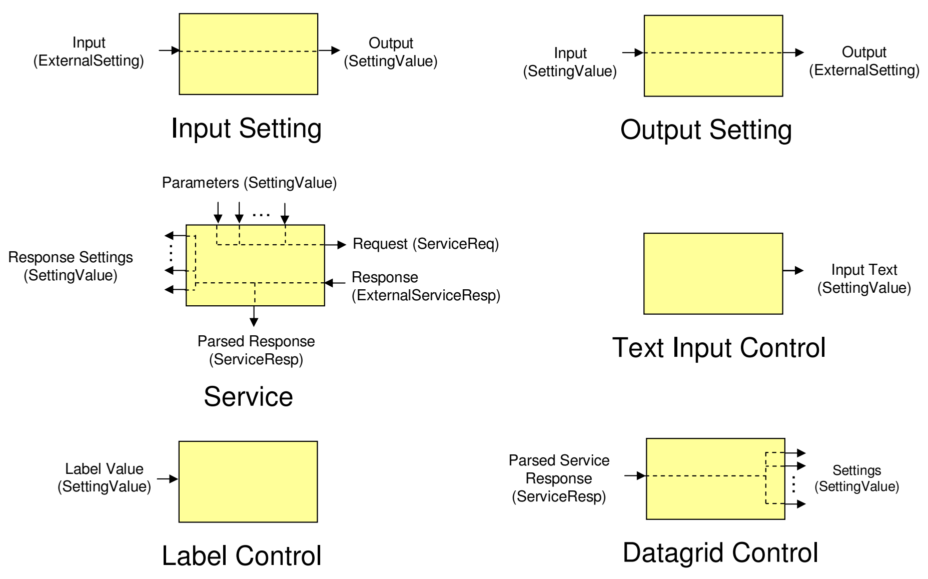


Figure 5.3: Modeling the elements of a monent

a box represent inputs, the arrows out of a box represent outputs, and the dashed lines represent information flow from inputs to outputs. Identifiers have the form (k, t, n), where k is a *kind*, t is a datatype (e.g. string, integer, zip-code), and n is a name. Kinds are used to encode rules about which inputs and outputs may be interconnected. Valid kinds include `ExternalSetting`, `SettingValue`, `ServiceReq`, `ServiceResp`, and `ExternalServiceResp`. When the kind is `ExternalSetting`, the name n represents the name of a monent setting. When the kind is `ServiceReq` or `ExternalServiceResp`, the name represents the URL of a service.

We now look in more detail at the individual element types of Figure 5.3. An input setting has a single input of kind `ExternalSetting` and a single output of kind `SettingValue`. Output settings are a mirror of inputs. This ensures that only output settings may connect to an input setting. Services take as their inputs one or more setting values. These values are used to form a request, of kind `ServiceReq`. The response, of kind `ExternalServiceResp`, enters the service and is parsed, resulting in a parsed response, of kind `ServiceResp`, and zero or more response settings, of kind `SettingValue`. Text input controls have a single output, whose value comes from the user. Label controls have a single input and display this value to the user. Finally, data grid controls accept a parsed service response as input and, based on a user's record selection, provide one or more settings, of kind `SettingValue`, as output.

Give a set of element interfaces $\mathbb{E}$, we can build a monent by taking the composition $\mathsf{compose}(\mathbb{E}, \mathsf{hide}(\mathbb{E}))$, where $\mathsf{hide}(\mathbb{E})$ is a function which returns the output identifiers for $\mathbb{E}$ which have kinds of either `SettingValue` or `ServiceResp`. This has the effect of hiding the internal connections of a monent. We say that a monent is *well-formed* if the following conditions are met:

1. Each output identifier in the composition is unique (inherited from the requirements for well-formed information flow interfaces).

2. All element inputs of kinds `SettingValue` and `ServiceResp` have corresponding outputs. This ensures that these inputs are connected internally within the monent and are not visible outside the composition.

3. The request output and response input of each service element must have identifiers with matching names — identifiers of the forms $(\texttt{ServiceReq}, d, n)$ and $(\texttt{ExternalServiceResp}, d', n)$, respectively. This ensures the request and response correspond to the same external web service. In addition, it restricts monents to have only one service element for a given URL.

Proposition 9 (Monent information flow)**.** *Given a set of monent elements* $\mathbb{E}$*, if the composition* $(\mathbb{I}_e, \mathbb{O}_e, F_e) = \mathsf{compose}(\mathbb{E}, \mathsf{hide}(\mathbb{E}))$ *is well formed, then:*

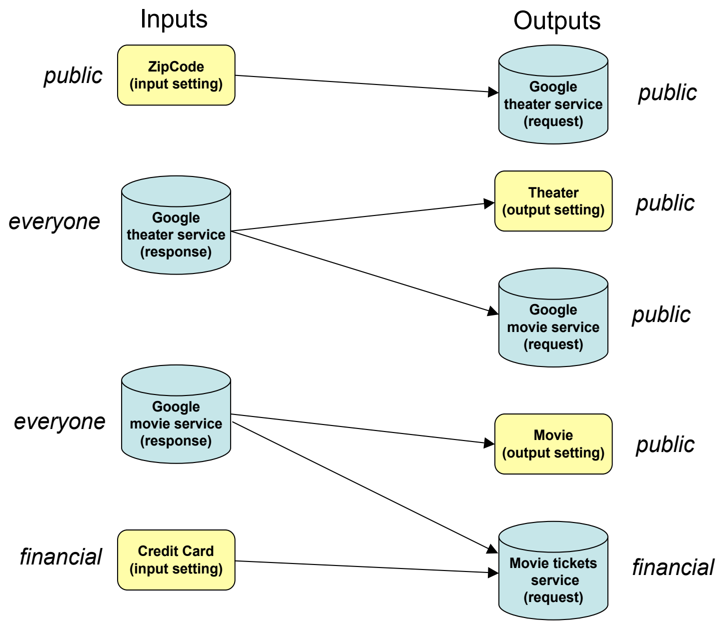

Inputs
Outputs
public
ZipCode (input setting)
Google theater service (request)
public
everyone
Google theater service (response)
Theater (output setting)
public
Google movie service (request)
public
everyone
Google movie service (response)
Movie (output setting)
public
financial
Credit Card (input setting)
Movie tickets service (request)
financial

Figure 5.4: Information flow relation for a monent

1. *The sets $\mathbb{I}_e$ and $\mathbb{O}_e$ only include kinds* ***ExternalSetting****,* ***ServiceReq****, and* ***ExternalServiceResp****.*

2. *The relation $F_e(i, o)$ is true for (i, o) if and only if a message arriving at input i may flow to input o, given the definitions of monent elements in Figure 5.3.*

The first condition follows from the well-formed monent requirements and the hide function. The second condition is a restatement of Proposition 5 for monents — since we are only restricting the set of valid element compositions, all the propositions for information flow interfaces apply to monents as well.

Example 8. *Figure 5.4 shows the information flow relation for the monent of Figure 5.2. To depict this relation, we create a vertex for each input or output and an edge from source s to destination d if $F(s, d)$ is true. We see that the two*

Google services each have two nodes, one for requests and one for responses. The two edges starting at the Google Theater *service response represent indirect information flow paths which pass through the Theaters data grid control.*

The Movie Tickets *service has only an output node, as it does not return any data, only a status response. The* Credit Card *setting has only one outbound edge, connecting it to the* Movie Tickets *service. Thus, we can conclude that the credit card number is not passed to any of the other websites.*

□

5.5.2 Monent composition

Given a collection $\mathbb{M}$ of monents composed from monent elements, we can compose these monents into a single monent: $(\mathcal{M}, F) = \mathsf{compose}(\mathbb{M}, \emptyset)$. We say that this composition is *well-formed* if each output identifier in the composition is unique. Since monents can be described using information flow interfaces, the propositions of section 5.4.1 apply to monent compositions as well.

Proposition 10. *Given a collection of monents* $\mathbb{M}$*, if the composition* $(\mathbb{I}_m, \mathbb{O}_m, F_m) = \mathsf{compose}(\mathbb{M}, \emptyset)$ *is well formed, then the relation* $F_m(i, o)$ *is true for* (i, o) *if and only if a message arriving at input* i *may flow to input* o*, given the definitions of monent elements in Figure 5.3.*

This follows from Propositions 5 and 9.

Proposition 11. *Monent composition is associative and commutative, provided the compositions are all well-formed.*

This follows from Proposition 6.

5.5.3 Security

We now consider the evaluation of security policies for monents.

5.5.3.1 The tag partial order

Monent security policies are specified by associating *tags* with each input and output. We assume that we are given a partial order $PO_{tag} = (\Theta_{tag}, \texttt{everyone}, \texttt{nobody}, \sqsubseteq_{tag})$ which describes the set of tags, the least-sensitive and most-sensitive tags, and an sensitivity ordering relation between tags. We write $[\![\theta]\!]$ to represent the set $\{\theta_i \in \Theta_{tag} | \theta_i \sqsubseteq_{tag} \theta\}$ of tags dominated by θ. In other words, $[\![\theta]\!]$ is the set of user categories represented by θ.

From PO_{tag}, we define a partial order for tag sets: $PO_{set} = (\Theta_{set}, \Theta_{tag}, \emptyset, \sqsubseteq_{set}$). Each element of Θ_{set} is a set of tags, with the least-sensitive element $\top = \Theta_{tag}$ (the set of all tags) and the most-sensitive element $\bot = \emptyset$. For consistency, we equate the set containing the `nobody` tag with the empty set: $\{\texttt{nobody}\} \equiv \emptyset$, and the set containing the `everyone` tag with the set of all tags: $\{\texttt{everyone}\} \equiv \Theta_{tag}$.

We define our element ordering operator $\sqsubseteq_{set}$ in terms of the single tag lattice comparison operator $\sqsubseteq_{tag}$ as follows:

$$\mathcal{T}_1 \sqsubseteq_{set} \mathcal{T}_2 = \begin{cases} \texttt{true} \text{ if } \forall \theta_1 \in \mathcal{T}_1 \ . \ \exists \theta_2 \in \mathcal{T}_2 | \theta_1 \sqsubseteq_{tag} \theta_2 \\ \texttt{false} \text{ otherwise} \end{cases}$$

In other words, for tag set $\mathcal{T}_1$ to be less than or equal to tag set $\mathcal{T}_2$, each element of $\mathcal{T}_1$ must be dominated by or equal to an element in $\mathcal{T}_2$. We extend $[\![\cdot]\!]$ to tags sets in the natural manner: $[\![\mathcal{T}]\!] = \{\theta_i \in \Theta_{tag} | \exists \theta_j \in \mathcal{T} \ . \ \theta_i \sqsubseteq_{tag} \theta_j\}$

Example 9. *The information flow analysis of Example 8 has shown that our* *user's credit card number will not be passed to other websites. However, how do we know that we can trust the movie ticket site itself? As shown in Figure 5.4, each input and output of the movie monent have been labeled (by a security policy) with tags. The inputs of the* GOOGLE THEATER *and* GOOGLE MOVIE *services have been labeled as* ***public****, meaning that we are willing to trust them with some non-identifying personal information (e.g. our zip code). We do not place any restrictions on the responses from these services and thus label them with* `everyone`*. We trust the* MOVIE TICKETS *service with our credit card and thus label it with* ***financial****. All the settings, except for* CREDIT CARD *, have been labeled* ***public****.*

To evaluate whether to activate this monent, we compare the tags associated with the source and target of each edge. If, for each edge, with source tag set $\mathbb{T}_s$ and target tag set $\mathbb{T}_t$, $[\![\mathbb{T}_t]\!] \subseteq [\![\mathbb{T}_s]\!]$, then the monent is sound. This is true for all the edges in Figure 5.4, and thus the monent is sound with respect to our input/output labeling.

$\square$

5.5.3.2 Sound composition

Given the tag set partial ordering PO_{set}, we can formally state the consequences of soundness with respect to a security policy.

Proposition 12. *If a monent $M = (\mathbb{I}_m, \mathbb{O}_m, F_m)$ is sound with respect to a security policy $(PO_{set}, \mathbb{T})$, then there does not exist an output $o \in \mathbb{O}_m$ such that there is an information flow path from i to o in M and $[\![\mathbb{T}(o)]\!] \subseteq [\![\mathbb{T}(i)]\!]$ is* ***false****.*

In other words, if a monent is sound with respect to the policy, then it does not permit information to flow outside the bounds established by the labeling of

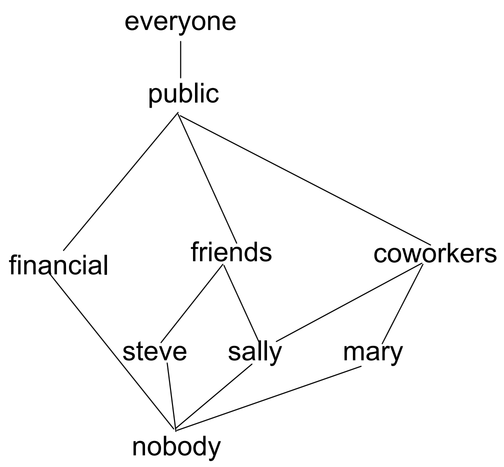


Figure 5.5: Relationships between tag elements

each input. This is a consequence of Proposition 9 and our partial ordering.

Since monents are describable using information flow interfaces, Propositions 7 and 8 apply as well:

Proposition 13 (Sound composition)**.** *1. If a set of monents $\mathbb{M}$ are individually sound with respect to a security policy $(PO_{set}, \mathbb{T})$, then the composition* $\mathsf{compose}(\mathbb{M}, \emptyset)$ *of these monents will also be sound.*

2. Given a set of monents $\mathbb{M}$, which are individually sound with respect to a security policy $(PO_{set}, \mathbb{T})$, and a second set $\mathbb{M}'$ of monents which are individually not sound with respect to the policy, the composition $\mathsf{compose}(\mathbb{M} \cup \mathbb{M}', \emptyset)$ *is not sound with respect to $(PO_{set}, \mathbb{T})$.*

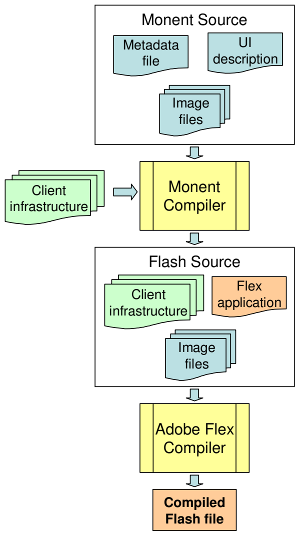


Figure 5.6: Compilation process

5.5.3.3 Tagging contacts

In addition to facilitating interactions with external websites, tags should facilitate interactions with a user's friends and colleagues. To do this, we treat contacts as categories for data and incorporate the user's contact list into the partial ordering of tags. Furthermore, users can label each contact with additional tags to indicate the membership of that contact in specific categories. For example, one might have tags such as `friends` or `coworkers`. If a contact is labeled with one of these categorization tags, the contact appears before the category in the partial ordering of tags.

Example 10. *Our user has three contacts:* Steve, *a friend,* Mary, *a coworker, and* Sally, *who is both a friend and a coworker. Figure 5.5 shows the partial ordering of tags if we create new tags* `friend` *and* `coworker` *and then label the contact entries for these people accordingly.* □

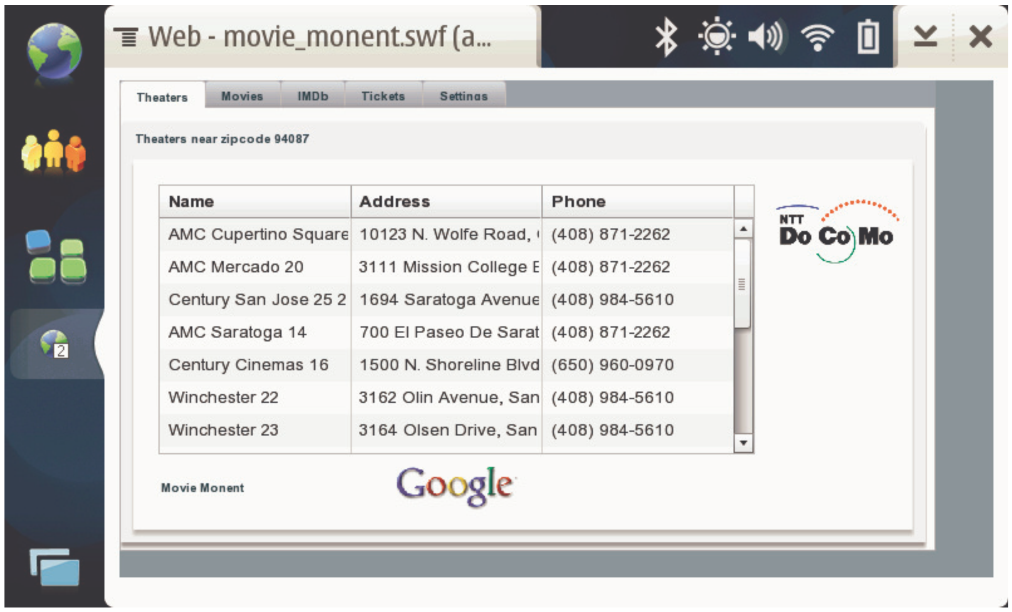
5.6 Implementation

I have implemented a prototype version of the monent framework, using the Adobe Flash^{TM} player as a target platform. Figure 5.6 shows how monents are compiled. A monent is specified in two files: a metadata file, which describes the settings and services used by the monent, and a UI description file, which describes the monent's UI controls. The UI description file uses a subset of Adobe's Flex XML language (without any embedded code). This allows the UI to be built using Adobe's visual designer. In addition, image files may be included in the monent. Settings, services, and UI controls may reference each other as described in Section 5.2.

Once completed, the monent is translated into a Adobe Flex^{TM} application. Each setting and service becomes an object in this application, and event-driven code is added to refresh/invalidate objects based on downstream data changes. Client-side infrastructure, including the security manager, is implemented in ActionScript (Adobe's version of JavaScript for the Flash player) and copied into the monent. An extra tab called *settings* is added to each monent, to enable users to change the values of the monent's input settings.

Next, Adobe's Flex compiler is run to create a compiled Flash file. This compiled file can be executed by any Flash player, such as a browser plug-in.

Composition A separate program implements monent composition. It takes as its inputs an arbitrary number of monent source definitions. These are combined into a single output definition using the approach described in Section 5.2.4. The resulting monent can then be compiled as described above.



Theaters near zipcode 94087

Name	Address	Phone
AMC Cupertino Square	10123 N. Wolfe Road, (	(408) 871-2262
AMC Mercado 20	3111 Mission College E	(408) 871-2262
Century San Jose 25 2	1694 Saratoga Avenue	(408) 984-5610
AMC Saratoga 14	700 El Paseo De Sarat	(408) 871-2262
Century Cinemas 16	1500 N. Shoreline Blvd	(650) 960-0970
Winchester 22	3162 Olin Avenue, San	(408) 984-5610
Winchester 23	3164 Olsen Drive, San	(408) 984-5610

Figure 5.7: The movie monent running on a Nokia N810 Internet Tablet

5.6.1 The movie monent

To demonstrate the framework, I implemented a subset of the movie monent, including reading live data from Google Movies and the Internet Movie Database (see Figure 5.7). Tickets are purchased by calling a simulated e-commerce web service. Extracting the relevant structured data from the unstructured HTML/JavaScript returned by Google and the IMDB is beyond the capabilities of the XPath-like notation I use to navigate structured documents. To address this, I have a separate proxy server which extracts the appropriate content using XSLT and Java and then returns an XML document to the monent.

The hostname of the proxy service is an input setting for the monent. This setting is part of the location for each service which uses the proxy. Thus, when the proxy is changed, the security policy is automatically reevaluated to see whether the monent can use that proxy. If the security check fails, the monent is disabled, except for the settings tab.

I ran the movie monent on a Nokia N810 Internet Tablet. The performance was acceptable – the security policy evaluation, request for movie data from Google, and rendering of the results all occurred within a few seconds.

Composition I originally built the movie monent in a monolithic manner, including all of the functionality directly within a single monent. I then partitioned it into three separate monents: one for Google Movies, one for the IMDB, and one for ticket purchasing. Using the monent composer, I combined these into a single monent with equivalent functionality. Performance of the composition was identical to that of the original monent, as the generated code is the same.

5.6.2 Monent sharing application

Developers at DoCoMo also built a simple web application to facilitate the sharing, configuration, and composition of monents. A list of available monents is maintained in the application's database and displayed as a selection of icons for the user. A user can choose to run, configure, or compose a monent. Running a monent causes the associated Flash file to be downloaded to the user's browser. Users can edit values and tags for a monent's settings (these are local to each user). Finally, a user can select two or more monents for composition. This causes the composer and compiler to be run on the server, generating a new monent. This new monent is then made available on the sharing application.

5.7 Related work

5.7.1 Formal models for components

Information flow interfaces were inspired in part by *Interface Automata* [AH01] and *Reactive Modules* [AH99]. Interface automata describe components as a set of input, output, and internal actions, along with an automata specifying the ordering requirements between actions. Composing two such interfaces yields a new interface which combines the requirements of the two source automata. Composition is successful as long as there exists *some* environment that can make the two source automata work together. Reactive modules can be used to model components with synchronous or asynchronous interactions using a hardware-like model for computation. Of particular interest are the three "spacial" operations defined on reactive modules: variable renaming, variable hiding, and parallel composition.

Our open approach to composition, where inputs and outputs are matched individually, and any unmatched inputs are sourced from the environment, comes from these models. We replaced the action ordering restrictions of interface automata with information flow properties. Clearly, these aspects of specification can be combined, if desired. From reactive modules, we took the "wire-like" nature of our interconnections and the hiding operator, which is necessary to ensure Proposition 13.

5.7.2 Security and process models of computation

Much work has focused on investigating the security aspects of process-based computation models. For example, the *security π-calculus* [HR02] extends the asynchronous π-calculus with a type system to enforce access control and information flow requirements, the *box-π-calculus* [SV99] models the use of program wrappers to mediate the interactions of untrusted programs, and [HW01] describes a type system for the π-calculus to ensure that untrusted data is not used in a trusted context. We use a restricted form of the π-calculus to describe the dynamic behavior of information flow interfaces. Given our focus on component-level composition, we chose to address security issues at the interface-level rather than in the behavioral model. Our security manager, which interprets and enforces security policies, assumes a similar role as the wrappers of the box-π-calculus.

The *Ambient calculus* [CG98] was created to model mobile code, such as code running on mobile devices or code that may be passed over a network. As such, it is particularly well suited for investigating approaches to address security issues for mobile code [LS00, BC01, BCC04]. Instead of explicitly modeling mobility of code, we defer evaluation of security policies until runtime and evaluate policies with respect to each component's current environment. Thus, we can simulate the mobility of components by evaluating them in each environment in which they execute, but we cannot model any restrictions in the movement of components. Without restricting code movement, it may be possible to circumvent an information flow policy by moving a component to an environment where the internal state of a monent can be accessed directly. To avoid this problem, our components either must not contain any internal persistent state or must be run only in an environment which prevents malicious access. We plan to extend our model to include component mobility, with a "sanitizing" step that removes sensitive data when crossing untrusted boundaries.

5.7.3 Information flow

Denning [Den76] first proposed using a lattice to model the security requirements of an information flow analysis. This approach was implemented in a modular program analysis by Myers and Liskov [ML97]. Each element of their lattice is a set of pairs, where the first of the pair is an *owner* and the second of the pair is a set of allowed *readers* for that owner. Data is permitted to flow from a source to a destination if the destination's label has fewer owners and/or fewer readers per owner. Our tag lattice simplifies this model by only tracking the allowed readers. This is reasonable for a lattice that is intended for the end user. Two interesting ideas originating in [ML97] are *declassification* and *label polymorphism*. Declassification is a reduction on information flow restrictions by a trusted component. We intend to investigate approaches to declassification as future work. Label polymorphism can be implemented by a security policy in the monent framework, if it is extended to permit the extraction of tags from runtime values.

Recently, the operating systems community has investigated the use of information flow as a process-level security mechanism [KYB07, EKV05]. However, labels in these frameworks are more like capabilities: they are created dynamically and are not visible to the end user.

5.7.4 Application isolation

The monent security model relies on the security manager's ability to identify and control the external interactions of each monent. Other work has focused on isolating untrusted components within a composition. Subspace [JW07] and MashupOS [WFH07] describe approaches to isolating JavaScript-based applications within a web browser. We avoid the need to use these techniques since monents are compiled from a high level description. However, the monent security model could be adapted to applications written directly in HTML and JavaScript using these techniques.

5.7.5 Presentation-layer integration

Key issues when designing a UI integration tool include the level of integration between components (e..g completely independent, separate but interlinked, or directly interlinked), the approach for handling update dependencies, and the security model. [YBS07] describes a presentation layer integration framework for web applications. Linking of components is specified in an XML *composition model*. At runtime, this model is interpreted by middleware, which automatically calls dependent components based on event notifications. The level of UI integration is similar to that provided by monents: each component's UI elements are distinct (in this case on a single web page), but components respond to changes in dependent components.

MashMaker [EG07] provides a high level programming language and end user development environment for building composite web applications. Composition in MashMaker requires more user involvement than our approach, but permits the merging of presentation elements from the constituent web services. Data dependencies are tracked by the underlying language, and dependent values are updated automatically.

Neither integration framework provides security guarantees. Compared to these frameworks, our approach sacrifices tight coupling of the UI elements (compared to MashMaker) and flexibility in component linking to achieve a much simpler composition model.

5.7.6 Automatic service composition

The web services community, has investigated the automatic (or semi-automatic) composition of web services (e.g. [RS05, SHP02, NBM07]). The automatic composition of presentation-layer components is considered in [DI05]. These approaches focus on either the matching and adaptation of complex arguments and protocols or the selection of services and links using semantic information. I avoid these challenges by using simple argument datatypes (with no subtyping or adaptation) and an open composition model, where each setting is matched independently, and unmatched settings are left as inputs.

5.8 Recap

In this chapter, we looked into issues related to secure end user composition. Certain classes of service-based applications, such as those handling sensitive financial or personal data, require security guarantees in order to be accepted by users. I defined the *information flow interface* as a new abstraction to externalize the information flow properties of a component. Such interfaces are used by a *security manager* to evaluate whether a composition satisfies a *security policy*.

I have instantiated this model in *monents*, which are simple user interfaces on top of external services, and can securely share/correlate data from different sources. Monents can be build by casual developers and composed by end users, without having to worry about many of the security issues faced when building on top of lower-level abstractions.

CHAPTER 6

Conclusion and Future Work

In this dissertation, we have seen how formal models case be used to understand important problems in service-based systems. Specifically, we looked at lost messages in asynchronous programming, inconsistent end-states in flow composition languages, in-sufficient privileges and unintended disclosures in access policy integration, and unintended disclosures in end-user composition.

For each issue, I defined new abstractions which limit the possible system behaviors in exchange for easier reasoning about the issue being addressed. These new abstractions convert issues which are difficult to detect in isolation to properties which can be enforced through compositional, syntax-directed analyses. These analyses have been implemented in four prototypes: TaskJava, BPELCheck, RoleMatcher, and Monents.

In this section, I recap my contributions and look at potential future work in each area. Then, I briefly consider how the approach I have taken can be broadened as a methodology to support the development of robust software.

6.1 Asynchronous Programming

In Chapter 2, I described the task programming model and its instantiation in the TaskJava extension to Java. This approach provides three advances over prior work: 1) a modular translation enabled by method annotations, 2) the idea of a linkable scheduler, which has been formalized through a two-level operational semantics and implemented in our TaskJava compiler, and 3) the design of a CPS translation that works with Java language features including exceptions and sub-classing.

TaskJava is the first step toward my goal of writing robust and reliable programs for large-scale asynchronous systems. I plan to improve our compiler implementation and extend the TaskJava language. For example, I would like to add Tame's fork and join constructs to TaskJava, in a manner that is compatible with exceptions, and supports the static guarantees currently provided by our language.

I also plan to investigate how the explicit control flow of TaskJava programs can improve static analysis tools. I expect analyses for TaskJava programs to be more precise when compared to analyses of general event-driven programs, which must reason about event flow through function pointers and objects. In addition, TaskJava's translation approach may also yield insights about how event-programming frameworks may better support analysis tools. For example, a TaskJava program interacts with the scheduler in two ways: through `spawn` and through `wait`. These interactions are both translated into event registrations. In a program written directly using callbacks, making the distinction between these cases explicit yields more information about the programmer's intent, which may help static analyses.

Finally, I plan to investigate a version of the TaskJava compiler for embedded systems. Many very-small embedded systems (e.g. those used for sensor networks) cannot use threads as a concurrency abstraction due memory usage and non-deterministic scheduling. TaskJava's `async` annotations will help in analyzing stack usage. If we restrict the TaskJava language by disallowing (non-tail) recursion and local variables used across asynchronous calls, we can establish a static memory bound for each task's continuation. Possible targets for an embedded version of TaskJava include Virgil [Tit06], a Java-like language with static memory allocation, and Squawk [SCC06], a Java Virtual Machine designed for small systems used in Sun Microsystems' SunSpot embedded processor boards [sun].

6.2 Consistency

I made two contributions in Chapter 3. First, I formalized a notion of consistency for interacting web services that can be locally checked, given the code for a process and conversation automata specifying the interactions with its peers. Second, I developed BPELCheck, a tool for BPEL process developers that statically checks BPEL processes for consistency violations.

I am pursuing several future directions to make BPELCheck more robust and usable. First, I currently abstract the data in the services to only track local variables of base type. Services typically interact through XML data, and processes query and transform the XML through XPath and XSLT. I plan to integrate reasoning about XML [FBS04a] into BPELCheck. Further, I assume *synchronous* messaging, but certain runtimes can provide asynchronous (queued) messages, for which an additional synchronizability analysis [FBS05] may be required.

Finally, I would like to investigate how our notion of consistency can be applied to systems which are not described using flow composition languages. For example, how can one capture the notion of consistency for systems like Amazon's service-oriented architecture? If consistency requirements are specified by developers, is a dynamic monitoring facility useful? Or, is a static program analysis needed to provide value to developers?

6.3 Relating Access Frameworks

Access control integration for systems interacting via service calls is an important industrial problem. Current solutions (e.g., from Securent, Vaau, and Aveksa) either perform expensive and error-prone manual integration, or ad hoc mining of access control rules from logs that are then centrally managed and enforced. In contrast, our global constraint analysis, together with our precise formulation of the semantics, allows us to make precise claims of correctness with respect to sufficiency and confidentiality.

In the course of doing this work, we encountered a number of subtle issues where the formal model gave us important insights into the problem. Examples include the definition of sufficiency and minimality as well as a categorization of the situations where data can be disclosed. We do not expect that this thesis will be the final word on this topic, but serve as a description of the problem and a starting point for refined solutions which come out of attempts to address these issues in real world situations.

6.3.1 Limitations

I am addressing some limitations in current work. In the event that no global role schema satisfying our constraints exists for a given system, I currently just report an error. Going forward, I would like to provide more guidance to the user, e.g., by providing feedback about which services and constraints actually cause the infeasibility. This information can be obtained as an *unsatisfiable core* from the SAT solver.

I can also attempt to relax constraints or drop inter-system calls and return to the user a solution which satisfies as many of the constraints as possible. This approach has been proposed for other access control interoperability situations [GQ96]. For example, upon finding that no full solution is possible, one might selectively relax the role separation constraints until a sufficient global schema is found. This variation of the global role schema inference problem can be reduced to pseudo-boolean constraints.

In a large collection of systems, the appropriate mappings between roles may not be obvious. If several candidate roles can satisfy the constraints for a given group, the system arbitrary picks one of them. The administrator may influence this process by using role ascriptions. In fact, our approach permits the administrator to pick any point on the spectrum from full role inference to the validation of a completely specified global role schema. However, manually providing ascriptions for all the roles in a large enterprise may be quite tedious. One solution is to use existing role mining techniques [VAG07, ZRE07, BGJ05], which look at the underlying object permissions, to automatically compute an initial set of ascriptions and run our algorithm using this initial guess. If these ascriptions do not admit a global role schema, the solver could relax them and consider other role mappings as needed.

6.3.2 Applications of global schema inference

As described in Section 4.6, global role schema inference is useful in situations where a packaged enterprise application must integrate with a number of other applications, the full set of which is not known in advance.

As shown by the data synchronization example of Section 4.2, global role schema inference should also be useful as a security management tool, permitting IT managers to define and implement a global access control policy for their unique set of deployed systems. A number of companies have recently introduced applications to manage enterprise-wide RBAC policies. These applications are based on a centrally defined and enforced policy, possibly defined with help from a role mining tool. In contrast, my approach leverages each application's existing RBAC infrastructure and simply tries to find a global policy consistent with the collection of individual policies. I believe that this is more practical, as local access policy enforcement in each application is not going away and can introduce problems if the global policy is inconsistent with respect to local policies. I envision a collection of tools to extract role interfaces from each application's metadata, maintain the global role schema, and keep the user-role assignments in each application consistent with the global schema.

Finally, global role schemas should be useful in the definition of access policies for enterprise web portals, such as the patient information portal described in Section 4.2, which combine content from several applications in a single web interface. Portal frameworks generally provide their own RBAC policy enforcement, which must be manually configured to be consistent with the individual applications being displayed in the portal. Our global schema tool could automatically populate the portal's RBAC database with an inferred schema based on the policies of the displayed applications.

6.4 Secure Composition

In Chapter 5, I presented *monents*, a framework for usable, safe, and composable mobile applications. Monents empower the end user by giving them control over customization, composition, and the sharing of their data. Software writers can focus on the creation of new components, specify their intended composite application declaratively, benefiting from a clear security model. The compiler then handles the generation of the final application that is runnable on mobile devices, and respects all security policies.

My formal interface model provides a very general framework for reasoning about information flow and admits a provably secure automatic composition algorithm. In addition, by separating the security policy from the definition of an interface, we allow security policies that are tailored to the environment of a component. Similarly, the same security policy may be applied to multiple components.

From my experiences with the movie monent, and other proposed examples, I believe that this approach to composition is sufficient to meet the expectations of mobile users in the real world. DoCoMo hopes to validate this through user experience studies.

To facilitate an ecosystem of reusable monents, we plan to provide developers with guidelines on naming conventions and a collection of predefined, domain-specific datatypes (e.g. for addresses, phone numbers, and other common settings). For the small fraction of the cases where this is not sufficient, we also could provide developers with more powerful composition tools that support renaming and mapping inputs/outputs. These tools would be used offline to build "adapter" monents bridging incompatible monents. Such adapters would then be composed by end users just like any other monents. Another tool might support better integration of monent user interfaces by specifying the visual layout of UI elements across a composition.

We would also like to simplify the extraction of structured content from web pages, borrowing ideas from MashMaker [EG07] and Koala [LLC07], a tool which captures web navigation actions into a descriptive and modifiable language.

Finally, we are looking into the integration of monents with communication applications like email and instant messaging. We believe that the context and shared state provided by monents can be a useful supplement to less structured message exchanges.

6.5 Beyond Design Patterns

Design Patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [GHJ95]. As such, design patterns provide guidance on how to address recurring issues, either due to language limitations (e.g. the *singleton* pattern) or due to fundamental design trade-offs (e.g. the *visitor* pattern as a solution to the *expression* problem). Patterns have been described to address architectural organization [SG96] and specific domains such as concurrent networked software [SSR00], enterprise applications [Fow02], and application integration [HW03].

The lightweight formal models described in this thesis, which might be called *design models*, combine the best aspects of design patterns and formal modeling approaches. This is achieved by modeling a class of systems rather than a specific instance of a system type. For example, in Chapter 3, we formalize languages for flow composition and consistency specifications, rather than a specific business process.

Like design patterns, the lightweight formal models presented in this thesis highlight important design decisions and are reusable in different contexts. Key aspects of each area were elucidated through the modeling process (e.g. the formalization of Chapter 2 made issues regarding error handling in event-driven programs clearer). The effort of formal modeling can be amortized over many system instantiations. Reusable formal models have been proposed in the past (e.g. [GD90]). However, these proposals largely focus on creating specialized models via refinement. In my approach, the model remains abstract, but key properties of well-formed systems are distilled for use by developers.

In fact, developers do not need to understand the formal model to take advantage of a design model and tools derived from such models. Two classic examples of formal models which see widespread use in industry are the relational data model [Dat82] and role-based access control [FK92], both based on set theory.

Design models go beyond design patterns in linking a system's static description to guarantees about its dynamic behavior. This is achieved by defining new abstractions (represented in the model by mathematical objects and properties) to be used by the programmer. I believe that, although better static analysis techniques and type systems can help for specific issues (e.g. concurrency), much of the progress in software robustness will occur through better abstractions and design models.

My thesis has shown how modern techniques in programming language semantics can be applied to create concise and meaningful models. Going forward, I would like to build on this work, creating a more systematic and practitioner-friendly approach to defining design models. Perhaps this may involve building a framework for Domain Specific Languages (DSLs), where design models can be implemented on top of (or mapped to) existing general purpose languages. I also hope to begin a catalog of design models which fully address the key issues faced by developers of enterprise software and service-based systems.

References

[AH99] R. Alur and T. Henzinger. "Reactive Modules." *Formal Methods in System Design*, **15**(1):7–48, July 1999.

[AH01] L. de Alfaro and T. Henzinger. "Interface automata." In *ESEC/FSE '01*, pp. 109–120. ACM, 2001.

[AHT02] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. "Cooperative Task Management without Manual Stack Management." In *Proc. Usenix Tech. Conf.*, 2002.

[And04] A. Andersen. *XACML Profile for Role Based Access Control (RBAC)*. OASIS, February 2004.

[App] "Developing Apple Dashboard Widgets." http://developer.apple.com/macosx/dashboard.html.

[App91] A. Appel. *Compiling with continuations*. Cambridge University Press, 1991.

[BBB02] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, and et al. "Web Services Conversation Language (WSCL) 1.0." Technical report, World Wide Web Consortium, March 2002. http://www.w3.org/TR/wscl10/.

[BC85] J. Bergeretti and B. Carré. "Information-flow and data-flow analysis of while-programs." *ACM Trans. Program. Lang. Syst.*, **7**(1):37–61, 1985.

[BC01] M. Bugliesi and G. Castagna. "Secure safe ambients." In *POPL '01*, pp. 222–235. ACM, 2001.

[BCB03] R. von Behren, J. Condit, and E. Brewer. "Why Events Are a Bad Idea (for high-concurrency servers)." In *HotOS IX*, 2003.

[BCC04] M. Bugliesi, G. Castagna, and S. Crafa. "Access control for mobile agents: The calculus of boxed ambients." *ACM Trans. Program. Lang. Syst.*, **26**(1):57–124, January 2004.

[BCH05] D. Beyer, A. Chakrabarti, and T. Henzinger. "Web service interfaces." In *WWW '05*, pp. 148–159. ACM, 2005.

[BCZ03] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. "Capriccio: scalable threads for internet services." In *SOSP '03*, pp. 268–281. ACM, 2003.

[Bet07] C. Betz. *Architectures and Patterns for IT Service Management, Resource Planning, and Governance: Making Shoes for the Cobbler's Children*. Morgan Kaufmann, 2007.

[BF00] M. Butler and C. Ferreira. "A Process Compensation Language." In *IFM '00: Integrated Formal Methods*, pp. 61–76. Springer, 2000.

[BF04] M. Butler and C. Ferreira. "An Operational Semantics for StAC, a Language for Modelling Long-Running Business Transactions." In *Coordination '04*, volume 2949 of *LNCS*, pp. 87–104. Springer, 2004.

[BFH03] T. Bultan, X. Fu, R. Hull, and J. Su. "Conversation specification: a new approach to design and analysis of e-service composition." In *WWW '03*, pp. 403–410. ACM, 2003.

[BFL96] M. Blaze, J. Feigenbaum, and J. Lacy. "Decentralized Trust Management." In *S& P '96*, p. 164. IEEE, 1996.

[BGJ05] E. Bertino, A. Ghafoor, J. Joshi, and B. Shafiq. "Secure Interoperation in a Multidomain Environment Employing RBAC Policies." *IEEE Transactions on Knowledge and Data Engineering*, **17**(11):1557–1577, November 2005.

[BHF04] M. Butler, C. A. R. Hoare, and C. Ferreira. "A Trace Semantics for Long-Running Transactions." In *Communicating Sequential Processes: The First 25 Years*, volume 3525/2005 of *LNCS*, pp. 133–150. Springer, 2004.

[BMM05] R. Bruni, H. Melgratti, and U. Montanari. "Theoretical foundations for compensations in flow composition languages." In *POPL '05*, pp. 209–220. ACM, 2005.

[Bon] David Bond. "Fizmez Web Server." http://sourceforge.net/projects/fizmezwebserver.

[BPE03] T. Andrews et al. "Business Process Execution Language for Web Services.", May 2003. http://dev2dev.bea.com/webservices/BPEL4WS.html.

[BPM] "Business Process Modeling Language (BPML)." http://www.bpmi.org.

[Bre] "BREW: Binary Runtime Environment for Wireless." http://brew.qualcomm.com.

[BSS96] P. Bonatti, M. Sapino, and V. Subrahmanian. "Merging heterogeneous security orderings." In *EOSRICS '96*, pp. 183–197, September 1996.

[CC07] L. Chen and J. Crampton. "Inter-domain role mapping and least privilege." In *SACMAT '07*, pp. 157–162. ACM Press, 2007.

[CCL06] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. "A Formal Account of Contracts for Web Services." *Web Services and Formal Methods*, pp. 148–162, 2006.

[CFH06] B. Carminati, E. Ferrari, and P. Hung. "Security Conscious Web Service Composition." In *ICWS '06*, pp. 489–496, 2006.

[CG98] L. Cardelli and A. Gordon. "Mobile Ambients." In *FoSSaCS '98*, pp. 140–155. Springer-Verlag, 1998.

[CGP08] G. Castagna, N. Gesbert, and L. Padovani. "A theory of contracts for web services." In *POPL '08*, pp. 261–272. ACM, 2008.

[CK05] R. Cunningham and E. Kohler. "Making events less slippery with EEL." In *HotOS X*, 2005.

[CRR02] S. Chaki, S.K. Rajamani, and J. Rehof. "Types as models: model checking message-passing programs." In *POPL '02*, pp. 45–57. ACM, 2002.

[Dat82] C. J. Date. "A formal definition of the relational model." *SIGMOD Rec.*, **13**(1):18–29, 1982.

[Den76] DE. Denning. "A lattice model of secure information flow." *Commun. ACM*, **19**(5):236–243, May 1976.

[DGJ98] J. Dingel, D. Garlan, S. Jha, and D. Notkin. "Reasoning about Implicit Invocation." In *FSE '98*, pp. 209–221. ACM, 1998.

[DHJ07] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store." In *SOSP '07*. ACM Press, October 2007.

[DI05] O. Díaz and A. Iturrioz, J.and Irastorza. "Improving portlet interoperability through deep annotation." In *WWW '05*, pp. 372–381. ACM, 2005.

[DJ06] S. Du and J. Joshi. "Supporting authorization query and inter-domain role mapping in presence of hybrid role hierarchy." In *SACMAT '06*, pp. 228–236. ACM Press, 2006.

[DM06] B. Dutertre and L. de Moura. "The Yices SMT solver." Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[EG07] R. Ennals and D. Gay. "User-friendly functional programming for web mashups." In *ICFP '07*, pp. 223–234. ACM, 2007.

[EKV05] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. "Labels and event processes in the asbestos operating system." In *SOSP '05*, pp. 17–30. ACM Press, 2005.

[ELL90] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. "A multidatabase transaction model for InterBase." In *VLDB '90*, pp. 507–518. Morgan Kaufmann, 1990.

[EM07] M. Emmi and R. Majumdar. "Verifying compensating transactions." In *VMCAI '07*. Springer, 2007.

[Eng00] R. Engelschall. "Portable Multithreading - The Signal Stack Trick for User-Space Thread Creation." In *Proc. USENIX Tech. Conf.*, June 2000.

[ES03] N. Een and N. Sörensson. "An extensible SAT-solver." In *SAT '03*, pp. 502–518, 2003.

[ext05] "eXtensible Access Control Markup Language (XACML) Version 2.03." OASIS Standard, February 2005.

[FBS04a] X. Fu, T. Bultan, and J. Su. "Analysis of interacting BPEL web services." In *WWW '04*, pp. 621–630. ACM, 2004.

[FBS04b] Xiang Fu, Tevfik Bultan, and Jianwen Su. "WSAT: A Tool for Formal Analysis of Web Services." In *CAV '04*, pp. 510–514, 2004.

[FBS05] Xiang Fu, Tevfik Bultan, and Jianwen Su. "Synchronizability of Conversations Among Web Services." *IEEE Trans. on Softw. Eng.*, **31**(12):1042–1055, December 2005.

[FK92] D. Ferraiolo and R. Kuhn. "Role-Based Access Control." In *15th National Computer Security Conference*, 1992.

[FMM06] J. Fischer, R. Majumdar, and T. Millstein. "Preventing Lost Messages in Event-driven Programming.", January 2006. http://www.cs.ucla.edu/tech-report/2006-reports/060001.pdf.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

[FUM03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. "Model-based Verification of Web Service Compositions." In *ASE '03*. IEEE, 2003.

[GD90] D. Garlan and N. Delisle. "Formal Specifications as Reusable Frameworks." In *VDM '90*, pp. 150–163. Springer, 1990.

[GFW99] S. Ganz, D. Friedman, and M. Wand. "Trampolined Style." In *ICFP '99*, pp. 18–27, 1999.

[GHJ95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison Wesley, 1995.

[GKH01] P. Graunke, S. Krishnamurthi, S. Van Der Hoeven, and M. Felleisen. "Programming the Web with High-Level Programming Languages." *LNCS*, **2028**:122–137, 2001.

[GKK03] D. Garlan, S. Khersonsky, and J.S. Kim. "Model Checking Publish-Subscribe Systems." In *SPIN '03*, LNCS 2648, pp. 166–180. Springer, 2003.

[GLB03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC Language: A Holistic Approach to Network Embedded Systems." In *PLDI '03*, pp. 1–11, June 2003.

[Goo] "Google Gadgets API Overview." http://www.google.com/apis/gadgets.

[GQ96] L. Gong and X. Qian. "Computational issues in secure interoperation." *Software Engineering, IEEE Transactions on*, **22**(1):43–52, 1996.

[GS87] H. Garcia-Molina and K. Salem. "Sagas." In *SIGMOD '87*, pp. 249–259. ACM, 1987.

[HA00] C. Hagen and G. Alonso. "Exception Handling in Workflow Management Systems." *IEEE Trans. Softw. Eng.*, **26**(10):943–958, 2000.

[Hig06] K. Higgins. "WGI Engineers Application Integration Strategy with SOA." *Network Computing*, June 2006.

[HL7] "HL7: Health Level Seven." http://www.hl7.org.

[HR02] M. Hennessy and J. Riely. “Information flow vs. resource access in the asynchronous pi-calculus.” *ACM Trans. Program. Lang. Syst.*, **24**(5):566–591, September 2002.

[HSW00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. “System architecture directions for networked sensors.” In *ASPLOS ’00*, pp. 93–104. ACM, 2000.

[HVK98] K. Honda, V. Vasconcelos, and M. Kubo. “Language primitives and type discipline for structured communication-based programming.” In *Programming Languages and Systems*, pp. 122–138. Springer-Verlag, 1998.

[HW01] M. Hepburn and D. Wright. “Trust in the pi-calculus.” In *PPDP ’01*, pp. 103–114. ACM, 2001.

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003.

[IPW01] A. Igarashi, B. Pierce, and P. Wadler. “Featherweight Java: a minimal core calculus for Java and GJ.” *ACM Trans. Program. Lang. Syst.*, **23**(3):396–450, 2001.

[JLB03] J.Augusto, M. Leuschel, M. Butler, and C. Ferreira. “Using the Extensible Model Checker XTL to Verify StAC Business Specifications.” In *AVoCS ’03*, 2003.

[JM07] R. Jhala and R. Majumdar. “Interprocedural analysis of asynchronous programs.” In *POPL ’07*, pp. 339–350. ACM, 2007.

[JW07] C. Jackson and H. Wang. “Subspace: secure cross-domain communication for web mashups.” In *WWW ’07*, pp. 611–620. ACM Press, 2007.

[KrK06] M. Krohn, E. Kohler, F. Kaashoek, and D. Mazieres. “The Tame event-driven framework.”, 2006. http://www.okws.org/doku.php?id=okws:tame.

[KSS03] M. Kuhlmann, D. Shohat, and G. Schimpf. “Role mining - revealing business roles for security administration using data mining technology.” In *SACMAT ’03*, pp. 179–186, New York, NY, USA, 2003. ACM Press.

[KYB07] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, Eddie Kohler, and R. Morris. "Information flow control for standard OS abstractions." In *SOSP '07*, pp. 321–334. ACM Press, 2007.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system." *Commun. ACM*, **21**(7):558–565, 1978.

[LLC07] G. Little, T. Lau, A. Cypher, J. Lin, Eb. Haber, and E. Kandogan. "Koala: capture, share, automate, personalize business processes on the web." In *CHI '07*, pp. 943–946. ACM Press, 2007.

[LMW02] N. Li, J. Mitchell, and W. Winsborough. "Design of a Role-Based Trust-Management Framework." In *S& P '02*, p. 114. IEEE, 2002.

[LS00] F. Levi and D. Sangiorgi. "Controlling interference in ambients." In *POPL '00*, pp. 352–364. ACM, 2000.

[LWM03] N. Li, W. Winsborough, and J. Mitchell. "Distributed credential chain discovery in trust management." *J. Comput. Secur.*, **11**(1):35–86, 2003.

[LZ04] P. Li and S. Zdancewic. "Advanced control flow in Java card programming." In *LCTES '04*, pp. 165–174. ACM Press, 2004.

[MFG04] J. Matthews, R. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen. "Automatically Restructuring Programs for the Web." *Automated Software Eng.*, **11**(4):337–364, October 2004.

[Mic] Sun Microsystems. "Enterprise Java Beans." http://java.sun.com/products/ejb/.

[Mil80] R. Milner. *A Calculus of Communicating Systems.* LNCS 92. Springer-Verlag, 1980.

[Mil99] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[ML97] A. Myers and B. Liskov. "A decentralized model for information flow control." In *SOSP '97*, pp. 129–142. ACM Press, 1997.

[ML00] A. Myers and B. Liskov. "Protecting privacy using the decentralized label model." *ACM Trans. Softw. Eng. Methodol.*, **9**(4):410–442, 2000.

[MP02] P. McDaniel and A. Prakash. "Methods and limitations of security policy reconciliation." In *S& P '02*, pp. 73–87. IEEE, 2002.

[NBM07] H. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. "Semi-automated adaptation of service interactions." In *WWW '07*, pp. 993–1002. ACM Press, 2007.

[NCM03] N. Nystrom, M.R. Clarkson, and A.C. Myers. "Polyglot: An Extensible Compiler Framework for Java." In *CC '03*, LNCS 2622, pp. 138–152. Springer, 2003.

[ONS07] K. Ono, Y. Nakamura, F. Satoh, and T. Tateishi. "Verifying the Consistency of Security Policies by Abstracting into Security Types." In *ICWS '07*, pp. 497–504, 2007.

[OPE] "OpenMRS." http://www.openmrs.org/.

[Osb02] S. Osborn. "Information flow analysis of an RBAC system." In *SACMAT '02*, pp. 163–168. ACM Press, 2002.

[PCM05] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, and M. Felleisen. "Continuations from generalized stack inspection." In *ICFP '05*, pp. 216–227, 2005.

[PDZ99] V.S. Pai, P. Druschel, and W. Zwaenepoel. "Flash: An Efficient and Portable Web Server." In *Proc. USENIX Tech. Conf.*, pp. 199–212. Usenix, 1999.

[PFF07] Marco Pistoia, Stephen J. Fink, Robert J. Flynn, and Eran Yahav. "When Role Models Have Flaws: Static Validation of Enterprise Security Policies." In *ICSE '07*, pp. 478–488. IEEE Computer Society, 2007.

[Pie02] B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PJ05] S. Piromruen and J. Joshi. "An RBAC Framework for Time Constrained Secure Interoperation in Multi-domain Environments." In *WORDS '05*, pp. 36–58. IEEE Computer Society, 2005.

[Que03] C. Queinnec. "Inverting back the inversion of control or, continuations versus page-centric programming." *SIGPLAN Not.*, **38**(2):57–64, 2003.

[Rep91] J. Reppy. "CML: A higher concurrent language." In *PLDI '91*, pp. 293–305, New York, NY, USA, 1991. ACM Press.

[RR02] Sriram K. Rajamani and Jakob Rehof. "Conformance Checking for Models of Asynchronous Message Passing Software." In *CAV*, LNCS 2404, pp. 166–179. Springer, 2002.

[RS05] J. Rao and X. Su. "A Survey of Automated Web Service Composition Methods." In *LNCS*, volume 3387/2005, pp. 43–54. Springer, 2005.

[SA] G. Shekhtman and M. Abbott. "State Threads Library for Internet Applications." http://state-threads.sourceforge.net.

[SAB02] H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. "Atomicity and isolation for transactional processes." *ACM Trans. Database Syst.*, **27**(1):63–116, 2002.

[SBM04] I. Singh, S. Brydon, G. Murray, V. Ramachandran, T. Violleau, and B. Stearns. *Design Web Services with the J2EE 1.4 Platform.* Addison-Wesley, 2004.

[SC85] A. Sistla and E. Clarke. "The complexity of propositional linear temporal logics." *J. ACM*, **32**(3):733–749, 1985.

[SCC06] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. "Java^{TM}on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine." In *VEE '06*, pp. 78–88. ACM, July 2006.

[SCF96] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. "Role-Based Access Control Models." *IEEE Computer*, **29**(2):38–47, 1996.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SHP02] E. Sirin, J. Hendler, and B. Parsia. "Semi-automatic Composition of Web Services using Semantic Descriptions." In *Web services: modeling, architecture and infrastructure workshop in ICEIS '03*, 2002.

[SIM07] M. Srivatsa, A. Iyengar, T. Mikalsen, I. Rouvellou, and J. Yin. "An Access Control System for Web Service Compositions." In *ICWS '07*, pp. 1–8, 2007.

[SM03] A. Sabelfeld and A. Myers. "Language-based information-flow security." *Selected Areas in Communications, IEEE Journal on*, **21**(1):5–19, January 2003.

[SP06] Randy Shoup and Dan Pritchett. "The eBay Architecture." Talk at SD Forum, November 2006. www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf.

[SSR00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley, 2000.

[sun] Sun Microsystems. "Project Sunspot." http://www.sunspotworld.com/.

[SV99] P. Sewell and J. Vitek. "Secure Composition of Insecure Components." In *CSFW '99.* IEEE Computer Society, 1999.

[THK94] K. Takeuchi, K. Honda, and M. Kubo. "An Interaction-based Language and its Typing System." In *PARLE '94*, pp. 398–413, London, UK, 1994. Springer-Verlag.

[Tit06] B. Titzer. "Virgil: objects on the head of a pin." In *OOPSLA '06*, pp. 191–208. ACM, 2006.

[VAG07] Jaideep V., Vijayalakshmi A., and Q. Guo. "The role mining problem: finding a minimal descriptive set of roles." In *SACMAT '07*, pp. 175–184. ACM Press, 2007.

[WCB01] M. Welsh, D. Culler, and E. Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." In *SOSP '01.* ACM, 2001.

[WFH07] H. Wang, X. Fan, J. Howell, and C. Jackson. "Protection and communication abstractions for web browsers in MashupOS." In *SOSP '07*, pp. 1–16. ACM, 2007.

[Wid] "WidSets." http://www.widsets.com/.

[YBS07] J. Yu, B. Benatallah, R. Saint-Paul, F. Casati, F. Daniel, and M. Matera. "A framework for rapid integration of presentation components." In *WWW '07*, pp. 923–932. ACM, 2007.

[YCC06] C. Ye, S.C. Cheung, and W.K. Chan. "Publishing and composition of atomicity-equivalent services for B2B collaboration." In *ICSE '06*, pp. 351–360. ACM, 2006.

[ZRE07] D. Zhang, K. Ramamohanarao, and T. Ebringer. "Role engineering using graph optimisation." In *SACMAT '07*, pp. 139–144. ACM Press, 2007.