Higher-order Behavioral Contracts for Distributed Components

Higher-order Behavioral Contracts for Distributed Components

Lucas Waye
Christos Dimoulas
and
Stephen Chong

TR-03-15

Computer Science Group
Harvard University
Cambridge, Massachusetts

Higher-order Behavioral Contracts for Distributed Components

Lucas Waye   Christos Dimoulas   Stephen Chong

Harvard University

{lwaye, chrdimo, chong}@seas.harvard.edu

Abstract

Inspired by the Design by Contract paradigm, we introduce CONSUL, a contract system for distributed components. CONSUL monitors distributed components at run time with higher-order behavioral contracts. Contract monitoring is local to a component, and the component is treated as a black box. Thus it does not disturb the highly decoupled nature of distributed components and allows heterogeneous implementation languages and platforms without modifications to a component's code. We describe the design, semantics and properties of CONSUL (adapter transparency and correct blame), and show that its contracts can capture and enforce precise and useful properties of a variety of off-the-shelf components.

1. Introduction

The Design by Contract methodology [21–23] facilitates the correct design of component-based software. It dictates that each component should come with a *contract*: a precise and enforceable specification of its expectations of and promises to other components. Contracts are usually expressed in a full-fledged programming language and are checked when components run. Thus they make it easy for programmers to state and enforce precise conditions on the correct use of a component. In addition to protecting components within an application, contracts enrich the documentation of components, eliminate the need for defensive code that hinders the readability and maintenance of components, and upon contract failures provide useful information (blame) that serves as a starting point for debugging. Since their introduction in Eiffel [21], contracts have been adapted with much success to diverse settings (e.g., [3, 4, 11, 17, 18]). In this paper, we describe CONSUL, a tool that brings the benefits of contracts to distributed components.

CONSUL, and any other contract system for distributed components, must satisfy a challenging set of requirements to be practical. First, when writing distributed applications, programmers wire together third-party remote components usually without having access or the ability to modify the components' source code. Even when they do have access, the source code of the components is often written in a multitude of languages. Thus, in contrast to a contract system for a programming language, a contract system for distributed components cannot depend on specific features of a component's language to describe and enforce properties of the component, i.e., it must be *component-language agnostic*.

Second, programmers usually do not have control over the deployment and configuration of a third-party component. This has three implications for a distributed contract system: (i) the contract system can only assume third-party components are *black boxes* that send and receive messages, i.e., contracts should express properties of the messages and information from messages alone should suffice to enforce the properties; (ii) the contract system must be able to check contracts under *partial deployment*, i.e., when some but not all components use it; (iii) the contract system should cause *no changes to communication patterns* between components, since doing so under partial deployment of the contract system may cause some components to fail.

Third, distributed components are often only loosely coupled. Their interfaces are typically specified using RPC frameworks (e.g., Thrift [1]), or web services standards (e.g., WSDL [8]) that describe the operations a component offers to its clients together with the structure of the messages the operations expect and return. These basic interface abstractions enable the re-use of components in diverse contexts since they do not assume that components are deployed on top of a particular middleware or distributed system (e.g., CORBA [24]). In turn, this enables programmers to easily compose components that assume different RPC frameworks, transfer protocols, etc., all within the same application. A contract system for distributed components should accommodate the *loose coupling of distributed components* and operate on top of popular interface abstractions. Moreover, as new interface abstractions are developed, a contract system should be *extensible with interface abstractions*.

CONSUL meets all of these requirements. CONSUL takes a black-box approach to contract checking and can enhance the interface of components without requiring code changes or a particular shared middleware, message protocol, etc. A CONSUL-enhanced component collects data from messages the component sends and receives and then determines whether a contract is satisfied based only on this local data. Moreover, CONSUL operates on top of popular interface abstractions such as Thrift and WSDL and users can extend

CONSUL with plugins for the interface abstraction of their choice. Finally, CONSUL can be partially deployed: some but not necessarily all components in a distributed application may use it while CONSUL remains transparent to components that do not use it.

***Properties enforceable by* CONSUL** CONSUL allows the specification and enforcement of *behavioral contracts* [5]: arbitrary computable properties of the messages that a component operation receives and sends. By contrast, most existing distributed component monitoring tools focus on syntactic properties of messages, communication patterns between components, or quality of service.

Existing systems that support behavioral contracts (e.g. [6]) limit expressiveness to first-order properties of messages, i.e, assertions. However, the documentation of popular distributed components are rife with descriptions of properties of messages that go beyond what assertions can express. For instance, the documentation of the Thrift API of the popular note taking service Evernote[1] states that the listLinkedNotebooks operation returns (among other data) a noteStoreURL, the address of a service that implements the NoteStore interface [10]. To express this property, a contract system must be able to associate noteStoreURL with a contract for the NoteStore interface. To enforce it, a contract system must be able to keep track of future messages sent to and from noteStoreURL and monitor that they conform with the contract for NoteStore. This is not a first-order property of noteStoreURL but rather a *higher-order* one. It describes what operations the server at noteStoreURL offers, what arguments they expect and what results they return. Note that the arguments or results may in turn implement NoteStore or some other higher-order specification. Thrift and other frameworks that describe only syntactic contracts or first-order behavioral contract systems for distributed components cannot express this property and would treat noteStoreURL as a string, rather than a reference to a component.

In contrast, CONSUL is a higher-order behavioral contract system for distributed components. It can express and enforce exactly the higher-order properties of messages that programmers can express in contract systems for higher-order programming languages. More concretely, with CONSUL the author of the Evernote API can state that noteStoreURL refers to a service that adheres to the NoteStore contract and CONSUL checks that any connections to noteStoreURL conform to that contract. In Section 2 we further describe how CONSUL can express and enforce behavioral contracts for an application like Evernote.

Validation Simultaneously addressing all the above requirements is complex and challenging. Moreover, even though contracts for higher-order functions [11] have been extensively studied over the last fifteen years, adapting these results for CONSUL is not straight-forward. Important semantic issues arises from the lack of linguistic mechanisms, such as proxies, that associate values and their contracts. Instead, CONSUL has to carefully infer and maintain the association between services and their contracts only from inspecting network messages. To ensure the correctness of CONSUL, we develop CC, a calculus that models the semantics of CONSUL. In Section 3 we present CC and its properties, namely that CONSUL is transparent to application components (under certain reasonable conditions) and that CONSUL assigns correct blame upon contract violations. These results help validate that our system meets the requirements for a practical contract monitoring system for distributed components. In addition, we have implemented CONSUL and used it to harden the interfaces of a variety of off-the-shelf components (Section 4), and evaluate the impact of CONSUL on their performance (Section 5).

[1] `https://evernote.com/`

2. CONSUL by Example

Evernote provides cloud-based storage of notes, organized into notebooks. Each user's notes and notebooks are stored in a particular sharded database called a *note store*. Client-side components implement tools for users to create and access notes and notebooks. Evernote allows a user to share her notebooks with other users. Thus, a note store contains the notes and notebooks a user has created, and a list of *shared notebooks* she has shared with other users and a list of *linked notebooks* that other users have shared with her.

Accessing a shared notebook may require contacting a note store that is different from the one the user contacts for her own notes. Figure 1 depicts some of the steps that a user must take to access a note from a notebook shared with her by another user [10]. Each box in the diagram represents a component, and arrows indicate RPC calls and returns. Labels in bold above an arrow indicate the operation being called, and labels below arrows describe the data transferred in the request or reply message. Component Client is the client-side component that interacts with the Evernote services. Component Server1 is the Evernote cloud service that implements the user's note store, and Server2 represents the note store where a shared notebook resides.

To access a shared notebook, the client retrieves from its own note store a list of linked notebooks (operation listLinkedNotebooks) and uses the information from a linked notebook to contact Server2 and authenticate to the shared notebook (authenticateToSharedNotebook).The client can then access the shared notebook (getSharedNotebookByAuth) and find a note within the notebook (findNotes).

There are many properties of this workflow that are necessary to successfully access a user's shared notebooks but stated only informally in the documentation. We focus here on two, a simple and a complex one: (1) the findNotes operation consumes a nonnegative integer offset argument and returns a list that has a length of at most maxNotes (another argument to the function); (2) listLinkedNotebooks returns URLs that are references to NoteStore services (i.e., services with a particular interface located at different endpoints) together with a shareKey per NoteStore service to be used when calling authenticateToSharedNotebook on that service.

CONSUL can capture both properties as contracts and enforce them at run time. CONSUL provides its own IDL to express contracts. The IDL does not focus on syntactic specifications (which IDLs such as Thrift's already handle) but instead on behavioral contracts. Its features are tailored to the contracts that CONSUL aims to express and enforce.

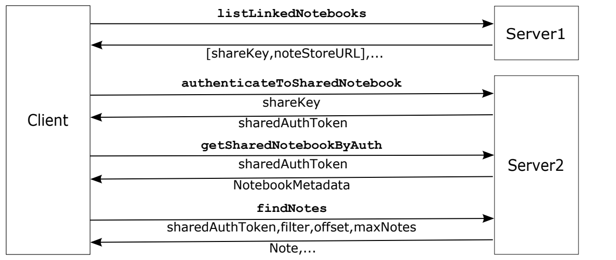

Figure 1. Evernote: access to a shared notebook

To express the first property as a CONSUL contract, we note that it corresponds to a first-order behavioral contract, i.e., the kind of contract that can be expressed with just a pre- and a post-condition for an operation. Lines 2–4 of Figure 2 show the part of the contract for findNotes. The keyword **service** defines a contract and names it, in this case NoteStore. A contract describes the interface of a component and contains signatures for all the operations the component provides. Each signature is followed by tags that state properties about the arguments and result of the operations. Figure 2 shows part of the contract for a note store service, such as Server1 and Server2, and shows the signature for findNotes, with two tags, **@requires** and **@ensures**, which define a pre-condition and a post-condition respectively. Pre- and post-conditions are specified as Python code, i.e., code between « and » in the contract is Python. The Python code in pre- and post-conditions can refer to the operation's arguments (e.g., offset and maxNotes in the snippet above). Post-conditions also have access to a special Python variable **result** that is bound to the result of an operation call. In the snippet, **@requires** checks that offset is nonnegative and **@ensures** checks that the length of **result** is less than or equal to maxNotes.

A CONSUL-enhanced component installs a network adapter as a stand-alone running process that intercepts all messages between the component and its peers. The network adapter checks messages against the contract. CONSUL treats all components as black boxes and does not require code modifications nor does it change how components interact. If a check fails, CONSUL logs the details of the contract and message involved, and *blames* the component that was responsible for the contract violation. Intuitively, CONSUL blames the component that vouched for the value to meet its contract that triggered the violation. For example if the Evernote client's adapter detects a violation of the pre-condition of findNotes, it blames the client. With this information, CONSUL helps localize the fault to facilitate the debugging and maintenance of distributed components.

To express the second property from above, the CONSUL IDL must be able to express that: (1) a datum from the result of a service operation corresponds to an endpoint of another service that adheres to a specific contract; (2) another datum, called hereafter a token, is coupled with an endpoint; and (3) operations of the endpoint expect as an argument the token that accompanies the endpoint. In terms of our Evernote example, the CONSUL NoteStore contract must capture that (1) listLinkedNotebooks returns noteStoreUrls that in turn adhere to the NoteStore contract; (2) each noteStoreUrl is coupled with a token shareKey; and (3) authenticateToSharedNotebook expects as its argument the token shareKey for the corresponding service's endpoint. In sum, the NoteStore contract is a higher-order contract, i.e., it captures a property that can only be checked when the client uses the note store service at a noteStoreUrl and not when listLinkedNotebooks returns the noteStoreUrl.

Lines 6–12 of Figure 2 show the part of the NoteStore contract for listLinkedNotebooks and authenticateToSharedNotebook. An **@identifies** tag that supplements the signature of an operation directs CONSUL to identify a datum from a request message for the operation as an endpoint of a service that adheres to a contract together with its accompanying token if any. We call the token that accompanies an endpoint the *index* of the endpoint's contract. Intuitively, an index denotes that an endpoint adheres to a family of contracts that all describe the same properties, but are distinguished by a particular value. An **@identifies** tag has two parts: a contract for an endpoint, and a Python snippet for extracting the endpoint and the index for its contract. In this snippet lines 7–8 extract a list of URLs from the result of listLinkedNotebooks (the Python code in between « and »; expression yield is special CONSUL syntax to identify an individual service) and declare that all of those identified services should satisfy the NoteStore contract, each indexed by the respective shareKey. The **@indexedby** tag that supplements the signature of an operation directs CONSUL to treat a datum from a request message for the operation as an index, i.e., the token distinguishes the member of the corresponding contract family CONSUL should check. Line 11 of figure 2 specifies that the index that CONSUL should use when checking the contract for the authenticateToSharedNotebook operation is shareKey.

To enforce this contract, a CONSUL adapter tracks where endpoints are identified so that the referrer of an endpoint can be blamed in the event of a contract failure of the post-condition of the contract of the endpoint. Furthermore CONSUL also distinguishes different referrers of an endpoint based on indices. Concretely, the adapter of the Evernote client records that, according to Server1, noteStoreUrl adheres to the NoteStore contract with index shareKey. Thus if a subsequent call to authenticateToSharedNotebook with the token received from listLinkedNotebooks as the argument, violates its post-condition (elided for conciseness from Fig-

```
service NoteStore {
  findNotes(authToken,filter,offset,maxNotes)
  @requires « offset >= 0 »
  @ensures  « len(result) <=  maxNotes »
  ...
  listLinkedNotebooks()
  @identifies NoteStore[] by «
    for nb in result: yield (nb.noteStoreUrl, index=nb.shareKey) »
  ...
  authenticateToSharedNotebook(shareKey)
  @indexedby « shareKey »
  ...
}
```

Figure 2. The NoteStore contract

$$\begin{array}{ll} \textbf{Processes} & P = \mathtt{B}^{D,U}_{in,out} \mid m \mid P \| P \\ \textbf{Messages} & m = x\langle\, \kappa, \widetilde{b}\,\rangle \mid \kappa\langle\, \widetilde{b}\,\rangle \end{array}$$

Figure 3. Core CC syntax

ure 2), the adapter of the client blames Server1, the provider of the reference to the note store at Server2 and of the index. If in contrast, the client forges a token and passes it to authenticateToSharedNotebook, the client's adapter blames the client for using an index the adapter does not know and thus absolves Server1 from any responsibility for the result of the operation. Hence, CONSUL contracts can express and enforce the second property of the workflow.

In the following section, we precisely and formally describe how the system monitors messages from black boxes, checks for errors according to the given contracts, and assigns blame upon contract violations.

3. CONSUL Formally

We gradually introduce CC, a formal model for CONSUL, along with its properties. This model highlights key aspects of the design of CONSUL, and demonstrates formally how CONSUL meets the goals we set in Section 1.

3.1 Core CC: Distributed Black Boxes

Figure 3 shows the syntax of Core CC. Processes P represent distributed black-box components and the messages they exchange. There are two kinds of basic processes: black boxes $\mathtt{B}^{D,U}_{in,out}$ and messages m. A compound process $P^1 \| P^2$ represents the parallel composition of processes P^1 and P^2. We assume the standard structural equivalence.

A black box $\mathtt{B}^{D,U}_{in,out}$ corresponds to an application component in CONSUL. To capture our assumption that we do not have access to a component's source code, black boxes are opaque and we can observe only messages they send and receive. Black boxes receive messages on their open ports and send messages to open ports of other black boxes. To model TCP-like communication, we distinguish between *request ports* and *reply ports*. A request port x corresponds to a well-known TCP endpoint, whereas a reply port κ corresponds to an ephemeral TCP port. Annotations D, U, in and out on black boxes represent information about the port interface of a black box: port domain D is the unique set of

$$\frac{\kappa \in D \quad \kappa \notin U \quad \kappa \notin in \quad in' = in \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \xrightarrow{!x\langle\, \kappa,\widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in',out} \| x\langle\, \kappa, \widetilde{b}\,\rangle}$$

$$\frac{x \in in \quad out' = out \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \| x\langle\, \kappa, \widetilde{b}\,\rangle \xrightarrow{?x\langle\, \kappa,\widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in,out'}}$$

$$\frac{\kappa \in out \quad out' = out \setminus \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \xrightarrow{!\kappa\langle\, \widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in,out} \| \kappa\langle\, \widetilde{b}\,\rangle}$$

$$\frac{\kappa \in in \quad \kappa \notin U \quad in' = in \setminus \{\kappa\} \quad U' = U \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \| \kappa\langle\, \widetilde{b}\,\rangle \xrightarrow{?\kappa\langle\, \widetilde{b}\,\rangle} \mathtt{B}^{D,U'}_{in',out}}$$

Figure 4. Core CC reduction rules

request and reply ports a black box can open; input ports $in \subseteq D$ are the open ports of a black box; used ports $U \subseteq D$ are the reply ports a black box has opened and closed; output ports out are the reply ports of other black boxes a black box has not replied to yet.

Messages are of the form $y\langle\, mc\,\rangle$ where y indicates the target port of the message and mc is the message content. Request messages have the form $x\langle\, \kappa, \widetilde{b}\,\rangle$, where κ indicates the the reply port for the request and $\widetilde{b}$ is an arbitrary bitstream (representing the message's payload).[2] Reply messages have the form $\kappa\langle\, \widetilde{b}\,\rangle$; they are messages sent to the reply port of a previous request message.

In Core CC, processes evolve when black boxes consume and spawn messages. Figure 4 shows the reduction rules for Core CC. The rules are of the form $P \xrightarrow{a} P'$.[3] The first rule shows that black box $\mathtt{B}^{D,U}_{in,out}$ can produce a request message $x\langle\, \kappa, \widetilde{b}\,\rangle$ using a fresh reply port $\kappa \in D \setminus (U \cup in)$. The resulting black box $\mathtt{B}^{D,U}_{in',out}$ records that it has opened reply port κ (i.e., $in' = in \cup \{\kappa\}$). The second rule shows that black box $\mathtt{B}^{D,U}_{in,out}$ can consume request message $x\langle\, \kappa, \widetilde{b}\,\rangle$ provided $x \in in$, and records in its port interface that it has not yet sent a reply message to port κ (i.e., $out' = out \cup \{\kappa\}$). The third rule shows that black box $\mathtt{B}^{D,U}_{in,out}$ can spawn a reply message to reply port $\kappa \in out$. To record the use of the request port, the rule removes κ from out. The final rule consumes a reply message on port $\kappa \in in$. To ensure that reply ports are used at most once, the rule removes κ from in and adds it to U.

For example, the following trace represents two black boxes where one sends a request on x with reply port κ and message body $\widetilde{b}$ (1); the other black box responds (2); and the resulting final state of the black boxes (3):

$$\mathtt{B}^{D^1,\emptyset}_{\emptyset,\emptyset} \| \mathtt{B}^{D^2,\emptyset}_{\{x\},\emptyset} \xrightarrow{!x\langle\, \kappa,\widetilde{b}\,\rangle} \mathtt{B}^{D^1,\emptyset}_{\{\kappa\},\emptyset} \| x\langle\, \kappa, \widetilde{b}\,\rangle \| \mathtt{B}^{D^2,\emptyset}_{\{x\},\emptyset} \xrightarrow{?x\langle\, \kappa,\widetilde{b}\,\rangle} \tag{1}$$

$$\mathtt{B}^{D^1,\emptyset}_{\{\kappa\},\emptyset} \| \mathtt{B}^{D^2,\emptyset}_{\{x\},\{\kappa\}} \xrightarrow{!\kappa\langle\, \widetilde{b}'\,\rangle} \mathtt{B}^{D^1,\emptyset}_{\{\kappa\},\emptyset} \| \mathtt{B}^{D^2,\emptyset}_{\{x\},\emptyset} \| \kappa\langle\, \widetilde{b}'\,\rangle \xrightarrow{?\kappa\langle\, \widetilde{b}'\,\rangle} \tag{2}$$

$$\mathtt{B}^{D^1,\{\kappa\}}_{\emptyset,\emptyset} \| \mathtt{B}^{D^2,\emptyset}_{\{x\},\emptyset} \tag{3}$$

$$
\begin{array}{llcl}
\textbf{Processes} & P & ::= & ... \mid \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B) \\
\textbf{Messages} & m & ::= & ... \mid \underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, l\,\rangle \mid \underline{\kappa}\langle \widetilde{b}, C, l\,\rangle \\
\textbf{Configurations} & C & : & z \mapsto z \\
\textbf{Specs} & S & : & id \mapsto (c, vp, pp, ip) \\
\textbf{Contracts} & c & ::= & \mathtt{given}\ \mathsf{e}, \widetilde{id}\ \mathtt{returns}\ \mathsf{e}, \widetilde{id} \\
\textbf{Value Plugins} & vp & : & \widetilde{b} \mapsto \widetilde{\mathsf{v}} \\
\textbf{Port Plugins} & pp & : & \widetilde{b} \mapsto (\widetilde{x}, \widetilde{\mathsf{v}_\bot}) \\
\textbf{Parameter Plugins} & ip & : & (\widetilde{b} \mapsto \mathsf{v}) \vee (\widetilde{b} \mapsto \bot) \\
\textbf{Registry} & R & : & y \mapsto re \\
\textbf{Registry Entry} & re & ::= & (\diamond, \widetilde{pi}) \mid (id, \overline{bi}, \widetilde{pi}) \\
\textbf{Blame Info} & bi & ::= & (\mathsf{v}_\bot, \overline{l}, l) \\
\textbf{Blame Log} & L & ::= & \widetilde{le} \\
\textbf{Blame Log Entry} & le & ::= & (\mathbf{pre} : x, id, \mathsf{v}_\bot, l) \\
 & & \mid & (\mathbf{post} : \kappa, id, \mathsf{v}_\bot, \overline{l}) \\
 & & \mid & (\mathbf{conflict} : x, id, \overline{l}, id, \overline{l}) \\
 & & \mid & (\mathbf{unknown}\ \mathsf{v} : x, id, \overline{l})
\end{array}
$$

Figure 5. CC syntax

3.2 Adding CONSUL Adapters

We now extend CC with CONSUL adapters. Intuitively, an adapter wraps around a black-box component. The adapter and the component it wraps are together called a CONSUL-enhanced component. The adapter intercepts messages the wrapped black box sends, performs contract checks, and forwards these messages to their destination (which is either a black box, or, if the destination is CONSUL-enhanced, the destination's adapter). The adapter also intercepts messages intended for the black box it wraps, performs contract checks, and forwards these messages to the wrapped black box. The adapter holds sufficient information to determine which messages to intercept, where to forward them, and which contracts to enforce on these messages.

Figure 5 presents the syntax of CC, extending the syntax of Core CC. An adapter process $\mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B)$ wraps an *adapter pool* B, which consists of a black box, and a (possibly empty) list of Core CC messages that the black box has just sent and have yet to be processed by the adapter, or that the adapter has forwarded to the black box. Each adapter has a label l that uniquely identifies it and a blame log L that records information about contract violations.

Adapter annotations D, U, in and out have the same meaning as for black boxes. Port domain D of an adapter is a superset of the port domain of the black box it wraps, and also contains *request adapter ports* $\underline{x}$ and *reply adapter ports* $\underline{\kappa}$. Adapter ports are used for communication between CONSUL-enhanced components. For the remainder of the paper z ranges over ports in general while y ranges over black-box ports and $\underline{y}$ over adapter ports.

The adapter's configuration C tells the adapter where to forward messages it intercepts. It is a map from ports to ports, such that if $C(z) = z'$ then the adapter should forward intercepted messages intended for z to port z'. Port z' is either equal to z (meaning the destination is not known to be CONSUL-enhanced) or is an appropriate adapter port of the destination's adapter (which allows the adapter to communicate directly with the destination's adapter).

A specification S is a map from contract identifiers to a tuple that consists of a service contract c and contract plugins vp, pp and ip to extract information from messages that the adapter should check against c. For example, in Figure 2, S would map identifier NoteStore to the contract presented in the figure and its plugins. An adapter's S includes all the contracts an adapter knows. A contract's plugins vp, pp and ip extract contract-relevant information from message payloads. Plugin vp consumes a bitstream and returns a sequence of values $\widetilde{\mathsf{v}}$, the arguments for the contract's predicates. Plugin ip returns the index v for the contract or $\bot$ if the contract does not expect an index. Plugin pp extracts the request ports mentioned in the payload, and, if relevant, indexes for the contracts on those request ports. Note that the plugins are typically automatically derived from the contract and a parser for the message protocol. For the example in Section 2, the plugins are derived automatically given a parser for Thrift messages. We assume plugins are correct.

In a contract `given` $pre, \widetilde{id}_{pre}$ `returns` $post, \widetilde{id}_{post}$, predicates pre and $post$ correspond to the pre-condition and post-condition respectively for the service offered on a port x. Thus, pre is the predicate that the payload of a request message to x must meet and $post$ is the predicate for the payload of the corresponding reply. Both predicates are total boolean functions, i.e., they terminate. The sequences of contract identifiers $\widetilde{id}_{pre}$ and $\widetilde{id}_{post}$ correspond to **@identifies** clauses in the CONSUL IDL and indicate the contracts that should be enforced on ports mentioned in the payload of request and reply messages. (Specification S is used to extract ports mentioned in the payloads.) CC contracts are higher-order: they can express not only first order properties of arguments and results of service calls but also the expected behavior of ports that service calls consume and return.

An adapter's registry R maps black-box ports to information about the port's contract. The domain of R includes all black-box ports in the adapter's input ports in, but may also include black-box ports of other components that the adapter learns about. If $R(y) = (id, \overline{bi}, \widetilde{pi})$ then id is the contract for y and $\overline{bi}$ is information the adapter collects for y to assign blame upon contract failures. Blame info bi is a triplet of an index $\mathsf{v}_\bot$, a set of labels $\overline{l}$, and a blame label l. Index $\mathsf{v}_\bot$ is an index the adapter has "learned" for the contract of port y (either a value v or the default index $\bot$). Labels $\overline{l}$ are the labels of adapters that have "promised" that the service on y given index $\mathsf{v}_\bot$ behaves according to contract id. Blame label l is the label of the adapter that has "promised" to use the service on y given index $\mathsf{v}_\bot$ as a service that behaves according to contract id. That is, $\overline{l}$ are the adapters to blame if the post-condition is violated, and l is blamed if the pre-conditions are violated. These labels are like the *server* and

[2] We use notation $\widetilde{t}$ and $\overline{t}$ to indicate a sequence and a set of ts respectively. Here b stands for bits.

[3] Label $a \in \{?m, !m\}$ indicates consumption or spawn of message m.

$$\frac{(y \notin in \text{ and } y \notin dom(R)) \text{ or } (y \notin in \text{ and } R(y) = (\diamond, \widetilde{pi}))}{\begin{array}{l}\mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B \| y\langle\, mc\,\rangle) \longrightarrow \\ \qquad \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B) \| y\langle\, mc\,\rangle\end{array}}$$

$$\frac{y \notin in \text{ and } y \in D}{\begin{array}{l}\mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B) \| y\langle\, mc\,\rangle \longrightarrow \\ \qquad \mathtt{A}^{D,U}_{in,out}\{C,S,R,l,L\}(B \| y\langle\, mc\,\rangle)\end{array}}$$

Figure 6. Rules for messages that bypass an adapter

client blame labels in formal models of higher-order contracts. Under partial deployment we use † as the label for all non-CONSUL-enhanced black boxes.

The registry is updated as the adapter learns about ports and their contracts from messages it intercepts. Multiple messages may talk about the same port (and so we have a set of server labels in the blame info instead of just one). If the messages contain contradictory information about port y then we replace $R(y)$ with a conflict entry $(\diamond, \widetilde{pi})$, record a contract conflict, and stop checking contracts for y.

Registry entries include provenance information $\widetilde{pi}$ which does not affect how adapters intercept messages, check contracts, assign blame, or forward messages. Provenance is a technical artifact to prove blame correctness. We describe provenance in Section 3.3.

Adapter messages $\underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, l\,\rangle$ and $\underline{\kappa}\langle\, \widetilde{b}, C, l\,\rangle$ are sent between adapters. In addition to the message payload $\widetilde{b}$, they include the label l of the sender adapter and a transmitted configuration C, which informs the receiver of other CONSUL-enhanced black boxes the sender knows of. The reduction semantics of CC include the reductions of Core CC (which allow a black box in a CONSUL-enhanced component to consume and spawn messages within its adapter pool). Additional reduction rules enable an adapter to isolate a black box in its pool from other black boxes and regulate how messages move in and out of the pool. There are 10 extra rules, in three groups: (1) rules for messages that bypass adapters; (2) rules for black-box messages that an adapter intercepts and forwards; (3) rules for sending and receiving adapter messages. We discuss all three groups but show only representative rules that capture the subtleties of CONSUL. The appendix includes all reduction rules.

Messages that Bypass Adapters Depending on an adapter's state, some messages to and from the adapter's black box are not intercepted by the adapter. Figure 6 shows these rules. For messages sent by an adapter's black box, this occurs when: (1) the target port is not an input port of the adapter (i.e., the message is not sent by the black box to itself); and either (2a) the adapter's registry does not know of a contract for the target port; or (2b) the adapter's registry has conflicting information for the contract for the target port. Condition (1) ensures that the adapter doesn't allow messages the black box sends to itself to escape the pool.

For messages sent to the adapter's black box, the adapter does not intercept the message if its registry indicates that no contract checking is required (since $y \notin in$ implies $y \notin dom(R)$). Note that the adapter allows messages into the pool only if they are intended for the black box component (which $y \in D \setminus in$ entails).

Black-Box Interaction When an adapter intercepts a message (that doesn't meet the bypass conditions above), the adapter checks the message against the appropriate contract before forwarding to its destination port y. Contract checking and message forwarding depend on whether the adapter's configuration C considers the destination to be CONSUL-enhanced: If $C(y) = y$ then y is not CONSUL-enhanced. Here, we describe reduction rules for this case (i.e., partial deployment), and consider the rules for communicating with a CONSUL-enhanced component below.

There are four rules in this group. The first fires when the adapter intercepts a request message that its black box sends to another black box. The second fires when the adapter receives a request message from a black box to the adapter's black box. The third fires when the adapter intercepts a reply message that its black box sends to another black box. The fourth fires when the adapter receives a reply message from a black box to the adapter's black box. Each of the four rules of this group is analogous to one of the four rules of Core CC. Similar to their Core CC analogues, the rules of this subsection update the in, U and out sets of an adapter as the adapter opens, receives or uses a reply port κ.

All these rules perform the same tasks to update the adapter's state before forwarding the message. We discuss these tasks for one of the rules and then explain how the details of the individual tasks differ between the four rules.

Figure 7 displays the rule that fires when the adapter intercepts a request message $x\langle\, \kappa, \widetilde{b}\,\rangle$ that its black box sends to another black box. The rule triggers when $x \notin in$ and $C(x) = x$. We walk through the six tasks the adapter completes before forwarding the message to its destination.

First, the adapter gets a reply port $\kappa \in D \setminus (in \cup U)$ and adds it to the input ports: $in' = in \cup \{\kappa\}$.

Second, the adapter parses the intercepted message to extract relevant information, using meta-function $message\_info$. This meta-function consumes the target port x of the intercepted message, the payload $\widetilde{b}$, the adapter's specifications S, and the adapter's registry R. The last argument to $message\_info$ indicates whether it should extract information for the **given** or the **returns** part of the contract, i.e., whether it is checking a pre- or post-condition.

Meta-function $message\_info$ returns the contract identifier id for x (according to registry R), the contract c that corresponds to id (according to S), and the index $\mathsf{v}_\perp$. It also uses registry R to return blame labels to use in the event of contract violations (the server blame labels $\overline{l^s}$ and the blame label l^c for x). It also returns arguments for the contract predicate $(\widetilde{\mathsf{v}})$, and any ports mentioned in the payload along with their contract identifiers and indexes ($\widetilde{x}$, $\widetilde{id^x}$, and $\widetilde{\mathsf{v}^x_\perp}$). It also returns provenance information $\widetilde{pi}$ that we ignore here.

$$\frac{\begin{array}{c} x \notin in \qquad C(x) = x \qquad \kappa \in D \qquad \kappa \notin in \qquad \kappa \notin U \qquad in' = in \cup \{\kappa\} \\ (id, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{id^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = message\_info(x, \widetilde{b}, S, R, given\_ids) \\ pr = identified\_ports(\widetilde{x}, \widetilde{id^x}, \widetilde{\mathsf{v}^x_\perp}, R, l) \\ L^{unk} = unk\_blame(x, id, \mathsf{v}_\perp, \{l^c\}, R) \qquad L^{pred} = pred\_blame(\mathbf{pre}, x, id, \mathsf{v}_\perp, pre(c), \widetilde{\mathsf{v}}, l^c) \\ L^{conf} = conf\_blame(pr.conf, R, in, \{l^c\}) \\ R^u = update\_registry(pr, \{l^c\}, l^c, R, in) \qquad C^u = update\_config(pr.new, C) \\ C' = C^u \cup \{\kappa \mapsto \kappa\} \quad R' = R^u \cup \{\kappa \mapsto (id, \perp, \overline{l^s}, l^c, \perp \rightsquigarrow \widetilde{pi})\} \quad L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B \| x\langle\, \kappa, \widetilde{b}\,\rangle) \longrightarrow \mathtt{A}^{D,U}_{in',out}\{C', S, R', L', l\}(B) \| x\langle\, \kappa, \widetilde{b}\,\rangle}$$

Figure 7. Rule for request message from CONSUL-enhanced black box to non CONSUL-enhanced black box

An invariant of CC is that if $in \in dom(R) \setminus in$ then client label l^c is the label l of the adapter. That is, the component is the client of all ports it knows about that aren't its own input ports. Similarly, if $in \in dom(R) \cap in$ then the label l of the adapter is in the set of server labels $\overline{l^s}$. That is, the component is the server responsible for all its input ports. We return to these invariants in Section 3.3.

Third, for each port (and the port's contract information) mentioned in the message payload, the adapter uses its registry R to determine whether (a) it is a new port that the adapter knew nothing about previously; (b) the port's contract information is consistent with what the adapter already knew about the port; or (c) the port's contract information conflicts with the adapter's prior knowledge. The third line of the rule carries out this task using meta-function $add\_identified\_ports$, which returns a record pr with three fields containing information about the new ports (field $pr.new$), consistent ports (field $pr.cons$), and conflicted ports ($pr.conf$).

Fourth, the adapter checks whether the adapter already knows the index $\mathsf{v}_\perp$ for the contract of x or not, and also checks the appropriate predicate of the contract. Meta-functions unk_blame and $pred\_blame$ perform these checks. Each returns an appropriate blame log entry if it detects a contract violation using its label arguments. Blame log entry $(\mathbf{unknown}\ \mathsf{v}_\perp : x, id, \{l^c\})$ is returned by unk_blame, indicating that index $\mathsf{v}_\perp$ for the contract contract id of port x is not known to the adapter and adapter l^c is responsible for this unknown index. Blame log entry $(\mathbf{pre} : x, id, \mathsf{v}_\perp, l^c)$ is returned by $pred\_blame$ indicating violation of the precondition of contract id given index $\mathsf{v}_\perp$ for for port x and adapter l^c is responsible for the violation.

Fifth, the adapter uses meta-function $conf\_blame$ to produce blame log entries for the conflicted ports in $pr.conf$ that are not in the input ports in of the adapter. An entry $(\mathbf{conflict} : x^{conf}, id^{old}, \overline{l^{old}}, id^{conf}, \{l^c\})$ says that the adapter has associated port x^{conf} with contract id^{old} according to adapters $\overline{l^{old}}$ while adapter l^c claims that x^{conf} has contract id^{conf}.

Sixth the adapter updates its registry R, its configuration C, and its blame log L accordingly. For new ports, meta-function $update\_registry$ adds new entries to the registry using the supplied server and client labels. For consistent ports, it adds the server and client labels to the existing registry entry. For conflicted ports, it typically replaces the existing registry entry with a conflict entry. However, if the conflicted port is in the input domain in of the adapter, it leaves the registry entry unchanged, since the adapter is responsible for the input port and should continue to enforce contracts on messages to it. Meta-function $update\_config$ adds entries $x^{new} \mapsto x^{new}$ to C for every newly identified port x^{new}. Finally, the adapter extends C and R with appropriate entries for the newly allocated reply port κ and adds any blame log entries to blame log L.

rule	**send request**	**receive request**	**send reply**	**receive reply**
condition	$x \notin in$, $C(x) = x$	$x \in in$, $C(x) = x$	$\kappa \in out$, $C(\kappa) = \kappa$	$\kappa \in in$, $C(\kappa) = \kappa$
opens port	κ	–	–	–
unk_blame	$\{l^c\}$	$\{l^c\}$	$\overline{l^s}$	$\overline{l^s}$
$pred\_blame$	**pre**, l^c	**pre**, l^c	**post**, $\overline{l^s}$	**post**, $\overline{l^s}$
$conf\_blame$	$\{l^c\}$	$\{l^c\}$	$\overline{l^s}$	$\overline{l^s}$
$update\_registry$	$\{l^c\}, l$	$\{l^c\}, l$	$\overline{l^s}, l^c$	$\overline{l^s}, l^c$

Figure 8. Differences between black-box interaction rules

After completing all six tasks, the adapter consumes the message $x\langle\, \kappa, \widetilde{b}\,\rangle$ from its pool and spawns it into its context.

The other three rules for black-box interaction follow the same structure as the rule in Figure 7. However, they differ from this rule as follows. First, they fire under different conditions. Second, they do not allocate a reply port κ. Third, they use different blame labels as arguments to the various meta-functions. Fourth, the rules that concern reply messages check the post-condition of the contract of the target port instead of the pre-condition and do not add to the adapter's registry and configuration entries for the reply port κ (those are already there due to the corresponding request message). Figure 8 summarizes the differences.

CONSUL-*enhanced Interaction* CC has four more rules that fire when an adapter forwards a message to a CONSUL-enhanced target or receives a message from a CONSUL-enhanced origin. These rules have the same structure as the corresponding rules for black-box interaction.

Figure 9 displays the CONSUL-enhanced interaction rule that corresponds to the black-box interaction rule from Figure 7. Notice that the rules are almost identical. They have only four differences. First, the rule in Figure 9 fires when $C(x) = \underline{x}$, i.e., when the adapter considers the target of the message to be CONSUL-enhanced. Second, the rule opens two new ports κ and $\underline{\kappa}$ and adds both to the input ports in of

$$\frac{\begin{array}{c} x \notin in \qquad C(x) = \underline{x} \qquad \kappa, \underline{\kappa} \in D \qquad \kappa, \underline{\kappa} \notin in \qquad \kappa, \underline{\kappa} \notin U \qquad in' = in \cup \{\kappa, \underline{\kappa}\} \\ (id, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{id^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = \mathit{message\_info}(x, \widetilde{b}, S, R, \mathit{given\_ids}) \\ pr = \mathit{identified\_ports}(\widetilde{x}, \widetilde{id^x}, \widetilde{\mathsf{v}^x_\perp}, R, l) \\ L^{unk} = \mathit{unk\_blame}(x, id, \mathsf{v}_\perp, \{l^c\}, R) \qquad L^{conf} = \mathit{conf\_blame}(pr.conf, R, in, \{l^c\}) \\ R^u = \mathit{update\_registry}(pr, \{l^c\}, l^c, R, in) \qquad C^u = \mathit{update\_config}(pr.new, C) \\ C' = C^u \cup \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \quad R' = R^u \cup \{\kappa \mapsto (id, \perp, \overline{l^s}, l^c, \perp \rightsquigarrow \widetilde{pi})\} \quad L' = L \cdot L^{unk} \cdot L^{conf} \\ C^t = \mathit{transmitted\_config}(pr.cons, C') \cup \{\underline{\kappa} \mapsto \kappa, \kappa \mapsto \underline{\kappa}, \underline{x} \mapsto x\} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B \| x\langle \kappa, \widetilde{b} \rangle) \longrightarrow \mathtt{A}^{D,U}_{in',out}\{C', S, R', L', l\}(B) \| \underline{x}\langle \underline{\kappa}, \widetilde{b}, C^t, l \rangle}$$

Figure 9. Rule for request message from CONSUL-enhanced black box to CONSUL-enhanced black box

the adapter. Third the rule does not check the pre-condition of contract id. This is because under CONSUL-enhanced interaction adapters share the burden of contract checking to reduce duplicate contract checks. Thus only the adapter of the destination of the intercepted message checks the pre-condition. Fourth, the adapter does not spawn the same message as the one it receives. Instead, it replaces the ports x and κ in the message with their adapter analogues $\underline{x}$ and $\underline{\kappa}$. In addition, the adapter supplements the payload of the message $\widetilde{b}$ with a configuration C^t. This configuration, produced by meta-function *transmitted_config*, includes all entries in the configuration C of the adapter that correspond to identified consistent ports. The adapter augments the transmitted configuration with entries for the newly opened reply ports κ and $\underline{\kappa}$ and, the target port of the intercepted message x. The inclusion of the transmitted configuration in adapter messages (1) contains sufficient information for the receiver adapter to translate the adapter message into a black-box one and, (2) informs the receiver adapter of CONSUL-enhanced black boxes they may ignored. When an adapter receives an adapter message, it merges the transmitted configuration into its configuration. Thus transmitted configurations promote CONSUL-interaction and reduce the cost of contract checking. We assume adapters that communicate using adapter messages are mutually trusted.

The other three rules for CONSUL-enhanced interaction follow exactly the same structure as the rule in Figure 9. They differ to each other in a similar manner that black-box interaction rules differ to each other. The corresponding table to that in Figure 8 has three important differences. First, when adapters consumed an adapter (black-box) message they check if their configuration maps the target adapter (black-box) port to a black-box (adapter) port. Second, when adapters consume an adapter message from their context (**receive request** and **receive reply**), they use the message label l^m as the client label for their various meta-functions. Third, **send request** and **receive reply** do not perform predicate checks. Finally, adapters translate an adapter message into a black box one and spawn the translated message in their adapter pool and vice versa for a black-box message consumed from an adapter's pool and forwarded to the adapter's context.

3.3 Properties of CC

In this section, we establish the two key properties of CC: *adapter transparency* and *correct blame assignment*.[4] First, though, we discuss certain conditions that adapters must satisfy to achieve these properties.

We capture invariants of CC processes using a well-formedness predicate. These include relationships between annotations on black boxes and adapters (e.g., $in \cap U = \emptyset$), uniqueness of adapter labels, and consistency of configurations C and specifications S across different adapters. Preservation of well-formedness is the basis for proving adapter transparency and correct blame assignment for CC. We provide the full definition of well-formedness in the appendix.

Theorem 1 (Preservation of Well-formedness). *If P is well-formed and $P \xrightarrow{\alpha} P'$ then P' is well-formed.*

3.3.1 Adapter Transparency

To prove that adapters are transparent (i.e., do not affect the messages black boxes send and receive, as described in Section 1), we define a notion of contextual equivalence between processes in CC.

Recall that in CC messages that black boxes consume and spawn appear as labels on reduction steps. We use these labels to define observational simulation and equivalence for CC. A process P simulates P' iff in any well-formed context $\mathcal{E}$, where $\mathcal{E} = [\bullet]$ or $\mathcal{E} = \mathcal{E} \| P$, when P, after possibly some reduction steps, takes a step with label α then P', after possibly some reduction steps, also takes a step with label α:

Definition 2 (Black Box Preorder $\sqsubseteq_{\mathtt{B}}$). *For P and P', $P \sqsubseteq_{\mathtt{B}} P'$ iff for all contexts $\mathcal{E}$ such that $\mathcal{E}[P]$ and $\mathcal{E}[P']$ are well-formed, if $\mathcal{E}[P] \xrightarrow{\widetilde{\beta}}_* \xrightarrow{\alpha} Q$ then $\mathcal{E}[P'] \xrightarrow{\widetilde{\gamma}}_* \xrightarrow{\alpha} Q'$ where $\alpha, \gamma \in \{?m, !m, \cdot\}$ and $\beta \in \{?m, !m\}$.*

Two processes are equivalent if one simulates the other:

Definition 3 (Black Box Equivalence $\cong_{\mathtt{B}}$). *Terms P and P' are black box equivalent, $P \cong_{\mathtt{B}} P'$, iff $P \sqsubseteq_{\mathtt{B}} P'$ and $P' \sqsubseteq_{\mathtt{B}} P$.*

With the definition of contextual equivalence in hand, we can show the transparency of adapters. This boils down to

[4] Complete formalisms and proof sketches are in the appendix.

proving that a wrapped black box is contextually equivalent to the black box without the wrapper:

Theorem 4. $\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}} \cong_{\mathtt{B}} \mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C, S, R, L, l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}})$

Adapter transparency has pragmatic significance for CONSUL. It implies that a user can add adapters to a collection of distributed communicating components without affecting their behavior.

3.3.2 Correct Blame Assignment

The pragmatic value of a contract system depends on the correctness of blame assignment. Informally, Dimoulas et al. [9] define that a contract system assigns blame correctly if it blames the component that vouched for the value to meet its contract that triggered the violation. Formally, they extend their contract calculus with provenance information and prove that the blame label reported upon contract violation matches the provenance of the value that triggered the violation. The provenance information is not used in contract enforcement, but provides a sound basis for specifying blame correctness.

We use the same approach and add provenance information to CC to specify correct blame. Due to the black-box nature of components in CC, we cannot use the same provenance tracking mechanism of Dimoulas et al. [9]. Instead, each adapter records in its registry R provenance information about ports it "learns" from intercepted messages. The recorded provenance information is the $\widetilde{pi}$ part of a registry entry re in Figure 5. An element pi of $\widetilde{pi}$ is either of the form $\mathsf{v}_\perp \rightsquigarrow l$ or $\mathsf{v}_\perp \rightsquigarrow \widetilde{pi}$. Provenance element $\mathsf{v}_\perp \rightsquigarrow l$ means that the adapter "learned" that the index $\mathsf{v}_\perp$ for the contract of a port directly from adapter l. Provenance element $\mathsf{v}_\perp \rightsquigarrow \widetilde{pi}$ means the provenance of index $\mathsf{v}_\perp$ is $\widetilde{pi}$. Thus $\widetilde{pi}$ is a forest where each tree describes the provenance of a known index for the contract of a port in the domain of R. The leaves of each tree are adapter labels; the labels of the origin of the contract of a port given an index. We use meta-function $\mathit{origin\_of}((y, \mathsf{v}_\perp), R)$ to obtain the set of labels that consist the *origin* of the contract of a non-conflicted port y given index $\mathsf{v}_\perp$ in registry R.

The reduction rules of CC compute provenance information for identified ports and newly opened reply ports and update the registry accordingly. The provenance of contracts of ports identified from request messages is the label of the adapter that sent the request. For example, in Figures 7 and 9 the last argument l of the call to the $\mathit{identified\_ports}$ meta-function is the label used to construct the provenance entries for the contracts of identified ports. Contracts of ports identified from reply messages and newly opened reply ports inherit the provenance of the corresponding request port. For example, in Figures 7 and 9, the rule uses the provenance entry $\widetilde{pi}$ for x to construct the provenance for the contract of newly opened x. Note that like Dimoulas et al. [9], the reduction rules of CC construct provenance information for the contracts of ports independently from the way they manipulate blame labels. Thus provenance information in CC is a sound basis for specifying and proving correct blame.

We first use a *blame-consistent with provenance* relation $\triangleright\mathcal{R}(P)$ (defined in the appendix) that requires for the registries of all the adapters in P, the origin of the contract of a non-conflicted port given an index matches the server blame labels for the the contract of the port given the same index. Blame consistency with provenance is an invariant of the semantics of CC and together with well-formedness guarantees correct blame.

Informally, correct blame assignment requires that if a well-formed and blame-consistent with provenance P takes a step and this step results in an adapter in P detecting contract violations, then if an additional entry in the blame log of the adapter after the step is (1) a violation of a pre-condition, CC blames always the sender of the request message that caused the violation; (2) a violation of a post-condition, CC blames the origin of the contract of the target port of the reply message that caused the violation. Formally, the correct blame theorem is (the actual theorem, provided in the appendix, includes also cases for blame due to conflicts and unknown indices):

Theorem 5. *Let* $P = \mathcal{E}[\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B)]$. *If* P *is well-formed,* $\triangleright\mathcal{R}(P)$ *and* $P \longrightarrow P'$ *where* $P' = \mathcal{E}'[\mathtt{A}^{D,U'}_{in',out'}\{C', S, R', L', l\}(B')]$ *and* $L' = L \cdot L^*$ *then:*

1. *if* $(\mathbf{pre} : x, id, \mathsf{v}_\perp, l^*)$ *then*
   (a) *if* $P = \mathcal{E}'[x\langle\, \kappa, \widetilde{b}\,\rangle \| \mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B)]$ *then* $l^* = \dagger$*;*
   (b) *if* $P = \mathcal{E}'[\underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, l^m\,\rangle \| \mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B)]$ *then* $l^* = l^m$*;*
   (c) *if* $P = \mathcal{E}[\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(x\langle\, \kappa, \widetilde{b}\,\rangle \| B')]$ *then* $l^* = l$
2. *if* $(\mathbf{post} : y, id, \mathsf{v}_\perp, \overline{l^*}) \in L^*$ *then* $\overline{l^*} = \mathit{origin\_of}((y, \mathsf{v}_\perp), R)$*.*

4. CONSUL in Practice

We have developed a prototype implementation for CONSUL. It consists of the network adapter that was formalized in Section 3, and an interposition library for redirecting connections through the adapter. The network adapter is approximately 3,800 non-empty lines of Python code. The adapter runs in its own OS process and is responsible for checking contracts and spawning messages. The adapter stores service records in its disk-backed store. Service records remain in memory until a limit in the cache is reached, or the network adapter process exits. When a cache miss occurs, the file store is checked for the service record and the service record is fetched to memory if found. We leave garbage collection of the store as future work. In our case studies all information in the store is ephemeral and can be safely collected after some time out. Protocols can be supported by CONSUL by writing a plugin for the tool. We have plugins for Thrift (in 150 lines of code), REST (100 lines) and SOAP (400 lines). To handle issues of encrypted communications (i.e., a component using SSL), the adapter and the component it enhances share certificates to allow the adapter to decrypt the message to check contracts. Upon deployment, CONSUL-enhanced components are loaded with an interposition library that ensures that any connect system call made by the application component is intercepted by CONSUL. The interposition library contacts the adapter redirector to check if a new connection should bypass the adapter.

We have used CONSUL to harden the interfaces of three real-world off-the-shelf distributed components: Evernote (from Section 2), the Twitter API, and an online correspondence chess service. CONSUL's contracts are expressive enough to capture properties of the components that were described only informally in their documentation. The complete CONSUL contracts for the three case studies and an additional one are in the appendix. We discuss their most interesting aspects in the remainder of the section.

4.1 Evernote

Evernote demonstrates how CONSUL copes with four of the challenges we discuss in Section 1: (i) the Evernote server is a black box for its clients; (ii) CONSUL can only be partially deployed as clients do not have access to Evernote servers; (iii) CONSUL cannot change the communication pattern between the Evernote server and its clients; and (iv) CONSUL has to interoperate with Evernote's Thrift-based API and also its simpler HTTPS protocol for OAuth requests.

Evernote documentation describes many first-order properties that both Evernote server components and clients have to defensively check. These include: non-empty strings, bounds checks on integers, malformed GUIDs, strings that are too long, missing parameters that could not be marked as required due to Thrift limitations, and strings not matching certain patterns (e.g. valid MIME type). CONSUL is able to enforce these properties and we wrote contracts for many of them (one of which was described in Section 2 in the findNotes operation). These contracts help the early detection of malformed messages.We also expressed properties about the correct use of tokens similarly to Section 2 despite some of these tokens being initially given over a different protocol.

4.2 Twitter

Twitter's REST API[5] allows access to information about tweets and followers, and is representative of many REST APIs. This case study highlights how we can re-express the API's example-based documentation as a precise and executable specification.

Property 1. (Well-formed data) The documentation requires that the operation to fetch tweets is given either a user ID or screen name. We encode this disjunctive requirement with a contract. We found that very few API libraries actually defensively checked this requirement. Instead they relied only on the server to report back an error message. The contract can also check for appropriate data types, like lists meeting a length requirement or proper date formats.

Property 2. (Valid OAuth tokens) Similar to Evernote, Twitter uses the OAuth protocol to perform authentication. We encoded the validity checks in a similar way to Evernote's use of tokens described in Section 2.

Property 3. (Valid User and Tweet IDs) Twitter makes extensive use of tokens to identify different types of records, like tweets and users. We found it useful to track these IDs to ensure the proper discovery and use of the various IDs. The following contract shows how we capture the proper use of IDs with the Twitter API for Tweet IDs.

```
service Twitter {
  /1.1/statuses/user_timeline(req)
  @identifies Tweet[] by « for t in result: yield (index='t_' + t.id) »

  /1.1/statuses/retweet/<id>.json(req)
  @indexedby « 't_' + id »
  @ensures « 'does not exist' not in result.get('errors') »
}
```

Here the operation name is the HTTP path and req is a structured representation of the request body. The index is derived from the id that comes from the request URL (which is bound to the variable id declared in the operation name).

4.3 Chess

Xfcc is a popular web service specification for playing correspondence chess[6] with many different service providers that follow the protocol. This case study showcases that CONSUL can check contracts with a diversity of server and client implementations and can find server and client violations of the specification. The standard is specified using WSDL with SOAP as the underlying message protocol. It supports two operations: GetMyGames returns all games the user is playing in and the state of the game, and MakeAMove performs a game action (e.g., move a piece, offer a draw).

Property 1. (Valid game IDs used) The Xfcc documentation states that if a client provides an invalid game ID in MakeAMove, then a return code InvalidGameID should be returned. We found that two popular Xfcc servers returned a database error page rather than the appropriate error code. We also found that a client was unable to interpret the return code, making a move look successful to the client user. We were able to express this contract in a similar way to how we express valid IDs in the Twitter case study above.

Property 2. (Accept draw when allowed) Xfcc allows a player to offer a draw and accept an offered draw (as long as another move was not made after the offer). To express the constraint we made the index based on the game ID, move count, and whether the draw was indeed offered. When a draw was offered, two services were identified: one for accepting the draw and one for not accepting the draw. When a draw was not offered, it would yield one service. When a client attempts to accept a draw for when not offered, the

[5] https://dev.twitter.com

[6] http://xfcc.org/

client will be blamed. When a server does not approve a valid draw acceptance, the server will be blamed.

5. Performance

We evaluate CONSUL's performance to understand its impact on components it enhances. We analyze the time, memory, and network overhead on the three case studies described in Section 4. For our experimental setup, we developed a test suite for each case study. Each workflow exercises the contracts from Section 4. Operations that identify a service create fresh tokens to grow the adapter's store. We developed mock servers to provide responses to the client requests in the test suite. We opt to mock servers for two reasons: (1) mocking the servers removes several sources of measurement noise; (2) we cannot deploy the real servers and performing our experiments on production servers violates their terms of use.

We collect four measurements for each test suite. First, we record the time each request takes for the test client and for the adapter performing the original request on the mocked server; the difference between these two measurements yields the latency the adapter creates. In our experimental setup, adapters check all contracts. Second, we record the amount of memory used by adapters and the size of client adapters' store. Finally, we measure the adapter-to-adapter traffic (not including the original request or response) in the TCP stream. Our experiments are set up with full deployment in order to create the most amount of adapter traffic. We take measurements only for the clients' adapters as they are the hub for all communication in our experiments.

We ran our experiments on a 3 GHz Intel Core i7 processor with 2 GB of DDR3 memory with all components communicating over the loopback adapter. Figure 10 shows our measurements for each test suite. The production server latency on the operations we tested was between 250ms – 500ms. The average adapter latencies for Twitter, Chess, and Evernote were 30ms, 46ms, and 53ms respectively. The amount of RAM used by the adapter and latency did not increase as requests were made. The rate of increase in the disk-backed store for Twitter, Chess, and Evernote was 6.8kb per request, 1.7kb per request, and 3.3kb per request respectively. The rate of increase in store size as well as network overhead was dependent on how many services were identified. The average network overhead for Twitter, Chess, and Evernote was 105kb per request, 76kb per request, and 92kb per request respectively. Variance in network overhead is due only to the different operations made and does not change over each set of requests in a trial. Network overhead and store size did not have a dominant effect on latency; Twitter had the largest store and highest average network overhead yet the lowest latency. Instead, we found that latency was largely dependent on the implementation of the network plugin; the REST plugin used a more efficient marshaller and handled sockets better than the other plugins.

6. Related work

Existing frameworks for the composition of distributed components can enforce higher-order behavioral contracts similar to those of CONSUL but assume that components are written and deployed in a particular manner. For example, CORBA [24] and Java RMI [26] require all components to use their libraries. CONSUL can support these middlewares given an appropriate CONSUL adapter plugin.

Behavioral Interface Specification Languages (BISLs), such as JML [18], have extensions for specifying and enforcing higher-order behavioral contracts for communicating components. However, these languages are tightly coupled with particular component-implementation languages or families of languages. For instance, JML is designed for Java programs and comes with particular features to handle inheritance. Also, tools based on these languages re-write programs to insert checking probes. Thus BISLs and their contract checking tools are not language-agnostic. In contrast, CONSUL and its IDL are language-agnostic and do not modify components' code. Some features of CONSUL's IDL, such as pre- and post-conditions, are common with most BISLs. Others, such as @identifies, are unique to CONSUL. Runtime verification tools, such as Monitor-Oriented Programming (MOP) [7], can in principal enforce higher-order behavioral contracts in a language agnostic manner (with appropriate plugins). However, these are general frameworks and building a contract system on top of them requires solving the semantic issues CONSUL solves.

Finally, there is much research on enforceable synchronization and quality of service specifications for distributed components that do not overlap with CONSUL's higher-order behavioral contracts. For example, finite state machines can constrain WSDL-defined interactions [19] and web browsing [12]. BPEL [16] is a specification language for the orchestration of web services. It assumes a common communication bus for all components in an application. Multi-party session types [13] assume a global choreography protocol that is broken into locally enforceable pieces. Extensions to multi-party session types check dynamically first-order behavioral contracts for messages [6] similar to CONSUL's pre- and post-conditions on operations. Dynamic verification of multi-party session types shares the same motivation as CONSUL (e.g., support for heterogeneous or black-box components) [14, 15]. Other tools express and enforce quality of service specifications [2, 25].

7. Conclusion

CONSUL enhances the interfaces of distributed components with higher-order behavioral contracts. CONSUL contracts express and enforce semantically rich properties of these components. To accommodate the heterogeneity of real-

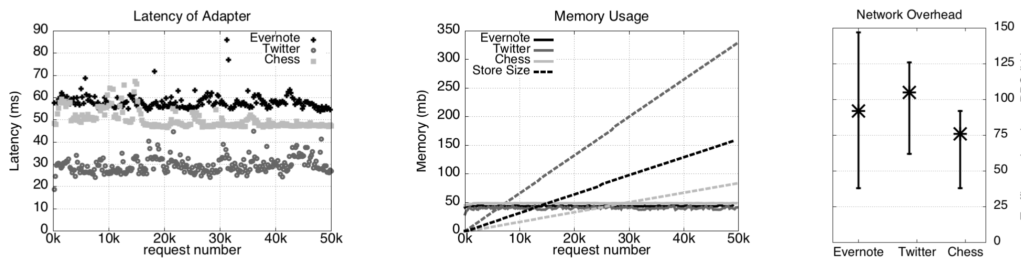

Figure 10. The charts show the time, memory, and network overhead for each case study. The left chart shows the latency of the adapter as requests are made. Each point is the average from the 250 requests around it. The middle chart shows the resident set size of the adapter and the dashed lines show the sizes of the store on disk. The right chart shows the average amount of adapter traffic per RPC. 95% confidence intervals are indicated by vertical bars.

world distributed components, CONSUL treats components as black boxes. It does not depend on or modify the source code of components and does not change the way distributed components communicate.

Thus CONSUL can improve the correct and reliable composition of distributed components, including legacy components, and facilitate their debugging and maintenance.

Acknowledgements

This research is supported in part by the National Science Foundation under Grant Nos. CCF-1421770, 1524052, 1054172, and 1237235.

References

[1] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007. URL `http://thrift.apache.org/static/files/thrift-20070401.pdf`.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Opearting Systems Design & Implementation*, pages 259–272. USENIX Association, 2004.

[3] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3):199–208, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer Berlin Heidelberg, 2004.

[5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7): 38–45, 1999.

[6] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *Proceedings of the 21st International Conference on Concurrency Theory*, pages 162–176, 2010.

[7] F. Chen and G. Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 569–588, 2007.

[8] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service definition language (WSDL). Technical report, 2001. URL `http://www.w3.org/TR/wsdl`.

[9] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *Proc. 38th SOSP*, pages 215–226, 2011. ISBN 978-1-4503-0490-0. .

[10] Evernote. Evernote developer documentation. URL `https://dev.evernote.com/doc/`.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, 2002.

[12] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering*, pages 235–244, 2010.

[13] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, 2008.

[14] R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and K. Honda. Practical interruptible conversations - distributed dynamic verification with session types and Python. In *Runtime Verification - 4th International Conference*, pages 130–148, 2013.

[15] L. Jia, H. Gommerstadt, and F. Pfenning. Monitors and blame assignment for higher-order session types. Technical Report 15-004, Carnegie Mellon University CyLab, 2015.

[16] M. B. Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS 2nd Edition*. Packt Publishing, 2006. ISBN 1904811817.

[17] R. Kramer. iContract - the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, 1998.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *Software Engineering Notes*, 31(3):1–38, 2006.

[19] Z. Li, Y. Jin, and J. Han. A runtime monitoring and validation framework for web service interactions. In *Proceedings of the Australian Software Engineering Conference*, pages 10–79, 2006.

[20] S. McGregor, T. Russ, N. Wilde, J. P. Gabes, W. Hutchinson, D. Duhon, and A. Raza. Experiences implementing interoperable SOA in a security-conscious environment. Technical Report S2ERC-TR-307, S2ERC, 2012.

[21] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[22] B. Meyer. Design by contract. In *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

[23] B. Meyer. Applying design by contract. *IEEE Computer*, 25 (10):40–51, 1992.

[24] Object Management Group. CORBA component model. Specification Version 3.3, 2012. URL `http://www.omg.org/spec/CORBA/3.3/`.

[25] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, pages 115–128. USENIX Association, 2006.

[26] J. Waldo. Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

[27] N. Wilde, J. Coffey, T. Reichherzer, and L. White. Open SOALab: Case study artifacts for SOA research and education. In *2012 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 59–60, 2012.

A. A Distributed Black Boxes Contract Calculus

A.1 The Core Distributed Black Boxes Calculus

A.1.1 Syntax

$$\begin{array}{llcl} \textbf{Processes} & P & = & \mathtt{B}^{D,U}_{in,out} \mid m \mid P \| P \\ \textbf{Messages} & m & = & x\langle\, \kappa, \widetilde{b}\,\rangle \mid \kappa\langle\, \widetilde{b}\,\rangle \end{array}$$

A.1.2 Evaluation Contexts

$$\textbf{Evaluation Contexts} \quad \mathcal{A} \quad = \quad [\bullet] \mid P \| [\bullet]$$

A.1.3 Reduction Semantics

$$\frac{x \in in \qquad out' = out \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \| x\langle\, \kappa, \widetilde{b}\,\rangle \xrightarrow{?x\langle\, \kappa, \widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in,out'}} \qquad \frac{\kappa \in D \qquad \kappa \notin U \qquad \kappa \notin in \qquad in' = in \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \xrightarrow{!x\langle\, \kappa, \widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in',out} \| x\langle\, \kappa, \widetilde{b}\,\rangle}$$

$$\frac{\kappa \in in \qquad \kappa \notin U \qquad in' = in \setminus \{\kappa\} \qquad U' = U \cup \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \| \kappa\langle\, \widetilde{b}\,\rangle \xrightarrow{?\kappa\langle\, \widetilde{b}\,\rangle} \mathtt{B}^{D,U'}_{in',out}} \qquad \frac{\kappa \in out \qquad out' = out \setminus \{\kappa\}}{\mathtt{B}^{D,U}_{in,out} \xrightarrow{!\kappa\langle\, \widetilde{b}\,\rangle} \mathtt{B}^{D,U}_{in,out} \| \kappa\langle\, \widetilde{b}\,\rangle}$$

A.2 The Distributed Black Boxes Contract Calculus

A.2.1 Syntax

Processes	P	$=$	$... \mid \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B)$
Black Box Pool	B	$=$	$\mathtt{B}^{D,U}_{in,out} \mid B \Vert x\langle \kappa, \widetilde{b} \rangle \mid B \Vert \kappa\langle \widetilde{b} \rangle$
Messages	m	$=$	$... \mid \underline{x}\langle \underline{\kappa}, \widetilde{b}, C, l \rangle \mid \underline{\kappa}\langle \widetilde{b}, C, l \rangle$
Ports	z	$=$	$y \mid \underline{y}$
Black Box Ports	y	$=$	$x \mid \kappa$
Adaptor Ports	$\underline{y}$	$=$	$\underline{x} \mid \underline{\kappa}$
Configurations	C	$:$	$z \mapsto z$
Specs	S	$:$	$id \mapsto (c, vp, pp, ip)$
Contracts	c	$=$	$\mathtt{given}\ \mathsf{e}, \widetilde{id}\ \mathtt{returns}\ \mathsf{e}, \widetilde{id}$
Value Plugins	vp	$:$	$\widetilde{b} \mapsto \widetilde{\mathsf{v}}$
Port Plugins	pp	$:$	$\widetilde{b} \mapsto (\widetilde{x}, \widetilde{\mathsf{v}_{\bot}})$
Parameter Plugins	ip	$:$	$(\widetilde{b} \mapsto \mathsf{v}) \vee (\widetilde{b} \mapsto \bot)$
Registry	R	$:$	$y \mapsto re$
Registry Entry	re	$=$	$(\diamond, \widetilde{pi}) \mid (id, \overline{bi}, \widetilde{pi})$
Blame Info	bi	$=$	$(\mathsf{v}_{\bot}, \bar{l}, l)$
Provenance Info	pi	$=$	$\mathsf{v}_{\bot} \rightsquigarrow l \mid \mathsf{v}_{\bot} \rightsquigarrow \widetilde{pi}$
Blame Log	L	$=$	$\widetilde{le}$
Blame Log Entry	le	$=$	$(\mathbf{pre} : x, id, \mathsf{v}_{\bot}, l) \mid (\mathbf{post} : \kappa, id, \mathsf{v}_{\bot}, \bar{l})$
		$\mid$	$(\mathbf{conflict} : x, id, \bar{l}, id, \bar{l}) \mid (\mathbf{unknown}\ \mathsf{v} : x, id, \bar{l})$

A.2.2 Evaluation Contexts

$$\textbf{Evaluation Contexts} \quad \mathcal{A} \quad = \quad ... \mid \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}([\bullet])$$

A.2.3 Reduction Semantics

$$\frac{(y \notin \mathit{in} \text{ and } y \notin \mathit{dom}(R)) \text{ or } (y \notin \mathit{in} \text{ and } R(y) = (\diamond, \widetilde{pi}))}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B \| y\langle\, \mathit{mc}\,\rangle) \longrightarrow \mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B) \| y\langle\, \mathit{mc}\,\rangle}$$

$$\frac{y \notin \mathit{in} \text{ and } y \in D}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B) \| y\langle\, \mathit{mc}\,\rangle \longrightarrow \mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, l, L\}(B \| y\langle\, \mathit{mc}\,\rangle)}$$

$$\frac{\begin{array}{c}
x \notin \mathit{in} \qquad C(x) = x \qquad \kappa \in D \qquad \kappa \notin \mathit{in} \qquad \kappa \notin U \qquad \mathit{in}' = \mathit{in} \cup \{\kappa\} \\
(\mathit{id}, \mathsf{v}_\perp, c, \bar{l}^s, l^c, \tilde{\mathsf{v}}, \tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = \mathit{message\_info}(x, \tilde{b}, S, R, \mathit{given\_ids}) \\
\mathit{pr} = \mathit{identified\_ports}(\tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, l) \\
L^{unk} = \mathit{unk\_blame}(x, \mathit{id}, \mathsf{v}_\perp, \{l^c\}, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{pre}, x, \mathit{id}, \mathsf{v}_\perp, \mathit{pre}(c), \tilde{\mathsf{v}}, l^c) \\
L^{conf} = \mathit{conf\_blame}(\mathit{pr}.\mathit{conf}, R, \mathit{in}, \{l^c\}) \\
R^u = \mathit{update\_registry}(\mathit{pr}, \{l^c\}, l^c, R, \mathit{in}) \qquad C^u = \mathit{update\_config}(\mathit{pr}.\mathit{new}, C) \\
C' = C^u \cup \{\kappa \mapsto \kappa\} \qquad R' = R^u \cup \{\kappa \mapsto (\mathit{id}, \perp, \bar{l}^s, l^c, \perp \rightsquigarrow \widetilde{pi})\} \qquad L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf}
\end{array}}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B \| x\langle\, \kappa, \tilde{b}\,\rangle) \longrightarrow \mathtt{A}^{D,U}_{\mathit{in}',\mathit{out}}\{C', S, R', L', l\}(B) \| x\langle\, \kappa, \tilde{b}\,\rangle}$$

$$\frac{\begin{array}{c}
x \in \mathit{in} \qquad C(x) = x \qquad \kappa \notin \mathit{out} \qquad \mathit{out}' = \mathit{out} \cup \{\kappa\} \\
(\mathit{id}, \mathsf{v}_\perp, c, \bar{l}^s, l^c, \tilde{\mathsf{v}}, \tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = \mathit{message\_info}(x, \tilde{b}, S, R, \mathit{given\_ids}) \\
\mathit{pr} = \mathit{identified\_ports}(\tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, \dagger) \\
L^{unk} = \mathit{unk\_blame}(x, \mathit{id}, \mathsf{v}_\perp, \{l^c\}, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{pre}, x, \mathit{id}, \mathsf{v}_\perp, \mathit{pre}(c), \tilde{\mathsf{v}}, l^c) \\
L^{conf} = \mathit{conf\_blame}(\mathit{pr}.\mathit{conf}, R, \mathit{in}, \{l^c\}) \\
R^u = \mathit{update\_registry}(\mathit{pr}, \{l^c\}, l, R, \mathit{in}) \qquad C^u = \mathit{update\_config}(\mathit{pr}.\mathit{new}, C) \\
C' = C^u \cup \{\kappa \mapsto \kappa\} \qquad R' = R^u \cup \{\kappa \mapsto (\mathit{id}, \perp, \bar{l}^s, l^c, \mathsf{v}_\perp \rightsquigarrow \widetilde{pi})\} \qquad L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf}
\end{array}}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B) \| x\langle\, \kappa, \tilde{b}\,\rangle \longrightarrow \mathtt{A}^{D,U}_{\mathit{in},\mathit{out}'}\{C', S, R', L', l\}(B \| x\langle\, \kappa, \tilde{b}\,\rangle)}$$

$$\frac{\begin{array}{c}
\kappa \in \mathit{out} \qquad C(\kappa) = \kappa \qquad \mathit{out}' = \mathit{out} \setminus \{\kappa\} \\
(\mathit{id}, \mathsf{v}_\perp, c, \bar{l}^s, l^c, \tilde{\mathsf{v}}, \tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = \mathit{message\_info}(\kappa, \tilde{b}, S, R, \mathit{returns\_ids}) \\
\mathit{pr} = \mathit{identified\_ports}(\tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, \widetilde{pi}) \\
L^{unk} = \mathit{unk\_blame}(\kappa, \mathit{id}, \mathsf{v}_\perp, \bar{l}^s, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{post}, \kappa, \mathit{id}, \mathsf{v}_\perp, \mathit{post}(c), \tilde{\mathsf{v}}, \bar{l}^s) \\
L^{conf} = \mathit{conf\_blame}(\mathit{pr}.\mathit{conf}, R, \mathit{in}, \bar{l}^s) \\
R' = \mathit{update\_registry}(\mathit{pr}, \bar{l}^s, l, R, \mathit{in}) \qquad C' = \mathit{update\_config}(\mathit{pr}.\mathit{new}, C) \\
L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf}
\end{array}}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B \| \kappa\langle \tilde{b}\,\rangle) \longrightarrow \mathtt{A}^{D,U}_{\mathit{in},\mathit{out}'}\{C', S, R', L', l\}(B) \| \kappa\langle \tilde{b}\,\rangle}$$

$$\frac{\begin{array}{c}
\kappa \in \mathit{in} \qquad C(\kappa) = \kappa \qquad \mathit{in}' = \mathit{in} \setminus \{\kappa\} \qquad U' = U \cup \{\kappa\} \\
(\mathit{id}, \mathsf{v}_\perp, c, \bar{l}^s, l^c, \tilde{\mathsf{v}}, \tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{pi}) = \mathit{message\_info}(\kappa, \tilde{b}, S, R, \mathit{returns\_ids}) \\
\mathit{pr} = \mathit{identified\_ports}(\tilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, \widetilde{pi}) \\
L^{unk} = \mathit{unk\_blame}(\kappa, \mathit{id}, \mathsf{v}_\perp, \bar{l}^s, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{post}, \kappa, \mathit{id}, \mathsf{v}_\perp, \mathit{post}(c), \tilde{\mathsf{v}}, \bar{l}^s) \\
L^{conf} = \mathit{conf\_blame}(\mathit{pr}.\mathit{conf}, R, \mathit{in}, \bar{l}^s) \\
R' = \mathit{update\_registry}(\mathit{pr}, \bar{l}^s, l, R, \mathit{in}) \qquad C' = \mathit{update\_config}(\mathit{pr}.\mathit{new}, C) \\
L' = L \cdot L^{pred} \cdot L^{conf}
\end{array}}{\mathtt{A}^{D,U}_{\mathit{in},\mathit{out}}\{C, S, R, L, l\}(B) \| \kappa\langle \tilde{b}\,\rangle \longrightarrow \mathtt{A}^{D,U'}_{\mathit{in}',\mathit{out}}\{C', S, R', L', l\}(B \| \kappa\langle \tilde{b}\,\rangle)}$$

$$\frac{\begin{array}{c} x \notin \mathit{in} \qquad C(x) = \underline{x} \qquad \kappa, \underline{\kappa} \in D \qquad \kappa, \underline{\kappa} \notin \mathit{in} \qquad \kappa, \underline{\kappa} \notin U \qquad \mathit{in}' = \mathit{in} \cup \{\kappa, \underline{\kappa}\} \\ (\mathit{id}, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{\mathit{pi}}) = \mathit{message\_info}(x, \widetilde{b}, S, R, \mathit{given\_ids}) \\ \mathit{pr} = \mathit{identified\_ports}(\widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, l) \\ L^{unk} = \mathit{unk\_blame}(x, \mathit{id}, \mathsf{v}_\perp, \{l^c\}, R) \qquad L^{conf} = \mathit{conf\_blame}(\mathit{pr.conf}, R, \mathit{in}, \{l^c\}) \\ R^u = \mathit{update\_registry}(\mathit{pr}, \{l^c\}, l^c, R, \mathit{in}) \qquad C^u = \mathit{update\_config}(\mathit{pr.new}, C) \\ C' = C^u \cup \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \qquad R' = R^u \cup \{\kappa \mapsto (\mathit{id}, \perp, \overline{l^s}, l^c, \perp \rightsquigarrow \widetilde{\mathit{pi}})\} \qquad L' = L \cdot L^{unk} \cdot L^{conf} \\ C^t = \mathit{transmitted\_config}(\mathit{pr.cons}, C') \cup \{\underline{\kappa} \mapsto \kappa, \kappa \mapsto \underline{\kappa}, \underline{x} \mapsto x\} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B \| x\langle \kappa, \widetilde{b} \rangle) \longrightarrow \mathtt{A}^{D,U}_{in',out}\{C', S, R', L', l\}(B) \| \underline{x}\langle \underline{\kappa}, \widetilde{b}, C^t, l \rangle}$$

$$\frac{\begin{array}{c} \underline{x} \in \mathit{in} \qquad C(\underline{x}) = x \qquad C^t(\underline{\kappa}) = \kappa \qquad \kappa, \underline{\kappa} \notin \mathit{out} \qquad \mathit{out}' = \mathit{out} \cup \{\kappa, \underline{\kappa}\} \\ (\mathit{id}, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{\mathit{pi}}) = \mathit{message\_info}(x, \widetilde{b}, S, R, \mathit{given\_ids}) \\ \mathit{pr} = \mathit{identified\_ports}(\widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, l^m) \\ L^{unk} = \mathit{unk\_blame}(x, \mathit{id}, \mathsf{v}_\perp, \{l^m\}, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{pre}, x, \mathit{id}, \mathsf{v}_\perp, \mathit{pre}(c), \widetilde{\mathsf{v}}, l^m) \\ L^{conf} = \mathit{conf\_blame}(\mathit{pr.conf}, R, \mathit{in}, \{l^m\}) \\ R^u = \mathit{update\_registry}(\mathit{pr}, \{l^m\}, l, R, \mathit{in}) \qquad C^u = \mathit{update\_config}(\mathit{pr.new}, C) \\ C' = C^u \oplus C^t \qquad R' = R^u \cup \{\kappa \mapsto (\mathit{id}, \perp, \overline{l^s}, l^m, \perp \rightsquigarrow \widetilde{\mathit{pi}})\} \qquad L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B) \| \underline{x}\langle \underline{\kappa}, \widetilde{b}, C^t, l^m \rangle \longrightarrow \mathtt{A}^{D,U}_{in,out'}\{C', S, R', L', l\}(B \| x\langle \kappa, \widetilde{b} \rangle)}$$

$$\frac{\begin{array}{c} C(\kappa) = \underline{\kappa} \qquad \kappa, \underline{\kappa} \in \mathit{out} \qquad \mathit{out}' = \mathit{out} \setminus \{\kappa, \underline{\kappa}\} \\ (\mathit{id}, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}) = \mathit{message\_info}(\kappa, \widetilde{b}, S, R, \mathit{returns\_ids}) \\ \mathit{pr} = \mathit{identified\_ports}(\widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, \widetilde{\mathit{pi}}) \\ L^{unk} = \mathit{unk\_blame}(\kappa, \mathit{id}, \mathsf{v}_\perp, \overline{l^s}, R) \qquad L^{conf} = \mathit{conf\_blame}(\mathit{pr.conf}, R, \mathit{in}, \overline{l^s}) \\ R' = \mathit{update\_registry}(\mathit{pr}, \overline{l^s}, l, R, \mathit{in}) \qquad C' = \mathit{update\_config}(\mathit{pr.new}, C) \qquad L' = L \cdot L^{unk} \cdot L^{conf} \\ C^t = \mathit{transmitted\_config}(\mathit{pr.cons}, C') \cup \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B \| \kappa\langle \widetilde{b} \rangle) \longrightarrow \mathtt{A}^{D,U}_{in,out'}\{C', S, R', L', l\}(B) \| \underline{\kappa}\langle \widetilde{b}, C^t, l \rangle}$$

$$\frac{\begin{array}{c} C(\underline{\kappa}) = \kappa \qquad \kappa, \underline{\kappa} \in \mathit{in} \qquad \mathit{in}' = \mathit{in} \setminus \{\kappa, \underline{\kappa}\} \qquad U' = U \cup \{\kappa, \underline{\kappa}\} \\ (\mathit{id}, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{\mathit{pi}}) = \mathit{message\_info}(\kappa, \widetilde{b}, S, R, \mathit{returns\_ids}) \\ \mathit{pr} = \mathit{identified\_ports}(\widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, R, \widetilde{\mathit{pi}}) \\ L^{unk} = \mathit{unk\_blame}(\kappa, \mathit{id}, \mathsf{v}_\perp, \overline{l^s}, R) \qquad L^{pred} = \mathit{pred\_blame}(\mathbf{post}, \kappa, \mathit{id}, \mathsf{v}_\perp, \mathit{post}(c), \widetilde{\mathsf{v}}, \overline{l^s}) \\ L^{conf} = \mathit{conf\_blame}(\mathit{pr.conf}, R, \mathit{in}, \overline{l^s}) \\ R' = \mathit{update\_registry}(\mathit{pr}, \overline{l^s}, l, R, \mathit{in}) \qquad C' = \mathit{update\_config}(\mathit{pr.new}, C) \oplus C^t \qquad L' = L \cdot L^{unk} \cdot L^{pred} \cdot L^{conf} \end{array}}{\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B) \| \underline{\kappa}\langle \widetilde{b}, C^t, l^m \rangle \longrightarrow \mathtt{A}^{D,U'}_{in',out}\{C', S, R', L', l\}(B \| \kappa\langle \widetilde{b} \rangle)}$$

A.2.4 Definitions of Metafunctions

The Get Message Info Metafunction

$\mathit{message\_info}(x, \widetilde{b}, S, R, \mathit{get\_ids}) \quad = \quad (\mathit{id}, \mathsf{v}_\perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{\mathit{pi}^{\mathsf{v}_\perp}})$

where $R(x) = (\mathit{id}, \overline{bi}, \widetilde{\mathit{pi}})$

and where $S(\mathit{id}) = (c, \mathit{vp}, \mathit{pp}, \mathit{ip})$, $\widetilde{\mathsf{v}} = \mathit{vp}(\widetilde{b})$, $(\widetilde{x}, \widetilde{\mathsf{v}^x_\perp}) = \mathit{pp}(\widetilde{b})$, $\widetilde{\mathit{id}^x} = \mathit{get\_ids}(c)$ and, $\mathsf{v}_\perp = \mathit{ip}(\widetilde{b})$

and where $\widetilde{\mathit{pi}^{\mathsf{v}_\perp}} = \mathit{filter\_provenence}(\widetilde{\mathit{pi}}, \mathsf{v}_\perp)$

and where $\mathsf{v}_\perp = \mathsf{v}$ and $(\mathsf{v}, \overline{l^s}, l^c) \in \overline{bi}$

or, $\mathsf{v}_\perp = \mathsf{v}$ and $(\mathsf{v}, \overline{l^{s'}}, l^c) \notin \overline{bi}$ and $(\perp, \overline{l^s}, l^c) \in \overline{bi}$

$\mathit{message\_info}(\kappa, \widetilde{b}, S, R, \mathit{get\_ids}) \quad = \quad (\mathit{id}, \perp, c, \overline{l^s}, l^c, \widetilde{\mathsf{v}}, \widetilde{x}, \widetilde{\mathit{id}^x}, \widetilde{\mathsf{v}^x_\perp}, \widetilde{\mathit{pi}^\perp})$

where $R(y) = (\mathit{id}, \overline{bi}, \widetilde{\mathit{pi}})$

and where $S(\mathit{id}) = (c, \mathit{vp}, \mathit{pp}, \mathit{ip})$, $\widetilde{\mathsf{v}} = \mathit{vp}(\widetilde{b})$, $(\widetilde{x}, \widetilde{\mathsf{v}^x_\perp}) = \mathit{pp}(\widetilde{b})$ and, $\widetilde{\mathit{id}^x} = \mathit{get\_ids}(c)$

and where $\widetilde{\mathit{pi}^\perp} = \mathit{filter\_provenence}(\widetilde{\mathit{pi}}, \perp)$

and where $(\perp, \overline{l^s}, l^c) \in \overline{bi}$

The Filter Provenance Metafunction

$$filter\_provenence(\emptyset, \mathsf{v}_\bot) = \emptyset$$

$$filter\_provenence(\{\mathsf{v}_\bot \rightsquigarrow l\} \cdot \widetilde{pi}, \mathsf{v}_\bot) = \{\mathsf{v}_\bot \rightsquigarrow l\} \cdot filter\_provenence(\widetilde{pi}, \mathsf{v}_\bot)$$

$$filter\_provenence(\{\mathsf{v}_\bot \rightsquigarrow \widetilde{pi'}\} \cdot \widetilde{pi}, \mathsf{v}_\bot) = \{\mathsf{v}_\bot \rightsquigarrow \widetilde{pi'}\} \cdot filter\_provenence(\widetilde{pi}, \mathsf{v}_\bot)$$

$$filter\_provenence(pi \cdot \widetilde{pi}, \mathsf{v}_\bot) = filter\_provenence(\widetilde{pi}, \mathsf{v}_\bot)$$
in any other case

The Identified Ports Metafunction

$$identified\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, R, ppi) = \{new = new; conf = conf; cons = cons\}$$
where $new = new\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$
$conf = conflicted\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$
$cons = consistent\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$

$$new\_ports(\emptyset, \emptyset, \emptyset, ppi, R) = \emptyset$$

$$new\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = new\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
if $x \in dom(R)$

$$new\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = \{(x, id, \mathsf{v}_\bot, \mathsf{v}_\bot \rightsquigarrow ppi)\} \cup new\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
in any other case

$$consistent\_ports(\emptyset, \emptyset, \emptyset, ppi, R) = \emptyset$$

$$consistent\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = \{(x, id, \mathsf{v}_\bot, \mathsf{v}_\bot \rightsquigarrow ppi)\} \cup consistent\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
if $R(x) = (id, \overline{bi}, \widetilde{pi})$

$$consistent\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = consistent\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
in any other case

$$conflicted\_ports(\emptyset, \emptyset, \emptyset, ppi, R) = \emptyset$$

$$conflicted\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = conflicted\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
if $x \notin dom(R)$ or $R(x) = (id, \overline{bi}, \widetilde{pi})$

$$conflicted\_ports(x\widetilde{x}, id\widetilde{id}, \mathsf{v}_\bot\widetilde{\mathsf{v}_\bot}, ppi, R) = \{(x, id, \mathsf{v}_\bot, \mathsf{v}_\bot \rightsquigarrow ppi)\} \cup conflicted\_ports(\widetilde{x}, \widetilde{id}, \widetilde{\mathsf{v}_\bot}, ppi, R)$$
in any other case

The Unknown Parameter Check and Blame Metafunction

$$unk\_blame(y, id, \mathsf{v}_\bot, \bar{l}, R) = \emptyset \quad \text{if } \mathsf{v}_\bot \in known\_params\_of(y, R)$$

$$unk\_blame(y, id, \mathsf{v}_\bot, \bar{l}, R) = \{(\mathbf{unknown}\ \mathsf{v}_\bot : y, id, \bar{l})\} \quad \text{in any other case}$$

A.2.5 The Known Parameters of Port According to Registry Metafunction

$$known\_params\_of(y, R) = \overline{\mathsf{v}_\bot}$$
if $R(y) = (id, \overline{bi}, \widetilde{pi})$
and where $\overline{bi} = \{(\mathsf{v}_\bot^1, \overline{l^{s^1}}, l^{c^1}), ..., (\mathsf{v}_\bot^n, \overline{l^{s^n}}, l^{c^n})\}$
and where $\overline{\mathsf{v}_\bot} = \{\mathsf{v}_\bot^1, .., .\mathsf{v}_\bot^n\}$

$$known\_params\_of(y, R) = \emptyset$$
in any other case

The Predicate Check and Blame Metafunction

$$pred\_blame(kind, y, id, \mathsf{v}_\bot, pred, \widetilde{\mathsf{v}}, b) = \emptyset \quad \text{if } (pred\ \widetilde{\mathsf{v}}) \Downarrow \mathsf{true}$$

$$pred\_blame(kind, y, id, \mathsf{v}_\bot, pred, \widetilde{\mathsf{v}}, b) = \{(kind : y, id, \mathsf{v}_\bot, b) \quad \text{if } (pred\ \widetilde{\mathsf{v}}) \Downarrow \mathsf{false}$$

The Conflict Blame Metafunction

$$conf\_blame(\emptyset, R, in, \bar{l}) \quad = \quad \emptyset$$

$$conf\_blame(\{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)\} \cup \overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)}, R, in, \bar{l}) \quad =$$
$$\{(\textbf{conflict} : x^{conf}, id^{old}, \overline{l^{old}}, id^{conf}, \bar{l})\} \cdot conf\_blame(\overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)}, R, in, \bar{l})$$
if $R(x^{conf}) = (id^{old}, \overline{bi}, \widetilde{pi'})$ and $x^{conf} \notin in$
and where $\overline{bi} = \{(\mathsf{v}_\perp^1, \overline{l^{old^1}}, l^{c^1}), ..., (\mathsf{v}_\perp^n, \overline{l^{old^n}}, l^{c^n})\}$ and $\overline{l^{old}} = \overline{l^{old^1}} \cup ... \cup \overline{l^{old^n}}$

$$conf\_blame(\{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)\} \cup \overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)}, R, in, \bar{l}) \quad =$$
$$conf\_blame(\overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi)}, R, in, \bar{l})$$
in any other case

A.2.6 The Blame Labels of Port According to Registry Metafunction

$$blame\_labels\_of(y, R) \quad = \quad \bar{l}$$
if $R(y) = (id, \overline{bi}, \widetilde{pi})$
and where $\overline{bi} = \{(\mathsf{v}_\perp^1, \overline{l^{s^1}}, l^{c^1}), ..., (\mathsf{v}_\perp^n, \overline{l^{s^n}}, l^{c^n})\}$
and where $\bar{l} = \{\overline{l^{s^1}}, .., \overline{l^{s^n}}\}$

$$blame\_labels\_of(y, R) \quad = \quad \emptyset$$
in any other case

The Update Registry Metafunction

$$update\_registry(pr, \overline{l^s}, l^c, R, in) \quad = \quad update\_registry\_aux(pr.cons, pr.conf, pr.new, \overline{l^s}, l^c, R, in)$$

$$update\_registry\_aux(\emptyset, \emptyset, \overline{(x^{new}, id^{new}, \mathsf{v}_\perp^{new}, \mathsf{v}_\perp \rightsquigarrow ppi^{new})}, \overline{l^s}, l^c, R, in) \;=\; R \cup \bar{r}$$
where $r = x^{new} \mapsto (id^{new}, \{(\perp, \overline{l^s}, l^c), (\mathsf{v}_\perp^{new}, \overline{l^s}, l^c)\}, \{\perp \rightsquigarrow ppi^{new}, \mathsf{v}_\perp \rightsquigarrow ppi^{new}\})$

$$update\_registry\_aux(\emptyset, \overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi^{conf})}, \overline{(x^{new}, id^{new}, \mathsf{v}_\perp^{new}, pi^{new})}, \overline{l^s}, l^c, R, in) \quad =$$
$$update\_registry\_aux(\emptyset, \emptyset, \overline{(x^{new}, id^{new}, \mathsf{v}_\perp^{new}, pi^{new})}, \overline{l^s}, l^c, R', in)$$
where for all $y \in dom(R)$ if $y \neq x^{conf}$ or $y \in in$ then $R'(y) = R(y)$
else if $R(y) = (id, \overline{bi}, \widetilde{pi})$ then $R'(y) = (\diamond, \widetilde{pi} \cdot pi^{conf})$
else if $R(y) = (\diamond, \widetilde{pi})$ then $R'(y) = (\diamond, \widetilde{pi})$

$$update\_registry\_aux(\overline{(x^{cons}, id^{cons}, \mathsf{v}_\perp^{cons}, pi^{cons})}, \overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi^{conf})}, \overline{(x^{new}, id^{new}, \mathsf{v}_\perp^{new}, pi^{new})}, \overline{l^s}, l^c, R, in) \;=$$
$$update\_registry\_aux(\emptyset, \overline{(x^{conf}, id^{conf}, \mathsf{v}_\perp^{conf}, pi^{conf})}, \overline{(x^{new}, id^{new}, \mathsf{v}_\perp^{new}, pi^{new})}, \overline{l^s}, l^c, R', in)$$
where for all $y \in dom(R)$ if $y \neq x^{cons}$ then $R'(y) = R(y)$ else $R'(y) = re$
and where if $R(y) = (id^{cons}, \{(\mathsf{v}_\perp^{cons}, \overline{l^{old}}, l^c)\} \cup \overline{bi}, \widetilde{pi})$
then $re = (id^{cons}, \{(\mathsf{v}_\perp^{cons}, \overline{l^s} \cup \overline{l^{old}}, l^c)\} \cup \overline{bi}, \widetilde{pi} \cdot pi^{cons})$
else if $R(y) = (id^{cons}, \{(\perp, \overline{l^{old}}, l^c)\} \cup \overline{bi}, \widetilde{pi})$
then $re = (id^{conf}, \{(\perp, \overline{l^{old}}, l^c), (\mathsf{v}_\perp^{cons}, \overline{l^s} \cup \overline{l^{old}}, l^c)\} \cup \overline{bi}, \widetilde{pi} \cdot pi^{cons})$

The Update Configuration Metafunction

$$update\_config(\overline{(x, id, \mathsf{v}_\perp, pi)}, C) \quad = \quad C \cup \overline{x \mapsto x}$$

The Transmitted Configuration Metafunction

$$transmitted\_config(\emptyset, C) \quad = \quad \emptyset$$
$$transmitted\_config(\{(x, id, \mathsf{v}_\perp, pi)\} \cup \overline{(x, id, \mathsf{v}_\perp, pi)}, C) \quad = \quad \{x \mapsto C(x)\} \cup transmitted\_config(\overline{(x, id, \mathsf{v}_\perp, pi)}, C)$$

The Configuration Merging Metafunction

$$C \oplus \emptyset \quad = \quad C$$
$$C \oplus \{z \mapsto z'\} \cup C' \quad = \quad C \oplus C' \qquad \text{if } C(z) \neq z$$
$$C \oplus \{z \mapsto z'\} \cup C' \quad = \quad ((C \setminus \{z \mapsto z\}) \cup \{z \mapsto z'\}) \oplus C' \qquad \text{if } C(z) = z$$

B. Well-formedness

B.0.7 Well-formed Global Configurations

$\boxed{U, in \Vdash \mathcal{C}}$

$$\frac{dom(\mathcal{C}) \cup rng(\mathcal{C}) = in \cup U}{U, in \Vdash \mathcal{C}}$$

B.0.8 Well-formed Global Specs

$\boxed{\Vdash \mathcal{S}}$

$$\frac{\forall(\texttt{given e}, \widetilde{id}_g \ \texttt{returns e}, \widetilde{id}_r, vp, pp, ip) \in rng(\mathcal{S}).\widetilde{id}_g \widetilde{id}_r \subseteq dom(\mathcal{S})}{\Vdash \mathcal{S}}$$

B.0.9 Well-formed Terms

$\boxed{D, U, in, out, \bar{l} \vdash^{\mathcal{C}}_{\mathcal{S}} P}$

$$\frac{}{\emptyset, \emptyset, \emptyset, \emptyset, \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} x\langle\, \kappa, \widetilde{b}\,\rangle} \qquad \frac{}{\emptyset, \emptyset, \emptyset, \emptyset, \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} \kappa\langle\, \widetilde{b}\,\rangle}$$

$$\frac{C \preceq \mathcal{C} \quad \{\underline{\kappa} \mapsto \kappa, \kappa \mapsto \underline{\kappa}, \underline{x} \mapsto x\} \subseteq C}{\emptyset, \emptyset, \emptyset, \{\kappa, \underline{\kappa}\}, \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} \underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, l\,\rangle} \qquad \frac{C \preceq \mathcal{C} \quad \{\underline{\kappa} \mapsto \kappa, \kappa \mapsto \underline{\kappa}\} \subseteq C}{\emptyset, \emptyset, \emptyset, \{\kappa, \underline{\kappa}\}, \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} \underline{\kappa}\langle\, \widetilde{b}, C, l\,\rangle}$$

$$\frac{\begin{array}{c} D = D^1 \uplus D^2 \quad U = U^1 \uplus U^2 \quad in = in^1 \uplus in^2 \quad out = out^1 \uplus out^2 \quad \bar{l} = \bar{l^1} \uplus \bar{l^2} \\ D^1, U^1, in^1, out^1, \bar{l^1} \vdash^{\mathcal{C}}_{\mathcal{S}} P^1 \qquad D^2, U^2, in^2, out^2, l^2 \vdash^{\mathcal{C}}_{\mathcal{S}} P^2 \end{array}}{D, U, in, out, \bar{l} \vdash^{\mathcal{C}}_{\mathcal{S}} P^1 \| P^2}$$

$$\frac{\begin{array}{c} in \subseteq D \quad U \subseteq D \quad in \cap U = \emptyset \quad out \cap D = \emptyset \\ in' \subseteq in \quad in' \cup U' \subseteq in \cup U \quad out \subseteq out' \end{array}}{D, U', in', out', \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} \mathtt{B}^{D,U}_{in,out}}$$

$$\frac{\begin{array}{c} in \subseteq D \quad U \subseteq D \quad in \cap U = \emptyset \quad out \cap D = \emptyset \quad out^A \subseteq out \quad l \neq \dagger \\ NewK(B, D) \cap UsedK(B, D) = \emptyset \quad NewK(B, D) \cap (U \cup in) = \emptyset \quad UsedK(B, D) \subseteq U \\ \mathcal{S} \vdash S \qquad \mathcal{C}, U, in \vdash C \qquad D, U, in \vdash^{l} R \\ D' = \{x \mid x \in D\} \quad U' = \{\kappa \mid \kappa \in U\} \quad in' = \{x \mid x \in in\} \quad out' = \{\kappa \mid \kappa \in out\} \\ D', U', in', out', \emptyset \vdash^{\mathcal{C}}_{\mathcal{S}} B \end{array}}{D, U, in, out, \{l\} \vdash^{\mathcal{C}}_{\mathcal{S}} \mathtt{A}^{D,U}_{in,out^A}\{C, S, R, L, l\}(B)}$$

B.0.10 The New Call-Back Ports Metafunction

$$\begin{array}{lcll} NewK(P^1 \| P^2, D) & = & NewK(P^1, D) \uplus NewK(P^2, D) & \\ NewK(x\langle\, \kappa, \widetilde{b}\,\rangle, D) & = & \{\kappa\} & \text{if } \kappa \in D \\ NewK(x\langle\, \kappa, \widetilde{b}\,\rangle, D) & = & \emptyset & \text{if } \kappa \notin D \\ NewK(\kappa\langle\, \widetilde{b}\,\rangle, D) & = & \emptyset & \\ NewK(P, D) & = & D & \text{in any other case} \end{array}$$

B.0.11 The Used Call-Back Ports Metafunction

$$\begin{array}{lcll} UsedK(P^1 \| P^2, D) & = & UsedK(P^1, D) \uplus UsedK(P^2, D) & \\ UsedK(x\langle\, \kappa, \widetilde{b}\,\rangle, D) & = & \emptyset & \\ UsedK(\kappa\langle\, \widetilde{b}\,\rangle, D) & = & \{\kappa\} & \text{if } \kappa \in D \\ UsedK(\kappa\langle\, \widetilde{b}\,\rangle, D) & = & \emptyset & \\ UsedK(P, D) & = & D & \text{in any other case} \end{array}$$

B.0.12 Well-formed Registries

$\boxed{D, in \vdash^{l} R}$

$$\frac{\begin{array}{c} dom(R) \cap D \subseteq in \cap U \\ \forall x \mapsto (id, \overline{bi}, \widetilde{pi}) \in R.\overline{bi} \neq \emptyset \text{ and } unique\_params(\overline{bi}) \\ \forall y \in in.y \in dom(R) \\ \forall x \in dom(R).x \in in \implies R(x) = (id, \{(\bot, \{l\} \cup \overline{l^{s}}, \dagger), (\mathsf{v}_{\bot}^{1}, \{l\} \cup \overline{l^{s^1}}, l^{c^1}), ..., (\mathsf{v}_{\bot}^{n}, \{l\} \cup \overline{l^{s^n}}, l^{c^n})\}, \widetilde{pi}) \\ \forall \kappa \in dom(R).\kappa \in in \cup U \implies R(\kappa) = (id, \{\bot, \overline{l^{s}}, l\}, \widetilde{pi}) \\ \forall x \in dom(R).x \notin D \implies R(x) = (id, \{(\bot, \overline{l^{s^{\bot}}}, l), (\mathsf{v}_{\bot}^{1}, \overline{l^{s^1}}, l), ..., (\mathsf{v}_{\bot}^{n}, \overline{l^{s^n}}, l)\}) \\ \forall \kappa \in dom(R).\kappa \notin D \implies R(\kappa) = (id, \{\bot, \{l\} \cup \overline{l^{s}}, l^{c}\}, \widetilde{pi}) \end{array}}{D, U, in \vdash^{l} R}$$

B.0.13 The Unique Parameters Metafunction

$$\begin{array}{lcl} unique\_params(\overline{bi}) & = & true \\ \multicolumn{3}{l}{\text{if } \forall (\mathsf{v}_{\bot}, \overline{l^{s}}, l^{c}) \in \overline{bi}.\overline{bi} = (\mathsf{v}_{\bot}, \overline{l^{s}}, l^{c}) \cup \overline{bi'} \text{ and } (\mathsf{v}_{\bot}, \overline{l^{s'}}, l^{c'}) \notin \overline{bi'}} \\ unique\_params(p) & = & false \\ \multicolumn{3}{l}{\text{in any other case}} \end{array}$$

B.0.14 Well-formed Local Specs

$\boxed{\mathcal{S} \vdash S}$

$$\frac{\begin{array}{c} S \subseteq \mathcal{S} \\ \forall (\texttt{given}\ \mathsf{e}, \widetilde{id}_{g}\ \texttt{returns}\ \mathsf{e}, \widetilde{id}_{r}, vp, pp, ip) \in rng(S).\widetilde{id}_{g}\widetilde{id}_{r} \subseteq dom(S) \end{array}}{\mathcal{S} \vdash S}$$

B.0.15 Well-formed Local Configurations

$\boxed{\mathcal{C}, U, in \vdash C}$

$$\frac{\begin{array}{c} C \preceq \mathcal{C} \\ \forall x \in in.\{x \mapsto x, \underline{x} \mapsto x\} \subseteq C \text{ where } \underline{x} \in in \\ \forall \underline{x} \in in.\{x \mapsto x, \underline{x} \mapsto x\} \subseteq C \text{ where } x \in in \\ \forall \kappa \in in.\{\kappa \mapsto \kappa\} \subseteq C \text{ or } \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \underline{\kappa} \in in \\ \forall \underline{\kappa} \in in.\{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \kappa \in in \\ \forall \kappa \in U.\{\kappa \mapsto \kappa\} \subseteq C \text{ or } \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \underline{\kappa} \in U \\ \forall \underline{\kappa} \in U.\{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \kappa \in U \\ \forall \kappa \in out.\{\kappa \mapsto \kappa\} \subseteq C \text{ or } \{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \underline{\kappa} \in out \\ \forall \underline{\kappa} \in out.\{\kappa \mapsto \underline{\kappa}, \underline{\kappa} \mapsto \kappa\} \subseteq C \text{ where } \kappa \in out \end{array}}{\mathcal{C}, U, in \vdash C}$$

B.0.16 Consistent Configurations

$\boxed{C \preceq C'}$

$$\frac{\begin{array}{c} \forall y \mapsto y \in C.y \mapsto y \in C' \text{ or } \{y \mapsto \underline{y}, \underline{y} \mapsto y\} \subseteq C' \\ \forall y \mapsto \underline{y} \in C.\{y \mapsto \underline{y}, \underline{y} \mapsto y\} \subseteq C' \\ \forall \underline{y} \mapsto y \in C.\{y \mapsto \underline{y}, \underline{y} \mapsto y\} \subseteq C' \end{array}}{C \preceq C'}$$

Definition 6 (Well-formed Terms)**.** *A term P is well-formed iff there exist $\mathcal{C}$, $\mathcal{S}$, D, U, in, out and $\bar{l}$ such that $U, in \Vdash \mathcal{C}, \Vdash \mathcal{S}$ and $D, U, in, out, \bar{l} \vdash^{\mathcal{C}}_{\mathcal{S}} P$.*

Theorem 7 (Preservation of Well-formedness)**.** *If $U, in \Vdash \mathcal{C}$, $\Vdash \mathcal{S}$ and $D, U, in, out, \bar{l} \vdash^{\mathcal{C}}_{\mathcal{S}} P$ and $P \xrightarrow{\alpha} P'$ where $\alpha \in \{?m, !m, \cdot\}$ then there exist $\mathcal{C}'$, U', in' and out' such that $U', in' \Vdash \mathcal{C}'$, $D, U', in', out', \bar{l} \vdash^{\mathcal{C}'}_{\mathcal{S}} P'$ and, $\mathcal{C} \preceq \mathcal{C}'$, $U \subseteq U'$, $in \cup U \subseteq in' \cup U'$ and $out \subseteq out'$.*

PROOF SKETCH. By straightforward case analysis on the form of the reduction rule $P \xrightarrow{\alpha} P'$. □

C. Monitor Erasure

$$\begin{array}{lrcl} \textbf{Contexts} & \mathcal{E} & = & [\bullet] \mid \mathcal{E} \| P \\ \textbf{Multihole Contexts} & \hat{\mathcal{E}} & = & [\bullet] \mid \hat{\mathcal{E}} \| \hat{\mathcal{E}} \mid \hat{\mathcal{E}} \| P \end{array}$$

Definition 8 (Black Box Preorder $\sqsubseteq_{\mathtt{B}}$)**.** *For P^1 and P^2, $P^1 \sqsubseteq_{\mathtt{B}} P^2$ iff for all contexts $\mathcal{E}$ such that $\mathcal{E}[P^1]$ and $\mathcal{E}[P^2]$ are well-formed, if $\mathcal{E}[P^1] \xrightarrow{\widetilde{\alpha}}_* \xrightarrow{\beta} Q^1$ then $\mathcal{E}[P^2] \xrightarrow{\widetilde{\gamma}}_* \xrightarrow{\beta} Q^2$ where $\alpha, \gamma \in \{?m, !m, \cdot\}$ and $\beta \in \{?m, !m\}$.*

Definition 9 (Black Box Equivalence $\cong_{\mathtt{B}}$)**.** *Terms P^1 and P^2 are black box equivalent, $P^1 \cong_{\mathtt{B}} P^2$, iff $P^1 \sqsubseteq_{\mathtt{B}} P^2$ and $P^2 \sqsubseteq_{\mathtt{B}} P^1$.*

Theorem 10. $\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}} \cong_{\mathtt{B}} \mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C,S,R,L,l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}})$

PROOF SKETCH. We perform the proof in two steps: we show that (1) $\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}} \sqsubseteq_{\mathtt{B}} \mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C,S,R,L,l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}})$ (Lemma 11) and, (2) $\mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C,S,R,L,l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}}) \sqsubseteq_{\mathtt{B}} \mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}}$ (Lemma 12). □

Lemma 11. $\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}} \sqsubseteq_{\mathtt{B}} \mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C,S,R,L,l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}})$

PROOF SKETCH. We set up the relation $\hat{\mathcal{R}}$ as the compatible closure over multihole contexts $\hat{\mathcal{E}}$ of the following relation $\mathcal{R}$:

- $B \;\mathcal{R}\; \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B)$
- $x\langle\, \kappa, \widetilde{b}\,\rangle \;\mathcal{R}\; \underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, \underline{l}\,\rangle$ if $C(\underline{x}) = x$ and $C(\underline{\kappa}) = \kappa$
- $\underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C, l\,\rangle \;\mathcal{R}\; \underline{x}\langle\, \underline{\kappa}, \widetilde{b}, C', l\,\rangle$ if $C \preceq C'$
- $\kappa\langle\, \widetilde{b}\,\rangle \;\mathcal{R}\; \underline{\kappa}\langle\, \widetilde{b}, C, \underline{l}\,\rangle$ if $C(\underline{\kappa}) = \kappa$
- $\underline{\kappa}\langle\, \widetilde{b}, C, l\,\rangle \;\mathcal{R}\; \underline{\kappa}\langle\, \widetilde{b}, C', l\,\rangle$ if $C \preceq C'$
  $\mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B) \;\mathcal{R}\; \mathtt{A}^{D,U'}_{in',out'}\{C',S,R,L',l\}(B)$ if $C \preceq C'$, $U \subseteq U'$, $in \subseteq in'$ and $out \subseteq out'$.

We show that: (1) if $P^1 \;\hat{\mathcal{R}}\; P^2$ and $P^1 \xrightarrow{\alpha} Q^1$ where $\alpha \in \{?m, !m\}$ then $P^2 \longrightarrow_* \xrightarrow{\alpha} \longrightarrow_* Q^2$ and $Q^1 \;\hat{\mathcal{R}}\; Q^2$; (2) if $P^1 \;\hat{\mathcal{R}}\; P^2$ and $P^1 \longrightarrow Q^1$ then $P^2 \longrightarrow_* Q^2$ and $Q^1 \;\hat{\mathcal{R}}\; Q^2$. We proceed with a straightforward case analysis on the steps $P^1 \xrightarrow{\alpha} Q^1$ and $P^1 \longrightarrow Q^1$. □

Lemma 12. $\mathtt{A}^{D^{\mathtt{A}},U^{\mathtt{A}}}_{in^{\mathtt{A}},out^{\mathtt{A}}}\{C,S,R,L,l\}(\mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}}) \sqsubseteq_{\mathtt{B}} \mathtt{B}^{D^{\mathtt{B}},U^{\mathtt{B}}}_{in^{\mathtt{B}},out^{\mathtt{B}}}$

PROOF SKETCH. The proof of this lemma is similar to the proof of Lemma 11. Instead, though, of the relation $\hat{\mathcal{R}}$, we employ its reverse $\hat{\mathcal{R}}^{-1}$. □

D. Contract Checking Correctness

$$\textbf{Registry Set} \quad \mathcal{R}(P) \quad = \quad \{(R \mid \mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B) \text{ occurs in } P\}$$

D.0.17 Registry Set Blame Consistent with Provenance

$\boxed{\rhd\mathcal{R}(P)}$

$$\frac{\forall R \in \mathcal{R}(P).\, \rhd R}{\rhd\mathcal{R}(P)}$$

D.0.18 Registry Blame Consistent with Provenance

$\boxed{\rhd R}$

$$\frac{\begin{array}{c}\forall(y, \mathsf{v}_\perp) \text{ such that } y \in dom(R) \text{ and } \mathsf{v}_\perp \in known\_params\_of(y, R).\\ blame\_labels\_of((y, \mathsf{v}_\perp), R) = origin\_of((y, \mathsf{v}_\perp), R)\end{array}}{\rhd R}$$

D.0.19 The Blame Labels of a Port–Parameter Pair According to Registry Metafunction

$$\begin{array}{lcl} blame\_labels\_of((y, \mathsf{v}_\perp), R) & = & \overline{l} \\ \quad \text{if } R(y) = (id, (\mathsf{v}_\perp, \overline{l}, l^c) \cup \overline{bi}, \widetilde{pi}) & & \\ blame\_labels\_of(y, R) & = & \emptyset \\ \quad \text{in any other case} & & \end{array}$$

D.0.20 The Origin of a Port–Parameter Pair According to Registry Metafunctions

$$\begin{array}{lcl} origin\_of((y, \mathsf{v}_\perp), R) & = & \\ \quad trans\_provenance(filter\_provenence(\widetilde{pi}, \mathsf{v}_\perp)) \cup trans\_provenance(filter\_provenence(\widetilde{pi}, \perp)) & & \\ \quad \text{if } R(y) = (\diamond, \widetilde{pi} \cdot pi) & & \\ origin\_of((y, \mathsf{v}_\perp), R) & = & \\ \quad trans\_provenance(filter\_provenence(\widetilde{pi}, \mathsf{v}_\perp)) \cup trans\_provenance(filter\_provenence(\widetilde{pi}, \perp)) & & \\ \quad \text{if } R(y) = (id, \overline{bi}, \widetilde{pi}) & & \end{array}$$

$$\begin{array}{lcl} origin\_for\_all(y, R) & = & trans\_provenance(\widetilde{pi}) \\ \quad \text{if } R(y) = (\diamond, \widetilde{pi} \cdot pi) & & \\ origin\_for\_all(y, R) & = & trans\_provenance(\widetilde{pi}) \\ \quad \text{if } R(y) = (id, \overline{bi}, \widetilde{pi}) & & \end{array}$$

$$\begin{array}{lcl} latest\_origin\_of(\mathsf{v}_\perp, R) & = & single\_trans\_provenance(pi) \\ \quad \text{if } R(y) = (\diamond, \widetilde{pi} \cdot pi) & & \\ latest\_origin\_of((y, \mathsf{v}_\perp), R) & = & single\_trans\_provenance(pi) \\ \quad \text{if } R(y) = (id, \overline{bi}, \widetilde{pi} \cdot pi) & & \end{array}$$

D.0.21 The Transitive Provenance Metafunctions

$$\begin{array}{lcl} single\_trans\_provenance(\mathsf{v}_\perp \leadsto l) & = & l \\ single\_trans\_provenance(\mathsf{v}_\perp \leadsto \widetilde{pi}) & = & trans\_provenance(\widetilde{pi}) \end{array}$$

$$trans\_provenance(pi^1 \cdot ... \cdot pi^n) \quad = \quad single\_trans\_provenance(pi^1) \cup ... \cup single\_trans\_provenance(pi^n)$$

Theorem 13 (Preservation of Registry Set Blame Consistency with Provenance)**.** *If a process P is well-formed, $\rhd\mathcal{R}(P)$ and, $P \longrightarrow P'$ then $\rhd\mathcal{R}(P')$.*

PROOF SKETCH. By straightforward case analysis on the form of the reduction rule $P \longrightarrow P'$. □

Theorem 14 (Correct Blame)**.** *Let $P = \mathcal{E}[\mathtt{A}^{D,U}_{in,out}\{C,S,R,L,l\}(B)]$. If P is well-formed, $\rhd\mathcal{R}(P)$ and $P \longrightarrow P'$ where $P' = \mathcal{E}'[\mathtt{A}^{D,U'}_{in',out'}\{C',S,R',L',l\}(B')]$ and $L' = L \cdot L^*$ then:*
1. if $(\textbf{pre} : x, id, \mathsf{v}_\perp, l^*)$ or $(\textbf{unknown}\ \mathsf{v} : x, id, \{l^*\}) \in L^*$ then

(a) if $P = \mathcal{E}'[x\langle\,\kappa, \widetilde{b}\,\rangle \| \mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B)]$ *then* $l^* = \dagger$*;*

(b) if $P = \mathcal{E}'[\underline{x}\langle\,\underline{\kappa}, \widetilde{b}, C, l^m\,\rangle \| \mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(B)]$ *then* $l^* = l^m$*;*

(c) if $P = \mathcal{E}[\mathtt{A}^{D,U}_{in,out}\{C, S, R, L, l\}(x\langle\,\kappa, \widetilde{b}\,\rangle \| B')]$ *then* $l^* = l$

2. if $(\mathbf{post} : y, id, \mathsf{v}_\perp, \overline{l^*}) \in L^*$ *then* $\overline{l^*} = \mathit{origin\_of}((y, \mathsf{v}_\perp), R)$*.*

3. if $(\mathbf{conflict} : x, id^\sharp, \overline{l^\sharp}, id^\flat, \overline{l^\flat}) \in L^*$ *then* $\overline{l^\sharp} = \mathit{origin\_for\_all}(y, R)$ *and* $\overline{l^\flat} = \mathit{latest\_origin\_of}(y, R')$

and L^* *does not contain any other kind of entries except those mentioned above.*

PROOF SKETCH. Parts (a) and (b) of the first conclusion of the theorem are direct consequences of the corresponding reduction rules. Parts (c) of the first conclusion of the theorem is a direct consequence of the well-formedness of P and Theorem 7. The second and third conclusions of the theorem are direct consequences of the corresponding reduction rules that produce these errors, the well-formedness of P, $\triangleright\mathcal{R}(P)$ and Theorem 13. $\square$

E. Full Specifications

E.1 Evernote

```

service UserStore {

    getNoteStoreUrl(authToken)
    @indexedby « authToken »
    @identifies ns:NoteStore[] by «
        yield (result, authToken)
    »

    getUser(authToken)
    @indexedby « authToken »
    @requires « len(authToken) > 0 »
}


service NoteStore {
    listNotebooks(authToken)
    @indexedby « authToken »
    @requires « len(authToken) < 0 »

    listLinkedNotebooks(authToken)
    @indexedby « authToken »
    @identifies noteStores:NoteStore[] by «
        for notebook in result:
            yield (notebook.noteStoreUrl, authToken + notebook.shareKey)
    »

    getSharedNotebookByAuth(authToken)

    authenticateToSharedNotebook(shareKey, authToken)
    @indexedby « shareKey + authToken »
    @identifies g:NoteStore[] by «
        yield (index=result.authenticationToken)
    »
    @requires «
        return len(shareKey) > 0
    »

    findNotes(authToken, filter, offset, maxNotes)
    @requires « offset >= 0 »
    @ensures «
        result.totalNotes <= maxNotes and result.totalNotes == len(result.notes)
    »
}
```

E.2 Twitter

```
service TwitterTweets {
    /1.1/friends/list.json(request)
    @indexedby « 'u' + request['args'].get('user_id', request['args'].get('screen_name')) »
    @requires « 'user_id' in request['args'] or 'screen_name' in request['args']»
    @requires « request['args'].get('count', 0) >= 0 »
    @requires « 'Authorization' in request['headers'] »
    @ensures « type(result['body']['users']) == list »
    @identifies t:TwitterTweets[] by «
        for f in result['body']['users']:
            yield (request['headers']['Host'], "u" + f['id_str'])
    »

    /1.1/statuses/user_timeline.json(request)
    @indexedby « 'u' + request['args'].get('user_id', request['args'].get('screen_name')) »
    @requires « 'Authorization' in request['headers'] »
    @requires « 'user_id' in request['args'] or 'screen_name' in request['args']»
    @identifies t:TwitterTweets[] by «
        for tweet in result['body']:
            yield (request['headers']['Host'], "t" + tweet['id_str'])
    »
    @ensures «
        'count' not in request['args'] or \
            len(result) <= max(200, request['args']['count']) »
    @ensures «
        for tweet in result['body']:
            assert rfc822.parsedate_tz(tweet['created_at']) != None
    »

    /1.1/statuses/retweet/<id>.json(request)
    @indexedby « "t" + request['args']['tweet_id'] »
    @ensures « 'errors' not in result['body'] »

}
```

E.3 Chess

```
service Chess {

    GetMyGames(username, password)
    @identifies g:Chess[] by «
        for game in result:
            nmoves = len(split('[0−9]+\. ', game['moves']))
            yield (index=str((game['id'], game['drawOffered'], nmoves)))
            if game['drawOffered']:
                yield (index=str((game['id'], False, nmoves)))
    »

    MakeGame(whitePlayer, blackPlayer)
    @identifies gas:Chess[] by «
        yield (index=str((result, False, 0)))
    »

    MakeAMove(username, password, gameId, resign, acceptDraw, movecount, myMove, offerDraw, claimDraw, myMessage)
    @indexedby « str((gameId, acceptDraw, movecount)) »
    @ensures « result != "NoDrawWasOffered" »
    @ensures « result != "InvalidGameID" »

}
```

F. Additional Case Study: Airline Reservations

The airline reservation open-source case study [20, 27] models an airline reservation system, and was developed to provide researchers and educators with a simple but complete service-oriented application. With respect to the challenges we discuss in Section 1, it demonstrates that (i) CONSUL can operate on top of yet another interface abstraction (SOAP with a WSDL interface); and (ii) CONSUL can express and enforce contracts between components implemented in different languages (Python and PHP for client and server respectively).

The text documentation of the case study informally describes the component interfaces, and the WSDL specification captures some of the interface's syntactic properties. However, WSDL cannot express many of the properties in the text documentation. We wrote CONSUL contracts for some of these additional properties.

Property 1. (Well-formed passenger information) The documentation states that passengers' first and last names cannot contain whitespace. This ensures consistent treatment of passenger names by all services and applies to all services that use passenger data.

With CONSUL, we are able to ensure that passenger information is well-formed by adding the following tags for the bookSeat operation to the contract of the airline server (We also add similar tags for other uses of passenger information.)

```
bookSeat(bookingRequestNumber, flightId, passenger)
@requires « ' ' not in passenger.firstName »
@requires « ' ' not in passenger.lastName »
```

We tested the effectiveness of this contract by removing the passenger data validity check from the reference implementation of the bookSeat operation. Similar defensive code that checks this property appears in multiple places in the case study's code base. Without CONSUL and the defensive code, a client is able to book a flight with invalid passenger information. With CONSUL and the above contract, the violation was reported and the client was blamed. Thus the CONSUL contract is as effective as defensive code injected in multiple places in the service's code.