Robust Distributed Programs by Design*

Mohammad Qudeisat†, Lukasz Ziarek♮, Patrick Eugster†
† Purdue University ♮ SUNY Buffalo
mqudeisa@cs.purdue.edu, lziarek@buffalo.edu, p@cs.purdue.edu

ABSTRACT

Failure handling and recovery represents a major hurdle in the design and development of distributed systems. Such systems perform coordinated interactions to recover from failures which include component crashes, communication failures, and local software errors. Session types have been proposed as a means of enforcing that components comply with interaction protocols. By verifying the interaction of distributed components against protocol specifications, many subtle interaction defects can be caught statically. However, session types have been mostly limited to enforce correct behavior of normal-flow interactions, or have focused on all-or-nothing recovery. In this paper, we propose protocol types, a novel variant of multi-party session types, for specifying normal and exceptional program flow in real-life fault-tolerant distributed protocols. Protocol types allow global reasoning about failures and recovery in a fine-grained manner. We demonstrate the benefits of protocol types through code quality and performance on distributed protocols such as Shibboleth or two phase commit.

1. INTRODUCTION

As underlined by the continuously growing interest in data-center-based cloud computing and the Internet of Things, distributed software systems are on the rise. Programming distributed systems, however, remains inherently difficult as corresponding programs are structured from multiple components that interact in complex manners, where communication and hosts may black out temporarily or permanently. Indeed, *partial failures* — the real possibility that certain components or interactions fail whilst the remaining ones must still fulfill certain invariants — are the most challenging design and implementation concern of distributed systems. A recent example of such a partial failure occurred in an Amazon data-center, eventually bringing several services like Instagram and Netflix to a halt [3]. Such partial failures significantly increase the burden on developers.

Following the pioneering work of Takeuchi, Honda, and Kubo [28], several approaches to automatically verifying that components abide to protocols by design — so-called *session types* [13, 29, 7, 16] — have been proposed to help develop correct distributed software. Session types focus on a core subset of distributed programs, namely the complex *interactions* between distributed components where many faults occur; by capturing such interactions, session types allow for individual components to be verified against global protocols in a way integrated with program compilation. However, despite distributed message passing systems being cited as motivating scenarios, both the theory and practice of session types have focused more on features typical of concurrent, centralized setups, or on high-level programming models such as Web Services. As a result, there are a number of limitations of session types that prevent their usage in real-world distributed software, including

(1) limited failure handling mechanisms and

(2) hardwired communication channels with semantics which do not match traditional network protocol stacks.

For example (1) session type incarnations typically support only un-named failures [9, 10], and all-or-nothing recovery for repeating an entire protocol [8] which does not allow for proper handling of many *partial* failure scenarios; (2) inherited from process algebras where session types emanated from, channels are first-class citizens to support session delegation and are shared between multiple receivers, which yields semantics that correspond to message queues, a very specific class of distributed messaging systems [5].

This paper thus introduces *protocol types* which can be viewed as a variant of session types with a novel set of features targeting core distributed protocols. Specifically, the contributions of this paper are:

- An expressive model of failure handling in protocols with features for fine-grained expressive handling of both "environment-induced" failures and application-level failures ("exceptions"). We define and illustrate our resulting syntax and semantics through examples.
- A compiler for automatically verifying that interaction upon partial failures in real-world protocols abides to protocol type specifications.
- A detailed empirical study that shows that a fault-tolerant program verified against protocols type retains asynchrony in that it imposes less unnecessary

*Financially supported by ERC Consolidator Award "Lightweight Verification of Software".

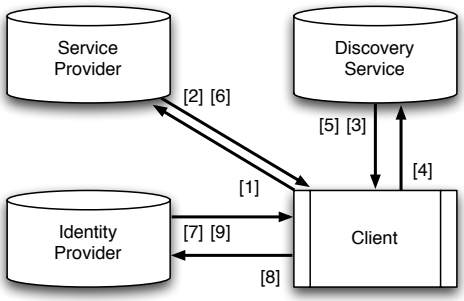


Figure 1: Shibboleth discovery service protocol

coordination among protocol participants than attempts to encode failures with basic session types. We further show by code quality analysis the benefits of our features for complexity of protocol types and code.

We motivate our protocol types in Section 2. Section 3 outlines their main features through examples. Section 4 discusses the design of these features and their semantics. Section 5 presents the implementation of protocol types. Section 6 evaluates them. Section 7 contrasts with related work. Section 8 presents final conclusions and future work.

Additional information including a formal characterication of type-checking in a core programming language for protocol types, further examples used in the evaluation, and a description of advanced features can be found on our website [1]. In the Appendix, we formalize the core of a programming language supporting our protocol types, and present type-checking of components' local protocol types against global protocol types, along with advanced features. A formally characterizes our protocol types. B and C present auxiliary definitions for our calculus. D discusses advanced features, and E presents additional session type examples used in the evaluation.

2. MOTIVATING EXAMPLE

To illustrate the benefits of protocol types consider the case of Shibboleth [2], a federated identity solution that provides single-sign-on capabilities for clients accessing various resources. Shibboleth can be viewed as a derivative of the Kerberos [22] protocol, and is among the world's most widely deployed federated identity solutions, connecting users to applications both within and between organizations. Shibboleth includes components acting in the roles of Identity Provider (IdP), Service Provider (SP), and Discovery Service (DS) in addition to clients. IdPs provide user information, SPs provide access to secure content based on that information provided by IdPs, and the DS provides a mechanism to seamlessly add or remove services and SPs.

```
protocol ShibbolethResourceRequest {
  participants: Client, SP
  //1. client requests resource
  Client -> SP: <HttpRequest>
  //2. response if client authorized
  SP -> Client: <HttpResponse>
}
```

Figure 2: Global session type of the Shibboleth resource request example

Table 1: Basic syntax for global session types

Basic session syntax	Purpose
A−>B: <T>	Send of message of type T from A to B
A :[...]∗	Loop with iterations controlled by A
A:{l1 :...,..., ln :...}	Branch with different labels l1-ln controlled by A

The discovery service protocol (DSP) for communicating between a client and the DS involves communication between all components of the Shibboleth system, following Figure 1. The DSP is composed of nine distinct steps. At the present it suffices to know that at one stage of the protocol, the client is authenticated with the single sign-on service and is redirected to an SP to access a resource (arrow [1]). Figure 2 shows this part of the protocol with basic multi-party session types (MPSTs) [4, 6, 11] which implicitly use one asynchronous communication channel *per participant pair*. Table 1 provides a syntax overview for such MPSTs. The client issues a request to the SP (A -> B: <T> denotes the sending of a value of type T from A to B) in the form of an HttpRequest and waits for a corresponding HttpResponse.

In this part of the protocol, two inherent failure scenarios that can occur are:

1. Although the client is authenticated, it might not be authorized to access the requested resource.
2. The client's authentication ticket may have expired, and thus the client must request a new ticket in order to gain access to resources at the SP.

There are different ways one can try to encode this with session types. The first scenario for instance can be encoded as a special value in the HttpResponse, however hiding the important distinction between success and failure. A clearer alternative, presented in Figure 3, is to have a branch guarded by the SP (SP: {...}) to make the three options explicit: (a) the regular scenario (NoFailure branch) leads to sending of the response; (b) the AuthorizationFailure aborts the attempt (see 1. above); (c) the ExpiredFailure represents the second above-mentioned failure scenario (see 2.). In this case the client communicates with the IdP to obtain a new ticket, and finally, indicates to the SP whether it wants to retry by controlling the loop (Client: [...]∗) around the protocol. This possibility of retrying also applies to (b), where the client might want to request a different resource instead.

Even this simple example illustrates several shortcomings of explicitly encoding failure handling with session types, an approach henceforth referred to as STX (*s*ession *t*ypes with e*x*plicit failure handling). First, adding failure scenarios adds substantial complexity and can lead to invalid paths. For instance, the possibility of retrying necessitates a loop around the entire session, which then also allows the failure-free path (NoFailure) to be repeated despite success. This is not in line with the intended protocol which ends upon success. Second, efficiency has to be considered when performing such an encoding. Even in the failure-free run — which can be assumed to be the common case — there is duplicated and redundant communication. Once the session is initiated, the first loop execution requires an additional message to be sent by the client to all others. In addition, there are now two messages — the branch label NoFailure (Line 8) as well as the actual response (Line 8) — which have to be

```
1  protocol RobustShibbolethResourceRequest {
2    participants: Client, SP, IdP
3    Client: [ // loop for retrying, if failed
4      // client requests resource
5      Client -> SP: <HttpRequest>
6      SP: {  // branch guarded by SP
7        // response if client authorized
8        NoFailure: SP -> Client: <HttpResponse>,
9        AuthorizationFailure: , // unauthorized
10       ExpiredFailure:       // ticket expired
11         // renew request
12         Client -> IdP: <HttpRequest>
13         // new ticket
14         IdP -> Client: <HttpResponse>
15     }
16   ]*
17 }
```

Figure 3: Global session type of the Shibboleth example with failure handling

transmitted between the SP and the client. If we inverse this last situation by having the SP send a HttpResponse which encodes the failure notifications as special values, as suggested at first, we can use the client as loop guard as follows to replace Lines 5-15 in Figure 3:

```
Client -> SP: <HttpRequest>
SP -> Client: <HttpResponse>
Client:                  // branch guarded by client
  NoFailure:             // no more messages needed
  AuthorizationFailure:
  ...
}
```

Besides losing clarity, the SP then inversely in the failure-free run waits for an additional branch label NoFailure that has to be sent by the client and received by the SP to proceed. Static or dynamic optimization might help in certain cases [20] but one can not take that for granted.

Challenges. Our protocol types provide expressive support for handling failures capturing both "environment-induced" failures (e.g., host or communication failures) as well as application-defined failures ("exceptions"). The main two challenges in the design of our support are:

Simplicity. We are interested in few features that capture a broad range of scenarios, and simplify global reasoning for the programmer in the presence of partial failures.

Efficiency. Our design should not over-constrain the system and hamper performance. In particular, we want to retain the potential for asynchronous execution rather than coupling components by coordination behind the scenes.

The biggest challenge comes from the seeming conflict between these two requirements. For instance, a facility which allows to reason in terms of atomic protocol blocks and performs automatic rollbacks upon failures in a transactional style would probably be easiest to use for a programmer, but such features are hard to implement efficiently at large scale and thus can hamper the asynchrony underlying many distributed systems.

In the following we describe our protocol types in more detail along with semantics balancing these two requirements.

3. PRIMER

We first present our support for handling failures in protocol types through examples, including Shibboleth as well as Two Phase Commit (2PC) [30]. Table 2 provides an overview of our protocol type syntax.

Table 2: Basic syntax for global protocol types

Protocol syntax	Purpose
A->B: <T> | F1 | ... | Fn	Send of message of type T from A to B with possible failures F1-Fn
A: [...]*	Loop, iterations controlled by A
A: {I1 :..., ..., In :...}	Branch with different labels I1-In controlled by A
try {...} handle(F1) {...} ... handle(Fn) {...}	Failure handling unit with handlers F1-Fn
A: retry	Retry of surrounding try block controlled by A

3.1 Shibboleth

Figure 4 shows the scenario corresponding to Figure 3 from Section 2 expressed with our proposed features. In short, message sends can now give rise to failures which are addressed via explicit failure handlers. Branches and loops do not have associated failures as these are captured by communication within them. We use a notation which is similar to the intuitive exception handling features in mainstream programming languages like Java and C++. try ... handle thus delimits a scope of failure handling, and several handle clauses for different types of failures can be paired up with a same try as usual.[1] (In contrast, previous models considered unnamed exceptions and thus single handlers, parameterized by shared channels over which exceptions are received [9, 10].) Here the SP can now immediately indicate a failure of either type AuthorizationFailure or ExpiredFailure — corresponding to the two failure scenarios of Section 2 respectively — as alternatives (|) to sending a response. As illustrated, the same message can yield different types of failures. In the case of a failure the execution moves forward to the corresponding handler (handle (...)). In contrast to the two STX encodings of these scenarios shown earlier, the intent is clear and there is no further need for communication between the client and the SP in the failure-free path. Similarly, the block terminates as it should and is not artificially constrained to be within a loop with the hidden invariant that the loop does not repeat upon success.

In many distributed protocols, retrying the protocol, or relevant sub-protocol, is a necessary recovery mechanism. In protocol types, retrying is thus supported by a corresponding keyword. The notation A: retry denotes that the next enclosing try {...} body may be repeated, and that the decision to do so is taken by A. Making repetition a choice rather than forcing it allows protocol types to avoid endless repetition. Assigning the duty of choosing the "retry path" to a particular participant is important for compile-time protocol verification and runtime enforcement, and is analogous to the assignment of branching and looping decisions to specific participants classically done in session types.

[1] We avoid the keyword **catch** to denote handlers to avoid confusion with the stronger (synchronous) semantics for exceptions in languages like C++ or Java.

```
protocol RobustShibbolethResourceRequest {
  participants: Client, SP, IdP
  try {
    // client requests resource
    Client -> SP: <HttpRequest>
    SP -> Client: <HttpResponse>
      | AuthorizationFailure | ExpiredFailure
  } handle(AuthorizationFailure) {
    Client: retry  // might req. diff. resource
  } handle(ExpiredFailure) {  // ticket expired
    Client -> IdP: <HttpRequest>  // renew req.
    IdP -> Client: <HttpResponse> // new ticket
    Client: retry
  }
}
```

Figure 4: Global protocol type of the Shibboleth resource request example with failures

Note that, though not shown, any failure message can also carry a value. For instance an ExpiredFailure could convey the time of expiration: ExpiredFailure <long>.

3.2 Two Phase Commit

Next consider the popular Two Phase Commit (2PC) protocol [30] used to decide on the outcome of distributed transactions executing across several servers. 2PC has several limitations, especially in an asynchronous failure-prone system assumed here, such as the dependence on a central coordinator [27, 15]. The goal here is not to debate these issues or possible solutions (e.g., coordinator replication). Figure 5 outlines a simplified version of such a protocol, encoded with our protocol types. For simplicity we focus on the case of two participants — which may fail — as that is enough to illustrate what we need, as also argued by Skeen and Stonebraker [27]. Malou et al. [11] deal in detail with sessions with parameterized (numbers of) participants.

Faithfully to the 2PC protocol, a TwoPC protocol type instance kicks off by having the coordinator send the identifier of a transaction (long) whose outcome (abort or commit) is to be voted upon by all participants. The coordinator waits for votes from each participant (true for commit, false for abort), based on which the coordinator sends the final decision to all participants. (Given the independence of the two sends from and to the coordinator respectively these can proceed in parallel.) Any other voting outcome than a unanimous commit (commit votes from all participants) must lead to aborting the transaction. If the coordinator times out on *any* of the responses then the protocol proceeds with the corresponding handle clause, leading to abort. In contrast to the previous examples, TimeoutFailure is raised by the "environment", which means that the runtime raises it. This is no different than a RemoteException in Java's remote method invocations which needs to be added to every remotely invocable method to convey errors like ConnectionExceptions, or SOAPExceptions in Web Services. With protocol types this is supported by declaring the corresponding failure as a subtype of a built-in InfrastructFailure .

In an asynchronous distributed system a TimeoutFailure does not necessarily imply a participant crash, and so we asynchronously notify both participants of the abort regardless of failures. The 2PC example points to the importance of choosing the semantics for failure handling. The coordinator always performs the same two sends of decisions to both

```
1  protocol TwoPC {
2    participants: Coord, Part1, Part2
3    Coord -> Part1: <long>
4    Coord -> Part2: <long>
5    try {
6      Part1 -> Coord: <bool>
7        | TimeoutFailure
8      Part2 -> Coord: <bool>
9        | TimeoutFailure
10     Coord -> Part1: <bool>
11     Coord -> Part2: <bool>
12   } handle(TimeoutFailure) { // abort to all
13     Coord -> Part1: <bool>
14     Coord -> Part2: <bool>
15   }
16 }
```

Figure 5: Global protocol type for the 2PC protocol with failure handling

participants, regardless of failures. Thus from the perspective of these participants there is no difference to replacing the entire try ... handle block on Lines 5-15 simply with

```
Part1 -> Coord: <bool>
Part2 -> Coord: <bool>
Coord -> Part1: <bool>
Coord -> Part2: <bool>
```

This would also avoid sending a failure message *and* an abort decision to participants in case of failure. The net difference, however, is that the coordinator can get stuck waiting for a vote from a participant which indeed failed. Based on the semantics expressed inherently with the example of Figure 5, a timeout on either participant constrains the coordinator to proceed to the handle clause. It also implies that any non-faulty participant knows to *not* expect two messages. In other terms they too proceed to the handler.

4. DESIGN

Next we discuss our design choices, with focus on synchronization and nesting. For brevity, we introduce small abstract examples.

4.1 Synchronization

Following the high-level presentation of failure semantics in the previous section, an obvious question is to ask under what conditions exactly participants occurring within a try {...} body skip forward to the corresponding handle clause, and when they do so. If a failure at any given point in such a try {...} body would immediately lead to aborting all subsequent communication in the body then that would trivially mean that all such communications would have to be *guarded* by the *absence* of failure. This would imply strong synchronization at runtime between recipients of failure-prone communications, senders, and receivers of any depending follow-up communications.

To not hamper asynchrony and thus efficiency, the basic semantics adopted for protocol types is to have failure notifications *follow the flow of communication*. That means that upon failure in a given try ... handle block *all communication causally depending on the failed communication in that block is immediately aborted*. Causal dependence follows the usual definition of causal ordering of events [19]: (a) a send from a participant causally precedes any subsequent send by that

participant, (b) a message send causally precedes its reception, and (c) if we have send or receive events e_1, e_2, e_3 such that e_1 precedes e_2 and e_2 precedes e_3, then e_1 precedes e_3.

From a participant P's perspective, this means that if P is on the receiving end of a causal chain of messages $(m_1, ..., m_n$ s.t. $\forall i \in [1..n] m_i = \mathsf{P}_i \text{->} \mathsf{P}_{i+1}$: ... , $\forall i < j \in [1..n]$ m_i occurs before m_j, and $\mathsf{P}_{n+1} = \mathsf{P})$ within a **try** ... **handle** block which starts at a transmission which can yield a failure (i.e., $m_1 = \mathsf{P}_1 \text{->} \mathsf{P}_2$: ... | F) then P is subject to being rolled forward to the corresponding **handle** (F) handler. Conceptually this does not add communication, as the failure notification or its absence, inherently, can be propagated along the usual communication path; in practice the failure notification can be sent directly to all targets though.

Other participants involved in a given **try** ... **handle** block whose communication is not in the causal extension of a given failure, on the other hand, can proceed with their remaining communication inside the **try** {...} body before being informed of the failure. We believe this is the right amount of synchrony and provides the programmer with most flexibility. Indeed, since the communication of such participants is not causally depending on failure-prone interactions, neither their success nor failure depends on those, so in any case they can proceed asynchronously until the point of synchronization at the end of the **try** {...} body. If no such synchronization is desired at all, then, given the independence of the said communication from the failure-prone parts, the programmer can also move them outside of the block; if they are failure-prone themselves they can be wrapped in their own **try** {...} block. For illustration consider the following abstract protocol fragment:

```
1 try {
2   A -> B: <T1> | F
3   C -> B: <T2>
4 } handle(F) {
5   ... D ...
6 }
```

Here the second send from C to B at Line 3 does not depend on the first send at Line 2. In the case of a failure F occurring at Line 3, the second send can thus proceed. (If one wanted to precondition that send upon the success of the first one, then a communication between B and C or possibly A and C would have to be inserted between Lines 2 and 3). As mentioned, since the second send at Line 3 does not causally depend on the first, the programmer can move it out of this **try** ... **handle** block, placing it either before or after. Inversely, if the programmer puts them together, he/she is expressing an "atomicity" constraint. Here this means that all participants may have to be eventually informed of an exception or its absence depending on any follow-up actions (e.g. in the failure handler).

Note that there is a difference between environment-induced and application-defined failures: the sender of a corresponding communication in the former case is not explicitly raising a failure notification and may not be aware of it — as is the case with TimeoutFailures in the 2PC example. As such, it is not considered for determining the set of causally depending participants and communications; only the receiver is considered, respecting the asynchronous nature of communication. There are two additions to these basic semantics which add flexibility. Any participant D which does not appear in a **try** {...} but appears in a given **handle**(F) clause, upon a F failure, will be informed like the other dependent participants in the body; D has to be informed in any case of the failure to participate in recovery actions. A second exemption is discussed in the context of advanced features D.

4.2 Nesting

Our model of protocol failures also supports nesting, in the sense that any **try** {...} block or **handle** ...{...} clause can contain another **try** ... **handle**. The rules for nested handling and propagation of failures/failure notifications is analogous to the rules for exception handling *within* method *bodies* in languages like Java or C++ (and not like the rules for propagation through nested method calls). That is, any failure which does not have a corresponding **handle** clause attached to its immediately surrounding **try** is propagated one scope outwards etc. If a given failure is not addressed by a corresponding handler within any enclosing scope, the protocol type is ill-formed and will be rejected by the compiler.

Nesting is orthogonal to our synchronization semantics, or put differently, composes straightforwardly with it. For deciding which participants to synchronize upon a given failure, all participants in the corresponding **try** {...} body are namely considered, which includes any participants in a nested **try** {...} block. Inversely, for any failure F not handled by any **handle** {...} clause of such a nested **try**, we consider the participants in the causal extension of the raising of F for determining the set of participants depending on the F after that nested **try**. Consider the following example:

```
1  try {
2    A -> B: <T1> | F1
3    try {
4      B -> C: <T2> | F2
5      B -> D: <T3> | F3
6    } handle(F2) {
7      ...
8    }
9    D -> A: <T4>
10   C -> A: <T5>
11 } handle(F1) {
12   ...
13 } handle(F3) {
14   ...
15 }
```

Upon a failure of type F1 at Line 2 the remainder of the protocol trivially gets aborted as all the other sends causally depend on it. Upon a failure F2 at Line 4, Line 5 is skipped, but Lines 9 and 10 are executed, as the failure F2 is handled "locally", which presumes that all invariants required for subsequent actions — as is common with exception handling — can be restored. Lastly, if a failure of type F3 is raised at Line 5, then the send at Line 9 will not take place as it causally derives from the failed send at Line 5. Thus, after the nested failure handler, we have "unfinished business" with F3. Inversely, the send at Line 10 can take place, since all its causal antecedents (Lines 2 and 4) succeeded.

4.3 Retrying

Consider the previous example. One possibility to establish the invariants required for continuing the protocol at Line 9 after a failure of type F2 occurred at Line 4 is to retry that nested (sub)protocol. As in the Shibboleth example presented in Section 3.1 (see Figure 4), this can be achieved by implementing the **handle**(F2) clause with B: **retry** such as to trigger a retry guarded for instance by B with the obvious meaning that the entire inner **try** {...} block is

re-attempted. The following thus replaces Lines 3-8 in the previous example:

```
try {
  B -> C: <T2> | F2
  B -> D: <T3> | F3
} handle(F2) {
  B: retry
}
```

It is important however to note that a retry can fail again and/or the participant guarding it can choose to not retry. The programmer is in that case, as always, still responsible for establishing any invariants necessary for any continuation of the protocol (e.g., for Lines 9 and 10 above). There is no implicit propagation of the handled failure in the case of a declined retry. Thus retrying is not a panacea, it's a feature that relieves the programmer from the burden of writing loops presented in the STX encoding of the Shibboleth example in Section 2 (see Figure 3), and ensuring that certain internal paths within such a loop reflecting success can *not* lead to retrying the loop.

The **retry** in a **handle**(F){...} body implicitly involves all participants appearing in the corresponding **try** {...} body in the recovery action. Thus in terms of synchronization, the first exemption to the basic semantics presented in Section 4.1 above applies here.

Introducing a dual primitive A: abort to the retry, which, if triggered, would skip any follow-up actions in a corresponding handler, on the other hand seems less useful, as it can be trivially implemented by a conditional branch guarding such follow-up actions. However, we discuss advantages of primitives for abort (and retry) sensitive to nesting as part of advanced features in D.

5. IMPLEMENTATION

Protocol types are implemented on top of the SessionJ [2] session types compiler and our previous work on the STing extension [1, 25, 26] to SessionJ. The SessionJ compiler supports bi-party interactions [18] and is written using the Polyglot [23] compiler framework. The compiler takes as input a session type specification as well as source files. SessionJ typechecks the input program against the session type specification and instruments the code with calls to the SessionJ runtime. The result is passed to a standard Java compiler.

To implement protocol types we first extended the SessionJ compiler to support MPSTs to allow for more interesting communication protocols to be constructed and then introduced our failure handling features. The input to our extended version of SessionJ, called ProtocolJ, is a non-standard Java file that contains a protocol type specification as well as a component implementation that utilizes the proposed language extensions. Just like SessionJ, ProtocolJ's output is a standard Java file that can be compiled using a standard Java compiler. The stages of the compilation process in ProtocolJ are described in Figure 6. Our compiler supports Java 4 collections but currently does not support generic types, since SessionJ uses a Polyglot version that does not support Java 5 generics. We are currently working on support for generic types can be achieved by upgrading the compiler to the newest release of Polyglot that has support for generic types.

In the parsing stage ProtocolJ processes the input file and

[2] http://code.google.com/p/sessionj/.

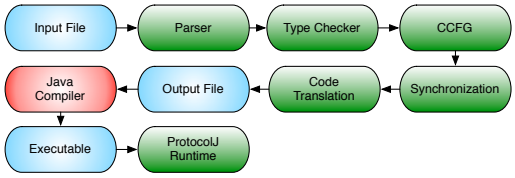


Figure 6: Work flow of ProtocolJ compilation process

protocol type specifications and constructs a global protocol structure and the AST for the input file. ProtocolJ synthesizes local protocol types for each participant from the global protocol type by projection (see C for formal specification).

The type-checking phase of the compiler verifies that the implementation of each participant conforms to its role as specified by the local protocol type. This includes checking that messages of the correct types are communicated with the correct participants at all protocol points. Moreover, the type checker uses type information to verify that failures specified in the protocol are handled properly, that is, all failures are handled at the points specified by the global protocol and their handlers implement a recovery protocol as specified by the global protocol type. The type checking algorithm is formalized and detailed in Appendix A.3.

To make sure that participants are correctly notified of failures and execute appropriate handlers, the compiler injects *synchronization messages*. These are determined by the synchronization inference and injection stage. The compiler first constructs a concurrent control flow graph (CCFG) of the global protocol type and determines the synchronization messages it needs to inject into the global protocol in order to notify participants of potential failures at the determined synchronization points. The compiler then injects message send and receive function calls to the runtime into the AST to automatically notify the participants that need to be notified of potential failures. Synchronization messages are inserted with one message send for each participant that needs to be notified of the failure (see [1]). After this, the compiler translates the AST into a standard Java file.

5.1 ProtocolJ Runtime

The compiler package includes runtime libraries that support message communication, synchronization and failure handling. At the beginning of the protocol execution the runtime automatically connects all distributed components to each other by deciding which components expect connection requests from which components in a way that avoids deadlocks. The runtime provides a set of API functions that enable the programs to exchange data messages (serializable Java objects), control messages (such as loop and branch decisions) as well as synchronization and failure messages.

The message communication portion of the runtime provides a set of functions to exchange data and control messages between participants. These include *send* and *receive*, which are used to exchange data messages, *inbranch* and *outbranch*, which are used to send and receive branch decision control messages, and *inwhile* and *outwhile* which are used to communicate the decision over a next loop iteration.

Whenever a failure occurs at runtime the automatically injected synchronization sends the necessary messages to the participants that need to be notified of the failure. These synchronization messages carry the type of the failure that took place. On the receiving end, whenever the runtime re-

ceives a failure message, it automatically raises an exception to move execution to the appropriate failure handler. Moreover, if execution proceeds normally without a failure at a synchronization point the runtime automatically sends a "No Failure" control message to any participants that would have been notified of a failure if one had occurred. If a participant receives a "No Failure" message, it simply continues its execution, representing the failure-free path of the protocol.

5.2 Static Synthesis of Synchronization

Concurrent control flow graphs. To automatically synthesize and insert synchronization protocols the compiler first must create a CCFG of the global protocol inferred from the protocol type. To that end the compiler first synthesizes a CFG for the global protocol type. The process of constructing the CFG is close to standard and is carried out as follows. First, the Start and End nodes are created, then a node for each message and control structure in the protocol is created. Two messages in the protocol are connected if they represent a message sequence. A message sequence is a series of message nodes with no interleaved control structures. Branches and loops are constructed in a way similar to standard CFGs with one distinction. A branch or a loop node is considered to be a message send from the branch or the loop guard to all other participants since the branch label and the loop decision are sent to every other participant in the protocol. For messages which can potentially generate failures, an edge is added from the message to the corresponding exception handler. Message sequences are collapsed into basic blocks. A node with no predecessor is connected to the Start node. Similarly, a node with no successor is connected to the End node. Once the CFG is generated, the compiler synthesizes a CCFG. Basic blocks in this CFG are split into concurrent component blocks, blocks which have disjoint communicating participants within this basic block. Figure 7 illustrates how a CFG and consequently a CCFG is generated for a simple example. The first step in the figure represents generating the CFG with all messages sequences collapsed into basic blocks. The second step in the figure represents generating a CCFG from a CFG by splitting basic blocks into concurrent blocks (represented by dashed lines).

Synchronization inference. The synchronization algorithm uses the CCFG in order to determine the synchronization message sends it needs to inject into the code. A synchronization message sends from the source of a failure to participant A is injected if one of the following three conditions is met: (1) A is involved in a message whose execution is directly or indirectly determined by whether the failure occurs, (2) A appears in the failure handler as a part of a communication message, and therefore must be notified to move to the handler upon failure occurrence, or (3) A appears anywhere in the **try** block and one of the block's handlers contains a **retry** operation, which means that all participants that appear in the **try** block must synchronize to the failure in order to be notified of a possible retry of the protocol block. Condition (1) is checked on a per basic block level. All participants within a basic block whose execution is dependent on whether a failure occurs is notified of the failure. Synchronization messages are inserted before each regular message within a basic block, to notify the participants engaged in that message. The last step in Figure 7 depicts the insertion of synchronization messages for each message that occurs after a message with a potential failure as well as all basic and control blocks in the CCFG that are control dependent on the block in which the failure occurs. Duplicate and unnecessary synchronizations are removed in a cleaning pass. Consider the synchronization messages insert in the last step in Figure 7, where X needs to notify Y, prior to Y's communication with X in the subsequent message. Notice that we do not need to notify X as it implicitly knows of the failure. This synchronization message, however, is spurious as Y is notified of the failure at the origin of the failure. This synchronization message can therefore be removed. Similarly, in the concurrent basic block, we inject synchronization messages prior to B's communication to C, for both B and C. A message notifying B of the failure is not necessary as we saw earlier and can be safely removed. Similarly, C needs to be notified only once of the failure and the second synchronization message can be removed. The resulting basic block with synchronization messages removed is given in the fourth step in Figure 7.

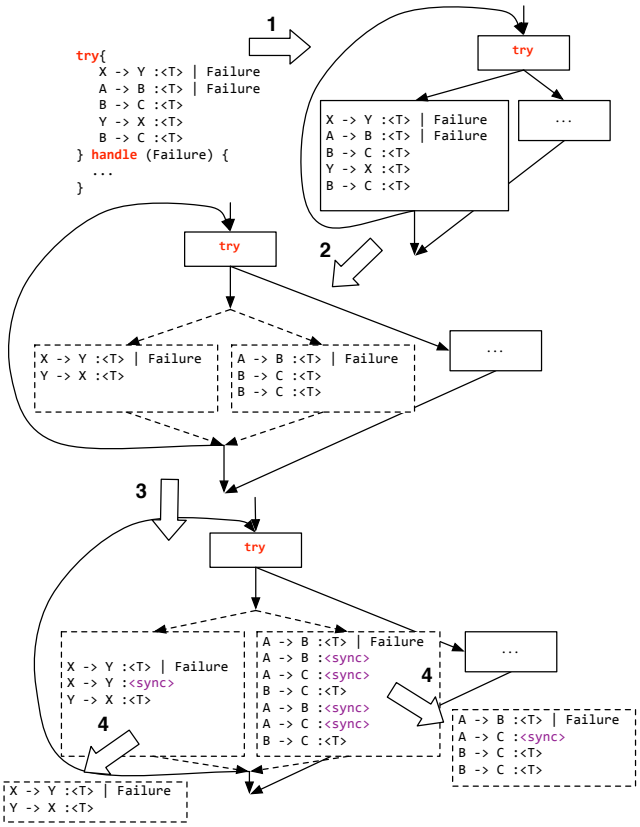


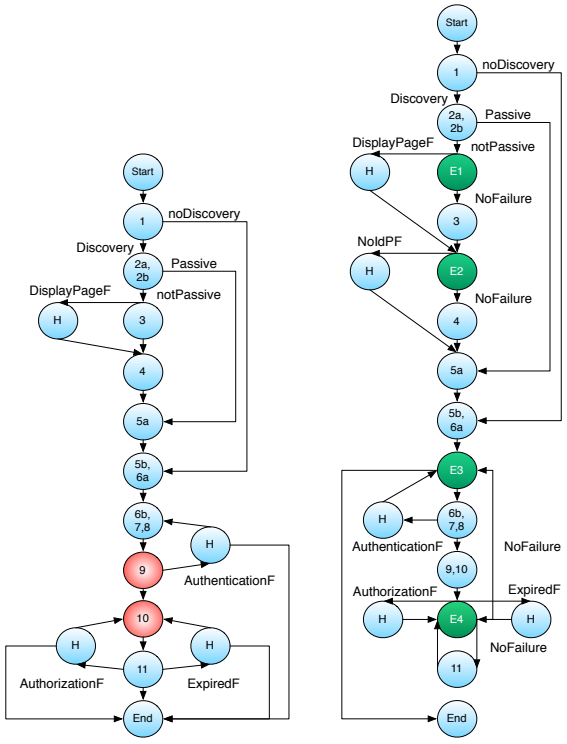
Figure 7: CCFG creation and synchronization inference

6. EVALUATION

We illustrate the benefits of our protocol types empirically both in terms of code quality and performance to gauge simplicity and efficiency (see Section 2).

6.1 Synopsis

We consider six benchmarks programs: Shibboleth and 2PC outlined earlier, 3PC [27], and 1PC [30], as well as Currency Broker and Buyer-Seller-Shipper examples inspired from previous work on session types with support for ex-



(a) With PT **(b)** With STX

Figure 8: State diagrams of Shibboleth discovery protocol

ception handling [9, 8]. Our extended report [1] provides brief descriptions of 3PC and the Currency Broker as well as the corresponding protocol types.

In the following we first assess the benefits of protocol types in terms of code quality and type (protocol description) complexity by comparing them to STX versions. Then, we also show that implementations based on protocol types provide better performance than STX implementations in that their execution is faster. For both code quality and performance comparisons we also include program versions obtained with *s*ession *t*ypes that are *a*gnostic to failures (STA), i.e., that do not deal with failures. This gives us a utopian baseline reflecting a world in which we need not worry about failures. The versions obtained with our protocol types are in the following referred to as PT for brevity.

6.2 Code Quality

To demonstrate the simplicity of devising fault-tolerant protocols with protocol types, we compare the quality of (protocol/session) types and their corresponding implementations to those of STX and STA. We gauge programmer effort by considering a number of statically determined code characteristics (1) lines of code (LoC) for type descriptions, (2) LoC for corresponding implementations, (3) nesting levels in types, (4) "maximum number" of messages (i.e., number of messages in the longest failure-free communication path), (5) number of distinct states (state here refers to a group of protocol operations that form a basic block in the protocol's CFG), (6) invalid paths between states in the protocol descriptions (i.e., paths permitted but not valid according to the protocol), and finally, (7) duplicated LoC due to explicit failure encoding (values for protocol types are always 0). The last two are moot in the case of STA.

Table 3: Code metrics

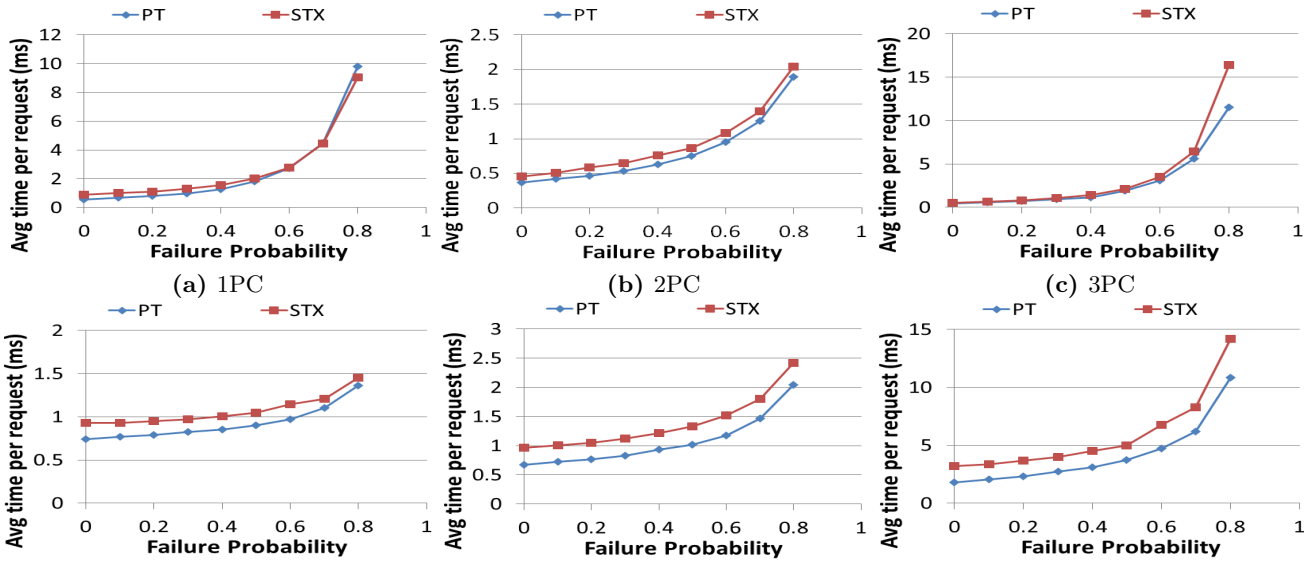
Protocol	Approach	Type lines	Code lines	Nesting levels	Msgs max.	States	Inv. paths	Dupl. lines
1PC	STA	8	19	0	4	1	-	-
	PT	14	36	1	7	4	0	0
	STX	16	59	4	12	6	1	8
2PC	STA	10	17	0	6	1	-	-
	PT	14	30	1	6	4	0	0
	STX	19	59	4	9	5	1	14
3PC	STA	14	21	0	10	1	-	-
	PT	45	139	3	12	16	0	0
	STX	49	184	9	18	26	1	23
Currency Broker	STA	13	34	1	7	3	-	-
	PT	24	67	3	11	8	0	0
	STX	22	68	4	14	8	0	0
Buyer Seller Shipper	STA	14	38	1	8	3	-	-
	PT	19	55	2	8	8	0	0
	STX	21	80	3	12	10	0	0
Shibboleth	STA	26	138	2	20	15	-	-
	PT	43	245	3	21	16	0	0
	STX	42	272	3	43	20	2	0

Table 3 summarizes the outcome of our static code quality evaluation. While the number of lines with PT (1) is close between PT and STX as specifying handlers also requires space, STX however leads to clearly increased numbers of LoC (2), and significantly higher nesting levels (3). The number of messages in the longest failure-free path is also clearly increased with STX (4); this is a hint to performance overhead which will be validated shortly. Another symptom is increased protocol complexity, which is demonstrated through an increasing number of different states (5). Figures 8a and 8b graphically illustrate this difference via protocol state transition diagrams for the discovery protocol of Shibboleth with PT and STX respectively. Notice that NoIdPFailure does not increase the number of states with PT as it is handled locally at the discovery service end. Moreover, we see that states 9 and 10 in Figure 8a are separate, but they appear in a single state in Figure 8b. The reason is that with protocol types the AuthenticationFailure is tied with the HttpRedirect message, which means that this message can potentially change the protocol execution path. In Figure 8b, however, the HttpRedirect message can be sent only after the execution path has been decided by the AuthenticationFailure branch path, and therefore message 9 is placed in the failure-free path along with message 10.

The most substantial increase in states with STX occurs in 3PC. This is largely due to disjoint paths through the protocol and corresponds directly to the large increase in nesting level and code duplication. Shibboleth in STX does not suffer from duplication but instead from invalid paths like 1PC, 2PC and 3PC (6). All *X*PC implementations exhibit significant code duplication (7) with STX. Figure 9 illustrates this for 2PC at the type level. The Currency Broker shows the least benefits for PT. We believe that this is largely due to its simplicity and focus on (few) application-level failures.

6.3 Performance Characteristics

As mentioned, simplicity for programmers can be easily achieved by proposing features which impose strong synchronization. We show that the less simple STX approach inversely adds much more overhead than PT. For our performance evaluation we ran successive rounds of the various



(a) 1PC (b) 2PC (c) 3PC

(d) Currency Broker (e) Buyer-Seller-Shipper (f) Shibboleth

Figure 10: Average times required to complete successful iterations of different protocols vs failure probability

```
1  protocol TwoPCExplicit {
2    participants: Coord, Part1, Part2
3    Coord -> Part1: <long>
4    Coord -> Part2: <long>
5    Part1: {
6      TimeoutFailure: //next 2 lines duplicated
7        Coord -> Part1: <Boolean>
8        Coord -> Part2: <Boolean>
9      NoFailure:
10       Part1 -> Coord: <Boolean>
11       Part2: {
12         TimeoutFailure: //next lines dupl. 7-8
13           Coord -> Part1: <Boolean>
14           Coord -> Part2: <Boolean>
15         NoFailure:
16           Part2 -> Coord: <Boolean>
17       }
18   }
19 }
```

Figure 9: 2PC with STX

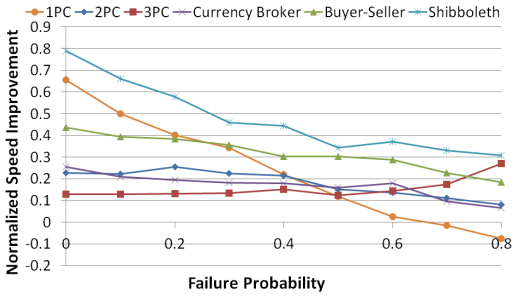


Figure 11: Normalized improvement of PT vs STX

protocols. All participants were executing on distinct machines as well as in distinct networks within campus with 1 - 3 hubs connecting each pair of networks. In these runs we varied the percent probability that *any* given communication that could result in a failure would actually raise a

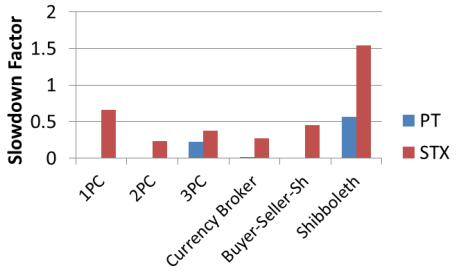


Figure 12: Overhead of PT and STX as compared to STA

failure notification. Thus as the percent probability of failures being raised increases, so does the number of times that portions of the protocol must be re-executed to achieve full completion of the protocol. The exponential trend of the graphs of Figure 10 is due to the fact that these protocols have more than one possible failure point. Thus as the failure probability increases for each of the failure points, the probability of a successful protocol execution exponentially decreases. All executions were repeated 10000 times and averaged. As the figure shows, in all reasonable ranges of failure probability PT clearly outperforms STX. For instance with Shibboleth, when the percent probability of a failure being raised at a communication point is 20% or below, PT takes only about 60-70% of the time of STX. These benefits are due to the absence of redundant messages which occur with manual encoding of failures. We note that this measurement approximates an upper bound on our performance gains as the protocols primarily perform communication.

Figure 11 normalizes the improvements of PT over STX. For instance with 2PC PT shaves off 9-22% of the time used by STX. For 2PC, Currency Broker, and Buyer-Seller-Shipper performance improvements of PT are consistent, saving between 9% and 42% over STX. In failure-free runs, PT only takes around 20% and 35% of the time STX takes for Shibboleth and 1PC respectively. These performance improvements come from a number of factors depending on

how protocols are implemented. For example, in the case of the STX implementation of 2PC, all protocol paths incur extra branch decision control messages that are used to notify the participants of whether or not a failure has occurred at each communication that can fail in addition to the actual sent message. In contrast PT implementations send either the message (in the absence of failure) or a failure message.

A second source of spurious messages that appears in the STX implementations is the way retries are implemented. Retries are implemented in STX using session loops. These add $2 \times (n-1)$ extra messages to the failure free path where n is the number of participants in the protocol. The first set of $n-1$ messages are the control messages sent by the loop guard to all participants to signal them to enter the first iteration of the loop, in order to proceed with the execution for the first time. Then, at the end of executing the subsession within the loop, an extra $n-1$ control messages are sent by the loop guard in order to signal whether to re-execute the loop body or not. This type of overhead occurs in STX for Shibboleth, Buyer-Seller-Shipper, 1PC, 2PC and Currency Broker examples. With PT, the participants proceed to execute the subprotocol within the **try** block without any exchange of messages. Moreover, **retry** control signals are sent only when a failure occurs and the handler offers the possibility of retrying the failure-prone subprotocol.

The third source of spurious messages is nesting. This is best demonstrated by the STX version of 3PC. Due to the deep nesting level, implemented using branch messages, the performance of the failure-free path is hampered by $n-1$ extra control messages for each nesting level: a branch guard has to send a control message to each of the $n-1$ participants to indicate whether or not a failure occurred.

The 1PC and 3PC protocols show interesting performance trends. Because of the simplicity of the 1PC protocol we see little performance improvement with PT over STX. This is because the number of spurious messages that are sent are very few. Moreover, as the failure probability increases the two implementations begin to converge and show similar performance results. For 3PC, on the other hand, PT shows consistent but humble improvements when the exception probability is below 60%. However, when the exception probability increases beyond the 60% point we see that PT performance improvements significantly increase. This is because the number of spurious messages exponentially increases as the exception probability increases.

Last but not least, Figure 12 focuses on failure-free runs, showing the overheads of PT and STX over STA. PT shows no overhead except for Shibboleth ($\sim$55%) and 3PC ($\sim$22%). In contrast, STX invariably incurs between $\sim$22% (2PC) and $\sim$152% (Shibboleth) overhead.

6.4 Threats to Validity and Discussion

We have shown that protocol types have advantages over manual encoding in basic session types both in terms of (1) *simplicity* and (2) *efficiency* on a range of different protocols. Although the examples are relatively small, they span a number of important protocol examples and families, and involve different failure models. We have encoded all of the examples in Java; however, we observe that most languages will contain primitives for loops and branches, while the protocol type syntax itself is language-independent, and *complexity* improvements for PT over STX can be generalized: with STX, a sequence of n failure-prone sends m_1, ..., m_n can translate to n nested branches, with the treatment of m_{i+1}, ..., m_n being duplicated at branch i at a degree proportionate to the number of different failures possibly occurring upon m_i, and every branch requires the sending of a label which is redundant with the subsequent message or failure. Furthermore, every loop introduced for retrying requires an additional multi-sending upon first execution. All of this translates to increased latency. Although the benchmarks show good percent improvement in the runtime of the protocols executed, we expect large, real-world systems to not always exhibit such performance improvements due to a lower ratio of communication to computation performed. However, we note that the benefits afforded by protocol types will be exhibited by reduced latency, a metric often of equal importance to raw throughput.

Last but not least, even if protocol types yield high LoC savings, we believe their main benefit lies in the reduction of invalid paths compared to implementing retrying via loops – these must be ruled out manually by the programmer which defies the purpose of the typing approach.

7. RELATED WORK

Session types. Honda et al. [16] and Bonelli et al. [7] extended the original bi-party session types to multi-party interaction. Honda et al. conduct a linearity analysis and prove progress of MPSTs. Linearity analysis in our system is superfluous since each participant has an implicitly defined, unshared, channel to every other participant. Bejleri and Yoshida's work [6] extends that of Honda et al. for synchronous communication among multiple interacting peers.

Operational semantics for asynchronous session types were first studied by Neubauer et al. [21]. Session types have been applied to functional [13] and object-oriented settings [12, 18], as well as others. Gay et al. [14] focus on modular implementation of sessions via objects. A Java implementation of binary session types was introduced by Hu, Yoshida, and Honda [18]; our implementation builds on it. Scribble [17] is an ongoing project on a session type based language and tool chain for large scale distributed applications.

Exception handling. Carbone et al. [9, 10] propose structured interactional exceptions for session types based asynchronous communication. When a process throws an exception, execution is interrupted at all participants involved in the conversation and they move to another dialogue. Exceptions can be nested through nested try blocks but raising an exception is not permitted to occur within an exception handler. The model supports only one kind of exception.

Hanazumi and Vieira de Melo [24] and Alexandar et al. [31] describe a method for exception handling based on Coordinated Atomic Actions (CAAs). Coordinated exception handling is achieved by satisfying a number of CAA properties on transactions, namely rollback and exception handling properties. When an exception is raised within a CAA or signaled to it, the participants handle the exception by executing the exception handling code for that CAA. If the exception is not handled within the CAA, it is propagated to other parts of the system. This method requires substantial programming effort. Also, transactional guarantees are not always needed or possible.

8. CONCLUSIONS AND OUTLOOK

To help the programmer combat partial failures in distributed systems we have proposed and presented protocol types. We are exploring several extensions to our work, e.g., assigning *different synchronization semantics* with different root failure and protocol types (subtyped by actual failure and protocol types to inherit corresponding semantics), notions of *nested protocols* to support different kinds of communication channels instead of the hardwired "−>".

9. REFERENCES

[1] Sting. `https://www.cs.purdue.edu/sting/`.

[2] *Shibboleth*. `http://www.shibboleth.net`.

[3] BBC News. Instagram, Vine and Netflix hit by Amazon Glitch. http://www.bbc.co.uk/news/technology-23839901, August 2013.

[4] L. Bettini, M. Coppo, L. D'Antoni, M. De Kuca, M. Dezani-Ciacaglini, and N. Yoshida. Global Progress in Dynamically Interleaved Multiparty Sessions. *CONCUR'08*, pages 418–433.

[5] B. Blakeley, H. Harris, and J.R.T. Lewis. Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development. *McGraw-Hill*, 1995.

[6] A. Bejleri and N. Yoshida. Synchronous Multiparty Session Types. *Electronic Notes in Theoretical Computer Science*, 241:pp. 3–33, 2009.

[7] E. Bonelli and A. Compagnoni. Multipoint Session Types for a Distributed Calculus. In *TGC*, pages 240–256, 2007.

[8] S. Capecchi, E. Giachino, and N. Yoshida. Global Escape in Multiparty Sessions. In *FSTTCS* , pages 338–351, 2010.

[9] M. Carbone, K. Honda, and N. Yoshida. Structured Interactional Exceptions in Session Types. In *CONCUR*, pages 402–417, 2008.

[10] M. Carbone, N. Yoshida, and K. Honda. Asynchronous Session Types: Exceptions and Multiparty Interactions. *Formal Methods for Web Services*, pages 187–212, 2009.

[11] P.-M. Deniélou, N. Yoshida, A. Bejleri, and R. Hu. Parameterised Multiparty Session Types. *Logical Methods in Computer Science*, 8(4), 2012.

[12] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys*, pages 177–190, 2006.

[13] S. J. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. Technical report, University of Glasgow, 2003.

[14] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular Session Types for Distributed Object-Oriented Programming. In *POPL*, pages 299–312, 2010.

[15] R. Guerraoui. Non-blocking Atomic Commit in Asynchronous Distributed Systems with Failure Detectors. *Distributed Computing*, 15(1):17–25, 2002.

[16] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL*, pages 273–284, 2008.

[17] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling Interactions with a Formal Foundation. In *ICDCIT* , pages 55–75, 2011.

[18] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP 2008*, pages 516–541, 2008.

[19] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] D. Mostrous and N. Yoshida. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In *TLCA*, pages 203–218, 2009.

[21] M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, pages 56–70, 2004.

[22] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, (9):33–38, September 1994.

[23] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. *CC* , pages 138–15, 2003.

[24] S. Hanazumi and A. C. Vieira de Melo Coordinating Exceptions of Java Systems: Implementation and Formal Verification. In *QATIC* , pages 338–351, 2012.

[25] K. C. Sivaramakrishnan, K. Nagaraj, L. Ziarek, and P. Eugster. Efficient Session Type Guided Distributed Interaction. In *COORDINATION*, pages 152–167, 2010.

[26] K.C. Sivaramakrishnan, L. Ziarek, K. Nagaraj, and P. Eugster. Efficient Sessions. *Science of Computer Programming*, 78(2):147–167, 2013.

[27] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Transactions on Software Engineering*, 9(3):219–228, May 1983.

[28] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE*, pages 398–413, 1994.

[29] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Software Components using Session Types. *Fundamenta Informaticae*, 73(4):pp. 583–598, 2006.

[30] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[31] J. Xu, A. B. Romanovsky, and B. Randell. Concurrent Exception Handling and Resolution in Distributed Object Systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(10):1019–1032, 2000.

APPENDIX

A. LANGUAGE

To clarify the details of our model of protocol failures, show how to use these features when implementing protocols, and to define how we have implemented protocol type verification we introduce a core calculus for multi-party protocol types with pair-wise channels and our primitives for failure handling. We provide typing rules for local protocols against global protocols after we introduce our language.

A.1 Syntax

The syntax and grammar of our core protocol type language is given in Figure 13. We use metavariables P to range over processes, p over process identifiers, e to range over expressions, v to range over values, l to range over labels, x to range over variables, m over messages, and h to range over choices – a pair consisting of a label and an expression. A program state includes a collection of processes and a protocol map ($\mathbf{S}$) which maps a given participant in a protocol (p) to a list of its peers ($\overline{p}$). Each process (P) is composed of a process identifier (p), and evaluation context ($E[e]$), and a message map (M). A message map is a mapping between process identifiers and a sequence of zero or more messages ($\overline{m}$). The message map represents a set of buffered, ordered message stream indexed by a given process. There are two types of messages: control messages, which represent coordinated control flow within a protocol, and value messages, raw sends of values between processes.

Our language contains primitives for sending values between participants in a protocol (**send**), receiving a value from a participant in a protocol (**recv**), performing a branch within a protocol (**outbranch**, **inbranch**), and looping within a protocol (**inwhile**, **outwhile**). Although it is not a part of the language the programmer writes programs in, we provide a utility function that allows sending a given message to a set of participants (**send**$(\overline{p}, m)$). Similarly, we provide a primitive that notifies a failure to a set of participants (**throw**$(\overline{p}, e)$). This set is determined by statically determining all targets for a failure notification from a protocol type. Figure 20 in B defines the relation $\rightsquigarrow$ for performing this identification.

Evaluation maps a program state to another. As we will detail shortly, evaluation is specified via two relations ($\Rightarrow$, $\Rightarrow_T$); the first represents the dynamic semantics of our MPSTs without support for failures ($\Rightarrow$), and the second ($\Rightarrow_T$) focuses on failures as an extension corresponding to boxed elements in Figure 13. All evaluation rules are applied up to commutativity of parallel composition ($\|$). We leverage evaluation contexts (E) to specify evaluation order.

A.2 Dynamic Semantics

There are eleven basic evaluation rules ($\Rightarrow$). The first two rules, [IFT] and [IFF] respectively, define conditionals and are standard. The first represents the choice of the **then** branch and the second the choice of the **else** branch. These two rules embody *local* control flow and do not directly affect other participants within a protocol. The rule [MULTI SEND] defines the behavior of our utility send function in terms of a sequence of sends. The rule [SEND] places the value being sent (v) in the map of the target process (p_2). Since our sends are both buffered and ordered we place the value (v) at the end of the sequences corresponding to p_1's stream in map M_2. Note that the rule for sending a value is thus global and atomic. The rule [RECV] simply removes the first message from the map (M) for the target process (p_2). The next two rules deal with starting and ending a protocol instance for a given participant.

The next four rules, [OB], [IB], [OW], [IW], define control flow for protocols and rely on control messages being passed between participants of the protocol. The rule [OB] and its corresponding receiver side rule [IB] represent an n-way choice. Each potential target for the choice is represented by a label (l). The rule [OB] specifies on the receiver side what branch target should be taken. A corresponding control message, comprised of the label (l), is sent to all other participants ($\overline{p_2}$) in the protocol. Similarly the two rules [OW], [IW] represent loops within a protocol. We define the while loop in terms of a standard rewrite using replication of the term and a branch. Notice that since this loop is *not* local additional send primitives are replicated in both branches. The **then** branch represents the decision to perform the loop and the **else** branch represents the decision to terminate the loop. We leverage the booleans **true** and **false** to represent the control messages for this decision respectively. The rule [IW] defines the looping construct for all other participants in the protocol. The looping predicate is not evaluated, but instead *received* as a control message from the specified participant (p_2). In much the same manner as [OW], the rule [IW] leverages replication and branches to define the loop.

The last two rules deal with starting and ending a session for a given participant. The rule [SESSION START] modifies the session map ($\mathbf{S}$) by placing a list of participants minus the current process identifier in the session map. This formulation allows us to quickly determine which participants need to receive a control message for branching and loop within a session. Conversely, the rule [SESSION END] clears the session map.

To this core calculus for protocols we add support for failures. Extensions and changes to the syntax and grammar are given in Figure 13 and denoted by boxes. We extend the definition of a process to include a failure stack σ and introduce a new failure notification value (**exc**(v)). Our formalism provides only one type of failure, but this failure can carry values. We observe that through value carrying failures we can encode different types of failures by casing on the value within a handle block. Failures are notified by evaluating a **throw** expression. Similarly, we introduce a new expression form (**try** { e } **handle**$(x : \tau)$ { e }), which defines a try block and a handle block. The variable x can be accessed in the expression within the handle block and is bound to the value carried by a failure that is handled by this handler.

Evaluation for failures is specified via a relation ($\Rightarrow_T$) that maps one program state to another. A program state is a collection of processes extended with failure stacks (σ) and a protocol map ($\mathbf{S}$). Notice **try** and **handle** are omitted as the evaluation rules define their evaluation explicitly.

Figure 15 defines the relation $\Rightarrow_T$ through five rules (two additional rules that supersede the rules [SEND] and [RECV] are given in C as the changes are uninteresting). The rule [PROC] defines the evaluation of a process (p_1) in terms of $\Rightarrow$. Notice that $\Rightarrow$ at most reduces two processes. The rule [TRY] installs a new failure handler on the stack of the process p. The failure handler contains the *continuation* of the

SYNTAX:

$$
\begin{array}{rcl}
S & ::= & \phi \mid \lceil p \rightarrow p' \rceil_\tau \mid \boxed{\lceil p \rightarrow p' \rceil_{\tau \mid \sharp:\tau'}} \mid S;S \mid T\{S\}C\{S\} \mid p\,\langle !\,\mu.S\rangle \mid p\,\langle !\,\overline{l;S}\rangle \\
s & ::= & \phi \mid !\,\lceil p \rceil_\tau \mid ?\,\lceil p \rceil_\tau \mid \boxed{!\,\lceil p \rceil_{\tau \mid \sharp:\tau'}} \mid s;s \mid T\{s\}C\{s\} \mid \boxed{\sharp\,\lceil \overline{p} \rceil_\tau} \\
 & \mid & !\,\mu.s \mid ?\,\lceil p \rceil\,\langle \mu.s\rangle \mid !\,l;s \mid ?\,\lceil p \rceil\,\langle \overline{l:s}\rangle \\
\\
P & ::= & \phi \mid P \| P \mid p\{e;\ M\} \mid \boxed{p\{e;\ M\}_\sigma} \\
v & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid \boxed{\mathsf{exc}(v)} \\
e & ::= & p \mid v \mid l \mid e;e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{begin}(e) \mid \mathsf{end} \mid \mathsf{send}(p,e) \mid \mathsf{recv}(p) \\
 & \mid & \mathsf{outwhile}(e,e) \mid \mathsf{inwhile}(p,e) \mid \mathsf{outbranch}(l,e) \mid \mathsf{inbranch}(p,\overline{h}) \\
 & \mid & \boxed{\mathsf{try}\ \{\ e\ \}\ \mathsf{handle}(x:\tau)\ \{\ e\ \}} \mid \boxed{\mathsf{throw}(p,\ e)} \\
h & ::= & l:e \\
m & ::= & l \mid v \mid \boxed{(v,i)}
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
P & \in & \mathit{Process} & l & \in & \mathit{Labels} \\
p & \in & \mathit{ProcessIdentifier} & m & \in & \mathit{Messages} \\
e & \in & \mathit{Expression} & x & \in & \mathit{Variables} \\
v & \in & \mathit{Values} & h & \in & \mathit{Choice} \\
\boxed{\mathsf{exc}(v)} & \in & \mathit{Failure} & s & \in & \mathit{LocalProtocolType} \\
\boxed{\sigma} & \in & \mathit{FailureStack} & S & \in & \mathit{GlobalProtocolType} \\
\tau & \in & \mathit{BaseType}
\end{array}
$$

$$
\begin{array}{rcl}
M & : & \mathit{Process} \rightarrow \mathit{MessageList} \\
\mathbf{S} & : & \mathit{ProcessIdentifier} \rightarrow \mathit{ProcessIdentifierList}
\end{array}
$$

EVALUATION CONTEXTS:

$$
E ::= \bullet \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid E;e \mid v;E \mid \mathsf{send}(p,E) \mid \boxed{\mathsf{throw}(p,E)}
$$

Figure 13: The syntax and grammar for our core language with message passing primitives.

try/handle expression (E), the handle expression e_2, and the variable for the handle (x) as well as the nesting level of the try block (i). The evaluation context which represents the continuation is removed from the expression. The rule [TRY COMP] will install this evaluation context and pop the failure handler from the stack. This rule defines the normal control flow through a try/handle expression (*i.e.*, no failure occurred). Notice that the rule [TRY COMP] creates a *synchronization* point at the end of the try blocks for all participants in the protocol instance. This guarantees that if a failure is notified by a participant all participants will receive it, either by rules [RECV EXC] or [ASYNC EXC]. The rule [THROW], abstractly, sends a failure notification to a set of participants ($\overline{p_2}$) determined from the protocol type (relation $\rightsquigarrow$ in Figure 20, B), and notifies the failure to the current process. Thus, the expression of the most closely enclosing handle is installed along with the continuation of the try/handle expression stored within the failure handler. The expression is rewritten so all occurrences of x are replaced by the value carried by the failure. The rule [RECV EXC] defines a non local failure raise via the installation of the continuation and handle expressions contained with handler of the local process that receives the failure. This rule is triggered only for the explicit receiver of the failure, all implicitly notified participants will receive the failure notification asynchronously via [ASYNC EXC]. Since failure notifications can be delivered asynchronously message maps must be cleaned appropriately for any unreceived messages at this nesting level (i) or higher. We define an auxiliary function Cl in C, which cleans M of all messages tagged with the current nesting level or higher.

A.3 Typing Rules

The grammar for our protocol types with failures is given in Figure 13. We use metavariables τ to range over base types, s to range over local protocol types, and S to range over global protocol types. A local protocol type can be a send of a message of type τ to a participant p ($!\ \lceil p \rceil_\tau$), a receive of a message of type τ from participant p ($?\ \lceil p \rceil_\tau$), a new "or" type which defines the send of a message of type τ or a failure $\sharp$ with a value of type τ' to participant p ($!\ \lceil p \rceil_{\tau \mid \sharp:\tau'}$), a sequence of local protocol types ($s;s$), a try/handle type ($T\{s\}C\{s\}$), or a failure type which specifies a target of participant p ($\sharp\ \lceil p \rceil_\tau$). Local protocol types can also be in out while type ($!\ \mu.s4$) or an in while type ($?\ \lceil p \rceil\ \langle \mu.s \rangle$). Similarly, they can also be an out branch ($!\ l;s$) or an in branch ($?\ \lceil p \rceil\ \langle \overline{l:s} \rangle$)

Global protocol types can be a communication type defining the flow of a message of type τ from participant p to p' ($\lceil p' \rightarrow p \rceil_\tau$), a communication type defining the flow of a message of type τ or a failure from participant p to p' ($\lceil p' \rightarrow p \rceil_{\tau \mid \sharp:\tau'}$), a sequence of global protocol types ($S;S$), a try/handle type ($T\{S\}C\{S\}$), a type defining a while loop ($p\,\langle !\ \mu.S \rangle$), or a type defining a conditional branch ($p\,\langle !\ \overline{l;S} \rangle$).

Figure 16 defines our local protocol typing rules and Figure 17 presents the unification of "or" types ($!\ \lceil p \rceil_{\tau \mid \sharp:\tau'}$). We introduce these since our model defines the flow of failure notifications along with messages. Conceptually "or" types are introduced by local branching control flow. Typing rule [TYPE IF] states that a branch expression has a local protocol type $s_1; s_4$ if the predicate expression has type s_1 under the type environment γ and the **then** branch and **else** branch

[IFT]

$$\frac{}{p\{E[\textsf{if true then } e_1 \textsf{ else } e_2];\ M\} \parallel P, \mathbf{S} \Rightarrow p\{E[e_1];\ M\} \parallel P, \mathbf{S}}$$

[IFF]

$$\frac{}{p\{E[\textsf{if false then } e_1 \textsf{ else } e_2];\ M\} \parallel P, \mathbf{S} \Rightarrow p\{E[e_2];\ M\} \parallel P, \mathbf{S}}$$

[MULTI SEND]

$$\frac{\overline{p} = p' :: \overline{p'}}{p\{E[\textsf{send}(\overline{p}, m)];\ M\} \parallel P, \mathbf{S} \Rightarrow p\{E[\textsf{send}(p', m); \textsf{send}(\overline{p'}, m)];\ M\} \parallel P, \mathbf{S}}$$

[SEND]

$$\frac{M_2' = M_2[p_1 \mapsto M_2(p_1) :: v]}{p_1\{E[\textsf{send}(p_2, v)];\ M_1\} \parallel p_2\{E[e];\ M_2\} \parallel P, \mathbf{S} \Rightarrow p_1\{E[\textsf{unit}];\ M_1\} \parallel p_2\{E[e];\ M_2'\} \parallel P, \mathbf{S}}$$

[RECV]

$$\frac{M(p_2) = v :: \overline{v} \quad M' = M[p_2 \mapsto \overline{v}]}{p_1\{E[\textsf{recv}(p_2)];\ M\} \parallel P, \mathbf{S} \Rightarrow p_1\{E[v];\ M'\} \parallel P, \mathbf{S}}$$

[OB]

$$\frac{\mathbf{S}(p_1) = \overline{p_2}}{p_1\{E[\textsf{outbranch}(l, e_1)];\ M\} \parallel P, \mathbf{S} \Rightarrow p_1\{E[\textsf{send}(\overline{p_2}, l); e_1];\ M\} \parallel P, \mathbf{S}}$$

[IB]

$$\frac{(l : e) \in \overline{h} \quad M(p_2) = l :: \overline{m} \quad M' = M[p_2 \mapsto \overline{m}]}{p_1\{E[\textsf{inbranch}(p_2, \overline{h})];\ M\} \parallel P, \mathbf{S} \Rightarrow p_1\{E[e];\ M'\} \parallel P, \mathbf{S}}$$

[OW]

$$\frac{\mathbf{S}(p_1) = \overline{p_2}}{\begin{array}{l} p_1\{E[\textsf{outwhile}(e_1, e_2)];\ M\} \parallel P, \mathbf{S} \Rightarrow \\ p_1\{E[\textsf{if } e_1 \textsf{ then send}(\overline{p_2}, \textsf{true}); e_2; \textsf{outwhile}(e_1, e_2) \textsf{ else send}(\overline{p_2}, \textsf{false})];\ M\} \parallel P, \mathbf{S} \end{array}}$$

[IW]

$$\frac{M(p_2) = v :: \overline{m} \quad M' = M[p_2 \mapsto \overline{m}]}{\begin{array}{l} p_1\{E[\textsf{inwhile}(p_2, e)];\ M\} \parallel P, \mathbf{S} \Rightarrow \\ p_1\{E[\textsf{if } v \textsf{ then } e; \textsf{inwhile}(p_2, e) \textsf{ else unit}];\ M'\} \parallel P, \mathbf{S} \end{array}}$$

[SESSION START]

$$\frac{}{p_1\{E[\textsf{begin}(\overline{p_2})];\ M\} \parallel P, \mathbf{S} \Rightarrow p_1\{E[\textsf{unit}];\ M\} \parallel P, \mathbf{S}[p_1 \mapsto \overline{p_2}/p_1]}$$

[SESSION END]

$$\frac{}{p\{E[\textsf{end}];\ M\} \parallel P, \mathbf{S} \Rightarrow p\{E[\textsf{unit}];\ M\} \parallel P, \mathbf{S}[p_1 \mapsto \phi]}$$

Figure 14: Operational semantics for a core protocol type calculus

[PROC]

$$\frac{p_1\{e_1;\ M_1\} \parallel p_2\{e_2;\ M_2\}, \mathbf{S} \Rightarrow p_1\{e_1';\ M_1'\} \parallel p_2\{e_2';\ M_2'\}, \mathbf{S}'}{p_1\{e_1;\ M_1\}_{\sigma_1} \parallel p_2\{e_2;\ M_2\}_{\sigma_2} \parallel P, \mathbf{S} \Rightarrow_T p_1\{e_1';\ M_1'\}_{\sigma_1} \parallel p_2\{e_2';\ M_2'\}_{\sigma_2} \parallel P, \mathbf{S}'}$$

[TRY]

$$\frac{||\sigma|| = i \quad \sigma' = (E[e_2], x, i+1) :: \sigma}{p\{E[\textsf{try}\ \{\ e_1\ \}\ \textsf{handle}(x:\tau)\ \{\ e_2\ \}];\ M\}_\sigma \parallel P, \mathbf{S} \Rightarrow_T p\{e_1;\ M\}_{\sigma'} \parallel P, \mathbf{S}}$$

[TRY COMP]

$$\frac{\begin{array}{c}P' = p_1\{E[\overline{v_1}];\ M_1\}_{\sigma_1'} \parallel \ldots \parallel p_n\{E[\overline{v_n}];\ M_n\}_{\sigma_n'} \\ \sigma_i = (E_i[e_i'], x, j) :: \sigma_i' \quad \mathbf{S}(p_1) = p_2 :: \ldots :: p_n\end{array}}{p_1\{\overline{v_1};\ M_1\}_{\sigma_1} \parallel \ldots \parallel p_n\{\overline{v_n};\ M_n\}_{\sigma_n} \parallel P, \mathbf{S} \Rightarrow_T P' \parallel P, \mathbf{S}}$$

[THROW]

$$\frac{\sigma_1 = (E'[e_2], x, i) :: \sigma_1' \quad \overline{p_2} \in \mathbf{S}(p_1)}{p_1\{E[\textsf{throw}(\overline{p_2}, \textsf{exc}(v))];\ M\}_{\sigma_1} \parallel P, \mathbf{S} \Rightarrow_T p_1\{\textsf{send}(\overline{p_2}, \textsf{exc}(v)); E'[e_2[v/x]];\ M\}_{\sigma_1'} \parallel P, \mathbf{S}}$$

[RECV EXC]

$$\frac{M(p_2) = (\textsf{exc}(v), i) :: \overline{(v, j)} \quad M' = M[p_2 \mapsto \overline{v}] \quad \sigma_1 = (E'[e_2], x, i) :: \sigma_1'}{p_1\{E[\textsf{recv}(p_2)];\ M\}_{\sigma_1} \parallel P, \mathbf{S} \Rightarrow_T p_1\{E'[e_2[v/x]];\ M'\}_{\sigma_1'} \parallel P, \mathbf{S}}$$

[ASYNC EXC]

$$\frac{M(p_2) = (\textsf{exc}(v), i) :: \overline{(v, j)} \quad M' = Cl(M[p_2 \mapsto \overline{v}], i) \quad \sigma = \sigma' :: (E'[e_2], x, i) :: \sigma''}{p_1\{E[e];\ M\}_\sigma \parallel P, \mathbf{S} \Rightarrow_T p_1\{E'[e_2[v/x]];\ M'\}_{\sigma''} \parallel P, \mathbf{S}}$$

Figure 15: Formal operational semantics for protocol failures.

[TYPE IF]

$$\frac{\gamma \vdash e_1 : s_1 \quad \gamma \vdash e_2 : s_2 \quad \gamma \vdash e_3 : s_3 \quad s_2 \diamond s_3 \vdash s_4}{\gamma \vdash \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3 : s_1; s_4}$$

[TYPE SEND]

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \textsf{send}(p, e) :! \lceil p \rceil_\tau}$$

[TYPE THROW]

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \textsf{throw}(\overline{p}, e) : \sharp\ \lceil \overline{p} \rceil_\tau}$$

[TYPE RECV]

$$\frac{}{\gamma \vdash \textsf{recv}(p) :? \lceil p \rceil_\tau}$$

[TYPE IW]

$$\frac{\gamma \vdash e : s}{\gamma \vdash \textsf{inwhile}(p, e) :? \lceil p \rceil\ \langle \mu.s \rangle}$$

[TYPE OW]

$$\frac{\gamma \vdash e_1 : \tau \quad \gamma \vdash e_2 : s_2}{\gamma \vdash \textsf{outwhile}(e_1, e_2) :! \mu.s_2}$$

[TYPE IB]

$$\frac{\overline{h} = (l_1 : e_1), ..., (l_n : e_n) \quad \gamma \vdash e_1 : s_1 \quad \ldots \quad \gamma \vdash e_n : s_n}{\gamma \vdash \textsf{inbranch}(p, \overline{h}) :? \lceil p \rceil\ \langle \{l_1 : s_1, ..., l_n : s_n\} \rangle}$$

[TYPE OB]

$$\frac{\gamma \vdash e : s}{\gamma \vdash \textsf{outbranch}(l, e) :! l; s}$$

[TYPE TRY HANDLE]

$$\frac{\gamma \vdash e_1 : s_1 \quad \gamma, x : \tau \vdash e_2 : s_2 \quad \forall\ (!\ \lceil p \rceil_{\tau' |\ \sharp\ : \tau''}) \in s_1 :\ \tau'' = \tau}{\gamma \vdash \textsf{try}\ \{\ e_1\ \}\ \textsf{handle}(x:\tau)\ \{\ e_2\ \} : T\{s_1\}C\{s_2\}}$$

[TYPE PROC]

$$\frac{\gamma \vdash e : s \quad s \equiv_p \Pi_p(S)}{p\{e;\ M\}_\sigma \sim S}$$

Figure 16: Local protocol typing rules

[OR EQUAL]

$$\frac{}{s \diamond s \vdash s}$$

[OR EXC SEND]

$$\frac{}{! \lceil p \rceil_{\tau} | \sharp : \tau' \diamond \sharp \lceil \overline{p} \rceil_{\tau'} \vdash ! \lceil p \rceil_{\tau} | \sharp : \tau'}$$

[OR EXC]

$$\frac{}{! \lceil p \rceil_{\tau} \diamond \sharp \lceil \overline{p} \rceil_{\tau'} \vdash ! \lceil p \rceil_{\tau} | \sharp : \tau'}$$

[OR SEQ]

$$\frac{s = s_1; s_2 \quad s' = s_1'; s_2' \quad s_1 \diamond s_1' \vdash s_1'' \quad s_2 \diamond s_2' \vdash s_2''}{s \diamond s' \vdash s_1''; s_2''}$$

[OR TRY HANDLE]

$$\frac{s_1 \diamond s_1' \vdash s_1'' \quad s_2 \diamond s_2' \vdash s_2''}{T\{s_1\}C\{s_2\} \diamond T\{s_1'\}C\{s_2'\} \vdash T\{s_1''\}C\{s_2''\}}$$

[OR OW]

$$\frac{s_1 \diamond s_2 \vdash s}{! \mu.s_1 \diamond ! \mu.s_2 \vdash ! \mu.s}$$

[OR IW]

$$\frac{s_1 \diamond s_2 \vdash s}{? \lceil p \rceil \langle \mu.s_1 \rangle \diamond ? \lceil p \rceil \langle \mu.s_2 \rangle \vdash ? \lceil p \rceil \langle \mu.s \rangle}$$

[OR OB]

$$\frac{s_1 \diamond s_2 \vdash s}{! l; s_1 \diamond ! l; s_2 \vdash ! l; s}$$

[OR IB]

$$\frac{s_1' \diamond s_1'' \vdash s_1 \quad \ldots \quad s_n' \diamond s_n'' \vdash s_n}{\{l_1 : s_1', \ldots, l_n : s_n'\} \diamond \{l_1 : s_1'', \ldots, l_n : s_n''\} \vdash ? \lceil p \rceil \langle \{l_1 : s_1, \ldots, l_n : s_n\} \rangle}$$

Figure 17: Unification of "or" types

can be unified to local protocol type s_4. We define the unification ($\diamond$) of two types through the rules [OR EQUAL], [OR EXC], [OR SEQ], and [OR TRY HANDLE]. If two types are equal then the unification of the two types is just the type itself [OR EQUAL]. A local protocol send type and a local protocol failure type can be unified into an "or" type if their targets (p) are the same (rule [OR EXC]). A sequence of local protocol types can be unified with another sequence of local protocol types into a new sequence if each of the constituent types in both sequences can be unified in rule [OR SEQ]. A local protocol try/handle type can be unified with another local protocol try/handle type if the types of the **try** and the types of the **handle** can be unified as shown in [OR TRY HANDLE]. The rules [OR OW], [OR IW], [OR OB], and [OR IB] state that loop and branch types can be unified if the sequences (s) associated with the labels for the branches or the body of the loop can be unified.

The type of a send expression in rule [TYPE SEND] is derived from the result type of the expression in the typing environment γ and the target of the send (p). Similarly the type of a throw expression in rule [TYPE THROW] is the failure type ($\sharp$), the targets of the **throw** ($\overline{p}$), and the type of the value stored in the failure (τ). The type of a receive expression in rule [TYPE RECV] is derived from the type of the message to be received and the process from which the message originates (p). The type of a try/handle expression in rule [TYPE TRY HANDLE] is a try/handle type ($T\{s_1\}C\{s_2\}$) where the type of the **try** expression is s_1 and the type of the **handle** expression is s_2. We state that a process adheres to a global protocol type ($p\{e;\ M\} \sim S$) if the process expression is type to s and s is equivalent to the projection of p from the global protocol type S.

Figure 18 defines rules for type checking a local protocol type against the projection of the global protocol type for that participant. Rule ([EQUIV SEQ]) defines equivalence over a type sequence. A local protocol type sequence is equivalent to a global projected protocol type sequence if each of their constituent elements are equivalent. A local protocol send type is equivalent to a global projected protocol communication type (rule [EQUIV SEND]) if the sender and receiver processes are equal. A local protocol send or failure type is equivalent to a global projected protocol communication or failure type (rule [EQUIV SEND EXC]) if the sender and receiver processes are equal. A local protocol receive type is equivalent to a global projected protocol communication type (rule [EQUIV RECV]) or to a global projected protocol communication or failure type (rule [EQUIV RECV EXC]) if the sender and receiver processes are equal. The rest of the rules are standard.

A.4 Meta-Theory

We state that a program is well-formed if it does not contain infinite loops and its expressions type-check. We define safety of a well-formed program P in terms of a global protocol type S. If the arity of the program and protocol type match and each process within the program type-checks against the global protocol type (i.e., the process is a well-typed participant in the global protocol S) then the program will complete (i.e., every process evaluates down to a value).

THEOREM 1 (SAFETY). *Given P s.t. P is well-formed and global protocol type S s.t. $||P|| = ||S||$, if $\forall\, p\{e;\ M\}_\sigma \in P$, $p\{e;\ M\} \sim S$, let $\boldsymbol{S} = \{p \mid p\{e;\ M\}_\sigma \in P\}$, then $P, \boldsymbol{S} \Rightarrow_T \overline{p\{v;\ M\}_\sigma}, \boldsymbol{S}$.*

B. TRACKING FAILURE DEPENDENCIES THROUGH TYPES

When a failure occurs, it is important to know whom to notify of the occurrence of the failure itself. Any computation that would have normally been executed if the failure had not occurred, must be examined. Specifically, if any communication action occurs in that computation, both the sender and received must be notified of the failure. Notice that the type structure of protocol types is sufficient to determine the communication dependencies on a given failure. We define two relations ($\succ$) and ($\rightsquigarrow$) to compute such dependencies. These are defined in Figures 19 and 20 respectively.

The relation ($\succ$) takes a global protocol type (S) and produces a sequence of processes ($\overline{p}$) that communicate within (S). A communication type, [DEP COMM], contains both a

[EQUIV SEQ]

$$\frac{s = s_1; s_2 \quad S = S_1; S_2 \quad s_1 \equiv_p S_1 \quad s_2 \equiv_p S_2}{s \equiv_p S}$$

[EQUIV TRY HANDLE]

$$\frac{s_1 \equiv_p S_1 \quad s_2 \equiv_p S_2}{T\{s_1\}C\{s_2\} \equiv_p T\{S_1\}C\{S_2\}}$$

[EQUIV SEND]

$$\frac{}{! \lceil p \rceil_\tau \equiv_{p'} \lceil p' \ \rightarrow \ p \rceil_\tau}$$

[EQUIV SEND EXC]

$$\frac{}{! \lceil p \rceil_{\tau \mid \sharp : \tau'} \equiv_{p'} \lceil p' \ \rightarrow \ p \rceil_{\tau \mid \sharp : \tau'}}$$

[EQUIV RECV]

$$\frac{}{? \lceil p \rceil_\tau \equiv_{p'} \lceil p \ \rightarrow \ p' \rceil_\tau}$$

[EQUIV RECV EXC]

$$\frac{}{? \lceil p \rceil_\tau \equiv_{p'} \lceil p \ \rightarrow \ p' \rceil_{\tau \mid \sharp : \tau'}}$$

[EQUIV OW]

$$\frac{s \equiv_p S}{! \ \mu.s \equiv_p p \ \langle \ ! \ \mu.S \rangle}$$

[EQUIV IW]

$$\frac{s \equiv_p S}{? \ \lceil p \rceil \ \langle \mu.s \rangle \equiv_{p'} p \ \langle \ ! \ \mu.S \rangle}$$

[EQUIV OB]

$$\frac{l; S \in \overline{l; S} \quad s \equiv_p S}{! \ l; s \equiv_p p \ \langle ! \ \overline{l; S} \rangle}$$

[EQUIV IB]

$$\frac{\overline{s} \equiv_p \overline{S}}{? \ \lceil p \rceil \ \langle \overline{l : s} \rangle \equiv_{p'} p \ \langle ! \ \overline{l; S} \rangle}$$

Figure 18: Type checking rules

[DEP COMM]

$$\frac{}{\lceil p \ \rightarrow \ p' \rceil_\tau \succ p :: p'}$$

[DEP COMM EX]

$$\frac{}{\lceil p \ \rightarrow \ p' \rceil_{\tau \mid \sharp : \tau'} \succ p :: p'}$$

[DEP LOOP]

$$\frac{S \succ \overline{p'}}{p \ \langle \ ! \ \mu.S \rangle \succ p :: \overline{p'}}$$

[DEP COND]

$$\frac{\overline{l; S} = l_1; S_1, ..., l_n; S_n \quad S_1 \succ p_1 ... S_n \succ p_n \quad \overline{p'} = p_1 :: ... :: p_n}{p \ \langle ! \ \overline{l; S} \rangle \succ p :: \overline{p'}}$$

[DEP TRY]

$$\frac{S_1 \succ \overline{p} \quad S_2 \succ \overline{p'}}{T\{S_1\}C\{S_2\} \succ \overline{p} :: \overline{p'}}$$

[DEP SEQ]

$$\frac{S_1 \succ \overline{p} \quad S_2 \succ \overline{p'}}{S_1; S_2 \succ \overline{p} :: \overline{p'}}$$

Figure 19: Linear dependence tracking.

send and a receiver. The rule [DEP COMM EX] is similar to [DEP COMM]. For loops, [DEP LOOP], the type of the body is analyzed. For conditionals, [DEP COND], each branch is analyzed separately and all are added together. This is necessary to capture dependencies on exceptions that propagate to outer contexts and, thus, can affect participants on each branch. Both the try and handle clauses are analyzed in [DEP TRY] and all communication actions are included. We assume that the compiler will validate that a failure can be raised in the **try** block, thus both paths (the standard and the exceptional) are feasible. Sequences are analyzed in the standard way in [DEP SEQ].

The relation ($\rightsquigarrow$) rewrites a stack of failure handlers (ϱ) and a global protocol type (S) to a stack of failure handlers (ϱ) and an annotated global protocol type ($S^{\overline{p}}$). The annotation on the type lists the dependencies for the type. Logically the dependencies are a list of processes that engage in communication *after* the computation described by (S). The rewrite rules are given in pairs to account for sequencing. The rule [RW COM] and rule [RW COM SEQ] define rewriting for basic communication types. In the case of a single communication, there are no additional dependencies other than the communication itself. For a communication sequence before a type (S), the dependencies are the processes engaged in communication within (S). The rules [RW COM EX] and [RW COM EX SEQ] are similar to their non failure counterparts, except that they also consider the most enclosing handler and the communications performed within the handler. The rules for loops, [RW LOOP] and [RW LOOP SEQ], introspect the body of the loop. The dependencies on the loop are the communicating processes occurring after the loop. The rules for conditionals, [RW COND] and [RW COND SEQ], are similar to those of loops. The most interesting rules are for try and handle types, [RW TRY] and [RW TRY SEQ]. Both these rules construct a stack of failure handlers and then calculate the dependencies for the try type and the handle type.

C. AUXILIARY RELATIONS

Figure 21 defines the filtering of a given participant's portion of a global protocol type from a global protocol type. Specifically given a process (*i.e.* participant) identifier, all relevant typing statements are filtered from the global protocol type. This relation is used to simplify the main projection rules, which project a local protocol type from a global protocol type.

The send rule for $\Rightarrow_T$ is the same as for $\Rightarrow$, except that each message is tagged with the nesting level at which it was sent. The nesting level is simply the size of the failure handling stack.

[RW COM]

$$\frac{}{(\varrho, \lceil p \ \rightarrow \ p' \rceil_\tau)_{\overline{p''}} \rightsquigarrow (\varrho, \lceil p \ \rightarrow \ p' \rceil_\tau^{\overline{p''}})}$$

[RW COM SEQ]

$$\frac{S \succ \overline{p''} \quad (\varrho, S) \rightsquigarrow (\varrho, S')}{(\varrho, \lceil p \ \rightarrow \ p' \rceil_\tau ; S)_{\overline{p'''}} \rightsquigarrow (\varrho, \lceil p \ \rightarrow \ p' \rceil_\tau^{\overline{p'''} :: \overline{p''}} ; S')}$$

[RW COM EX]

$$\frac{S \succ \overline{p''}}{(C\{S\} :: \varrho, \lceil p \ \rightarrow \ p' \rceil_{\tau | \sharp : \tau'})_{\overline{p'''}} \rightsquigarrow (\varrho, \lceil p \ \rightarrow \ p' \rceil_{\tau | \sharp : \tau'}^{\overline{p'''} :: \overline{p''}})}$$

[RW COM EX SEQ]

$$\frac{S \succ \overline{p''} \quad (s, S)_{\overline{p'''} :: \overline{p'}} \rightsquigarrow (s, S')}{(C\{S\} :: \varrho, \lceil p \ \rightarrow \ p' \rceil_{\tau | \sharp : \tau'} ; S)_{\overline{p'''}} \rightsquigarrow (C\{S\} :: \varrho, \lceil p \ \rightarrow \ p' \rceil_{\tau | \sharp : \tau'}^{\overline{p''}} ; S')}$$

[RW LOOP]

$$\frac{(\varrho, S_1)_{\overline{p'}} \rightsquigarrow (\varrho, S_1')}{(\varrho, p \ \langle \ ! \ \mu.S \rangle)_{\overline{p'}} \rightsquigarrow (\varrho, p \ \langle \ ! \ \mu.S \rangle^{\overline{p'}})}$$

[RW LOOP SEQ]

$$\frac{S_2 \succ \overline{p'} \quad (\varrho, p \ \langle \ ! \ \mu.S_1 \rangle)_{\overline{p''} :: \overline{p'}} \rightsquigarrow (\varrho, p \ \langle \ ! \ \mu.S_1' \rangle) \quad (\varrho, S_2)_{\overline{p''} :: \overline{p'}} \rightsquigarrow (\varrho, S_2')}{(\varrho, p \ \langle \ ! \ \mu.S \rangle)_{\overline{p''}} \rightsquigarrow (\varrho, p \ \langle \ ! \ \mu.S_1' \rangle^{\overline{p'}} ; S_2')}$$

[RW COND]

$$\frac{(\varrho, S_1)_{\overline{p}} \rightsquigarrow (\varrho, S_1')}{(\varrho, p \ \langle ! \ \overline{l; S_1} \rangle)_{\overline{p}} \rightsquigarrow (\varrho, p \ \langle ! \ \overline{l; S_1} \rangle^{\overline{p}})}$$

[RW COND SEQ]

$$\frac{S_2 \succ \overline{p'} \quad (\varrho, p \ \langle ! \ \overline{l; S_1} \rangle)_{\overline{p'} :: \overline{p''}} \rightsquigarrow (\varrho, p \ \langle ! \ \overline{l; S_1'} \rangle) \quad (\varrho, S_2)_{\overline{p'} :: \overline{p''}} \rightsquigarrow (\varrho, S_2')}{(\varrho, p \ \langle ! \ \overline{l; S_1} \rangle ; S_2)_{\overline{p''}} \rightsquigarrow (\varrho, p \ \langle ! \ \overline{l; S_1'} \rangle^{\overline{p'}} ; S_2')}$$

[RW TRY]

$$\frac{(C\{S_2\} :: \varrho, S_1)_{\overline{p}} \rightsquigarrow (C\{S_2\} :: \varrho, S_1') \quad (\varrho, S_2)_{\overline{p}} \rightsquigarrow (\varrho, S_2')}{(\varrho, T\{S_1\}C\{S_2\})_{\overline{p}} \rightsquigarrow (\varrho, T\{S_1'\}C\{S_2'\}^{\phi})}$$

[RW TRY SEQ]

$$\frac{S \succ \overline{p} \quad (C\{S_2\} :: \varrho, S_1)_{\overline{p'} :: \overline{p}} \rightsquigarrow (C\{S_2\} :: \varrho, S_1') \quad (\varrho, S_2)_{\overline{p'} :: \overline{p}} \rightsquigarrow (\varrho, S_2') \quad (\varrho, S) \rightsquigarrow (\varrho, S')}{(\varrho, T\{S_1\}C\{S_2\} ; S)_{\overline{p'}} \rightsquigarrow (\varrho, T\{S_1'\}C\{S_2'\}^{\overline{p}} ; S')}$$

Figure 20: Type dependence construction.

[FILTER SEQ]

$$\frac{\Pi_p(S_1) = S_1' \quad \Pi_p(S_2) = S_2'}{\Pi_p(S_1; S_2) = S_1'; S_2'}$$

[FILTER EMPTY]

$$\frac{p \neq p_1 \quad p \neq p_2}{\Pi_p(\lceil p_1 \ \rightarrow \ p_2 \rceil_\tau) = \phi}$$

[FILTER SEND]

$$\frac{}{\Pi_p(\lceil p \ \rightarrow \ p_1 \rceil_\tau) = \lceil p \ \rightarrow \ p_1 \rceil_\tau}$$

[FILTER RECV]

$$\frac{}{\Pi_p(\lceil p_1 \ \rightarrow \ p \rceil_\tau) = \lceil p_1 \ \rightarrow \ p \rceil_\tau}$$

[FILTER EMPTY EXC]

$$\frac{p \neq p_1 \quad p \neq p_2}{\Pi_p(\lceil p_1 \ \rightarrow \ p_2 \rceil_\tau | \ \sharp \ ) = \phi}$$

[FILTER SEND EXC]

$$\frac{}{\Pi_p(\lceil p \ \rightarrow \ p_1 \rceil_\tau | \ \sharp \ ) = \lceil p \ \rightarrow \ p_1 \rceil_\tau | \ \sharp}$$

[FILTER RECV EXC]

$$\frac{}{\Pi_p(\lceil p_1 \ \rightarrow \ p \rceil_\tau | \ \sharp \ ) = \lceil p_1 \ \rightarrow \ p \rceil_\tau | \ \sharp}$$

[FILTER LOOP]

$$\frac{}{\Pi_p(p_1 \ \langle \ ! \ \mu.S \rangle) = p \ \langle \ ! \ \mu.S' \rangle}$$

[FILTER BRANCH]

$$\frac{}{\Pi_p(p_1 \ \langle ! \ \overline{l; S} \rangle) = p \ \langle \ ! \ \mu.S' \rangle}$$

[FILTER TRY HANDLE]

$$\frac{\Pi_p(S_1) = S_1' \quad \Pi_p(S_2) = S_2' \quad S_1' \neq \phi \quad S_2' \neq \phi}{\Pi_p(T\{S_1\}C\{S_2\}) = T\{S_1'\}C\{S_2'\}}$$

[FILTER TRY HANDLE EMPTY]

$$\frac{\Pi_p(S_1) = \phi \quad \Pi_p(S_2) = \phi}{\Pi_p(T\{S_1\}C\{S_2\}) = \phi}$$

Figure 21: Filter of a participants portion of a global protocol type.

[SEND]

$$\frac{i = ||\sigma_1|| \quad M_2' = M_2[p_1 \mapsto M_2(p_1) :: (v, i)]}{p_1\{E[\textbf{send}(p_2, v)];\ M_1\}_{\sigma_1} \parallel p_2\{E[e];\ M_2\}_{\sigma_2} \parallel P, \mathbf{S} \Rightarrow p_1\{E[\textbf{unit}];\ M_1\}_{\sigma_1} \parallel p_2\{E[e];\ M_2'\}_{\sigma_2} \parallel P, \mathbf{S}}$$

Similarly, the receive rule simply ignores the tagged nesting level when processing a message.

[RECV]

$$\frac{M(p_2) = (v, i) :: \overline{(v, j)} \quad M' = M[p_2 \mapsto \overline{v}]}{p_1\{E[\textbf{recv}(p_2)];\ M\}_\sigma \parallel P, \mathbf{S} \Rightarrow p_1\{E[v];\ M'\}_\sigma \parallel P, \mathbf{S}}$$

Rule [CLEAN] takes a message map (M) and a nesting level (i) and removes all messages from the message map that were tagged with the nesting level or higher.

[CLEAN]

$$\frac{k \geq i > j \quad M = \{p_1 \mapsto \overline{(v, i)} :: \overline{(v, j)}, ..., p_n \mapsto \overline{(v, k)} :: \overline{(v, j)}\}}{Cl(M, i) = \{p_1 \mapsto \overline{(v, j)}, ..., p_n \mapsto \overline{(v, j)}\}}$$

D. ADVANCED FEATURES

This section discusses syntactic extensions to the basic features.

D.1 Extending Synchronization

One could consider exploiting the first addition to our basic semantics explained at the end of Section 4.1 to extend synchronization, when desired, to participants which are neither in the causal path of a failure nor appear in a given **handle** clause by artifically adding them to such a clause with a "bogus" communication. Since we believe it is important to give the programmer the choice on the extent of synchronization, the desire to thus add a participant does not seem absurd at all. To avoid however adding such artificial sends, which would have to be chosen carefully to not introduce synchronization beyond the extent desired, we allow **handle** handlers to be annotated with any number of participants (**handle**<P$_1$,...,P$_n$>) which have to be synchronized and informed upon a corresponding failure. This is the second exemption to the original rule of Section 4.1. If the occurrence of D in the example of Section 4.1 was using artificial communication, then the following is better:

```
try {
  A -> B: <T1> | F
  C -> B: <T2>
} handle<D>(F) {
  ...
}
```

D.2 Local Failures

For flexibility we also support one exemption from the propagation semantics presented in the context of nesting in Section 4.2.

In many cases failures — especially application-induced ones — denote specific values which prompt the recipient to choose an alternate treatment. Such a participant might very well be able to define a purely *local* treatment which overcomes the exceptional state. In other terms, there are scenarios where a recipient can deal entirely locally with a failure, e.g., by adopting a default value instead of the expected one. To capture these scenarios, a failure F can be wrapped in angle brackets, i.e., written [F]. Correspondingly, no failure handler has to be defined for it, and even if there is one in scope, the failure notification will not be propagated. In fact such a notation, e.g.,

```
A -> B: <T> | [F]
```

can be viewed as syntactic sugar for

```
try {
  A -> B: <T> | F
} handle(F) {}
```

that is, wrapping the corresponding send with an empty failure handler. It is easy to see how this can be straightforwardly composed with non-local failures thus ensuring orthogonality, e.g.,

```
A -> B: <T> | [F1] | F2
```

D.3 Explicitly Propagating Failures and Multi-level Retries

In effect, the notation A –> B: <T> | F can be viewed as a shortcut for a branching guarded by A that contains in one branch a subsequent send of a value of type T to B and in the other one an explicit raising of a failure of type F on B. As mentioned, the inherent support allows for communications to be combined. It seems natural to also allow an explicit raising of a failure without alternative sending A –> B: F. In addition, the possibility of explicitly propagating failures or notifying new failures from within **handle** clauses — as known from exceptions in mainstream languages — is another useful feature. Here, in contrast to the above, an explicit destination is not needed: the set of causally depending communications/participants is derived from the set used for the **handle** clause itself. In the example below, Line 8 will be performed, while Line 9 will be aborted in the case of a failure of type F.

```
1  try {
2    A –> B: <T1>
3    try {
4      A –> B: <T2> | F
5    } handle(F) {
6      B: throw F
7    }
8    C –> B: <T3>
9    A –> C: <T4>
10   ...
11 } handle(F) {
12   A: retry
13 }
14 ...
```

In the protocol *implementation* B can re-raise the very instance of F raised by A, or raise a new one. This mechanism is very convenient for implementing multi-level retries. The F failure propagated at Line 6 is namely used to delegate the retry decision to the outer scope, at Line 12, which allows the first send at Line 2 also to be repeated. A new failure type (e.g., FDeleg) could also be introduced for clarity.

We are currently considering adding syntactic sugar which would avoid the throwing of F and the introduction of the outer handler (and thus the outer **try**) by replacing Line 6 by A: **retry**(1). This would thus have the same effect of automatically transferring control 1 scope outwards and issuing the retry from there. In that light, we are also considering introducing support for multi-level aborts (e.g., A: abort(1) which would similarly save the introduction of a failure and a failure handler which in this case would be empty.

E. ADDITIONAL EXAMPLES

E.1 Three Phase Commit

Three Phase Commit (3PC) [27] is a refinement of 2PC which aims at supporting the case of coordinator failure (in addition to participant failure). The motivation is that if the coordinator fails together with at least one participant then it may be impossible for the participants to know what the outcome was/should have been. By catering for coordinator failures as well, the corresponding protocol type becomes vastly more complex — even for the case of only two participants — than the corresponding protocol type for 2PC (cf. Figure 5). This 3PC protocol type is defined in Figure 22. It was used for our evaluation.

E.2 Currency Broker

The buyer controls a loop where it keeps getting currency quotes until it gets an acceptable one or chooses to exit without buying. The broker simply keeps getting quotes from a seller and adds a brokerage amount to it and sends the updated quote to the buyer. If the buyer gets a good quote, it consults the user on whether to buy or not. If the user chooses to buy, the transaction begins with the broker buying from the third party seller and then selling to the buyer. During the time the user is consulted, the currency price may have changed. In this case, a failure is notified. When there is a price change, the user is notified and consulted again and the transaction is completed with the new price.

```
protocol 3PC {
  participants: Coordinator, Participant1, Participant2
  // Phase 1: send a ''CanCommit?'' request
  try {
    Coordinator -> Participant1: <Commit_Request>
      | CoordinatorFailure
    Coordinator -> Participant2: <Commit_Request>
      | CoordinatorFailure
    try {
      Participant1 -> Coordinator: <Can_Commit>
        | ParticipantFailure
      Participant2 -> Coordinator: <Can_Commit>
        | ParticipantFailure
      // Phase 2: either all participants agreed
      // to commit or some have refused
      try {
        Coordinator -> Participant1: <PrepOrAbort>
          | CoordinatorTimeoutFailure
        Coordinator -> Participant2: <PrepOrAbort>
          | CoordinatorTimeoutFailure
        Coordinator: {
          Commit: // In case we sent a Commit message
          try {
            Participant1 -> Coordinator: <ACK>
              | TimeoutFailure
            Participant2 -> Coordinator: <ACK>
              | TimeoutFailure
          } handle(TimeoutFailure) {
            Coordinator -> Participant1: <Abort>
            Coordinator -> Participant1: <Abort>
          }
          // Phase 3: all participants have
          // ACKed the Prepare message
          Coordinator -> Participant1: <Commit>
          Coordinator -> Participant2: <Commit>
          Participant1 -> Coordinator: <Committed>
          Participant2 -> Coordinator: <Committed>
          , Abort: //do nothing
        }
      } handle(CoordinatorTimeoutFailure) {}
    } handle(ParticipantFailure<Participant1>) {
      Coordinator -> Participant2: <Abort>
    } handle(ParticipantFailure<Participant2>) {
      Coordinator -> Participant1: <Abort>
    }
  } handle(CoordinatorFailure) {
    /* Coordination failure */ }
}
```

Figure 22: Global protocol type for 3PC

```
protocol CurrencyBroker {
  participant: Buyer, Broker, Seller
  Buyer: [                                  // Keep getting quotes
    Buyer -> Seller: <String>     // Request a quote
    Seller -> Broker: <Double>    // Seller's price
    Broker -> Buyer: <Double>     // Add 5% commission
    Buyer: {
      Accept:
        try {
          Broker -> Seller: <String>  //Sell
          Seller -> Broker: <String>
            | PriceChangeFailure
          Broker -> Buyer:
            <String>,           //Order Confirmation
        } handle(PriceChangeFailure) {
          Buyer: retry; // Retry only if you accept new quote
        }
      ,Reject:                                        // Do nothing
    }
  ]*
}
```

Figure 23: Global protocol type for the currency broker protocol