Compositional Programming and Testing of Dynamic Distributed Systems

ANKUSH DESAI, University of California, Berkeley, USA
AMAR PHANISHAYEE, Microsoft Research, USA
SHAZ QADEER, Microsoft Research, USA
SANJIT A. SESHIA, University of California, Berkeley, USA

A real-world distributed system is rarely implemented as a standalone monolithic system. Instead, it is composed of multiple independent interacting components that together ensure the desired system-level specification. One can scale systematic testing to large, industrial-scale implementations by decomposing the system-level testing problem into a collection of simpler component-level testing problems.

This paper proposes techniques for *compositional* programming and testing of distributed systems with two central contributions: (1) We propose a module system based on the theory of compositional trace refinement for *dynamic* systems consisting of asynchronously-communicating state machines, where state machines can be dynamically created, and communication topology of the existing state machines can change at runtime; (2) We present ModP, a programming system that implements our module system to enable compositional reasoning (assume-guarantee) of distributed systems.

We demonstrate the efficacy of our framework by building two practical fault-tolerant distributed systems, a transaction-commit service and a replicated hash-table. ModP helps implement these systems *modularly* and validate them via *compositional testing*. We empirically demonstrate that the abstraction-based compositional reasoning approach helps amplify the coverage during testing and scale it to real-world distributed systems. The distributed services built using ModP achieve performance comparable to open-source equivalents.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Software testing and debugging**; *Abstraction, modeling and modularity*; • **Networks →** *Protocol testing and verification*;

Additional Key Words and Phrases: domain-specific language; distributed systems; event-driven programming; actors; module system; compositional verification; systematic testing

ACM Reference Format:
Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 159 (November 2018), 30 pages. https://doi.org/10.1145/3276529

1 INTRODUCTION

Distributed systems are notoriously hard to get right. Programming these systems is challenging because of the need to reason about numerous control paths resulting from the myriad interleaving of messages and failures. Unsurprisingly, it is easy to introduce subtle errors while improvising to fill in gaps between high-level protocol descriptions and their concrete implementations. These problems have been highlighted by creators of large-scale distributed systems [Chandra et al. 2007].

Authors' addresses: Ankush Desai, University of California, Berkeley, USA, ankush@eecs.berkeley.edu; Amar Phanishayee, Microsoft Research, USA, amar@microsoft.com; Shaz Qadeer, Microsoft Research, USA, qadeer@microsoft.com; Sanjit A. Seshia, University of California, Berkeley, USA, sseshia@eecs.berkeley.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).
2475-1421/2018/11-ART159
https://doi.org/10.1145/3276529

Existing validation methods for distributed systems fall into two categories: *proof-based verification* and *systematic testing*. Researchers have used theorem provers to construct correctness proofs of both single-node systems [Chen et al. 2015; Hawblitzel et al. 2014; Klein et al. 2009; Wang et al. 2014] and distributed systems [Hawblitzel et al. 2015; Padon et al. 2016; Wilcox et al. 2015]. To prove a safety property on a distributed system, one typically needs to formulate an inductive invariant. Moreover, the inductive invariant often uses quantifiers, leading to unpredictable verification time and requiring significant manual assistance. While invariant synthesis techniques show promise, the synthesis of quantified invariants for large-scale distributed systems remains difficult. In contrast to proof-based verification, systematic testing explores behaviors of the system in order to find violations of safety specifications [Guo et al. 2011; Killian et al. 2007b; Yang et al. 2009]. Systematic testing is attractive to programmers as it is mostly automatic and needs less expert guidance. Unfortunately, even state-of-the-art systematic testing techniques scale poorly with increasing system complexity.

A distributed system is rarely implemented as a standalone monolithic system. Instead, it is composed of multiple independent interacting components that together ensure the desired system-level specification (e.g., our case study in Figure 1). One can scale systematic testing to large, industrial-scale implementations by decomposing the system-level testing problem into a collection of simpler component-level testing problems. Moreover, the results of component-level testing can be lifted to the whole system level by leveraging the theory of assume-guarantee (AG) reasoning [Abadi and Lamport 1995; Alur and Henzinger 1999; McMillan 2000]. We present a programming and testing framework, ModP (Modular P)[1], based on the principles of AG reasoning for *dynamic distributed systems.* ModP occupies a spot between proofs and black-box monolithic testing in terms of the trade-off between validation coverage and programmer effort.

Actors [Agha 1986; Akka 2017; Armstrong 2007; Bykov et al. 2010; Pony 2017] and state machines [Desai et al. 2013; Harel 1987; Killian et al. 2007a] are popular paradigms for programming distributed systems. These programming models support features like dynamic creation of machines (processes), directed messaging using machine references (as opposed to broadcast), and dynamic communication topology as references to machines can flow through the system (essential for modeling non-determinism like failures). ModP supports the actor [Agha 1986] model of computation and proposes extensions to make it amenable to compositional reasoning.

These dynamic features have an important impact on assume-guarantee (AG) reasoning, which typically relies on having explicit component interfaces – e.g., wires between circuits or shared variables between programs [Alur and Henzinger 1999; Lynch and Tuttle 1987]. In dynamic distributed systems, *interfaces between modules can change* as new state machines instances are created, or communication topology changes and this dynamic behavior depends on the context of a module. While some formalisms for AG reasoning [Attie and Lynch 2001; Fisher et al. 2011] support such dynamic features, they do not provide a programming framework for building practical dynamic distributed systems. Thus, to the best of our knowledge, ModP is the first system that supports assume-guarantee reasoning in a practical programming language with these dynamic features.

ModP introduces a *module system* to compositionally build a distributed system. A *ModP module* is a collection of dynamically-created and concurrently-executing state machines whose semantics is a collection of traces over externally visible actions. We formalize *refinement* as trace containment and define the semantics of ModP modules so that the composition of modules P and Q behaves like language intersection over the traces of P and Q. ModP also provides operators for *hiding actions* of a module, to construct a more abstract module. To ensure that compositional refinement

[1]ModP stands for Modular P and is available as part of the P programming framework [P-GitHub 2018]. We implemented our module system on top of the P [Desai et al. 2013] programming language.

holds in the presence of hiding, an especially challenging problem in a language where permission (machine-reference) to send events flows dynamically across machines, we use a methodology based on *permission-based capabilities control* [Hennessy and Riely 2002; Riely and Hennessy 1998]. Finally, ModP introduces a notion of *interfaces* as a proxy for state machines. Instead of creating state machines directly, ModP requires creating a machine indirectly as an instantiation of an interface, with the binding from an interface to the machine specified explicitly by the programmer. Separating the specification of the interface binding from the code that instantiates it allows flexibility in specializing machines and substituting one machine for another.

We have implemented ModP on top of P [Desai et al. 2013], a state machine based programming language that supports the dynamic features required for building realistic asynchronous systems. P has been used for implementing Windows device drivers [Desai et al. 2013] and for programming safe robotics systems [Desai et al. 2017a,b]. The ModP compiler generates code for compositional testing, which involves both safety and refinement testing of the decomposed system. We empirically demonstrate that ModP's abstraction-based decomposition helps the existing P systematic testing (both explicit and symbolic execution) back-ends to scale to large distributed systems.

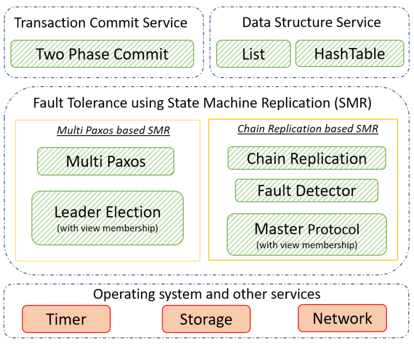

Fig. 1. Fault-Tolerant Distributed Services

Figure 1 shows two large distributed services that are representative of challenges in real-world distributed systems: (i) atomic commit of updates to decentralized, partitioned data using two-phase commit [Gray and Lamport 2006], and (ii) replicated data structures such as hash-tables and lists. These services use State Machine Replication (SMR) for fault-tolerance. Protocols for SMR, such as Multi-Paxos [Lamport 1998] and Chain-Replication [van Renesse and Schneider 2004], in turn use other protocols like leader election and fault detectors. To evaluate ModP, we implemented each sub-protocol (diagonal lines) as a separate module and performed compositional reasoning at each layer of the protocol stack. The AG approach would be to test each of the sub-protocol in isolation using abstractions of the other protocols. For example, when testing the two-phase commit protocol, we replace the Multi-Paxos based SMR implementation with its single process linearizability abstraction. Our evaluation demonstrates that such abstraction based decomposition provides orders of magnitude test-coverage amplification compared to monolithic testing. Further, our approach for checking refinement through testing is effective in finding errors in module abstractions. We compare the performance of the hash-table distributed service against its open-source counterpart by benchmarking it on a cluster.

To summarize, this paper makes the following novel contributions:

1. We present a new theory of compositional refinement and a module system for the assume-guarantee reasoning of dynamic distributed systems;
2. We implement a programming framework, ModP, that leverages this theory to enable compositional systematic-testing of distributed systems, and
3. Using ModP, we build two fault-tolerant distributed services for demonstrating the applicability of compositional programming and testing; we present an empirical evaluation of the systematic testing and runtime performance of these distributed services that combine 7 different protocols.

2 OVERVIEW

We illustrate the ModP framework for compositionally implementing, specifying, and testing distributed systems by developing a simple client-server application.

2.1 Basic Programming Constructs in ModP

ModP builds on top of P [Desai et al. 2013], an actor-oriented [Agha 1986] programming language where actors are implemented as state machines. A ModP program comprises state machines communicating asynchronously with each other using events accompanied by typed data values. Each machine has an input buffer, event handlers, and a local store. The machines run concurrently, receiving and sending events, creating new machines, and updating the local store.

We introduce the key constructs of ModP through a simple client-server application (see Figure 2) implemented as a collection of ModP state machines. In this example, the client sends a request to the server and waits for a response; on receiving a response from the server, it computes the next request to send and repeats this in a loop. The server waits for a request from the client; on receiving a request, it interacts with a helper protocol to compute the response for the client.

```
 1 /* Events */
 2 event eRequest : RequestType;
 3 event eResponse: ResponseType;
 4 ...
 5 /* Types */
 6 type RequestType =
 7 (source: ClientIT, reqId:int, val: int);
 8 type ResponseType = (resId: int, success: bool);
 9
10 machine ClientImpl receives eResponse;
11 sends eRequest; creates ServerToClientIT;
12 {
13   var server : ServerToClientIT;
14   var nextId, nextVal : int;
15   start state Init {
16     entry {
17       server = new ServerToClientIT;
18       goto StartPumpingRequests;
19     }
20   }
21   state StartPumpingRequests {
22     entry {
23       if(nextId < 5) //send 5 requests
24       {
25         send server, eRequest, (source = this,
26                 reqId = nextId, val = nextVal);
27         nextId++;
28       }
29     }
30     on eResponse do (payload: ResponseType) {
31       /* compute nextVal */
32       goto StartPumpingRequests;
33     }
34   }
35 }
```

(a) Client State Machine

```
 1 /* Interfaces */
 2 interface ServerToClientIT receives eRequest;
 3 interface ClientIT receives eResponse;
 4 interface HelperIT receives eProcessReq;
 5
 6 machine ServerImpl
 7 sends eResponse, eProcessReq;
 8 receives eRequest, eReqSuccess, eReqFail;
 9 creates HelperIT;
10 {
11   var helper: HelperIT;
12   start state Init {
13     entry {
14       helper = new HelperIT;
15       goto WaitForRequests;
16     }
17   }
18
19   state WaitForRequests {
20     on eRequest do (payload: RequestType) {
21       var client: ClientIT;
22       var result: bool;
23       client = payload.source;
24       /* interacts with the helper machine */
25       send helper, eProcessReq,
26                 (payload.reqId, payload.val);
27       ...
28       /* outcome: result = true or false*/
29       send client, eResponse,
30       (resId = payload.reqId, success = result);
31     }
32   }
33 }
34 machine HelperImpl receives eProcessReq;
35 sends eReqSuccess, eReqFail, ..; creates .. ;
36 { /* body */ }
```

(b) Server State Machine

Fig. 2. A Client-Server Application using ModP State Machines

Events and Interfaces. An event declaration has a name and a payload type associated with it. Figure 2a (line 2) declares an event `eRequest` that must be accompanied by a tuple of type `RequestType`. Figure 2a (line 6) declares the named tuple type `RequestType`. ModP supports primitive types like int, bool, float, and complex types like tuples, sequences and maps. Each interface declaration has an interface name and a set of events that the interface can receive. For example, the interface `ClientIT` declared at Figure 2b (line 3) is willing to receive only event `eResponse`. Interfaces are like symbolic names for machines. In ModP, unlike in the actor model where an instance of an actor is created using its name, an instance of a machine is created indirectly by performing **new** of an interface and linking the interface to the machine separately. For example, execution of the statement `server = new ServerToClientIT` at Figure 2a (line 17) creates a fresh instance of machine

ServerImpl and stores a unique reference to the new machine instance in server. The link between ServerToClientIT and ServerImpl is provided separately by the programmer using the **bind** operation.

Machines. Figure 2a (line 10) declares a machine ClientImpl that is willing to receive event eResponse, guarantees to send no event other than eRequest, and guarantees to create (by executing **new**) no interface other than ServerToClientIT. The body of a state machine contains variables and states. Each state can have an entry function and a set of event handlers. The entry function of a state is executed each time the machine transitions into that state. After executing the entry function, the machine tries to dequeue an event from the input buffer or blocks if the buffer is empty. Upon dequeuing an event from the input queue of the machine, the attached handler is executed. Figure 2a (line 30) declares an event-handler in the StartPumpingRequests state for the eResponse event, the payload argument stores the payload value associated with the dequeued eResponse event. The machine transitions from one state to another on executing the **goto** statement. Executing the statement **send** t,e,v adds event e with payload value v into the buffer of the target machine t. Sends are buffered, non-blocking, and directed. For example, the send statement Figure 2a (line 25) sends eRequest event to the machine referenced by the server identifier. In ModP, the type of a machine-reference variable is the name of an interface (Section 3.2).

Next, we walk through the implementation of the client (ClientImpl) and the server (ServerImpl) machines in Figure 2. Let us assume that the interfaces ServerToClientIT, ClientIT, and HelperIT are programmatically linked to the machines ServerImpl, ClientImpl, and HelperImpl respectively (we explain these bindings in Section 2.2). A fresh instance of a ClientImpl machine starts in the Init state and executes its entry function; it first creates the interface ServerToClientIT that leads to the creation of an instance of the ServerImpl machine, and then transitions to the StartPumpingRequests state. In the StartPumpingRequests state, it sends a eRequest event to the server with a payload value and then blocks for a eResponse event. On receiving the eResponse event, it computes the next value to be sent to the server and transitions back to the StartPumpingRequests state. The this keyword is the "self" identifier that references the machine itself. The ServerImpl machine starts by creating the HelperImpl machine and moves to the WaitForRequests state. On receiving a eResponse event, the server interacts with the helper machine to compute the result that it sends back to the client.

Dynamism. Two key features lead to dynamism in this model of computation, making compositional reasoning challenging: **(1)** *Machines can be created dynamically* during the execution of the program using the **new** operation that returns a reference to the newly-created machine. **(2)** References to machines are first-class values, and the payload in the sent event can contain references to other machines. Hence, the *communication topology can change dynamically* during the execution of the program.

2.2 Compositional Programming using ModP Modules

```
module ClientModule = {
  ClientIT -> ClientImpl
};
module ServerModule = {
  ServerToClientIT -> ServerImpl,
  HelperIT -> HelperImpl
};
//C code generation for the implementation.
implementation app: ClientModule || ServerModule;

module ServerModule' = {
  ServerToClientIT -> ServerImpl',
  HelperIT -> HelperImpl
};
implementation app': ClientModule || ServerModule';
```

Fig. 3. Modular Client-Server Implementation

ModP allows the programmer to decompose a complex system into simple components where each component is a ModP module. Figure 3 presents a modular implementation of the client-server application. A primitive module in ModP is a set of bindings from interfaces to state machines. ServerModule is a primitive module consisting of machines ServerImpl and HelperImpl where the ServerImpl machine is bound to the ServerToClientIT interface and the HelperImpl machine is bound to the HelperIT interface. The compiler ensures that the creation of an interface leads to the creation of a machine to which it binds. For example, creation of the `ServerToClientIT` interface (executing **`new`** `ServerToClientIT`) by any machine inside the module or by any machine in the environment (i.e., outside `ServerModule`) would lead to the creation of an instance of `ServerImpl`.

The client-server application (Figure 2) can be implemented modularly as two separate modules `ClientModule` and `ServerModule`; these modules can be implemented and tested in isolation. Modules in ModP are *open systems*, i.e., machines inside the module may create interfaces that are not bound in the module. Similarly, machines may send events to or receive events from machines that are not in the module. For example, the `ClientImpl` machine in `ClientModule` creates an interface `ServerToClientIT` that is not bound to any machine in `ClientModule`, it sends `eRequest` and receives `eResponse` from machines that are not in `ClientModule`.

Composition in ModP (denoted ||) is supported by type checking. If the composition type checks (Section 4) then the composition of modules behaves like language intersection over the traces of the modules. The compiler ensures that the joint actions in the composed module (`ClientModule || ServerModule`) are linked appropriately, e.g., the creation of the interface `ServerToClientIT` (Figure 2a line 18) in `ClientModule` is linked to `ServerImpl` in `ServerModule` and all the sends of `eRequest` events are enqueued in the corresponding `ServerImpl` machine. Note that the indirection enabled by the use of interfaces is critical for implementing the key feature of *substitution* required for modular programming, i.e., the ability to seamlessly replace one implementation module with another. For example, `ServerModule'` (Figure 3 line 11) represents a module where the server protocol is implemented by a different machine `ServerImpl'`. In module `ClientModule || ServerModule'`, the creation of an interface `ServerToClientIT` in the client machine is linked to machine `ServerImpl'`. The *substitution* feature is also critical for compositional reasoning, in which case, an implementation module is replaced by its abstraction. The compiler generates C code for the module in the **`implementation`** declaration.

2.3 Compositional Testing using ModP Modules

Monolithic testing of large distributed systems is prohibitively expensive due to an explosion of behaviors caused by concurrency and failures. The ModP approach to this problem is to use the principle of assume-guarantee reasoning for decomposing the monolithic system-level testing problem into simpler component-level testing problems; testing each component in isolation using abstractions of the other components.

Spec machines. In ModP, a programmer can specify temporal properties via specification machines (monitors). **`spec`** `s` **`observes`** `E1, E2 { .. }` declares a specification machine `s` that observes events `E1` and `E2`. If the programmer chooses to attach s to a module `M`, the code in `M` is instrumented automatically to forward any event-payload pair (e, v) to s if e is in the observes list of s; the handler for event e inside s executes synchronously with the delivery of e. The specification machines observe only the output events of a module. Thus, specification machines introduce a publish-subscribe mechanism for monitoring events to check temporal specifications while testing a ModP module. The module constructor **`assert`** `s` **`in`** `P` attaches specification machine `s` to module `P`. In Figure 4a, `ReqIdsAreMonoInc` and `ResIdsAreMonoInc` are specification machines observing events `eRequest` and `eResponse` to assert the safety property that the `reqId` and `resId` in the payload of these events are always monotonically increasing. Note that `ReqIdsAreMonoInc` is a property of the client machine and `ResIdsAreMonoInc` is a property of the server machine.

In ModP, abstractions used for assume-guarantee reasoning are also implemented as modules. For example, `AbstractServerModule` is an abstraction of the `ServerModule` where the `AbstractServerImpl` machine implements an abstraction of the interaction between `ServerImpl` and `HelperImpl` machine. The `AbstractServerImpl` machine on receiving a request sends back a random response.

ModP enables decomposing the monolithic problem of checking: `(`**`assert`** `ReqIdsAreMonoInc, ResIdsAreMonoInc` **`in`** `ClientModule || ServerModule)` into four simple proof obligations. ModP allows

```
machine AbstractServerImpl receives eRequest;
sends eResponse;
{
  start state Init {
    on eRequest do (payload: RequestType) {
    send payload.source, eResponse,
    (resId = payload.reqId,success = choose());
    }
  }
}
spec ReqIdsAreMonoInc observes eRequest {
  var prevId : int;
  start state Init {
    on eRequest do (payload: RequestType) {
      assert(payload.reqId == prevId + 1);
      prevId = payload.reqId;
    }
  }
}
spec ResIdsAreMonoInc observes eResponse
{
  var prevId : int;
  start state Init {
    on eResponse do (payload: ResponseType) {
      assert(payload.resId == prevId + 1);
      prevId = payload.resId;
    }
  }
}

```

(a) Abstraction and Specifications

```
 1 module AbstractServerModule = {
 2   ServerToClientIT -> AbstractServerImpl
 3 };
 4
 5 module AbstractClientModule = {
 6   ClientIT -> AbstractClientImpl
 7 };
 8
 9 /* Compositional Safety Checking */
10 //Test: ClientModule.
11 test test0: (assert ReqIdsAreMonoInc in ClientModule)
12                 || AbstractServerModule;
13 //Test: ServerModule.
14 test test1: (assert ResIdsAreMonoInc in ServerModule)
15                 || AbstractClientModule;
16
17 /* Circular Assume-Guarantee */
18 //Check that client abstraction is correct.
19 test test2: ClientModule || AbstractServerModule
20             refines
21             AbstractClientModule || AbstractServerModule;
22 //Check that server abstraction is correct.
23 test test3: AbstractClientModule || ServerModule
24             refines
25             AbstractClientModule || AbstractServerModule;
26
27 // Create abstract module using Hide
28 module hideModule = hide X in AbstractServerModule;
29
30 test test4: ClientModule || ServerModule
31             refines
32             AbstractClientModule || hideModule;
```

(b) Test Declarations for Compositional Testing

Fig. 4. Compositional Testing of the Client-Server Application using ModP Modules

the programmer to write each obligation as a test-declaration. The declaration **test** tname: P introduces a safety test obligation that the executions of module P do not result in a failure/error. The declaration **test** tname: P **refines** Q introduces a test obligation that module P refines module Q. The notion of refinement in ModP is trace-containment based only on externally visible actions, i.e., P refines Q, if every trace of P projected onto the visible actions of Q is also a trace of Q. ModP automatically discharges these test obligations using systematic testing. Using the theory of compositional safety (Theorem 5.3), we decompose the monolithic safety checking problem into two obligations (tests) test0 and test1 (Figure 4b). These tests use abstractions to check that each module satisfies its safety specification. Note that interfaces and the programmable bindings together enable substitution during compositional reasoning. For example, ServerToClientIT gets linked to ServerImpl in implementation but to its abstraction AbstractServerImpl during testing.

Meaningful testing requires that these abstractions used for decomposition be sound. To this end, ModP module system supports circular assume-guarantee reasoning (Theorem 5.4) to validate the abstractions. Tests test2 and test3 perform the necessary refinement checking to ensure the soundness of the decomposition (test0,test1). The challenge addressed by our module system is to provide the theorems of compositional safety and circular assume-guarantee for a dynamic programming model of ModP state machines. ModP module system also provides module constructors like **hide** for hiding events (interfaces) and **rename** for renaming of conflicting actions for more flexible composition. Hide operation introduces private events (interfaces) into a module, it can be used to convert some of the visible actions of a module into private actions that are no longer part of its visible trace. For example, assume that modules AbstractServerModule and ServerModule use event X internally for completely different purposes. In that case, the refinement check between them is more likely to hold if X is not part of the visible trace of the abstract module. Figure 4b (line 28-33) show how hide can be used in such cases. Ensuring compositional refinement for a dynamic language like ModP is particularly challenging in the presence of private events (Section 4.2)

2.4 Roadmap

ModP's module system supports two key theorems for the compositional reasoning of distributed systems: *Compositional Safety* (Theorem 5.3) and *Circular Assume-Guarantee* (Theorem 5.4). We use Section 3 through Section 5.1 to build up to these theorems. The module system formalized in this paper can be adapted to any actor-oriented programming language provided certain extensions can be applied. We describe these extensions that ModP state machines make to the P state machines in Section 3. When defining the operational semantics of a module and to ensure that *composition is intersection*, it is essential that constructed modules be well-formed. Section 4 presents the type-checking rules to ensure well-formedness for a module. Section 5.1 presents the operational semantics of a well-formed module. Finally, we describe how we apply the theory of compositional refinement to test distributed systems (Section 6) and present our empirical results (Section 8).

3 ModP STATE MACHINES

A module in ModP is a collection of the dynamic instances of ModP state machines. In this section, we describe the extensions ModP state machines makes to P state machines in terms of syntactic constructs and semantics. These extensions to P state machines are required for defining the operational semantics of ModP modules and making them amenable to compositional reasoning.
`(Extension 1)`: we add interfaces that are symbolic names for machines. In ModP, as described in Section 2.1, an instance of a machine is created indirectly by performing **new** of an interface (instead of **new** of a machine in P).
`(Extension 2)`: we extend P machines with annotations declaring the set of receive, send and create actions the dynamic instance of that machine can perform. These annotations are used to statically infer the actions a module can perform based on the actions of its comprising machines.
`(Extension 3)`: we extend the semantics of **send** in P to provide the guarantee that a ModP state machine can never receive an event (from any other machine) that is not listed in its receive set. This is achieved by extending machine identifiers with permissions (more details in Section 3.2).

3.1 Semantics of ModP State Machines

Let $\mathcal{E}$ represent the set of names of all the events. *Permissions* is a nonempty subset of $\mathcal{E}$; Let $\mathcal{K}$ represent the set of all permissions ($2^{\mathcal{E}} \setminus \{\emptyset\}$). Let $\mathcal{I}$ and $\mathcal{M}$ represent the sets of names of all interfaces and machines, respectively; these sets are disjoint from each other. Let $\mathcal{S}$ represent the set of all possible values the local state of a machine could have during execution. The local state of a machine represents everything that can influence the execution of the machine, including control stack and data structures. The buffer associated with a machine is modeled separately. Let $\mathcal{B}$ represent the set of all possible buffer values. The input buffer of a machine is a sequence of $(e, v) \in \mathcal{E} \times \mathcal{V}$ pairs, where $\mathcal{V}$ represent the set of all possible payloads that may accompany any event in a send action. Let $\mathcal{Z}$ be the set of all the machine identifiers.

A ModP state machine is a tuple (*MRecvs*, *MSends*, *MCreates*, *Rem*, *Enq*, *New*, *Local*) where:

1. $MRecvs \subseteq \mathcal{E}$ is the nonempty set of events received by the machine.
2. $MSends \subseteq \mathcal{E}$ is the set of all events sent by the machine.
3. $MCreates \subseteq \mathcal{I}$ is the set of interfaces created by the machine.
4. $Rem \subseteq \mathcal{S} \times \mathcal{B} \times \mathbb{N} \times \mathcal{S}$ is the transition relation for removing a message from the input buffer. If $(s, b, n, s') \in Rem$, then the n-th entry in the input buffer b is removed and the local state moves from s to s'.
5. $Enq \subseteq \mathcal{S} \times \mathcal{Z} \times \mathcal{E} \times \mathcal{V} \times \mathcal{S}$ is the transition relation for sending a message to a machine. If $(s, id, e, v, s') \in Enq$, then event e with payload v is sent to machine id and the local state of the sender moves from s to s'.

6. *New* $\subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ is the transition relation for creating an interface. If $(s, i, s') \in New$, then the machine linked against interface i is created and the machine moves from s to s'.
7. *Local* $\subseteq \mathcal{S} \times \mathcal{Z} \times \mathcal{S} \times \mathcal{Z}$ is the transition relation for local computation in the machine. The state of a machine is a pair $(s, id) \in \mathcal{S} \times \mathcal{Z}$. The first component s is the machine local-state. The second component id is a placeholder used to store the identifier of a freshly-created machine or to indicate the target of a send operation. If $(s, id, s', id') \in Local$, then the state can move from (s, id) to (s', id'), which allows us to model the movement of machine identifiers from s to id and vice-versa. The role of id will become clearer when we use it to define the operational semantics of the module (Section 5.1).

We refer to components of machine $m \in \mathcal{M}$ as $MRecvs(m)$, $MSends(m)$, $MCreates(m)$, $Rem(m)$, $Enq(m)$, $New(m)$, and $Local(m)$ respectively. We use $IRecvs(i)$ to refer to the receive set corresponding to an interface $i \in \mathcal{I}$.

3.2 Machine Identifiers with Permissions

A machine can send an event to another machine only if it has access to the receiver's machine identifier. The capability of a machine to send an event to another machine can change dynamically as machine identifiers can be passed from one machine to another. There are two key properties required for the compositional reasoning of communicating state machines using our module system: **(1)** *a machine never receives an event that is not in its receive set*, this property is required when formalizing the open module semantics of ModP modules and its receptiveness to input events (Section 5.1); **(2)** *the capability to send a private (internal) event of a module does not leak outside the module*, this property is required to ensure that compositional refinement in the presence of private events (Section 4.2). These properties are particularly challenging in the presence of machine-identifier that can flow freely. Our solution is similar in spirit to permissions based capability control for π-calculus [Hennessy and Riely 2002; Pierce and Sangiorgi 1996] where permissions are associated with channels or locations and enforced using type-systems.

We concretize the set of machine identifiers $\mathcal{Z}$ as $\mathcal{I} \times \mathbb{N} \times \mathcal{K}$. For our formalization, we are interested in machine identifiers that are embedded inside the data structures in a machine local-state $s \in \mathcal{S}$ or value $v \in \mathcal{V}$. Instead of formalizing all datatypes in ModP, we assume the existence of a function ids such that $ids(s)$ is the set containing all machine identifiers embedded in s and $ids(v)$ is the set containing all machine identifiers embedded in v. An identifier $(i, n, \alpha) \in \mathcal{Z}$ refers to the n-th instance of an interface represented by $i \in \mathcal{I}$. We refer to the final component α of a machine identifier as its *permissions*. The set α represents all the events that may be sent via this machine identifier using the **send** operation. The creation of an interface `I` returns a machine identifier $(\mathtt{I}, n, \beta) \in \mathcal{Z}$ referencing to the n-th instance of interface `I` where β represents the receive set associated with the interface `I` ($\beta = IRecvs(\mathtt{I})$). The ModP compiler checks that if an interface `I` is bound to `M` in a module, then the received events of `I` are contained in the received events of `M` ($IRecvs(\mathtt{I}) \subseteq MRecvs(\mathtt{M})$). Hence, the events that can be sent using an identifier is a subset of the events that the machine can receive. This mechanism ensures that a machine never receives an event that it has not declared in its receive set. Note that the permissions embedded in a machine identifier control the capabilities associated with that identifier.

In order to control the flow of these capabilities, ModP requires the programmer to annotate each event with a set $\mathcal{A} \in 2^{\mathcal{K}}$ of allowed permissions. For an event e, the set $\mathcal{A}(e)$ represents any permission that the programmer can put inside the payload accompanying e i.e., if v represents any legal payload value with e then $\forall(\_, \_, \alpha) \in ids(v), \alpha \in \mathcal{A}(e)$. In other words, $\mathcal{A}(e)$ represents the set of permissions that can be transferred from one machine to another using event e.

Finally, the modified send operation **send** `t,e,v` succeeds only if: **(1)** `e` is in the permissions of machine identifier `t`, to ensure `t` receives only those events that are in its receives set, and **(2)** all permissions embedded in `v` are in $\mathcal{A}$(`e`), the send fails otherwise (captured as the `(SendOk)` condition when defining the semantics of send in Section 5.1). This changed semantics of **send** based on permission-based capability control plays a crucial role in ensuring well-formedness of the hide operation that adds private events to a module (Section 4.2).

To statically check the permission that is passed using an event, we need to reflect the permission of a machine-reference stored in a variable in the variable's type. Recollect that, the type of a machine-reference variable is the name of an interface (Figure 2). An interface type represents the set of machine-identifiers whose permission is the receives events set of the interface. In other words, the type of a machine-identifier represents the permission stored in it. Thus, the static type of the payload associated with an event can be used to infer the permissions embedded in it and the check **(2)** above for the correctness of the send operation can be performed statically. We do not present the state-machine level typing rules for performing these checks statically because of space constraints; instead, they are described as dynamic checks when presenting the operational semantics in Section 5.1.

Remark 3.1. The module system formalized in this paper can be adopted to any actor-oriented programming language whose semantics is as described in Section 3.1 and can be extended with the three features `(Extension 1)` – `(Extension 3)`.

4 MODULES

ModP seeks to manage the complexity of a distributed system by designing it in a structured way, at different levels of abstractions and modularly as the composition of interacting modules. Figure 5 presents the expression language supported by ModP module system for module construction.

$$
\begin{array}{l}
\alpha \in 2^{\mathcal{E}} \quad \beta \in 2^{\mathcal{I}} \quad i, i', i_1, .., i_k \in \mathcal{I} \quad m_1, .., m_k \in \mathcal{M} \\
\begin{array}{rcl}
P, Q \in \mathit{ModuleExpr} & ::= & \textbf{bind}\ i_1 \rightarrow m_1, .., i_k \rightarrow m_k \\
& | & P \parallel Q \\
& | & \textbf{hide}\ \alpha\ \textbf{in}\ P \\
& | & \textbf{hide}\ \beta\ \textbf{in}\ P \\
& | & \textbf{rename}\ i \rightarrow i'\ \textbf{in}\ P
\end{array}
\end{array}
$$

Fig. 5. Module constructors

The **bind** constructor creates a primitive module as a collection of machines $m_1, \ldots, m_k$ bound to interfaces $i_1, \ldots, i_k$ respectively (syntax is a bit different from the examples in Section 2). The composition ($\|$) constructor builds a complex module from simpler ones. The **hide** constructor creates an abstraction of a module, by converting some of its visible actions to private actions. The **rename** operation enables reuse of modules (and resolution of conflicting actions) when composing modules to create larger ones. The module language enables programmatic construction of modules, reuse of module expressions and ease of assembling modules for compositional reasoning (Section 5.2).

Well-formed module. In the ModP module system, a module P is a syntactic expression and its well-formedness is checked using the judgment $P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P$. If module P satisfies the judgment then we read it as: *Module P is well-formed with private events EP_P, private interfaces IP_P, interface definition map I_P, interface link map L_P, events received ER_P, events sent ES_P, and interfaces created IC_P.* The judgment derives the components on the right-hand side which are used for defining the operational semantics of a well-formed module (Section 5.1). We use $dom(x)$ and $codom(x)$ to refer to the domain and codomain of any map x.

We next describe the components on the right-hand side of the judgment:

1. ***Private events.*** $EP_P \in 2^{\mathcal{E}}$ represents the private events for module P, these events must not cross the boundary of module P i.e. if a machine in P sends event $e \in EP_P$, then the target must

be some machine in P and, if a machine in P receives $e \in EP_P$, the sender must be some machine in P. The send of a private event is an internal (invisible) action of a module.

2. ***Private interfaces.*** $IP_P \in 2^{\mathcal{I}}$ represents the interfaces that are declared private in P; the creation of any interface in IP_P is an internal (invisible) action of P.
3. ***Interface definition map.*** $I_P \in \mathcal{I} \rightarrow \mathcal{M}$ interface definition map that binds an interface name i to a machine name $I_P[i]$. Recollect that in the ModP model of computation, dynamic instances of machines are created indirectly using interfaces. An interface definition map (I_P) is a collection of bindings from interface names to machine names. These bindings are initialized using the **bind** operation, so that if $(i, m) \in I_P$ then the creation of an interface i in module P leads to the creation of an instance of m.
4. ***Interface link map.*** $L_P \in \mathcal{I} \rightarrow \mathcal{I} \rightarrow \mathcal{I}$ is the *interface link map* that maps each interface $i \in dom(I_P)$ to a machine link map that binds interfaces created by the code of machine $I_P[i]$ to an interface name. If the statement **`new`** x is executed by an instance of machine $I_P[i]$, an interface actually created in lieu of the interface name x is provided by the machine specific link map $L_P[i]$. If $(x, x') \in L_P[i]$, then the compiler interprets x in statement **`new`** x in the code of machine $I_P[i]$ as creation of interface x', creating an instance of machine $I_P[x']$.

The last three components of the judgment can be inferred using the first four components:

5. ***Events received.*** $ER_P \in 2^{\mathcal{E}}$ represent the set of events received by module P. It is inferred as the set of non-private events received by machines in P, $ER_P = \bigcup_{m \in codom(I_P)} MRecvs(m) \setminus EP_P$.
6. ***Events sent.*** $ES_P \in 2^{\mathcal{E}}$ represent the set of events sent by module P. It is inferred as the set of non-private events sent by machines in P, $ES_P = \bigcup_{m \in codom(I_P)} MSends(m) \setminus EP_P$.
7. ***Interfaces created.*** $IC_P \in 2^{\mathcal{I}}$ represent the set of interfaces created by module P. It is inferred as the set of interfaces created by machines in P (interpreted based on its link map), $IC_P = \bigcup_{(i,m) \in I_P, x \in MCreates(m)} \{L_P[i][x]\}$.

Exported interfaces. The domain of the interface definition map after removing the private interfaces is the set of exported interfaces for module P; these interfaces can be created either by P or its environment.

Input and output actions. The *input events* of module P are the events that are received but not sent by P i.e. $ER_P \setminus ES_P$. The *input interfaces* of P are the set of interfaces that are exported but not created by P i.e. $dom(I_P) \setminus (IP_P \cup IC_P)$. The *output events* of P are the sent events i.e. ES_P and the *output interfaces* are the created non-private interfaces of P i.e. $IC_P \setminus IP_P$. Informally, the *input actions* of a module is the union of its input events and input interfaces, the *output actions* of a module is the union of its output events and output interfaces (formally defined in Definition 5.2).

In the rest of this section, we describe the various module constructors and present the static rules to ensure that the constructed module satisfies: (1) well-formedness conditions (`WF1` – `WF3`) required for defining the semantics of a module, and (2) the compositionality Theorems 5.1- 5.2.

Note. For simplicity, when describing the static rules we do not provide the derivation for the last three components of the judgment as they can be inferred, but we use them above the line.

4.1 Primitive Module

In ModP, a primitive module is constructed using the **bind** operation. Programmatically initializing I_P using **bind** operation enables linking the creation of an interface `I` to either a concrete machine `Impl` for execution or an abstract machine `Abs` for testing, a key feature required for substitution during compositional reasoning.

(Bind)

$$\frac{f = \{(i_1, m_1), \ldots, (i_n, m_n)\} \quad f \subseteq \mathcal{I} \rightarrow \mathcal{M}^{(b1)} \qquad \forall (i, m) \in f.\ IRecvs(i) \subseteq MRecvs(m)^{(b2)}}{\textbf{bind}\ i_1 \rightarrow m_1, \ldots, i_n \rightarrow m_n \vdash \{\}, \{\}, f, \{(i, x, x) \mid (i, m) \in f \wedge x \in MCreates(m)\}}$$

Rule Bind presents the rule for **bind** $i_1 \rightarrow m_1, \ldots, i_k \rightarrow m_k$ that constructs a primitive module by binding each interface i_k to machine m_k for $k \in [1, n]$. These bindings are captured in f; condition (b1) checks that f is a function. Condition (b2) checks that the received events of an interface are contained in the received events of the machine bound to it (ensures (WF1) below). The resulting module does not have any private events and interfaces. The function f is the interface definition map and the interface link map for interface $i \in dom(f)$ contains the identity binding for each interface created by $f(i)$ (ensures (WF2) below). The first entry for name x ever added to $L_P[i]$ is the identity map (x, x); subsequently, if interface x is renamed to x' (using **rename** constructor), this entry is updated to (x, x').

Well-formedness condition (WF1) helps ensure that a machine-identifier obtained by creating an interface can be used to send only those events that are in the receives set of the target machine ((SendOk) in Section 3.2).

(WF1) ***Interface definition map is consistent:*** *For each $(i, m) \in I_P$, we have $IRecvs(i) \subseteq MRecvs(m)$.*

Well-formedness condition (WF2) ensures that the link map lookups used during the create action always succeed.

(WF2) ***Interface link map is consistent:*** *The domains of I_P and L_P must be identical and for each $(i, m) \in I_P$ and $x \in MCreates(m)$, we have $x \in dom(L_P[i])$.*

4.2 Hiding Events and Interfaces

Hiding events and interfaces in a module allow us to construct a more abstract module [Attie and Lynch 2001]. There are two reasons to construct a more abstract version of a module P by hiding events or interfaces. First, suppose we want to check that another module `ServerModule` refines `AbstractServerModule`. But the event `X` is used for internal interaction among machines, for completely different purposes, in both `ServerModule` and `AbstractServerModule`. Then, the check that `ServerModule` refines `AbstractServerModule` is more likely to hold since sending of `X` is not a visible action of `AbstractServerModule`. Second, hiding helps make a module more composable with other modules. To compose two modules, the sent events and created interfaces of one module must be disjoint from the sent events and created interfaces of the other (Section 4.3). This restriction is onerous for large systems consisting of many modules, each of which may have been written independently by a different programmer. To address this problem, we relax disjointness for private events and interfaces, thus allowing incompatible modules to become composable after hiding conflicting events and interfaces.

To illustrate hiding of an event and an interface, we revisit the `ServerModule` in Figure 3. To legally hide an event in a module, it must be both a sent and received event of the module.

```
module HE_Server = hide eProcessReq, eReqSuccess, eReqFail in ServerModule
```

Module `HE_Server` is well-formed and `eProcessReq`, `eReqSuccess`, `eReqFail` become private events in it. A send of event `eProcessReq` is a visible action in `ServerModule` but a private action in `HE_Server`.

To hide an interface, it must be both an exported and created interface of that module.

```
module HI_Server = hide HelperIT in HE_Server
```

Module `HI_Server` is well-formed and interface `HelperIT` becomes a private interface in it. Creation of interface `HelperIT` is a visible action in `HE_Server` but a private action in `HI_Server`. *Hiding makes events and interfaces private to a module and converts output actions into internal actions.* All interactions between the server and the helper machine in `HI_Server` are private actions of the module.

Avoiding private permission leakage. Not requiring disjointness of private events creates a possibility for programmer error and a challenge for compositional refinement. When reasoning about a module P in isolation, only its input events (that are disjoint from private events) would be considered as input actions. This is based on the assumption that private events of a module are exchanged only within a module, in other words, a private event of a module can never be sent by any machine outside the module to any machine inside the module.

Recollect that a machine can send only those events to a target machine that are in the permission set of the reference to the target machine (Section 3.2). Suppose a machine `M` in module `P` has a private event e in its set of received events. Any machine that possesses a reference to an instance of `M` could send e to this instance. If such a reference were to leak outside the module `P` to a machine in a different module, it would create an obstacle to reasoning about `P` separately (and proving the compositionality theorems for a module with private events), since the environment may now send private events targeted at a machine inside `P`. ModP ensures that such leakage of a machine reference with permissions containing a private event cannot happen.

In ModP, there are two ways for permissions to become available to a machine: (1) by creating an interface, or (2) by sending permissions to the machine in the payload accompanying some event. To tackle private permission leakage through (1), ModP requires that an input interface not have a private event in its set of received events so that an interface with private permissions cannot be created from outside the module. This is ensured by the condition `(he2)` below. To tackle private permission leakage through (2), ModP enforces that (a) each send of event e adheres to the specification `(SendOk)` in Section 3, and (b) the set of private events is disjoint from any permission in $\mathcal{A}(e)$ for any non-private event e (ensure `(WF3)` below). Together, these two checks ensure that permission containing a private event does not leak outside the module through sends.
(`WF3`) ***Permissions to send private events does not leak:*** *For all* $e \in ER_P \cup ES_P$ *and* $\alpha \in \mathcal{A}(e)$*, we have* $\alpha \cap EP_P = \emptyset$. This is a static check asserting the capabilities that can leak outside the module.

$$\textbf{(HideEvent)}\quad (A \Delta B = (A \setminus B) \cup (B \setminus A))$$
$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \qquad \alpha \subseteq ER_P \cap ES_P^{\texttt{(he1)}} \\ \forall x \in IC_P \Delta dom(I_P).\ IRecvs(x) \cap \alpha = \emptyset^{\texttt{(he2)}} \\ \forall e \in (ER_P \cup ES_P) \setminus \alpha.\ \forall \alpha' \in \mathcal{A}(e).\ \alpha \cap \alpha' = \emptyset^{\texttt{(he3)}} \end{array}}{\textbf{hide } \alpha \textbf{ in } P \vdash EP_P \cup \alpha, IP_P, I_P, L_P}$$

$$\textbf{(HideInterface)}$$
$$\frac{P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \quad \beta \subseteq dom(I_P) \cap IC_P^{\texttt{(hi1)}}}{\textbf{hide } \beta \textbf{ in } P \vdash EP_P, IP_P \cup \beta, I_P, L_P}$$

Rule HideEvent handles the hiding of a set of events α in module P. This rule adds α to EP_P. Condition `(he1)` checks all events in β are both sent and received by module P; this condition is required to ensure that the resulting module is an abstraction of P. Conditions `(he2)` and `(he3)` together ensure that once an event e becomes private, any permission containing e cannot cross the boundary of the resulting module (ensure `(WF3)`). Rule HideInterface handles the hiding of a set of interfaces β in module P. This rule adds β to IP_P. Condition `(hi1)` is similar to the condition `(he1)` of rule HideEvent; this condition ensures that the resulting module is an abstraction of P.

4.3 Module Composition

Module composition in ModP enforces an extra constraint that the output actions of the modules being composed are disjoint. The requirement of output disjointness i.e. output actions of `P` and `Q` be disjoint in order to compose them is important for compositional reasoning, especially to ensure that *composition is intersection* (Theorem 5.1). For defining the open system semantics of a module P (Section 5.1), we require P to be *receptive only* to its input actions (sent by its environment). In other words, for the input actions, `P` assumes that its environment will not send it any event sent by `P` itself. Similarly, `P` assumes that its environment will not create an interface that is created by `P` itself. Any input action of `P` that is an output action of `Q` is an output action of $P \parallel Q$ and hence not an input action of $P \parallel Q$. This property ensures that by composing `P` with a module `Q` (that outputs some input action of `P`), we achieve the effect of constraining the behaviors of `P`. Thus, the composition is a mechanism used to introduce details about the environment of a component, which constrains its behaviors (*composition is intersection*), and ultimately allows us to establish the safety properties of the component.

However, composition inevitably makes the size of the system larger thus making the testing problem harder. Hence, we need abstractions of components to allow precise yet compact modeling of the environment. If one component is replaced by another whose traces are a subset of the former, then the set of traces of the system only reduces, and not increases, i.e., no new behaviors are added (*trace containment is monotonic with respect to composition* Theorem 5.2). This permits refinement of components in isolation.

(Composition) $(A \Delta B = (A \setminus B) \cup (B \setminus A))$

$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \quad Q \vdash EP_Q, IP_Q, I_Q, L_Q, ER_Q, ES_Q, IC_Q \qquad dom(I_P) \cap dom(I_Q) = \emptyset^{(c1)} \\ (ER_P \cup ER_Q \cup ES_P \cup ES_Q) \cap (EP_P \cup EP_Q) = \emptyset^{(c2)} \qquad \forall x \in (dom(I_P) \Delta IC_P) \cup (dom(I_Q) \Delta IC_Q).\ IRecvs(x) \cap (EP_P \cup EP_Q) = \emptyset^{(c3)} \\ \forall e \in ER_P \cup ER_Q \cup ES_P \cup ES_Q.\ \forall \alpha \in \mathcal{A}(e).\ \alpha \cap (EP_P \cup EP_Q) = \emptyset^{(c4)} \qquad IC_P \cap IC_Q = \emptyset^{(c5)} \qquad ES_P \cap ES_Q = \emptyset^{(c6)} \end{array}}{P \parallel Q \vdash EP_P \cup EP_Q, IP_P \cup IP_Q, I_P \cup I_Q, L_P \cup L_Q}$$

Rule Composition handles the composition of P and Q. Condition (c1) enforces that the domains of I_P and I_Q are disjoint, thus preventing conflicting interface bindings. Conditions (c2) ensures that the input and output actions of P are not hidden by private events of Q and vice-versa. Conditions (c3) and (c4) together check that private permissions of $P \parallel Q$ do not leak out. Condition (c3) checks that creation of an input interface of P does not leak permission containing a private event of Q and vice-versa. Condition (c4) checks that non-private events sent or received by P do not leak a permission containing a private event of Q and vice-versa (ensure (WF3)). Condition (c5) checks that created interfaces are disjoint; condition (c6) checks that sent events are disjoint. Composition is associative and commutative.

Example. If the conditions (c1) to (c6) hold then the composition of two modules is a union of its components. The composition operation acts as a language intersection. Consider the example of `ClientModule || ServerModule` from Figure 3. The interface `ServerToClientIT` is an input interface of `ServerModule` but becomes an output (no longer input) interface of `ClientModule || ServerModule`. Similarly, `eResponse` is an input event of `ClientModule` but becomes an output event of the composed module. Also, the union of the link-map and the interface definition maps ensures that the previously unbounded interfaces in link-map are appropriately bound after composition.

4.4 Renaming Interfaces

```
1 interface ServerToClientIT' receives eRequest,eReqFail;
2 interface HelperIT' receives eProcessReq;
3
4 machine HelperImpl' receives eProcessReq;
5 sends ..; creates ..; { /* body */ }
6
7 module ServerModule' = {
8   ServerToClientIT'->ServerImpl, HelperIT->HelperImpl
9 };
10
11 module allServers = ServerModule ||
12   rename HelperIT -> HelperIT' in ServerModule';
```

Fig. 6. Renaming Interfaces Module Constructor

The **rename** module constructor allows us to rename conflicting interfaces before composition. The example in Figure 6 builds on top of the Client-Server example in Section 2. In module `ServerModule'`, the interface `ServerToClientIT'` is bound to machine `ServerImpl`. The creation of `HelperIT` interface (Figure 2b line 14) in `ServerImpl` machine is bound to `HelperImpl` machine in both `ServerModule` and `ServerModule'`. But, it is not possible to compose modules `ServerModule` and `ServerModule'` because of the conflicting bindings of interface `HelperIT` (rule Composition (c1)).

In Figure 6 (line 12), the interface name `HelperIT` is renamed to `HelperIT'`. The rename module constructor updates the interface binding `(HelperIT->HelperImpl)` to `(HelperIT' -> HelperImpl)` and the interface link map entry of `(ServerToClientIT'->HelperIT->HelperIT)` to `(ServerToClientIT'->HelperIT ->HelperIT')`. As a result, the composition of modules `ServerModule` and `ServerModule'` is now possible.

Recollect that each module has an *interface link map* (Section 4) that maintains a machine specific mapping from the interface created by the code of a machine to the actual interface to be created in lieu of the **new** operation. The *interface link map* plays a critical role enable renaming of interfaces without changing the code of the involved machines. The execution of **new** `HelperIT` (Figure 2b line

14) in `ServerImpl` still leads to the creation of `HelperImpl` machine because of the indirection in the interface link map, and the corresponding visible action is creation of interface `HelperIT'`.

(Rename)

$$\frac{\begin{array}{c} P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P \quad i \in dom(I_P) \cup IC_P(\mathtt{r1}) \quad IRecvs(i) = IRecvs(i')(\mathtt{r2}) \\ i' \in \mathcal{I} \setminus (dom(I_P) \cup IC_P)(\mathtt{r3}) \quad A = \{x \mid x' \in IP_P \wedge x = ite(x' = i, i', x')\}(\mathtt{r4}) \quad B = \{(x, y) \mid (x', y) \in I_P \wedge x = ite(x' = i, i', x')\}(\mathtt{r5}) \\ C = \{(x, y, z) \mid (x', y, z') \in L_P \wedge x = ite(x' = i, i', x') \wedge z = ite(z' = i, i', z')\}(\mathtt{r6}) \end{array}}{\textbf{rename } i \longrightarrow i' \textbf{ in } P \vdash EP_P, A, B, C}$$

Rule Rename handles the renaming of interface i to i' in module P. Condition `(r1)` checks that i is well-scoped; the set of $dom(I_P) \cup IC_P$ is the universe of all interfaces relevant to P. Condition `(r2)` checks that the set of received events of i and i' are the same. Condition `(r3)` checks that i' is a new name different from the current set of interfaces relevant to P. Together with condition `(b2)` in rule Bind, this condition ensures that the set of received events of an interface is always a subset of the set of received events of the machine bound to it. Condition `(r4)` calculates in A the renamed set of private interfaces. Condition `(r5)` calculates in B the renamed interface defintion map. Condition `(r6)` calculates in C the renamed interface link map.

5 COMPOSITIONAL REASONING USING ModP MODULES

The ModP module system allows compositional reasoning of a module based on the principles of assume-guarantee reasoning. For assume-guarantee reasoning, the module system must guarantee that *composition is intersection* (Theorem 5.1), i.e., traces of a composed module are entirely determined by the traces of the component modules. We achieve this by first ensuring that a module is well-formed (Section 4), and then defining the operational semantics (as a set of traces) of a well-formed module such that its trace behavior (observable traces) satisfies the compositional trace semantics required for assume-guarantee reasoning.

In Section 4, a ModP module is described as a syntactic expression comprising of the module constructors listed in Figure 5. If the static rules are satisfied then any constructed module P is well-formed and can be represented by its components $(EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P)$. In this section, we present the operational semantics of a well-formed module (Section 5.1) that help guarantee the key compositionality theorems described in Section 5.2.

5.1 Operational Semantics of ModP Modules

A key requirement for assume-guarantee reasoning [Alur et al. 1998; Lynch and Tuttle 1987] is to consider each component as an *open system* that continuously reacts to input that arrives from its environment and generates outputs. The transitions (executions) of a module include non-deterministic interleaving of possible environment actions. Each component must be modeled as a labeled state-transition system so that traces of the component can be defined based only on the externally visible transitions of the system.

We refer to components on the right hand side of the judgment $P \vdash EP_P, IP_P, I_P, L_P, ER_P, ES_P, IC_P$ (Section 4) when defining the operational semantics of a well-formed module P. We present the *open system* semantics of a *well-formed* module P as a labeled transition system.

Configuration. A configuration of a module is a tuple (S, B, C): **(1)** The first component S is a partial map from $\mathcal{I} \times \mathbb{N}$ to $\mathcal{S} \times \mathcal{Z}$. If $(i, n) \in dom(S)$, then $S[i, n]$ is the state of the n-th instance of machine $I_P[i]$. The state $S[i, n]$ has two components, local state $s \in \mathcal{S}$ and a machine identifier $id \in \mathcal{Z}$ (as described in Section 3.1). **(2)** The second component B is a partial map from $\mathcal{I} \times \mathbb{N}$ to $\mathcal{B}$. If $(i, n) \in dom(B)$, then $B[i, n]$ is the input buffer of the n-th instance of the machine $I_P[i]$. **(3)** The third component C is a map from $\mathcal{I}$ to $\mathbb{N}$. $C[i] = n$ means that there are n dynamically created instances of interface i.

We present the operational semantics of a well-formed module P as a transition relation over its configurations (Figure 7). Let (S_P, B_P, C_P) represent the configuration for a module P. A transition is represented as $(S_P, B_P, C_P) \xrightarrow{a} (S'_P, B'_P, C'_P) \cup \{error\}$ where a is the label on a transition indicating the type of step being taken. The initial configuration of any module P is defined as (S^0_P, B^0_P, C^0_P) where S^0_P and B^0_P are empty maps, and C^0_P maps each element in its domain $(\mathcal{I})$ to 0.
Rules for local computation: Rules (R1)-(R2) present the rules for local computation of a machine. Rule Internal picks an interface i and instance number n and updates $S[i, n]$ according to the transition relation *Local*, leaving B and C unchanged. The map I_P is used to obtain the concrete machine corresponding to the interface i. Rule Remove-Event updates $S[i, n]$ and $B[i, n]$ according to the transition relation $(s, b, pos, s') \in Rem(I_P[i])$, the entry in pos-th position of $B[i, n]$ is removed and the local state in $S[i, n]$ is updated to s' leaving the machine identifier (id) unchanged. The transition for both these rules is labeled with ϵ to indicate that the computation is local and is an internal transition of the module P.
Rules for creating interfaces: Let $s_0 \in \mathcal{S}$ represent a state such that $ids(s_0) = \emptyset$. Let $b_0 \in \mathcal{B}$ be the empty sequence over $\mathcal{E} \times \mathcal{V}$. Rules (R3)-(R8) present the rules for interface creation. In all the rules, I_P is used to look-up the machine name corresponding to an interface bound in module P. The environment of P triggers the first two rules, and the last four are triggered by P itself. The rule Environment-Create creates an interface that is neither created nor exported by P; consequently, it updates C by incrementing the number of instances of i but leaves S and B unchanged. The rule Input-Create creates an interface i exported by P that is not created by P. The instance number of the new interface is $C[i]$; its local-store is initialized to (s_0, id) where id in this case stores the "self" identifier that references the machine itself. Note that the environment cannot create an interface that is also created by P, which is based on the key assumption of *output disjointness* required for compositional reasoning (Section 4.3). The rule Create-Bad creates a transition into *error* if the interface i' being created by machine (i, n) violates the predicate $CreateOk(m, x) = x \in MCreates(m)$. Thus, machine (i, n) may only create machines in $MCreates(I_P[i])$.

We use machine (i, n) to refer to the n-th instance of the machine $I_P[i]$. Output-Create-Outside allows machine (i, n) to create an interface i'' that is not implemented inside P, indicated by $i'' \notin dom(I_P)$. Create of interface i'' will get bound to an appropriate machine when P is composed with another module Q that has binding for i'' i.e. $i'' \in dom(I_Q)$. The predicate $CreateOk(m, x) = x \in MCreates(m)$ checks that if a machine m performs **new** x then x belongs to its creates set. Thus, machine (i, n) may only create machines in $MCreates(I_P[i])$. A well-formed module satisfies the condition (WF1) together with the property that machines cannot create identifiers out of thin air to guarantee that the set of permissions in any machine identifier is a subset of the received events of the machine referenced by that identifier.

The rule Output-Create-Inside allows the creation of an interface that is exported by P. An interesting aspect of this rule is that the machine identifier made available to the creator machine has permission $IRecvs(i'')$ but the "self" identifier of the created machine is the entire receive set which may contain some private events in addition to all events in $IRecvs(i'')$. Allowing extra private events in the permission of the "self" identifier is useful if the machine wants to send permissions to send private events to a sibling machine inside P. In all these rules, the link map (L_P) is used to look up the interface i'' to be created corresponding to **new** i'. The condition (WF2) holds for any well-formed module and guarantees that this lookup always succeeds.
Rules for sending events: Rules (R9)-(R13) present the rules for sending events. The environment of P triggers the first rule, and the last two are triggered by P itself. The rule Input-Send enqueues a pair (e, v) into machine (i, n) if $e \in MRecvs(I_P[i])$ and e is neither private in P nor sent by P and v does not contain any machine identifiers with private events in its permissions. First, an event

(Internal)(R1)

$$\frac{S_P[i, n] = (s, id) \qquad (s, id, s', id') \in Local(I_P[i])}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i, n) \mapsto (s', id')], B_P, C_P)}$$

(Remove-Event)(R2)

$$\frac{\begin{array}{c} S_P[i, n] = (s, id) \qquad B_P[i, n] = b \\ (s, b, pos, s') \in Rem(I_P[i])) \qquad b' = rem(b, pos) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i, n) \mapsto (s', id)], B_P[(i, n) \mapsto b'], C_P)}$$

(Environment-Create)(R3)

$$\frac{i \in \mathcal{I} \setminus (IC_P \cup dom(I_P)) \qquad n = C_P[i]}{(S_P, B_P, C_P) \xrightarrow{i} (S_P, B_P, C_P[i \mapsto n + 1])}$$

(Input-Create)(R4)

$$\frac{i \in dom(I_P) \setminus IC_P \qquad n = C_P[i] \qquad id = (i, n, IRecvs(i))}{(S_P, B_P, C_P) \xrightarrow{i} (S_P[(i, n) \mapsto (s_0, id)], B_P[(i, n) \mapsto b_0], C_P[i \mapsto n + 1])}$$

(Create-Bad)(R5)

$$\frac{\begin{array}{c} S_P[i, n] = (s, \_) \qquad (s, i', \_) \in New(I_P[i]) \\ \neg CreateOk(I_P[i], i') \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} error}$$

(Output-Create-Outside)(R6)

$$\frac{\begin{array}{c} S_P[i, n] = (s, \_) \qquad (s, i', s') \in New(I_P[i]) \qquad CreateOk(I_P[i], i') \\ i'' = L_P[i][i'] \qquad n' = C_P[i''] \\ i'' \notin dom(I_P) \qquad id' = (i'', n', IRecvs(i'')) \end{array}}{(S_P, B_P, C_P) \xrightarrow{i''} (S_P[(i, n) \mapsto (s', id')], B_P, C_P[i'' \mapsto n' + 1])}$$

(Output-Create-Inside)(R7)

$$\frac{\begin{array}{c} S_P[i, n] = (s, \_) \qquad (s, i', s') \in New(I_P[i]) \qquad CreateOk(I_P[i], i') \\ i'' = L_P[i][i'] \qquad i'' \in dom(I_P) \setminus IP_P \qquad n' = C_P[i''] \qquad id' = (i'', n', IRecvs(i'')) \qquad id'' = (i'', n', MRecvs(I_P[i''])) \end{array}}{(S_P, B_P, C_P) \xrightarrow{i''} (S_P[(i, n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], \quad B_P[(i'', n') \mapsto b_0], C_P[i'' \mapsto n' + 1])}$$

(Create-Private)(R8)

$$\frac{\begin{array}{c} S_P[i, n] = (s, \_) \qquad (s, i', s') \in New(I_P[i]) \\ CreateOk(I_P[i], i') \qquad i'' = L_P[i][i'] \qquad i'' \in IP_P \qquad n' = C_P[i''] \qquad id' = (i'', n', IRecvs(i'')) \qquad id'' = (i'', n', MRecvs(I_P[i''])) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i, n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], B_P[(i'', n') \mapsto b_0], C_P[i'' \mapsto n' + 1])}$$

(Input-Send)(R9)

$$\frac{\begin{array}{c} B_P[i, n] = b \qquad e \in MRecvs(I_P[i]) \setminus (EP_P \cup ES_P) \\ v \in \mathcal{V} \qquad \forall (i', n', \alpha') \in ids(v).\ \alpha' \in \mathcal{A}(e) \wedge n' < C_P[i'] \end{array}}{(S_P, B_P, C_P) \xrightarrow{((i,n),e,v)} (S_P, B_P[(i, n) \mapsto b \odot (e, v)], C_P)}$$

(Send-Bad)(R10)

$$\frac{\begin{array}{c} S_P[i, n] = (s, id_t) \qquad id_t = (\_, \_, \alpha_t) \\ (s, id_t, e, v, \_) \in Enq(I_P[i]) \qquad \neg SendOk(I_P[i], \alpha_t, e, v) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} error}$$

(Output-Send-Outside)(R11)

$$\frac{S_P[i, n] = (s, id_t) \qquad id_t = (i_t, n_t, \alpha_t) \qquad i_t \notin dom(I_P) \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v)}{(S_P, B_P, C_P) \xrightarrow{((i_t,n_t),e,v)} (S_P[(i, n) \mapsto (s', id_t)], B_P, C_P)}$$

(Output-Send-Inside)(R12)

$$\frac{\begin{array}{c} S_P[i, n] = (s, id_t) \\ id_t = (i_t, n_t, \alpha_t) \qquad i_t \in dom(I_P) \qquad e \in ES_P \qquad b_t = B_P[i_t, n_t] \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v) \end{array}}{(S_P, B_P, C_P) \xrightarrow{((i_t,n_t),e,v)} (S_P[(i, n) \mapsto (s', id_t)], B_P[(i_t, n_t) \mapsto b_t \odot (e, v)], C_P)}$$

(Send-Private)(R13)

$$\frac{\begin{array}{c} S_P[i, n] = (s, id_t) \\ id_t = (i_t, n_t, \alpha_t) \qquad i_t \in dom(I_P) \qquad e \in EP_P \qquad b_t = B_P[i_t, n_t] \qquad (s, id_t, e, v, s') \in Enq(I_P[i]) \qquad SendOk(I_P[i], \alpha_t, e, v) \end{array}}{(S_P, B_P, C_P) \xrightarrow{\epsilon} (S_P[(i, n) \mapsto (s', id_t)], B_P[(i_t, n_t) \mapsto b_t \odot (e, v)], C_P)}$$

Fig. 7. Rules for operational semantics of ModP modules

that is sent by P is not considered as an input event, which is safe since rules of *output-disjointness* (Section 4.3) forbid composing P with another module that sends an event in common with P. Second, only an event in the receives set of a machine is considered as an input event, because any machine can send only those events that are in the permission of an identifier and the permission set of an identifier is guaranteed to be a subset of the receives set of the machine referenced by it (based on (WF1)). Finally, private events or payload values with private events in its permissions are not considered as input because permission to send a private event cannot leak out of a well-formed module (based on (WF3)).

Before executing a send statement the target machine identifier is loaded into the local store represented by id_t using an internal transition. The predicate $SendOk(\hat{m}, \alpha, e, v) = e \in MSends(\hat{m}) \wedge$

$e \in \alpha \land \forall(\_, \_, \beta) \in ids(v).\ \beta \in \mathcal{A}(e)$) captures the `(SendOk)` specification described in Section 3.2. Thus, machine (i, n) may only send events declared by it in $MSends(I_P[i])$ and allowed by the permission α_t of the target machine and should not embed machine identifiers with private permissions in the payload v. Note that the dynamic check `(SendOk)` helps guarantee the well-formedness condition `(WF3)` and also ensures that a module receives only those events from other modules that are its input events (and is expected to be receptive against).

The rule Output-Send-Outside sends an event to machine outside P whereas rules Output-Send-Inside and Send-Private send an event to some machine inside P. In the former, the target machine m_t is not in the domain of I_P, whereas in the latter cases the target machine is inside the module and hence present in the domain of I_P. For Send-Private, the label on the transition is ϵ as a private event is sent. For brevity, we refer to a configuration (S^k, B^k, C^k) as G^k.

Definition 5.1 (Execution). An *execution* of P is a finite sequence $G^0 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ for some $n \in \mathbb{N}$ such that $G^i \xrightarrow{a_i} G^{i+1}$ for each $i \in [0, n)$.

Let $execs(P)$ represent the set of all possible executions of the module P. An execution is *unsafe* if $G^n \xrightarrow{\epsilon} error$; otherwise, it is *safe*. The module P is *safe*, if for all $\tau \in execs(P)$, τ is a safe execution. The signature of a module P is the set of labels corresponding to all externally visible transitions in executions of P.

Definition 5.2 (Module-Signature). The signature of a module P is the set $\Sigma_P = (\mathcal{I} \setminus IP_P) \cup ((\mathcal{I} \times \mathbb{N}) \times (ES_P \cup ER_P) \times \mathcal{V})$. The signature is partitioned into the *output signature* $(IC_P \setminus IP_P) \cup ((\mathcal{I} \times \mathbb{N}) \times ES_P \times \mathcal{V})$ and the *input signature* $(\mathcal{I} \setminus IC_P) \cup ((\mathcal{I} \times \mathbb{N}) \times (ER_P \setminus ES_P) \times \mathcal{V})$.

The transitions in an execution labeled by elements of the output signature are the **output actions** whereas transitions labeled by elements of the input signature are the **input actions**.

Definition 5.3 (Traces). Given an execution $\tau = G^0 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ of P, the trace of τ is the sequence σ obtained by removing occurrences of ϵ from the sequence $a_1, \ldots, a_{n-1}$.

Let $traces(P)$ represents the set of all possible traces of P. Our definition of a trace captures externally visible operations that add dynamism in the system like machine creation and sends with a payload that can have machine-references. If $\sigma \in traces(P)$ then $\sigma[\Sigma_P]$ represents the projection of trace σ over the set Σ_P where if $\sigma = a_0, \ldots, a_n$, then $\sigma[\Sigma_P]$ is the sequence obtained after removing all a_i such that $a_i \notin \Sigma_P$.

Definition 5.4 (Refinement). The module P *refines* the module Q, written $P \preceq Q$, if the following conditions hold: (1) $IC_Q \setminus IP_Q \subseteq IC_P \setminus IP_P$, (2) $dom(I_Q) \setminus IP_Q \subseteq (dom(I_P) \cup IC_P) \setminus IP_P$, (3) $ES_Q \subseteq ES_P$, (4) $ER_Q \subseteq ER_P \cup ES_P$ (note that (1)-(4) together imply $\Sigma_Q \subseteq \Sigma_P$), (5) and for every trace σ of P the projection $\sigma[\Sigma_Q]$ is a trace of Q.

5.2 Assume-Guarantee Reasoning

The two fundamental compositionality results required for assume-guarantee reasoning are:

Theorem 5.1 (**Composition Is Intersection**). *Let P, Q and $P||Q$ be well-formed modules. For any $\pi \in \Sigma^*_{P||Q}$, $\pi \in traces(P||Q)$ iff $\pi[\Sigma_P] \in traces(P)$ and $\pi[\Sigma_Q] \in traces(Q)$.* ***(Proof in [Desai et al. 2018])***

Theorem 5.1 states that composition of modules behaves like language intersection, the traces of the component modules completely determine traces of a composed module.

Theorem 5.2 (**Composition Preserves Refinement**). *Let P, Q, and R be well-formed modules such that $P||Q$ and $P||R$ are well-formed. Then $R \preceq Q$ implies that $P||R \preceq P||Q$.* ***(Proof in [Desai et al. 2018])***

Theorem 5.2 states that parallel composition is monotonic with respect to trace inclusion i.e. if one module is replaced by another whose traces are a subset of the former, then the set of traces of the resultant composite module can only be reduced.

Theorems 5.1 and 5.2 form the basis of our theory of compositional refinement and are used for proving the principles of circular assume-guarantee reasoning underlying our compositional testing methodology (Theorems 5.3-5.4). We introduce a generalized composition operation $\| \mathcal{P}$, where $\mathcal{P}$ is a non-empty set of modules. This operator represents the composition of all modules in $\mathcal{P}$. The binary parallel composition operator is both commutative and associative. Thus, $\| \mathcal{P}$ is a module obtained by composing modules in $\mathcal{P}$ in some arbitrary order. Let $\mathcal{P}$ and $\mathcal{Q}$ be set of modules. We say that $\mathcal{P}$ is a subset of $\mathcal{Q}$ if $\mathcal{P}$ can be obtained by dropping modules in $\mathcal{Q}$.

Theorem 5.3 (**Compositional Safety**). *Let $\| \mathcal{P}$ and $\| \mathcal{Q}$ be well-formed. Let $\| \mathcal{P}$ refine each module $Q \in \mathcal{Q}$. Suppose for each $P \in \mathcal{P}$, there is a subset $\mathcal{X}$ of $\mathcal{P} \cup \mathcal{Q}$ such that $P \in \mathcal{X}$, $\| \mathcal{X}$ is well-formed, and $\| \mathcal{X}$ is safe. Then $\| \mathcal{P}$ is safe.* ***(Proof in [Desai et al. 2018])***

When using Theorem 5.3 in practice, modules in $\mathcal{P}$ and $\mathcal{Q}$ typically consists of the implementation and abstraction modules respectively. When proving the safety of any module $P \in \mathcal{P}$, it is allowed to pick any modules in $\mathcal{Q}$ for constraining the environment of P. To use Theorem 5.3, we need to show that $\| \mathcal{P}$ refines each module $Q \in \mathcal{Q}$ which requires reasoning about all modules in $\mathcal{P}$ together. The following theorem shows that the refinement between $\| \mathcal{P}$ and $\mathcal{Q}$ can also be checked compositionally.

Theorem 5.4 (**Circular Assume-Guarantee**). *Let $\| \mathcal{P}$ and $\| \mathcal{Q}$ be well-formed. Suppose for each module $Q \in \mathcal{Q}$ there is a subset $\mathcal{X}$ of $\mathcal{P} \cup \mathcal{Q}$ such that $Q \notin \mathcal{X}$, $\| \mathcal{X}$ is well-formed, and $\| \mathcal{X}$ refines Q. Then $\| \mathcal{P}$ refines each module $Q \in \mathcal{Q}$.* ***(Proof in [Desai et al. 2018])***

Theorem 5.4 states that to show that $\| \mathcal{P}$ refines $Q \in \mathcal{Q}$, any subset of modules in $\mathcal{P}$ and $\mathcal{Q}$ can be picked as long as Q is not picked. Therefore, it is possible to perform sound *circular* reasoning, i.e., use Q_1 to prove refinement of Q_2 and Q_2 to prove refinement of Q_1. This capability of circular reasoning is essential for compositional testing of the distributed systems we have implemented.

Note that $\| \mathcal{P}$ refines every submodule of $\mathcal{Q}$ is implied by $\| \mathcal{P}$ refines module $\| \mathcal{Q}$. If $\| \mathcal{P}$ refines $\| \mathcal{Q}$, then using Theorem 5.1, $\| \mathcal{P}$ would refine each individual submodule in $\mathcal{Q}$ as well. Similarly, if $\| \mathcal{P}$ refines every submodule of $\mathcal{Q}$ and $\| \mathcal{Q}$ is a well-formed module, then $\| \mathcal{P}$ refines module $\| \mathcal{Q}$.

6 FROM THEORY TO PRACTICE

Theorems 5.3 and 5.4 indicate that there are two kinds of obligations that result from assume-guarantee reasoning—*safety* and *refinement*. Although these obligations can be verified using proof techniques, the focus of ModP is to use systematic testing to falsify them. ModP allows the programmer to write each obligation as a test declaration. The declaration `test tname: P` introduces a safety test obligation that the executions of module P do not result in a failure (module `P` is safe). The declaration `test tname: P refines Q` introduces a test obligation that module `P` refines module `Q`. These test obligations are automatically discharged using ModP's systematic testing engine (Section 7).

Fault-tolerant services software stack. We illustrate using the protocol stack in Figure 1, how we used ModP to test a complex distributed system compositionally. We implemented two distributed services: (i) distributed atomic commit of updates to partitioned data using two-phase commit [Bernstein et al. 1986; Gray and Lamport 2006], and (ii) distributed data structures: hash-table and list. These distributed services use State Machine Replication (SMR) for fault-tolerance [Schneider 1990].

We implement distributed transaction commit using the two-phase commit protocol, which uses a single *coordinator* state machine to atomically commit updates across multiple *participant* state machines. Hashtable and list are implemented as deterministic state machines with PUT and GET operations. These services by themselves are not tolerant to node failures. We use SMR to make the two-phase commit and the data structures fault-tolerant by replicating the deterministic coordinator, participant, and hash-table (list) state-machines across multiple nodes. We implemented Multi-Paxos [Lamport 1998] and Chain-Replication [van Renesse and Schneider 2004] based SMR, these protocols guarantee that a consistent sequence of events is fed to the deterministic (replicated) state machines running on multiple nodes. These events could be operations on a data-structure or operations for two-phase-commit. Multi-Paxos and Chain-Replication, in turn, use different sub-protocols. Though both these protocols provide linearizability guarantees their implementations are very different with distinct fault models and hence acts as an excellent case study for module (protocol) substitution. The protocols in the software stack use various OS services like timers, network channels, and storage services which are not implemented in ModP. We provide *over approximating models* for these libraries in ModP which are used during testing but replaced with the library, and OS calls for real execution. We implemented the safety specifications (as spec. machines) of all the protocols as described in their respective paper. The table below shows examples of specifications checked for some of the distributed protocols.

Protocol	**Specifications**
2PC	Transactions are atomic [Gray 1978] (2PCSpec)
Chain Repl.	All invariants in [van Renesse and Schneider 2004], cmd-log consistency (CRSpec)
Multi-Paxos	Consensus requirements [Lamport 2001], log consistency [Van Renesse and Altinbuken 2015] (MPSpec)

Fig. 8. Specifications checked for each protocol

Compositionally testing transaction-commit service. Figure 9 presents a simplified version of the test-script used for compositionally testing the transaction-commit service.

```
 1 //monolithic testing of software stack
 2 test mono: (assert 2PCSpec in 2PC) || MultiPaxosSMR || OSServAbs;
 3
 4 //Decomposition using compositional safety
 5 test t1: (assert 2PCSpec in 2PC) || SMRLinearizAbs || OSServAbs;
 6 test t2: SMRClientAbs || MultiPaxosSMR || OSServAbs;
 7 test t3: SMRClientAbs || MultiPaxosSMR || OSServAbs
 8          refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
 9 test t4: 2PC || SMRLinearizAbs || OSServAbs
10          refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
11 //Multi Paxos linearizability as specification machine
12 test t5: SMRClientAbs || assert MPSec in MultiPaxosSMR || OSServAbs;
13
14 //test chain replication SMR
15 test t6: SMRClientAbs || ChainRepSMR || OSServAbs
16 test t7: SMRClientAbs || ChainRepSMR || OSServAbs
17          refines SMRClientAbs || SMRLinearizAbs || OSServAbs;
18 //Chain replication linearizability as specification machine
19 test t8: SMRClientAbs || assert CRSpec in ChainRepSMR || OSServAbs;
20
21 //test 7
22 module LHS = ChainRepSMR || SMRClientAbs || TestDriver || OSServAbs;
23 module RHS =
24     // hide replicated machine creation operation
25     (hide SMRReplicatedMachineInterface in
26     //hide events used for interaction with replicated machine
27     (hide eSMRReplicatedMachineOperation, eSMRReplicatedLeader in
28       SMRClientAbs || TestDriver || SMRReplicated || OSServAbs));
29 test t7:  LHS refines RHS;
```

Fig. 9. Compositional Testing of Transaction Commit Service

The modules 2PC, MultiPaxosSMR, ChainRepSMR represent the implementations of the two-phase commit, Multi-Paxos based SMR, and Chain-Replication based SMR protocols respectively. The module SMRLinearizAbs represent the linearizability abstraction of the SMR service, both Multi-Paxos based SMR and Chain-Replication based SMR provide this abstraction. The module SMRClientAbs represent the abstraction of any client of the SMR service. OSServAbs implements the models for mocking OS services like timers, network channels, and storage. A failure injector machine that randomly halts machines in the program is also added as part of the OSServAbs. There are two sets of implementation modules $\mathcal{P}_m$ ={2PC, MultiPaxosSMR, OSServAbs} or $\mathcal{P}_c$ ={2PC, ChainRepSMR, OSServAbs} representing the Multi-Paxos and Chain-Replication based versions. The set of abstraction modules is Q ={SMRClientAbs, SMRLinearizAbs, OSServAbs}. The test obligation mono represents the monolithic testing problem for transaction-commit service.

Similar to *property-based testing* [Arts et al. 2008], the programmer can attach specifications to modules under test using the **assert** constructor (e.g., Figure 9-line 5). Using Theorem 5.3, we can decompose the monolithic problem into safety tests `t1` and `t2` under the assumption that each module in $\mathcal{P}_m$ refines each module in $\mathcal{Q}$. This assumption is then validated using the Theorem 5.4 and tests `t3`, `t4`. The power of compositional reasoning is substitutability; if the programmer wants to migrate the transaction commit service from using Multi-Paxos to use Chain-Replication then he just needs to validate `ChainRepSMR` in isolation using tests `t6` and `t7`. The tests `t5` and `t8` are substitutes for the refinement checks `t4` and`t7` since the spec. machines (from the table) assert the linearizability abstraction of these protocols.

The test declarations used in practice are a bit more involved than Figure 9. There are two main points: (1) For each test declaration, the programmer provides a finite test harness module comprising non-deterministic machines that close the module under test by either supplying inputs or injecting failures. The programmer may provide a collection of test harnesses modules for each test declaration to cover various testing scenarios for each test obligation. (2) In some cases, the module constructors like hide and rename have to be used to make modules composable or create the right projection relation. Figure 9 (line 22-29) represent the test-script we used to perform test `t7`. We had to hide internal events sent to the replicated machine to create the right projection relation for refinement.

7 ModP TOOL CHAIN

In this section, we describe the implementation of the ModP toolchain (Figure 10). The ModP toolchain is available as part of the P programming framework [P-GitHub 2018].

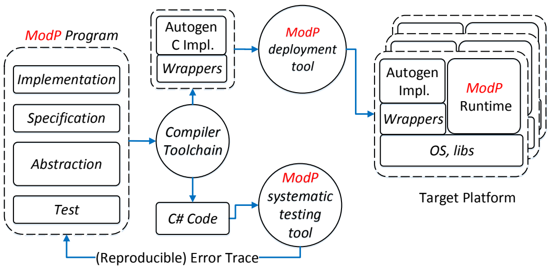

Fig. 10. ModP Programming Framework

Compiler. A ModP program comprises four blocks — implementation modules, specifications monitors, abstraction modules and tests. The compiler static-analysis of the source code not only performs the usual type-correctness checks on the code of machines but also checks that constructed modules are well-formed, and test declarations are legal. The compiler generates code for each test declaration; this generated code makes all sources of nondeterminism explicit and controllable by the systematic testing engine, which generates executions in the test program checking each execution against implicit and explicit specifications. *For each test declaration, the compiler generates a standalone program that can be independently analyzed by the back-end systematic testing engine.* The compiler also generates C code which is compiled and linked against the ModP runtime to generate application executables.

Systematic testing engine. The ModP systematic testing engine efficiently enumerates executions resulting from scheduling and explicit nondeterministic choices. The ModP compiler generates a standalone program for each safety test declaration. We reuse the existing P testing backends for safety test declarations with modifications to take into account the extensions to P state machines. There are two backends provided by P: **(1)** a sampling-based testing engine that explicitly sample executions using delay-bounding based prioritization [Desai et al. 2015], and **(2)** a symbolic execution engine with efficient state-merging using MultiSE [Sen et al. 2015; Yang et al. 2017].

We extended the sampling based testing engine to perform refinement testing of ModP programs based on trace containment. Our algorithm for checking $P \preceq Q$ consists of two phases: **(1)** In the first phase, the testing engine generates all possible visible traces of the abstraction module Q and compactly caches them in memory. The abstraction modules are generally small, and hence, all the traces of Q can be loaded in memory for all our experiments. **(2)** In the second phase, the testing engine performs stratified sampling of the executions in P, and for each terminating execution checks if the visible trace is contained in the cache (traces of Q). A safety bug is reported as a sequence of visible actions that lead to an error state. In the case of refinement checking, the tool returns a visible trace in implementation that is not contained in the abstraction.

Distributed runtime. Figure 11 shows the structure of a ModP application executing on distributed nodes. We believe that the *multi-container* runtime is a generic architecture for executing programs with distributed state-machines. Each node hosts a collection of *Container* processes. *Container* is a way of grouping collection of ModP state machines that interact closely with each other and must reside in a common fault domain. Each Container process hosts a *listener*, whose job is to forward events received from other containers to the state machines within the container. State machines within a container are executed concurrently using a thread pool and as an optimization interacts without serializing/deserializing the messages.

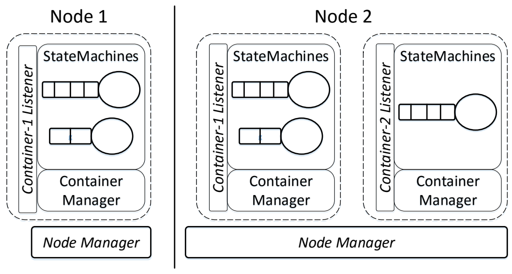

Fig. 11. Structure of ModP application

Each node runs a *NodeManager* process which listens for requests to create new Container processes. Similarly, each Container hosts a single *ContainerManager* that services requests for creations of new state machines within the container. In the typical case, each node has one NodeManager process and one Container process executing on it, but ModP also supports a collection of Containers per node enabling emulation of large-scale services running on only a handful of nodes. A ModP state machine can create a new container by invoking runtime's `CreateContainer` function. A state machine can create a *new* local or remote state machine by specifying the hosting container's ID. Hence, the ModP runtime enables the programmer to distribute state-machines across distributed nodes and also group them within containers for optimizing the performance.

In summary, the runtime executes the generated C representation of the ModP program and has the capability to (1) create, destroy, and execute distributed state machines, (2) efficiently communicate among state machines that can be distributed across physical nodes, (3) serialize data values before sends and deserialize them after receives.

8 EVALUATION

We empirically evaluate ModP framework by compositionally implementing and testing the fault-tolerant distributed services software stack (Figure 1). The goal of our evaluation is twofold: **(1)** Demonstrate that the theory of compositional refinement helps scale systematic testing to complex large distributed systems. We show that compositional testing leads to test-amplification in terms of both: increasing the test-coverage and finding more bugs (faster) than the monolithic testing approach (Section 8.2). We present anecdotal evidence of the benefits of refinement testing. It helps find bugs that would have been missed otherwise when performing abstraction-based compositional testing. **(2)** Demonstrate that the performance of the (rigorously tested) distributed services built using ModP is comparable to the corresponding open-source baseline. We evaluate the performance of the hash-table distributed service by benchmarking it on Azure cluster (Section 8.3).

8.1 Programmer Effort

The Table below shows a five-part breakdown, in source lines of ModP code, of our implementation of the distributed service. The Impl. column represents the detailed implementation of each module whose – generated C code can be deployed on the target platform. Specs. column represents

Protocol	Impl.	Specs.	Abst.	Test Driver	Test Decls
2 Phase Commit	441	61	41	35	128
Chain Rep. SMR	1267	220	173	130	105
Multi-Paxos SMR	1617	101	121	92	90
Data structures	276	25	-	89	25
Total	**3601**	**Others = 1436**			

Fig. 12. Source lines of ModP code

the component-level temporal properties (monitors). Abst. column represents abstractions of the modules used when testing other modules. The Driver column represents the different finite test-harnesses written for testing each protocol in isolation. The last column represents the test declarations across protocols to compositionally validate the "whole-system" level properties as described in Section 5.2.

8.2 Compositional Testing

The goal of our evaluation is to demonstrate the benefits of using the theory of compositional refinement in testing distributed systems, and hence, we use the same backend engine (Section 7) for testing both the monolithic test declaration and the corresponding compositional test declarations. We use the existing systematic testing engine of P that supports state-of-the-art search prioritization [Desai et al. 2015] and other efficient bug-finding techniques for analyzing the test declarations. Note that the approach used for analyzing the test declarations is orthogonal to the benefits of using compositional testing.

Compositional reasoning led to the state-space reduction and hence amplification of the test-coverage, uncovering 20 critical bugs in our implementation of the software stack. To highlight the benefits of using ModP-based compositional reasoning, we present two results in the context of our case-study: (1) abstractions help amplify the test-coverage for both the testing backends, the prioritized execution sampling and symbolic execution (Section 7), and (2) this test-coverage amplification results in finding bugs faster than the monolithic approach. For monolithic testing, we test the module constructed by composing the implementation modules of all the components.

Test-amplification via abstractions. Using abstractions simplifies the testing problem by reducing the state-space. The reduction is obtained because a large number of executions in the implementations can be represented by an exponentially small number of abstraction traces.

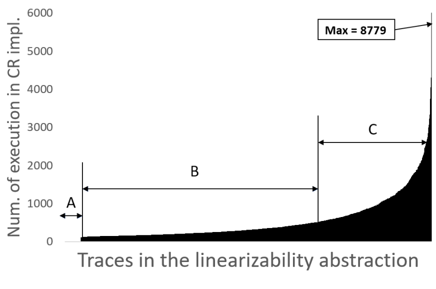

Fig. 13. Test-Amplification via Abstractions: Chain-Replication Protocol

To show the kind of amplification obtained for the sampling based testing approach, we conducted an experiment to count the number of unique executions in the implementation of a protocol that maps to a trace in its abstraction. Figure 13 present the graph for the Chain-Replication (CR) protocol with a finite test-harness that randomly pumps in 5 update operations. The `x-axis` represents the traces in the abstraction sorted by `y-axis` values, where the `y-axis` represents the number of executions in the implementation that maps (projects) to the trace in abstraction. The linearizability abstraction (guaranteed by Chain-Replication protocol) has 1931 traces for the finite test-harness, and there were exponentially many executions in the CR implementation. We sampled 10^6 unique executions in the CR implementation for this experiment.

The graph in Figure 13 is highly skewed and can be divided into three regions of interest: region (A) correspond to those traces in the abstraction to which no execution mapped from the samples set of 10^6 implementation executions which could be either because these traces correspond to a very low probability execution in implementation or are false positives; region (B) represent those traces that correspond to low probability executions in the implementation; region (C) represent those executions that may lead to a lot of redundant explorations during monolithic testing. Using linearizability abstraction helps in mitigating this skewness and hence increases the probability of exploring low probability behaviors in the system leading to amplification of test-coverage (as in some cases exploring one execution in the abstraction is equivalent to exploring approx. 8779 executions in the implementation).

Protocol	Schedules Explored	
	Monolithic	CST
MPaxos (bug1)	13	**11**
2PC (bug2)	1944	**19**
ChainR (bug3)	2018	**13**
MPaxos (bug4)	NF	**91**
T2PC (bug5)	NF	**112**
ChainR (bug6)	NF	**187**
ChainR (bug7)	NF	**782**
MPaxos (bug8)	NF	**2176**

Fig. 14. CST vs. Monolithic Testing. (NF: Bug not found)

Next, we show that the compositional testing approach helps the sampling based back-end to find bugs faster. We randomly chose 8 bugs (out of 20) that we found in different protocols during the development process. We compared the performance of compositional testing (CST) against the monolithic testing approach where the entire protocol stack is composed together and considered as a single monolithic system. We use the number of schedules explored before finding the bug as the comparison metric. Figure 14 shows that ModP-based compositional approach helps the sampling based back-end find bugs faster than the monolithic approach and in most cases, the monolithic approach fails to find the bug even after exploring 10^6 different schedules.

P also supports a symbolic execution back-end that uses the MultiSE [Sen et al. 2015; Yang et al. 2017] based approach for state-merging. To evaluate the test amplification obtained for the symbolic execution back-end, we compared the performance of the testing engine for the monolithic testing problem and its decompositions from Figure 9. We performed the test `mono` using the symbolic engine for a finite test-harness where the `2PC` performs 5 transactions. The symbolic engine could not explore all possible execution of the problem even after 10 hrs. We performed the tests `t1`, `t2`, `t5`, `t8` (for the same finite test-harness) and the symbolic engine was able to explore all possible executions for each decomposed test in 1.3 hours (total). The upshot of our module system is that we can get complete test-coverage (guaranteeing absence of bugs) for a finite test-harness which was not possible when doing monolithic testing.

We describe a few of these bugs in detail to illustrate the diversity of bugs found in practice.

1. ChainR (bug7) represents a consistency bug that violates the update propagation invariant in [van Renesse and Schneider 2004]. The bug was in the chain repair logic and can be reproduced only when an intermediate node in the chain that has uncommitted operations, first becomes a tail node because of tail failure and then a head node on the head failure. This specific scenario could not be uncovered using monolithic testing but is triggered when testing the Chain-Replication protocol in isolation because of the state-space reduction obtained using abstractions.
2. MPaxos (bug4) represents a bug in our acceptor logic implementation that violates the P2c invariant in [Lamport 2001]. For this bug to manifest, it requires multiple leaders (proposers) in the Multi-Paxos system to make a decision based on an incorrect promise from the acceptor. In a monolithic system, because of the explosion of non-deterministic choices possible the probability of triggering a failure that leads to choosing multiple leaders is extremely low. When compositionally testing Multi-Paxos, we compose it with a coarse-grained abstraction of the leader election protocol. The abstraction non-deterministically chooses any Multi-Paxos node as a leader and hence, increasing the probability of triggering a behavior with multiple leaders.
3. Meaningful testing requires that the abstractions used during compositional reasoning are sound abstractions of the components being replaced. We were able to uncover scenarios where bugs could have been missed during testing because of an unsound abstraction. The linearizability abstraction was used when testing the distributed services built on top of SMR. Our implementation of the abstraction guaranteed that for every request there is a single response. For Chain-Replication protocol (as described in [van Renesse and Schneider 2004]), in a rare scenario when the tail node of the system fails and after the system has recovered, there is a possibility that a request may be responded multiple times. Our refinement checker was able

to find this unsound assumption in the linearizability abstraction which led to modifying our Chain-Replication implementation. This bug could have caused an error in the client of the Chain-Replication protocol as it was tested against the unsound linearizability abstraction.

During compositional systematic testing, abstractions are used for decomposition. False positives can occur if the abstractions used are too coarse-grained and contain behaviors not present in the implementation. The number of false positives uncovered during compositional testing was low (4) compared to the real bugs that we found. We think that this could be because the protocols that we considered in this paper have well-studied and known abstractions.

8.3 Performance Evaluation

We would like to answer the question: Can the distributed applications build modularly using ModP with the aim of scalable compositional testing rival the performance of corresponding state-of-the-art implementations? We compare the performance of the code generated by ModP for the fault-tolerant hash-table built using Multi-Paxos against the hash-table built using the popular open-source reference implementation of Multi-Paxos from the EPaxos codebase [Moraru et al. 2013a,b]. All benchmarking experiments for the distributed services were run on A3 Virtual Machine (with 4-core Intel Xeon E5-2660 2.20GHz Processor, 7GB RAM) instances on Azure.

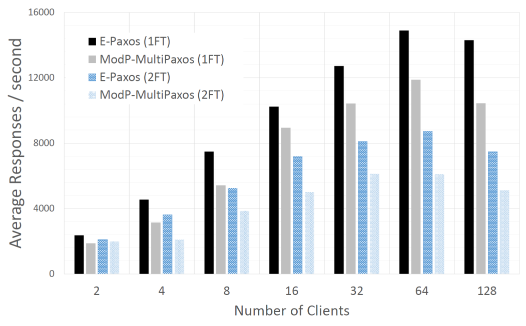

Fig. 15. Performance of ModP HashTable using Multi-Paxos (MP) is comparable with an open source baseline implementation (mean over 60s close-loop client runs).

To measure the update throughput (when there are no node failures in the system), we use clients that pump in requests in a closed loop; on getting a response for an outstanding request, the client goes right back to sending the next request. We scale the workload by changing the number of parallel clients from 2 to 128. For the experiments, each replica executes on a separate VM. Figure 15 summarizes our result for one fault-tolerant (1FT = 3 paxos nodes) and two fault-tolerant (2FT = 5 paxos nodes) hash-tables. We find the systematically tested ModP implementation achieves between 72%(2FT, 64 clients) to 80% (1FT, 64 clients) of peak throughput of the open source baseline (EPaxos codebase [Moraru et al. 2013a,b]). The open source implementation of the E-Paxos protocol suite is highly optimized and implemented in Go language (1169 LOC). We believe that the current performance gap between the two implementations can be further reduced by engineering our distributed runtime. The high-level points we would like to convey from these performance number is that it is possible to build distributed services using ModP that are rigorously tested and have comparable performance to the open source counterpart.

9 RELATED WORK

Assume-Guarantee reasoning has been implemented in model checkers [Alur et al. 1998; McMillan 1992, 2017] and successfully used for hardware verification [Eiríksson 2000; Henzinger et al. 1999; McMillan 2000] and software testing [Blundell et al. 2006]. However, the present paper is the first to apply it to distributed systems of considerable complexity and dynamic behavior. We next situate ModP with related techniques for modeling and analysis of distributed systems.

Formalisms and programming models. We categorize the formalisms for the modeling and compositional analysis of dynamic systems into three foundational approaches: process algebras, reactive modules [Alur and Henzinger 1999], and I/O automata [Lynch and Tuttle 1987].
(1) Process algebra. In the process algebra approach deriving from Hoare's CSP [Hoare 1978] and Milner's CCS [Milner 1982], the π-calculus [Milner et al. 1992; Pierce and Turner 2000] has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication. The popular approach to reasoning about behavior in these formalisms is the notions of equivalence and congruence: weak and strong bisimulation, which involves examining the state transition structure of the two systems. There's also extensive literature on observational equivalence in π-calculus based on trace inclusion [Cortier and Delaune 2009]. Extensions of π-calculus such as asynchronous π-calculus, distributed join calculus [Fournet and Gonthier 1996; Fournet et al. 1996], Dπ-calculus [Riely and Hennessy 1998] deal with distributed systems challenges like asynchrony and failures respectively. ModP chooses *Actors* [Agha 1986] as its model of computation, and our theory of compositional refinement uses trace inclusion based only on the externally visible behavior as it dramatically simplifies our *refinement testing* framework. In ModP, abstractions (modules) are state machines capable of expressing arbitrary trace properties. More recent work like session types [Castagna et al. 2009; Dezani-Ciancaglini and De'Liguoro 2009; Honda et al. 2016] and behavioral-types [Ancona et al. 2016] that have their roots in process calculi can encode abstractions in the type language (e.g., [Brady 2016]).
(2) Reactive modules. Reactive modules [Alur and Henzinger 1999] is a modeling language for concurrent systems. Modules communicate via single-writer multiple-reader shared variables and a global clock drives each module in lockstep. Dynamic Reactive Modules [Fisher et al. 2011] (DRM) is an extension of Reactive Modules with support for the dynamic creation of modules and dynamic topology. Dynamic discrete systems [Fisher et al. 2011] gives the semantics of dynamic reactive modules to model the creation of module instances and the refinement relation between dynamic reactive modules is defined using a specialized notion of transition system refinement. DRM does not formalize a compositionality theorem for the hide operation. Also, our module system is novel compared to DRM because of the fundamental differences in the supported programming model.
(3) I/O automata. Dynamic I/O automata (DIOA) [Attie and Lynch 2001] is a compositional model of dynamic systems, based on I/O automata [Lynch and Tuttle 1987]. DIOA is primarily a (set-theoretic) mathematical model, rather than a programming language or calculus. Our notion of parallel composition, trace monotonicity, and trace inclusion based on externally visible actions is inspired from DIOA and is formalized for the compositional reasoning of actor programs. ModP incorporates these ideas into a practical programming framework for building distributed systems.

Verification of distributed systems. There has been a lot of work towards reasoning about concurrent systems using program logics deriving from Hoare logic [Floyd 1993; Hoare 1969] – which includes rely-guarantee reasoning [Gavran et al. 2015; Vafeiadis and Parkinson 2007; Xu et al. 1997] and concurrent separation logic [Feng et al. 2007; Leino and Müller 2009; O'Hearn 2007]. Actor services [Summers and Müller 2016] propose program logic for modular proofs of *actor* programs. DISEL [Sergey et al. 2018] provides a language to implement and verify distributed systems compositionally. The goal of these techniques is similar to ours, enable compositional reasoning; they decompose reasoning along the syntactic structure of the program and emphasize modularity principles that allow proofs to be easily constructed, maintained and reused. They require fine-grained specifications at the level of event-handler, in our case programmer writes specifications for components as abstractions. The focus on compositional testing instead of proof allows us to attach an abstraction to an entire protocol rather than individual actions within that protocol (e.g., Send-hooks in DISEL), thereby reducing the annotations required for validation. The goal of this paper is to scale automated testing to large distributed services and to achieve this goal we develop a theory of assume-guarantee reasoning for actor programs.

Many recent efforts like IronFleet [Hawblitzel et al. 2015], Verdi [Wilcox et al. 2015], and Ivy [Padon et al. 2016] have produced impressive proofs of correctness for the distributed system, but the techniques in these efforts do not naturally allow for horizontal composition. McMillan [McMillan 2016] extended Ivy with a specification idiom based on reference objects and circular assume-guarantee reasoning to perform modular verification of a cache-coherence protocol.

Systematic testing of distributed systems. Researchers have built testing tools [Lauterburg et al. 2009; Sen and Agha 2006] for automated unit testing of Java actor programs. Mace [Killian et al. 2007a], TeaPot [Chandra et al. 1999] and P [Desai et al. 2013] provide language support for implementation, specification and systematic testing of asynchronous systems. MaceMC [Killian et al. 2007b] and MoDist [Yang et al. 2009] operate directly on the implementation of a distributed system and explore the space of executions to detect bugs in distributed systems. DistAlgo [Liu et al. 2012] supports asynchronous communication model, similar to ours, and allows extraction of efficient distributed systems implementation from the high-level specification. None of these programming frameworks tackle the challenges of compositional testing addressed in this paper. The conclusion of most of the researchers who developed these systems is similar to ours: monolithic testing of distributed systems does not scale [Guo et al. 2011].

McCaffrey's article [McCaffrey 2016] provides an excellent summary of the approaches used in the industry for systematic testing of distributed systems. *Manual-targeted testing* is an effective technique where an expert programmer provides manually crafted test-cases for finding critical bugs. However, it requires considerable expertise and manual effort. ModP's focus is on scaling automated testing and hence do not consider manual-target testing as a baseline for comparison. *Property-based testing* is another popular approach in industry for the semi-automatic testing of distributed systems (e.g., QuickCheck) [Arts et al. 2008; Hughes et al. 2016]). ModP's compositional testing approach, as well as the monolithic testing method we compare it to, can both be viewed as property-based testing since they assert the safety properties specified as monitors given a non-deterministic test harness. The compositional testing methodology described in this paper is orthogonal to the technique used for analyzing the test declarations; other approaches such as manual-targeted or property-based testing can also be used for discharging the test declarations.

10 CONCLUSION

ModP is a new programming framework that makes it easier to build, specify, and compositionally test asynchronous systems. It introduces a module system based on the theory of compositional trace refinement for the actor model of computation. We use ModP to implement and validate a practical distributed systems protocol stack. ModP is effective in finding bugs quickly during development and get orders of magnitude more test-coverage than monolithic approach. The distributed services built using ModP achieve performance comparable to state-of-the-art open source equivalents.

ACKNOWLEDGMENTS

We sincerely thank Rajeev Alur, Daniel Fremont, Jay Lorch, Nikhil Swamy, and Damien Zufferey for their comments on earlier drafts of this paper. We thank the anonymous reviewers for the careful reading and constructive suggestions that helped improve the presentation. The work of the first and last authors was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the DARPA BRASS and Assured Autonomy programs, and by Toyota under the iCyPhy center.

REFERENCES

Martín Abadi and Leslie Lamport. 1995. Conjoining Specifications. *ACM Trans. Program. Lang. Syst.* (1995).

Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

Akka. 2017. Akka Programming Language. http://akka.io/. (2017).

Rajeev Alur and Thomas A. Henzinger. 1999. Reactive Modules. *Formal Methods in System Design* 15, 1 (1999), 7–48.

Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. 1998. MOCHA: Modularity in Model Checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings.* 521–525.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J Gay, Nils Gesbert, Elena Giachino, Raymond Hu, et al. 2016. Behavioral types in programming languages. *Foundations and Trends® in Programming Languages* 3, 2-3 (2016), 95–230.

Joe Armstrong. 2007. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf.

Thomas Arts, Laura M Castro, and John Hughes. 2008. Testing erlang data types with quviq quickcheck. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG.* ACM, 1–8.

PaulC. Attie and NancyA. Lynch. 2001. Dynamic Input/Output Automata: A Formal Model for Dynamic Systems. In *CONCUR 2001*, KimG. Larsen and Mogens Nielsen (Eds.). Springer Berlin Heidelberg.

Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Colin Blundell, Dimitra Giannakopoulou, and Corina S. Păsăreanu. 2006. Assume-guarantee Testing. *SIGSOFT Softw. Eng. Notes* (2006).

Edwin Brady. 2016. State Machines All The Way Down An Architecture for Dependently Typed Applications. (2016).

Sergey Bykov, Alan Geller, Gabriel Kliot, James Larus, Ravi Pandya, and Jorgen Thelin. 2010. *Orleans: A Framework for Cloud Computing.* Technical Report.

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of session types. In *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming.* ACM, 219–230.

S. Chandra, B. Richards, and J. R. Larus. 1999. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering* (1999).

Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07).* ACM, New York, NY, USA, 398–407.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP).*

Véronique Cortier and Stéphanie Delaune. 2009. A method for proving observational equivalence. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE.* IEEE, 266–276.

Ankush Desai, Tommaso Dreossi, and Sanjit A. Seshia. 2017a. *Combining Model Checking and Runtime Verification for Safe Robotics.*

Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI.*

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. *Compositional Programming and Testing of Dynamic Distributed Systems.* Technical Report UCB/EECS-2018-95. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-95.html

Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. 2015. Systematic Testing of Asynchronous Reactive Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* ACM, New York, NY, USA.

Ankush Desai, Indranil Saha, Jianqiao Yang, Shaz Qadeer, and Sanjit A. Seshia. 2017b. DRONA: A Framework for Safe Distributed Mobile Robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems (ICCPS '17).* ACM, New York, NY, USA.

Mariangiola Dezani-Ciancaglini and Ugo De'Liguoro. 2009. Sessions and session types: An overview. In *International Workshop on Web Services and Formal Methods.* Springer, 1–28.

Ásgeir Th. Eiríksson. 2000. The Formal Design of 1M-gate ASICs. *Form. Methods Syst. Des.* (2000).

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *European Symposium on Programming.* Springer, 173–188.

Jasmin Fisher, Thomas A. Henzinger, Dejan Nickovic, Nir Piterman, Anmol V. Singh, and Moshe Y. Vardi. 2011. *Dynamic Reactive Modules.*

Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification.* Springer, 65–81.

Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. 1996. A Calculus of Mobile Agents. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR '96).*

Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. 2015. Rely/guarantee reasoning for asynchronous programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. London, UK, UK, 393–481.

Jim Gray and Leslie Lamport. 2006. Consensus on Transaction Commit. *ACM Trans. Database Syst.* 31, 1 (March 2006), 133–160.

Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011.* 265–278.

David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* (1987).

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles.*

Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-end Security via Automated Full-system Verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI).*

Matthew Hennessy and James Riely. 2002. Resource Access Control in Systems of Mobile Agents. *Inf. Comput.* 173, 1 (Feb. 2002), 82–120. https://doi.org/10.1006/inco.2001.3089

Thomas A. Henzinger, Xiaojun Liu, Shaz Qadeer, and Sriram K. Rajamani. 1999. Formal Specification and Verification of a Dataflow Processor Array. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design.*

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* (1969).

C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* (1978).

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (March 2016).

John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on.* IEEE, 135–145.

Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007a. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007.* 179–188.

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007b. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Symposium on Networked Systems Design and Implementation.*

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP).*

Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (Dec. 2001).

S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. 2009. A Framework for State-Space Exploration of Java-Based Actor Programs. In *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on.*

K Rustan M Leino and Peter Müller. 2009. A basis for verifying multi-threaded programs. In *European Symposium on Programming*. Springer, 378–393.

Yanhong A Liu, Scott D Stoller, Bo Lin, and Michael Gorbovitski. 2012. From clarity to efficiency for distributed algorithms. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 395–410.

Nancy A. Lynch and Mark R. Tuttle. 1987. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC).*

Caitie McCaffrey. 2016. The Verification of a Distributed System. *Commun. ACM* 59, 2 (Jan. 2016).

Kenneth McMillan. 2016. Modular specification and verification of a cache-coherent interface. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design.* FMCAD Inc, 109–116.

Kenneth Lauchlin McMillan. 1992. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Ph.D. Dissertation. Pittsburgh, PA, USA.

Kenneth L. McMillan. 2000. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.* 37, 1-3 (2000), 279–309.

Kenneth Lauchlin McMillan. 2017. SMV Model Checker. http://www.kenmcmil.com/smv.html. (2017).

R. Milner. 1982. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (Sept. 1992).

Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013a. EPaxos Code. https://github.com/efficient/epaxos/. (2013).

Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013b. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP).*

Peter W O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theoretical computer science* 375, 1-3 (2007), 271–307.

P-GitHub. 2018. The P Programming Langugage. https://github.com/p-org/P. (2018).

Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16).*

Benjamin Pierce and Davide Sangiorgi. 1996. Typing and Subtyping for Mobile Processes. In *Mathematical Strustures In Computer Science.* 376–385.

Benjamin C. Pierce and David N. Turner. 2000. Proof, Language, and Interaction. Chapter Pict: A Programming Language Based on the Pi-Calculus.

Pony. 2017. Pony Programming Langugage. https://www.ponylang.org. (2017).

James Riely and Matthew Hennessy. 1998. A Typed Language for Distributed Mobile Processes (Extended Abstract). In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98).*

Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319.

Koushik Sen and Gul Agha. 2006. Automated Systematic Testing of Open Distributed Programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering.*

Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 842–853.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and Proving with Distributed Protocols. In *45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '18).* ACM.

Alexander J. Summers and Peter Müller. 2016. Actor Services. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632.*

Viktor Vafeiadis and Matthew Parkinson. 2007. A marriage of rely/guarantee and separation logic. In *International Conference on Concurrency Theory.* Springer, 256–271.

Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015).

Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In *Proc. 6th USENIX OSDI.* San Francisco, CA.

Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. 2014. Jitk: A Trustworthy In-kernel Interpreter Infrastructure. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI).*

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Tom Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

Qiwen Xu, Willem-Paul de Roever, and Jifeng He. 1997. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing* 9, 2 (1997), 149–174.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI).*

Jianqiao Yang, Ankush Desai, and Koushik Sen. 2017. Multi-Path Symbolic Execution for P Language. https://github.com/thisiscam/MultiPathP. (2017).