Compositional Programming and Testing of Dynamic Distributed Systems

Ankush Desai
Amar Phanishayee
Shaz Qadeer
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-95
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-95.html

July 30, 2018
Copyright © 2018, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part 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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.
Compositional Programming and Testing of Dynamic Distributed Systems

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

Real-world distributed systems are rarely built as a monolithic system. Instead, they are a composition of multiple interacting components that together ensure the desired system specification. Programming these systems is challenging as one must deal with both concurrency and failures. This paper proposes techniques for building reliable 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 (assume-guarantee) testing of distributed systems.

We demonstrate the efficacy of our framework by building two practical distributed systems, a fault-tolerant transaction commit service and a fault-tolerant replicated hash-table. Our framework helps implement these systems modularly and validate them via compositional systematic testing. We empirically demonstrate that using abstraction-based compositional reasoning 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.

ACM Reference Format:

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 protocol descriptions and their concrete implementations [Chandra et al. 2007].

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; Leroy 2009; Sigurbjarnarson et al. 2016; Wang et al. 2014; Yang and Hawblitzel 2010] 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 built 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, 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 a changing communication topology as references 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 clear 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 are instantiated 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 composition of modules $P$ and $Q$ behaves like language intersection over the traces of $P$ and $Q$. ModP provides operators for hiding actions of a module, to construct a more abstract module. To ensure that compositional refinement 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.

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, failure detectors, and network channels. 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 demonstrate that the case study software stack implementation is realistic and sufficiently detailed.

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.

![Fig. 2. A Client-Server Application using ModP State Machines](image)

**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 (details in Section 2.2).

**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. Each time the machine transitions into a state, the entry function of that state is executed. 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 an 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 an eResponse event, the server interacts with the helper machine to compute the result that it sends back to the client.

**Dynamism.** There are two key features that 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

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. 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 machine.

Fig. 3. Modular Client-Server Implementation

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 (typing rules for module constructors are defined in 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. The compiler generates C code for the module in the implementation declaration.

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.

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.

```
1 machine AbstractServerImpl receives eRequest;
2 sends eResponse;
3 {
4     start state Init {
5         on eRequest do (payload: RequestType) {
6             send payload.source, eResponse,
7             (resId = payload.reqId,success = choose());
8         }
9     }
10 }
11 spec ReqIdsAreMonoInc observes eRequest {
12     var prevId : int;
13     start state Init {
14         on eRequest do (payload: RequestType) {
15             prevId = payload.reqId;
16             assert(prevId <= prevId + 1);
17         }
18     }
19 }
20 spec ResIdsAreMonoInc observes eResponse {
21     var prevId : int;
22     start state Init {
23         on eResponse do (payload: ResponseType) {
24             prevId = payload.resId;
25             assert(prevId <= prevId + 1);
26         }
27     }
28 }
```

(a) Abstraction and Specifications

```
1 module AbstractServerModule = {
2     ServerToClientIT -> AbstractServerImpl
3 };
4 module AbstractClientModule = {
5     ClientIT -> AbstractClientImpl
6 };
7 module AbstractServerModule || AbstractClientModule
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     /* Circular Assume-Guarantee */
17     // Check that client abstraction is correct.
18     test test2: ClientModule || AbstractServerModule
19     | | AbstractServerModule;
20     // Check that server abstraction is correct.
21     test test3: AbstractServerModule || ServerModule
22     | | AbstractClientModule;
23     // Create abstract module using Hide
24     module hideModule = hide X in AbstractServerModule;
25     module hideModule = hide X in AbstractServerModule;
26     test test4: ClientModule || ServerModule
27     | | hideModule;
```

(b) Test Declarations for Compositional Testing

Fig. 4. Compositional Testing of the Client-Server Application using ModP Modules
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 simply 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 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 are 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 privates events (interfaces) into a module, it can be used to converts 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. For defining the operational semantics of a module and to ensure that composition is intersection, it is essential that constructed modules are 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 make 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 (Section 4) and making them amenable to compositional reasoning (Section 5.2).

(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 ($\mathcal{MRecvs}, \mathcal{M Sends}, \mathcal{M Creates}, \mathcal{Rem}, \mathcal{Enq}, \mathcal{New}, \mathcal{Local}$) where:

1. $\mathcal{MRecvs} \subseteq \mathcal{E}$ is the nonempty set of events received by the machine.
2. $\mathcal{M Sends} \subseteq \mathcal{E}$ is the set of all events sent by the machine.
3. $\mathcal{M Creates} \subseteq \mathcal{I}$ is the set of interfaces created by the machine.
4. $\mathcal{Rem} \subseteq \mathcal{S} \times \mathcal{B} \times \mathcal{M} \times \mathcal{S}$ is the transition relation for removing a message from the input buffer. If $(s, b, n, s') \in \mathcal{Rem}$, then the $n$-th entry in the input buffer $b$ is removed and the local state moves from $s$ to $s'$.
5. $\mathcal{Enq} \subseteq \mathcal{S} \times \mathcal{Z} \times \mathcal{E} \times \mathcal{V} \times \mathcal{S}$ is the transition relation for sending a message to a machine. If $(s, id, e, v, s') \in \mathcal{Enq}$, then event $e$ with payload $v$ is sent to machine $id$ and the local state of the sender moves from $s$ to $s'$.
6. $\mathcal{New} \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{S}$ is the transition relation for creating an interface. If $(s, i, s') \in \mathcal{New}$, then the machine linked against interface $i$ is created and the machine moves from $s$ to $s'$.
7. Local ⊆ S × Z × S × Z is the transition relation for local computation in the machine. The state of a machine is a pair (s, id) ∈ S × 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′) ∈ 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 ∈ 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 ∈ 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 Z as I × N × K. For our formalization, we are interested in machine identifiers that are embedded inside the data structures in a machine local state s ∈ S or a value v ∈ 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, α) ∈ Z refers to the n-th instance of an interface represented by i ∈ I. We refer to the final component α of a machine identifier as its permissions. This 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 (I, n, β) ∈ Z referencing to the n-th instance of interface I where β represents the receive set associated with the interface I (β = IRecvs(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(I) ⊆ MRecvs(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 A ∈ 2^K of allowed permissions. For an event e, the set 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 ∀(_, _, α) ∈ ids(v), α ∈ A(e). In other words, 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 A(e), the send fails otherwise (captured as the Send0k).
condition when defining the semantics of send in Section 5.1). This changed semantics of send based on permission-based capability control plays a key 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 that can be embedded in it and the check (2) above for the correctness of the send operation can be performed statically.

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 MODULAS

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 simpler modules. Figure 5 presents the expression language supported by ModP module system for module construction.

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 $P \vdash EP_P, IP_P, IP, IP_{ER}, IP_{ES}, IP_{IC}$. 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 $IP$, interface link map $IP$, events received $IP_{ER}$, events sent $IP_{ES}$, and interfaces created $IP_{IC}$. 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 $\text{dom}(x)$ and $\text{codom}(x)$ 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. $EP_P \in 2^I$ 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. $IP_P \in 2^I$ 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. $IP : I \rightarrow M$ interface definition map that binds an interface name $i$ to a machine name $IP[i]$. Recollect that in 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 \texttt{bind} operation, so that if \( (i, m) \in I_P \) then creation of an interface \( i \) in module \( P \) leads to the creation of an instance of \( m \).

4. \textbf{Interface link map.} \( I_P \in \mathcal{I} \rightarrow \mathcal{I} \rightarrow \mathcal{I} \) is the \textit{interface link map} that maps each interface \( i \in \text{dom}(I_P) \) to a machine link map that binds interfaces created by the code of machine \( I_P[i] \) to an interface name. If the statement \texttt{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 \texttt{new} \( x \) in the code of machine \( I_P[i] \) as creation of interface \( x' \), creating an instance of machine \( I_P[i] \).

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

5. \textbf{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 \( P \), \( ER_P = \bigcup_{m \in \text{codom}(I_P)} M\text{Recvs}(m) \setminus EP_P \).

6. \textbf{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 \( P \), \( ES_P = \bigcup_{m \in \text{codom}(I_P)} M\text{Sends}(m) \setminus EP_P \).

7. \textbf{Interfaces created.} \( IC_P \in 2^I \) 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), \( IC_P = \bigcup_{(i, m) \in IC, x \in M\text{Creates}(m)} \{ L_P[i][x] \} \).

\textit{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.

\textbf{Input and output actions}. The \textit{input events} of module \( P \) are the events that are received but not sent by \( P \) i.e. \( ER_P \setminus ES_P \). The \textit{input interfaces} of \( P \) are the set of interfaces that are exported but not created by \( P \) i.e. \( \text{dom}(I_P) \setminus (IP_P \cup IC_P) \).
The \textit{output events} of \( P \) are the set of events i.e. \( ES_P \) and the \textit{output interfaces} are the created non-private interfaces of \( P \) i.e. \( IC_P \setminus IP_P \).

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 \(5.1 - 5.2\).

\textbf{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 \textbf{Primitive Module}

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

\begin{equation}
\begin{aligned}
\text{BIND} & : (i_1, m_1), \ldots , (i_n, m_n) \rightarrow \forall (i, m) \in f . I_{\text{Recvs}}(i) \subseteq M_{\text{Recvs}}(m) \setminus h \neg x \in M_{\text{Creates}}(m).
\end{aligned}
\end{equation}

Rule \texttt{BIND} presents the rule for \texttt{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 \(i \in \text{dom}(f)\) 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 \texttt{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).

\textbf{(WF1) Interface definition map is consistent:} For each \((i, m) \in I_P\), we have \(I_{Recvs}(i) \subseteq M_{Recvs}(m)\).

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

\textbf{(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 \(x \in M_{Creates}(m)\), we have \(x \in \text{dom}(L_P[i])\).

### 4.2 Hiding Events and Interfaces

Hiding events and interfaces in a module allows 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 \(AbstractServerModule\) 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.

\begin{verbatim}
module HE_Server = hide eProcessReq, eReqSuccess, eReqFail in ServerModule
\end{verbatim}

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 in a module, it must be both an exported and created interface of the module.

\begin{verbatim}
module HI_Server = hide HelperIT in HE_Server
\end{verbatim}

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.

\textbf{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 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 does 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 (1) each send of event e adheres to the specification (SendOk) in Section 3, and (2) 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 a module with private events 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.

\[
(HIDEEVENT) \quad (AA \cap B = (A \setminus B) \cup (B \setminus A))
\]

\[
\begin{align*}
P \ni EP_P, IP_P, LP, ER_P, ES_P, IC_P \quad &\forall x \in IC_P \Delta dom(I_P), \text{IRecevs}(x) \cap \alpha = \emptyset \quad (he1) \\
&\forall e \in (ER_P \cup ES_P) \setminus \alpha, \forall \alpha' \in \mathcal{A}(e), \alpha \cap \alpha' = \emptyset \quad (he2)
\end{align*}
\]

\[
\begin{array}{c}
hide \alpha \text{ in } P \ni EP_P \cup \alpha, IP_P, LP, \text{L_P} \\
hide \beta \text{ in } P \ni EP_P, IP_P \cup \beta, IP_P, LP
\end{array}
\]

\[
(HIDEINTERFACE) \quad P \ni EP_P, IP_P, LP, ER_P, ES_P, IC_P \quad \beta \subseteq \text{dom}(I_P) \cap IC_P \quad (hi1)
\]

Rule HIDEEVENT handles the hiding of a set of events \( \alpha \) in module \( P \). This rule adds \( \alpha \) to \( EP_P \). Condition (he1) checks all events in \( \beta \) 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 \( \beta \) in module \( P \). This rule adds \( \beta \) 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 a 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.

**Rule COMPOSITION**. (AΔB = (A \ B) ∪ (B \ A))

\[
P \parallel Q \rightarrow P \parallel P \parallel Q = P \parallel Q \parallel P = \emptyset \quad (c1)
\]

**Condition** (c1) enforces that the domains of P and Q are disjoint, thus preventing conflicting interface bindings. Conditions (c2) ensures that the input and output actions of P do not hide by private events of Q and vice-versa. Conditions (c3) and (c4) together check that private permissions of P || Q do not leak out. Condition (c3) checks that creation of an input interface of P does not leak a 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

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.

```
1 interface ServerToClientIT' receives eRequest, eReqFail;
2 interface HelperIT' receives eProcReq;
3
4 machine HelperImpl' receives eProcReq;
5 sends ..; creates ..; // "body of ..."
6
7 module ServerModule' =
8     
9     8 ServerToClientIT'->HelperIT', HelperIT->HelperImpl1;
10    
11 module allServers = ServerModule' ||
12 rename HelperIT -> HelperIT' in ServerModule';
```

Fig. 6. Renaming Interfaces Module Constructor

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 condition (c1)). Interface renaming comes to the rescue in such a situation.

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 redirection in the interface link map, and the corresponding visible action is creation of interface HelperIT'.
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) ∪ IC_P is the universe of all interfaces relevant to P. Condition (r2) checks that i' is a new name different from the current set of interfaces relevant to P. Condition (r3) checks that the set of received events of i and i' are the same. 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 definition 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 completely 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, LP, 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 ⊢ EP_P, IP_P, LP, 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 I × N to S × Z. If (i, n) ∈ 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 ∈ S and a machine identifier id ∈ Z (as described in Section 3.1).
2. The second component B is a partial map from I × N to B. If (i, n) ∈ 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 I to 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 $(I)$ 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 $e$ to indicate that the computation is local and is an internal transition of the module $P$.

**Rules for creating interfaces:** Let $s_0 \in S$ represent a state such that $ids(s_0) = \emptyset$. Let $b_0 \in B$ be the empty sequence over $E \times V$. 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 first two rules are triggered by the environment of $P$ 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 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 first rule is triggered by the environment of $P$ 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.
\(\text{INTERNAL}(R1)\)
\[S_p[i, n] = (s, id) \quad \text{if} \quad (s, id, s', id') \in \text{Local}(I_p[i])\]
\(\rightarrow\)
\[S_p[i, n] \rightarrow (s', id'), B_p [i, n] \]
\(\text{ENVIRONMENT-CREATE}(R3)\)
\[i \in I \cap \text{dom}(I_p) \quad n \in C_p[i] \]
\(\rightarrow\)
\[S_p[i, n] \rightarrow (s', id'), B_p [i, n] \]
\(\text{CREATE-BAD}(R5)\)
\[S_p[i, n] = (s, id) \quad (s, id, s', id') \in \text{New}(I_p[i]) \]
\(\rightarrow\)
\[\text{error}\]
\(\text{OUTPUT-CREATE-INSIDE}(R7)\)
\[\text{CreateOk}(I_p[I], i') \]
\(\rightarrow\)
\[S_p[i, n] = (s, id) \quad (s, id, s', id') \in \text{New}(I_p[I]) \]
\(\rightarrow\)
\[\text{error}\]
\(\text{CREATE-PRIVATE}(R8)\)
\[\text{CreateOk}(I_p[I], i') \]
\(\rightarrow\)
\[S_p[i, n] = (s, id) \quad (s, id, s', id') \in \text{New}(I_p[I]) \]
\(\rightarrow\)
\[\text{error}\]
\(\text{INPUT-SEND}(R9)\)
\[B_p[i, n] = b \quad e \in \text{MRecvs}(I_p[I]) \]
\(\rightarrow\)
\[S_p[i, n] = (s, id) \quad (s, id, e, v) \in \text{SendOk}(I_p[I], i', e, v) \]
\(\text{SEND-BAD}(R10)\)
\[S_p[i, n] = (s, id) \quad \text{id} = (s, id, e, v) \quad (s, id, e, v) \in \text{SendOk}(I_p[I], i', e, v) \]
\(\rightarrow\)
\[\text{error}\]
\(\text{OUTPUT-SEND-OUTSIDE}(R11)\)
\[S_p[i, n] = (s, id) \quad \text{id} = (s, id, e, v) \quad (s, id, e, v) \in \text{SendOk}(I_p[I], i, e, v) \]
\(\rightarrow\)
\[\text{error}\]
\(\text{SEND-PRIVATE}(R13)\)
\[\text{id} = (s, id, e, v) \quad (s, id, e, v) \in \text{SendOk}(I_p[I], i, e, v) \]
\(\rightarrow\)
\[\text{error}\]

Fig. 7. Rules for operational semantics of ModP modules

First, an event that is sent by \(P\) is not considered as an input event, which is safe since rules of \(\text{output-disjointness}\) (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 \(\text{WF1}\)). 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 \(\text{WF3}\)).

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 \(\text{SendOk}(\hat{m}, a, e, v) = e \in \text{MSends}(\hat{m})\)∧
\( e \in a \land \forall (\_, \_, \beta) \in \text{ids}(v). \beta \in \mathcal{A}(e) \) captures the \((\text{SendOk})\) specification described in Section 3.2.

Thus, machine \((i, n)\) may only send events declared by it in \(\text{MSend}(I_p[i])\) and allowed by the permission \(\alpha_i\) of the target machine and should not embed machine identifiers with private permissions in the payload \(v\). Note that the dynamic check \((\text{SendOk})\) helps guarantee the well-formedness condition \((\text{WF}3)\) 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 \((\text{OUTPUT-Send-Outside})\) sends an event to machine outside \(P\) whereas rules \((\text{OUTPUT-Send-Inside})\) and \((\text{Send-Private})\) send an event to some machine inside \(P\). In the former, the target machine \(m_i\) is not in the domain of \(I_p\), whereas in the later cases the target machine is inside the module and hence present in domain of \(I_p\). For \((\text{Send-Private})\), the label on the transition is \(e\) 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 \(\text{execs}(P)\) represent the set of all possible executions of the module \(P\). An execution is unsafe if \(G^n \xrightarrow{.} \text{error}\); otherwise, it is safe. The module \(P\) is safe, if for all \(\tau \in \text{execs}(P)\), \(\tau\) 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 \(\Sigma_P = (I \setminus I_P) \cup ((I \times \mathbb{N}) \times (\text{ES}_P \cup \text{ER}_P) \times \mathcal{V})\). The signature is partitioned into the output signature \((\text{IC}_P \setminus I_P) \cup ((I \times \mathbb{N}) \times \text{ES}_P \times \mathcal{V})\) and the input signature \((I \setminus \text{IC}_P) \cup ((I \times \mathbb{N}) \times (\text{ER}_P \setminus \text{ES}_P) \times \mathcal{V})\).

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 \(\tau\) is the sequence \(\sigma\) obtained by removing occurrences of \(e\) from the sequence \(a_1, \ldots, a_{n-1}\).

Let \(\text{traces}(P)\) 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 \(\sigma \in \text{traces}(P)\) then \(\sigma[\Sigma_P]\) represents the projection of trace \(\sigma\) over the set \(\Sigma_P\) 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) \(\text{IC}_Q \setminus I_P \subseteq \text{IC}_P \setminus I_P\), (2) \(\text{dom}(I_Q) \setminus I_P \subseteq (\text{dom}(I_P) \cup \text{IC}_P) \setminus I_P\), (3) \(\text{ES}_Q \subseteq \text{ES}_P\), (4) \(\text{ER}_Q \subseteq \text{ER}_P \cup \text{ES}_P\) (note that (1)-(4) together imply \(\Sigma_Q \subseteq \Sigma_P\)), (5) and for every trace \(\sigma\) 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 \(\pi \in \Sigma_P^{\ast}Q, \pi \in \text{traces}(P||Q)\) iff \(\pi[\Sigma_P] \in \text{traces}(P)\) and \(\pi[\Sigma_Q] \in \text{traces}(Q)\). (Proof in Appendix)

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

**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 \(P \preceq Q\) implies that \(P||R \preceq P||Q\). (Proof in Appendix)
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. These theorems form the basis of our theory of compositional refinement and are used for proving the theorems underlying our compositional testing methodology (Theorems 5.3 and 5.4).

We present the two key theorems that describe the principles of circular assume-guarantee reasoning used for analysis of ModP systems. First, we introduce a generalized composition operation $\|P$, where $P$ is a non-empty set of modules. This operator represents the composition of all modules in $P$. The binary parallel composition operator is both commutative and associative. Thus, $\|P$ is a module obtained by composing modules in $P$ in some arbitrary order. Let $P$ and $Q$ be set of modules, we say that $P$ is a subset of $Q$, if $P$ can be obtained by dropping some of the modules in $Q$.

**Theorem 5.3 (Compositional Safety).** Let $\|P$ and $\|Q$ be well-formed. Let $\|P$ refine each module $Q \in Q$. Suppose for each $P \in P$, there is a subset $X$ of $P \cup Q$ such that $P \in X$, $\|X$ is well-formed, and $\|X$ is safe. Then $\|P$ is safe. (Proof in Appendix)

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

**Theorem 5.4 (Circular Assume-Guarantee).** Let $\|P$ and $\|Q$ be well-formed. Suppose for each module $Q \in Q$ there is a subset $X$ of $P \cup Q$ such that $Q \notin X$, $\|X$ is well-formed, and $\|X$ refines $Q$. Then $\|P$ refines each module $Q \in Q$. (Proof in Appendix)

Theorem 5.4 states that to show that $\|P$ refines $Q \in Q$, any subset of modules in $P$ and $Q$ 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 $\|P$ refines every submodule of $Q$ is implied by $\|P$ refines module $\|Q$. If $\|P$ refines $\|Q$ (a well-formed module), then using Theorem 5.1, $\|P$ would refine each individual submodule in $Q$ as well. Similarly, if $\|P$ refines every submodule of $Q$ and $\|Q$ is a well-formed module, then $\|P$ refines module $\|Q$.

**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 $\text{test tname: P}$ introduces a safety test obligation that the executions of module $P$ do not result in a failure (module $P$ is safe). The declaration $\text{test tname: P refines Q}$ introduces a test obligation that module $P$ refines module $Q$. These test obligations are automatically discharged using ModP’ systematic testing engine (Section 7).

We illustrate using the protocol stack in Figure 1, how we used ModP to compositionally test a complex distributed system. We implemented two distributed services: (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit [Bernstein et al. 1986;
Gray 1978; 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 a good 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 library and OS calls for real execution. We implemented each of the complex protocol described above as a separate module.

We implemented the safety specifications (as spec. machines) of all the protocols as described in their respective paper. Table on the side shows examples of specifications checked for some of the distributed protocols.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>Transactions are atomic [Gray 1978] (2PC.Spec)</td>
</tr>
<tr>
<td>Chain Rep.</td>
<td>All invariants in [van Renesse and Schneider 2004], cmd-log consistency (CRSpec)</td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>Consensus requirements [Lamport 2001], log consistency [Van Renesse and Altinbuken 2015] (MPSpec)</td>
</tr>
</tbody>
</table>

Fig. 8. Compositional Testing of Transaction Commit Service

1 // molochic 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 // MultiPaxos linearizability as specification machine
12 test t5: SMRClientAbs || assert MPSpec 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 SMReplicatedMachineInterface in
26 // hide events used for interaction with replicated machine
27 (hide eSMReplicatedMachineOperation, eSMReplicatedLeader in SMRClientAbs)
28 test t7: LHS refines RHS;
the Multi-Paxos and Chain-Replication based versions. The set of abstraction modules is $Q = \{\text{SMRClientAbs, SMRLinearizAbs, OSServAbs}\}$. The test obligation $\text{mono}$ 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 8-line 5). Using Theorem 5.3, we can decompose the problem into safety tests $t_1$ and $t_2$ under the assumption that each module in $P_m$ refines each module in $Q$. This assumption is then validated using the Theorem 5.4 and tests $t_3$, $t_4$. 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 $t_6$ and $t_7$. The tests $t_5$ and $t_8$ are substitutes for the refinement checks $t_4$ and $t_7$ 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 8. 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 8 (line 22-29) represent the test-script we used to perform test $t_7$. 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 9).

**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 non-determinism 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 $P \leq Q$ 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 10 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.

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 common 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 simulation of large-scale services running on only a handful of nodes. A ModP state machine can create a fresh container by invoking runtimes’ 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 a 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 ModP-based 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 reliable
(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 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Phase Commit</td>
<td>441</td>
<td>61</td>
<td>41</td>
<td>35</td>
<td>128</td>
</tr>
<tr>
<td>Chain Rep. SMR</td>
<td>1267</td>
<td>220</td>
<td>173</td>
<td>130</td>
<td>105</td>
</tr>
<tr>
<td>Multi-Paxos SMR</td>
<td>1617</td>
<td>101</td>
<td>121</td>
<td>92</td>
<td>90</td>
</tr>
<tr>
<td>Data structures</td>
<td>276</td>
<td>25</td>
<td>-</td>
<td>89</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>3601</td>
<td>Others</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

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 11 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 linearizability abstraction, 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.

![Fig. 11. Test-Amplification via Abstractions: Chain Replication Protocol](image-url)
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 11 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 and 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).

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 is considered as a single monolithic system. We use number of schedules explored before finding the bug as the comparison metric. Figure 12 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 8. 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 $t_1$, $t_2$, $t_5$, $t_8$ (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.

Examples of bugs found. We describe 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 having 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 was 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 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.

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 13 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

![Fig. 13. Performance of ModP HashTable using Multi-Paxos (MP) is comparable with an open source baseline implementation (mean over 60s close-loop client runs).](image-url)
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 [Eiriksson 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. Formalisms for the modeling and compositional analysis of dynamic systems can be categorized 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 \( \pi \)-calculus [Milner et al. 1992; Pierce and Turner 1997] has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication. The popular approach for reasoning about behavior in these formalisms is the notions of equivalence and congruence: weak and strong bisimulation, etc., which involves examining the state transition structure of the two systems being compared. There’s also a very large literature on observational equivalence in \( \pi \)-calculus based on trace inclusion [Cortier and Delaune 2009]. Extensions of \( \pi \)-calculus such as asynchronous \( \pi \)-calculus, distributed join calculus [Fourney and Gonthier 1996; Fournet et al. 1996], D\( \pi \)-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 greatly 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’Liguro 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. Communication between modules is done via single-writer multiple-reader shared variables and a shared global clock drives each module in lockstep. Dynamic Reactive Modules [Fisher et al. 2011] (DRM) is a dynamic extension of Reactive Modules with support for the dynamic creation of modules and dynamic topology. The semantics of dynamic reactive modules are given by dynamic discrete systems [Fisher et al. 2011] 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; Vafeeidis and Parkinson 2007; Xu et al. 1997] and concurrent separation logic [Feng et al. 2007; Leino and Müller 2009; O’Hearn...
Actor services [Summers and Müller 2016] propose program logic for modular proofs of actor programs. DISEL [Sergey et al. 2018] provides a language to compositionally implement and verify distributed systems. 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. But 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 tool) [Arts et al. 2008; Brown et al. 2014; 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.
REFERENCES

1338 ACM, New York, NY, USA, Article 1, 1 pages. https://doi.org/10.1145/2596631.2596633
Compositional Programming and Testing of Dynamic Distributed Systems


The assumption that machine identifiers cannot appear "out-of-thin-air" is formalized as follows.

A MODEL OF COMPUTATION (CONTD.)

Machine identifiers cannot be created out of thin air. A state machine can get access to a machine identifier either through a remove transition (Rem) where some other machine sent the identifier as a payload or through create transition (New) where it creates an instance of a machine. The assumption that machine identifiers cannot appear "out-of-thin-air" is formalized as follows.

For all $m \in \mathcal{M}$, $s, s' \in \mathcal{S}$, $id, id' \in \mathbb{Z}$, $e \in \mathcal{E}$, $v \in \mathcal{V}$, $i \in I$, $b \in B$, and $n \in \mathbb{N}$:

1. $(s, id, s', id') \in \text{Local}(m) \Rightarrow ids(s') \cup \{id'\} \subseteq ids(s) \cup \{id\}$.
2. $(s, b, n, s') \in \text{Rem}(m) \Rightarrow ids(s') \subseteq ids(s) \cup \{ids(v) \mid \exists e. b[n] = (e, v)\}$.
3. $(s, id, e, v, s') \in \text{Enq}(m) \Rightarrow ids(v) \cup ids(s') \subseteq ids(s)$.
4. $(s, i, s') \in \text{New}(m) \Rightarrow ids(s') \subseteq ids(s)$.

Invariants for Executions of a Module. During the execution of a module $P$, all reachable configurations $(S_P, B_P, C_P)$ satisfy the following invariants:

**Lemma A.1: Invariants for Executions of a Module**

Let $P$ be a well formed module. For any execution $\tau \in \text{execs}(P)$ where $\tau$ is a sequence of global configurations $G_0 \xrightarrow{a_0} G_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G_n$, all global configurations $G_i$ satisfy the invariants:

1. $\text{dom}(S_P) = \text{dom}(B_P)$
2. $\forall (i, n) \in \text{dom}(B_P). i \in \text{dom}(l_P) \land n < C[i]$ 
3. $\forall i \in \text{dom}(l_P). C[i] = \text{card} \{n \mid (i, n) \in \text{dom}(B_P)\}$
4. $\forall (x, n, \alpha) \in ids(S_P) \cup ids(B_P). x \in \text{dom}(l_P) \Rightarrow (x, n) \in \text{dom}(B_P)$
5. $\forall (x, n, \alpha) \in ids(S_P) \cup ids(B_P). n < C_P[x]$

**Proof.** The invariants (I1) - (I5) are inductive and can be proved by performing induction over the length of the execution $\tau$ for all the transitions (rules) defined in ModP operations semantics. \hfill $\square$
B THEOREMS AND PROOFS

The module system proposed in this paper provide the following important top-level lemmas:

1. **Composition Is Intersection**: Composition behaves like language intersection. This is captured by the Lemma B.1, which asserts that traces of a composed module are completely determined by the traces of the component modules. This Lemma forms the basis and used by the rest of the lemmas.

2. **Composition Preserves Refinement**: The traces of a composed module is a subset of the traces of each component module. Hence, the composition of two modules creates a new module which is equally or more detailed than its components. This is captured by the Lemma B.4.

3. **Circular Assume-Guarantee**: Lemma B.5 states that to show \( P \) refines \( Q \in Q \), any subset of modules in \( P \) and \( Q \) 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 \( Q_2 \) and \( Q_2 \) to prove \( Q_1 \).

4. **Compositional Safety Analysis**: Lemma B.6 talks about implementation modules in \( P \) and abstraction modules in \( Q \). When proving safety of any module \( P \in P \), it is allowed to pick any modules in \( Q \) for constraining the environment of \( P \).

5. **Hide Event Preserves Refinement**: Lemma B.7 states that the hide event operation preserves refinement, is compositional and create a sound abstraction of the module.

6. **Hide Interface Preserves Refinement**: Lemma B.8 states that the hide interface operation preserves refinement, is compositional and create a sound abstraction of the module.

---

Definitions. We present the definitions needed for the formalisms and proofs in this section.

1. Let \( G \) be the set of all possible configurations. For a configuration \( G = (S, B, C) \), we refer to its elements as \( G_S, G_B, \) and \( G_C \) respectively.

2. Let \( \text{last} \) be a function that given an execution which is a sequence of alternating global configuration and transition labels returns the last global configuration state. If \( \tau = G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n \) then \( \text{last}(\tau) = G^n \)

3. Let \( \text{trace}(\tau_P) \) represent the trace corresponding to execution \( \tau_P \in \text{execs}(P) \).

4. Two configurations \( G, G' \in \hat{G} \) are compatible, if the following conditions hold:
   
   \( (1) \ \forall (i, n) \in (\text{dom}(G_S) \cap \text{dom}(G'_S)), G_S[i, n] = G'_S[i, n], \)
   
   \( (2) \ \forall (i, n) \in (\text{dom}(G_B) \cap \text{dom}(G'_B)), G_B[i, n] = G'_B[i, n], \)
   
   \( (3) \ \forall i \in (\text{dom}(G_C) \cap \text{dom}(G'_C)), G_C[i] = G'_C[i] \)
   
   Informally, two configurations are compatible, if each element in the configurations agree on the common values in their domain.

5. Let \( \text{union} \) be a partial function from \( (\hat{G} \times \hat{G}) \) to \( \hat{G} \) satisfying the following properties:

   1. \( (G, G') \in \text{dom}(\text{union}) \) iff \( G \) and \( G' \) are compatible.
   2. \( (G^p, G^q, G^c) \in \text{union} \) iff \( G^c = (G^p_S \cup G^q_S, G^p_B \cup G^q_B, G^p_C \cup G^q_C) \).
**Lemma B.1: Compositional Is Intersection**

Let $P$, $Q$ and $P||Q$ be well-formed. For any $\pi \in \Sigma_s^{*||Q}$, $\pi \in \text{traces}(P||Q)$ iff $\pi[\Sigma_P] \in \text{traces}(P)$ and $\pi[\Sigma_Q] \in \text{traces}(Q)$.

**Proof.** We prove this lemma by proving two simpler lemmas, Lemma B.2 and Lemma B.3. The proof is decomposed into the following two implications:

**Forward Implication for traces:**
If $\sigma \in \text{traces}(P||Q)$ then the projection $\sigma[\Sigma_P] \in \text{traces}(P)$ and the projection $\sigma[\Sigma_Q] \in \text{traces}(Q)$. This follows from the Lemma B.2.

**Backward Implication for traces:**
If there exists a sequence $\sigma \in \Sigma_s^{*||Q}$ such that $\sigma[\Sigma_P] \in \text{traces}(P)$ and $\sigma[\Sigma_Q] \in \text{traces}(Q)$, then $\sigma \in \text{traces}(P||Q)$. This follows from the Lemma B.3.

**Proof.** We perform induction over the length of execution $\tau_c$ of the composed module $P||Q$.

**Inductive Hypothesis:** For every execution $\tau_c \in \text{execs}(P||Q)$, there exists an execution $\tau_p \in \text{execs}(P)$ such that $\text{trace}(\tau_p)[\Sigma_P] = \text{trace}(\tau_c)[\Sigma_P]$ and there exists an execution $\tau_q \in \text{execs}(Q)$ such that $\text{trace}(\tau_q)[\Sigma_Q] = \text{trace}(\tau_c)[\Sigma_Q]$, and $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$.

We refer to the elements of the global configuration $\text{last}(\tau_c)$ as $\text{last}(\tau_c)_S$, $\text{last}(\tau_c)_B$, and $\text{last}(\tau_c)_C$.

**Base case:** The base case for the inductive proof is for an execution $\tau_c$ of length 0, $\tau_c \in \text{execs}(P||Q)$. The projection of the execution $\tau_c$ over the alphabet of the individual modules results in a execution of length zero which belongs to the set of executions of all the modules. We know that, for the base case there exists an execution $\tau_p \in \text{execs}(P)$ and $\tau_q \in \text{execs}(Q)$ of length zero such that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$. Hence, the inductive hypothesis holds for the base case.

**Inductive case:** Let us assume that the hypothesis holds for any execution $\tau_c \in \text{execs}(P||Q)$. Let $\tau_p$ and $\tau_q$ be the corresponding executions for module $P$ and $Q$ such that $\text{trace}(\tau_c)[\Sigma_P] = \text{trace}(\tau_p)[\Sigma_P]$, $\text{trace}(\tau_c)[\Sigma_Q] = \text{trace}(\tau_q)[\Sigma_Q]$ and $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$. To prove that the hypothesis is inductive we show that it also holds for the execution $\tau'_c \in \text{execs}(P||Q)$ where $\tau'_c = \tau_c \xrightarrow{a} G$ and $\tau'_p, \tau'_q$ be the corresponding executions of $P$ and $Q$.

We perform case analysis for all possible transitions labels $a$.

- $a = \epsilon$
  This is the case when the composed module $P||Q$ takes an invisible transition. Lets say $n$-th instance of an interface $i$ identified by $(i, n) \in \text{dom}(\text{last}(\tau_c)_S)$ made an invisible transition. This could be because the machine took any of the following transitions: INTERNAL, REMOVE-EVENT, CREATE-BAD, OUTPUT-CREATE-3, SEND-BAD, and Output-Send-3.

Consider the case when $i \in \text{dom}(I_P)$ i.e. machine corresponding to interface $i$ is implemented in module $P$. 

Based on the assumption that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$, we know that $\text{last}(\tau_c)_S[i, n] = \text{last}(\tau_p)_S[i, n]$ and $\text{last}(\tau_c)_B[i, n] = \text{last}(\tau_p)_B[i, n]$. Hence, if machine instance $(i, n)$ in $P||Q$ can make an invisible transition $a$ when in global configuration $\text{last}(\tau_c)$, then the same invisible transition can be taken by module $P$ in configuration $\text{last}(\tau_p)$. Hence, $\text{trace}(\tau'_c)[\Sigma_P] = \text{trace}(\tau'_p)[\Sigma_P]$ (where $\tau'_p = \tau_p \xrightarrow{a} G'$). Since $a = \epsilon$, $\text{trace}(\tau_q)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q]$

Note that the invisible transitions do not change the map $C$. Since, the module $P||Q$ and $P$ took the same transition $a$ and configuration of module $Q$ has not changed, the resultant configurations satisfy the property $\text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau_q))$.

The same analysis can be applied to the case when $m \in \text{dom}(M_Q)$.

• $a = i$ where $i \in I$

This is the case when the composed module or the environment takes the visible transition of creating an interface $i$. We perform case analysis for all such possible transitions:

1. **Environment-Create**
   Consider the case when the environment of module $P||Q$ takes a transition to create an interface $i$. If it is created by $P||Q$ using the Environment-Create, then it can be created by $P$ and $Q$ only using the Environment-Create rule. This comes from the fact that $i$ does not belong to $\text{IC}_{P||Q}$ and $\text{dom}(I_{P||Q})$.

Hence the environment of both $P$ and $Q$ can take the transition and the resultant executions $\tau'_p, \tau'_q$ will satisfy the condition $\text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau'_q))$, $\text{trace}(\tau'_c)[\Sigma_P] = \text{trace}(\tau'_p)[\Sigma_P], \text{trace}(\tau'_c)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q]$.

2. **Input-Create**
   Our definition of composition and compatibility guarantees that if $P||Q$ is well-formed then:
   1. $\text{dom}(I_{P||Q}) = \text{dom}(I_P) \cup \text{dom}(I_Q)$
   2. $\text{dom}(I_P) \cap \text{dom}(I_Q) = \emptyset$

Hence, if the composed module $P||Q$ receives an input create request for $i \in \text{dom}(I_P) \subset \text{IC}_{I_P}$ from the environment, then either $i \in \text{dom}(I_P)$, or $i \in \text{dom}(I_Q)$. Also, since $i \notin \text{IC}_{P||Q}$, it implies that $i \notin \text{IC}_Q$ and $i \notin \text{IC}_P$.

Consider the case when $i \in \text{dom}(I_P)$. Based on the assumption that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$, we know that $\text{last}(\tau_c)_S[i, n] = \text{last}(\tau_p)_S[i, n]$ and $\text{last}(\tau_c)_B[i, n] = \text{last}(\tau_p)_B[i, n]$. Hence, if $P||Q$ takes the visible Input-Create transition $i$, when in global configuration $\text{last}(\tau_c)$, then the same transition can be taken by module $P$ in configuration $\text{last}(\tau_p)$. $i \in \Sigma_Q$ (we know that $i \notin (\text{dom}(I_Q) \cup \text{IC}_Q)$), hence $Q$ takes the Environment-Create transition. The resultant executions $\tau'_c, \tau'_p$ and $\tau'_q$ satisfy the condition that $\text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau'_q))$. Also, $\text{trace}(\tau'_c)[\Sigma_P] = \text{trace}(\tau'_p)[\Sigma_P]$ and $\text{trace}(\tau'_c)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q]$ since all modules took the same labeled transition.

The same analysis can be applied to the case when $i \in \text{dom}(I_Q)$.

3. **Output-Create-1**
   This is the case when a machine instance $(i', n) \in \text{dom}(\text{last}(\tau_c)_S)$ creates an interface $i$ and $i \notin \text{dom}(I_{P||Q})$ which means that interface $i$ is implemented by some machine in the environment of $P||Q$.

Consider the case when $i' \in \text{dom}(I_P)$ (which implies that $i' \notin \text{dom}(I_Q)$), since $i \notin \text{dom}(I_{P||Q})$ we know that $i \notin \text{dom}(I_P)$ and $i \notin \text{dom}(I_Q)$. 
Based on the assumption that \( \text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q)) \) we know that 
\[ S[\tau_c][i, n] = S[\tau_p][i, n] \] and 
\[ B[\tau_c][i, n] = B[\tau_p][i, n] \] and hence if \( P||Q \) takes the visible OUTPUT-Create-1 transition when in global configuration \( \text{last}(\tau, k) \), the same transition can be taken by module \( P \) in configuration \( \text{last}(\tau_p k') \).

\[ i \in \Sigma_Q \] and hence the environment of module \( Q \) creates an interface \( i \) (ENVIRONMENT-CREATE) and the resultant executions satisfy the condition that \( \text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau'_q)) \).

4. OUTPUT-CREATE-2

Similar analysis can be applied to prove that our inductive hypothesis holds when the composed module \( P||Q \) takes an OUTPUT-CREATE-2 transition.

\[ \square \]

**Lemma B.3**

For every pair of executions \( \tau_p \in \text{execs}(P) \) and \( \tau_q \in \text{execs}(Q) \), if there exists \( \sigma \in \Sigma_{P||Q}^* \) such that \( \sigma[\Sigma_p] = \text{trace}(\tau_p)[\Sigma_p] \) and \( \sigma[\Sigma_Q] = \text{trace}(\tau_q)[\Sigma_Q] \), then there exists an execution \( \tau_c \in \text{execs}(P||Q) \) such that \( \text{trace}(\tau_c)[\Sigma_{P||Q}] = \sigma \).

**Proof.** Given a pair of executions \((p, q)\) and \((p', q')\), we define a partial order over pair of executions as \((p, q) \preceq (p', q')\) iff \( p \) is a prefix of \( p' \) and \( q \) is a prefix of \( q' \). We perform induction over the pair of executions of module \( P \) and \( Q \) using the partial order.

**Inductive Hypothesis:** For any pair of executions \((\tau_p, \tau_q)\) of modules \( P \) and \( Q \) respectively, if there exists \( \sigma \in \Sigma_{P||Q}^* \) such that \( \sigma[\Sigma_p] = \text{trace}(\tau_p)[\Sigma_p] \) and \( \sigma[\Sigma_Q] = \text{trace}(\tau_q)[\Sigma_Q] \) then there exists an execution \( \tau_c \in \text{execs}(P||Q) \) such that \( \text{trace}(\tau_c)[\Sigma_{P||Q}] = \sigma \) and \( \text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q)) \).

**Base case:** The inductive hypothesis hold trivially for the base case when the length of the executions \( \tau_p, \tau_q \) of modules \( P, Q \) is zero.

\( \text{trace}(\tau_p)[\Sigma_p] = \text{trace}(\tau_q)[\Sigma_p] = \epsilon \) (\( \epsilon \in \Sigma_{P||Q}^* \)).

We know that: there exists \( \tau_p = (S_0^p, B_0^p, C_0^p) \in \text{execs}(P) \), there exists \( \tau_q = (S_0^q, B_0^q, C_0^q) \in \text{execs}(Q) \) and there exists \( \tau_c = (S_0^c, B_0^c, C_0^c) \in \text{execs}(P||Q) \).

Hence, there exists an execution \( \tau_c \in \text{execs}(P||Q) \) such that \( \text{trace}(\tau_c)[\Sigma_{P||Q}] = \epsilon \).

Finally, we have \( \text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q)) \) as:

- \( S_0^c = S_0^p = S_0^q = S_0 \) (empty map)
- \( B_0^c = B_0^p = B_0^q = B_0 \) (empty map)
- \( C_0^c = C_0^p = C_0^q = C_0 \) (all elements map to 0)

**Inductive case:** Let us assume that the hypothesis holds for any pair of executions \((\tau_p, \tau_q)\) and any \( \sigma \). To prove that the hypothesis is inductive, we show that the hypothesis holds for the next pair of executions in the partial order \((\tau'_p, \tau'_q), (\tau_p, \tau_q)\) and \((\tau'_p, \tau'_q)\) where \( \tau'_p = \tau_p \xrightarrow{a} G' \), \( \tau'_q = \tau_q \xrightarrow{a} G'' \) and \( \tau'_c = \tau_c \xrightarrow{a} G''' \).

Just to provide an intuition, \((\tau'_p, \tau'_q)\) represents the case when module \( P \) takes a transition with label \( a \) and \( a \notin \Sigma_Q \), similarly \((\tau_p, \tau'_q)\) represents the case when module \( Q \) takes a transition with label \( a \) and \( a \notin \Sigma_P \). \((\tau'_p, \tau'_q)\) represents the case when module \( P \) and \( Q \) both take transition with label \( a \), as \( a \in \Sigma_P, a \in \Sigma_Q \).
We perform case analysis for all possible transitions taken by module $P$ and module $Q$. We provide a proof for one such case:

1. Let us consider the case when module $P$ takes a transition OUTPUT-SEND-1 with label $a = ((i_t, n_t), e, v)$. Let $(i, n) \in \text{dom}(\text{last}(\tau_P))$ be the machine that takes this transition. Hence, $\sigma' = \sigma[a]$ and $\text{trace}(\tau_Q'[\Sigma_P]) = \sigma'[\Sigma_P]$. Let us consider the case when $i_t \in \text{dom}(I(Q))$, and $e \in M\text{Recs}(i_t) \setminus (E_P \cup E_Q)$ (input event of $Q$). Based on the assumption that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$ and the invariants I1-I6 about the state configurations, we know that $(i_t, n_t) \in \text{dom}(\text{last}(\tau_q))$. Hence, $Q$ can take an INPUT-SEND transition with label $a = ((i_t, n_t), e, v)$ and therefore $\text{trace}(\tau_Q'[\Sigma_Q]) = \sigma'[\Sigma_Q]$.

Finally, using same assumption $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$ and the invariants I1-I6, the composed module $P||Q$ can take the transition OUTPUT-SEND-2 with the same label $a = ((i_t, n_t), e, v)$. Hence, $\text{trace}(\tau_Q'[\Sigma_P||Q]) = \sigma'$. The resultant executions still satisfy the condition that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p'), \text{last}(\tau_q'))$.

Note: Proving that executions of modules satisfy the property $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))$ helps us prove a stronger property than what is needed for the lemma.

Similar analysis was performed for all possible transitions taken by modules $P$ and $Q$. □

**Lemma B.4: Composition preserves refinement**

Let $P$, $Q$, and $R$ be three modules such that $P||Q$ and $R$ are composable. Then the following holds: (1) $P||R \preceq P$ and (2) $P \preceq Q$ implies that $P||R \preceq Q||R$

**Proof.** (1) follows directly from the Lemma B.1. For (2), let $\sigma$ be a trace of $P||R$, then we know that $\sigma[\Sigma_P]$ is a trace of $P$ and $\sigma[\Sigma_R]$ is a trace of $R$. We know that, $P \preceq Q$ therefore $\sigma[Q]$ is a trace of $Q$ and using the Lemma B.1 $\sigma[\Sigma_Q||R]$ is a trace of $Q||R$. □

**Lemma B.5: Circular Assume-Guarantee**

Let $\|P\|$ and $\|Q\|$ be well-formed. Suppose for each module $Q \in Q$ there is a subset $X$ of $P \oplus Q$ such that $Q \subseteq X$, $\|X\|$ is well-formed, and $\|X\|$ refines $Q$. Then $\|P\|$ refines each module $Q \in Q$.

**Proof.** Definitions:

- Let $Q$ be a collection of ($n > 1$) composable modules represented by the set $\{Q_1, Q_2, ... Q_n\}$.
- Let $P$ be a collection of ($n' > 1$) composable modules represented by the set $\{P_1, P_2, ... P_{n'}\}$.

In this proof, we refer to $\|P\|$ (composition of all modules in $P$) as module $P$.

- Let $\forall k. X_k$ be a subset of $P \oplus Q$.

Let us assume that $\forall Q_k \in Q$ there exists a $X_k$ such that $X_k \subseteq Q_k$. 

---

We prove our inductive hypothesis by performing induction over the length of execution $\tau_P$.

- **Base case:** The base case is one where the length of execution $\tau_P$ is 0. The inductive hypothesis trivially holds for the base case.

- **Inductive case:** Let us assume that the inductive hypothesis holds for any execution $\tau_p \in \text{execs}(P)$ of length $k$. To prove that the hypothesis is inductive, we show that the hypothesis also holds for any execution $\tau'_p$ where $\tau'_p = \tau_p \xrightarrow{a} G$.

We have to perform the case analysis for all possible transition labels $a$. We provide a proof for some of these cases:

- $a = \epsilon$ (Invisible transition)  
  It can be easily seen that the inductive hypothesis holds for the case when the module $P$ takes an invisible transition.

- $a = i$ where $i \in I$ (creation of an interface)  
  $a$ can be equal to $i$ because of any of the following cases: (1) module $P$ creates an interface using the transitions: OUTPUT-CREATE-1, OUTPUT-CREATE-2 or (2) the environment creates it using the transitions: ENVIRONMENT-CREATE, INPUT-CREATE.

Let us consider the case when $a = i$ because $P$ executes the OUTPUT-CREATE-1 transition. Recollect that $P$ is a composition of modules $P_1, P_2, ... P'_n$. Using Lemma B.2, we can decompose the execution $\tau_P$ of module $P$ ($\tau_P \in \text{execs}(P)$) into the executions $\tau_{P_1}, \tau_{P_2}, ...$ of the component modules such that for all $P_k \in P$, $\text{trace}(\tau_P)[\Sigma_{P_k}] = \text{trace}(\tau_{P_k})[\Sigma_{P_k}]$.

From the operational semantics of OUTPUT-CREATE-1, we know that $i \in IC_P$ and $i \notin \text{dom}(I_P)$.

Let us consider the case when there exists a module $P_k \in P$ such that $i \in IC_{P_k}$, and from the definition of composition we know that $\forall j, j \neq k, i \notin IC_{P_j}$.

If $\exists j$, s.t. $j \neq k \land i \in \Sigma_{P_j}$ then $P_j$ can take the ENVIRONMENT-CREATE transition to match the visible action $a = i$.

If $i \in IC_Q$, then for some $Q_k \in Q, i \in IC_{Q_k} - (1)$.

If $\forall Q_k \in Q_k, i \notin IC_Q$, then all $Q_k$ can take the ENVIRONMENT-CREATE transition to match the visible action $a = i$.

Let us consider the case when only (1) is true. Since $i \in IC_{Q_k}$ and $X_k \leq Q_k$ we have $i \in IC_{X_k}$.

Note that $P$ and $Q$ are well formed modules. Since (1) $Q_k \notin X_k$ (2) $\forall j, j \neq k. i \notin IC_{P_j} \land i \notin IC_{Q_j}$, we know that $P_k \in X_k$.

Using Lemma B.3, and the fact that $X_k \leq Q_k$, we know that for any given $\tau'_{P_k} \in \text{execs}(P_k)$ there exist $\tau'_{Q_k}$ such that $\text{trace}(\tau'_{P_k})[\Sigma_{Q_k}] = \text{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$.

Finally, we know that:

1. Inductive hypothesis holds for any execution $\tau_P$ and $\tau'_P = \tau_P \xrightarrow{i} G$ (Output-CREATE-1)
2. $i \in IC_{Q_k}$ and $i \in IC_{P_k}$.
3. $\forall j, j \neq k. i \notin IC_{P_j}$ and $\forall j, j \neq k. i \notin IC_{Q_j}$.
4. there exists an execution $\tau'_{P_k} \in \text{execs}(P_k)$ such that $\text{trace}(\tau'_{P_k})[\Sigma_{P_k}] = \text{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$. 

5. there exists an execution $\tau'_{Q_k} \in \text{execs}(Q_k)$ such that $\text{trace}(\tau'_{P_k})[\Sigma_{Q_k}] = \text{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$

Hence, we can conclude that for the execution $\tau'_{P}$ there exists an execution $\tau'_{Q_k}$ such that $\text{trace}(\tau'_{P})[\Sigma_{Q_k}] = \text{trace}(\tau'_{Q_k})[\Sigma_{Q_k}]$.

And using (3), we also know that for all $Q_j \in Q_k$, $\text{trace}(\tau'_{Q_j})[\Sigma_{Q_j}] = \text{trace}(\tau_{Q_j})[\Sigma_{Q_j}]$.

Hence, the inductive hypothesis holds for the execution $\tau'_{P}$.

We do similar analysis to prove the other cases.

\[\square\]

**Lemma B.6: Compositional Safety Analysis**

Let $\parallel P$ and $\parallel Q$ be well-formed. Let $\parallel P$ refine each module $Q \in Q$. Suppose for each $P \in P$, there is a subset $X$ of $P \oplus Q$ such that $P \in X$, $\parallel X$ is well-formed, and $\parallel X$ is safe. Then $\parallel P$ is safe.

**Proof.** We describe a proof strategy using contradiction for a simplified system consisting of two implementation modules $P_1, P_2$ and two abstraction modules $Q_1, Q_2$. For such a system, the theorem states that if $P_1 \parallel P_2 \leq Q_1, P_1 \parallel P_2 \leq Q_2$ and $P_1 \parallel Q_2, Q_1 \parallel P_2$ are safe then $P_1 \parallel P_2$ is safe.

Let's say that there exists an error execution in $\tau^e$ in $P_1 \parallel P_2$. Using the compositional refinement Lemma, we can decompose the execution $\tau^e$ into $\tau^e_1$ of $P_1$ and $\tau^e_2$ of $P_2$. Let's say the error was because of module $P_1$ taking a transition and hence $\tau^e_1$ is an error trace.

We know that $P_1 \parallel Q_2$ is safe which means that for all executions of module $P_1 \parallel Q_2$ there is no execution of $P_1$ that is equal to $\tau^e_1$ after decomposition.

The above condition also implies that in the composed module $P_1 \parallel P_2$, module $P_2$ using an output action is triggering an execution in $P_1$ which results in execution $\tau^e_1$. And this output action is not triggered by $Q_2$ in the composition $P_1 \parallel Q_2$.

The above condition implies that $P_1 \parallel P_2 \leq Q_2$ does not hold which is a contradiction.

We generalized this proof strategy for proving the given lemma.

\[\square\]

**Lemma B.7: Hide Event Preserves Refinement**

For all well-formed modules $P$ and $Q$ and a set of events $\alpha$, if $(\text{hide } \alpha \text{ in } P)$ and $(\text{hide } \alpha \text{ in } Q)$ are well-formed, then (1) $P \leq (\text{hide } \alpha \text{ in } P)$ and (2) if $P \leq Q$, then $(\text{hide } \alpha \text{ in } P) \leq (\text{hide } \alpha \text{ in } Q)$.

**Proof.** Let $hP = (\text{hide } \alpha \text{ in } P)$ and $hQ = (\text{hide } \alpha \text{ in } Q)$.

We perform induction over the length of execution $\tau_{hP}$ of module $hP$.

**Inductive Hypothesis:** For every execution $\tau_p \in \text{execs}(hP)$, there exists an execution $\tau_p \in \text{execs}(hQ)$ such that $\text{trace}(\tau_{hP})[\Sigma_{hQ}] = \text{trace}(\tau_{hQ})[\Sigma_{hQ}]$.

We prove our inductive hypothesis by performing induction over the length of execution $\tau_p$.

- **Base case:** The base case is trivially satisfied by an execution of length zero.
- **Inductive case:** Let us assume that the hypothesis holds for any execution $\tau_{hP} \in \text{execs}(hP)$ and the corresponding execution of module $hQ$ be $\tau_{hQ} \in \text{execs}(hQ)$.
To prove that the hypothesis is inductive we show that it also holds for the execution 
\( \tau'_{hP} \in \text{execs}(hP) \) where \( \tau'_{hP} = \tau_{hP} \rightarrow G \) and \( \tau'_{hQ} \) be the resultant executions of \( hQ \). 
Hide operation only converts visible actions into internal actions. Hence, it can be easily shown that any execution of \( hP \) is also an execution of \( P \), similarly for module \( hQ \) and \( Q \), every execution of \( hQ \) is an execution of \( Q \).
The above property, along with the fact that \( P \leq Q \) helps us conclude that the inductive hypothesis always holds.

**Lemma B.8: Hide Interface Preserves Refinement**

For all well-formed modules \( P \) and \( Q \) and a set of interfaces \( \alpha \), if \( (\text{hide } \alpha \text{ in } P) \) and \( (\text{hide } \alpha \text{ in } Q) \) are well-formed, then (1) \( P \leq (\text{hide } \alpha \text{ in } P) \) and (2) if \( P \leq Q \), then \( (\text{hide } \alpha \text{ in } P) \leq (\text{hide } \alpha \text{ in } Q) \).

**Proof.** The proof is similar to the proof for Lemma B.7.

\[ \square \]