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-2017-163
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-163.html

November 21, 2017
Compositional Programming and Testing of Dynamic Distributed Systems

Ankush Desai  
University of California, Berkeley

Shaz Qadeer  
Microsoft Research, Redmond

Amar Phanishayee  
Microsoft Research, Redmond

Sanjit A. Seshia  
University of California, Berkeley

Abstract

Distributed systems are notoriously difficult to get right as they must deal with 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 created dynamically 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) reasoning of distributed systems.

We demonstrate the efficacy of our methodology by building two practical distributed systems, a fault-tolerant transaction commit service and fault-tolerant distributed data structures. Our framework helps implement these systems modularly and validate them via compositional systematic testing. We empirically demonstrate that using abstractions based compositional reasoning helps amplify the coverage during testing. We show that monolithic testing cannot find more than half the critical bugs found using ModP within equivalent time budgets.

1 Introduction

Distributed systems are notoriously hard to get right as they must correctly handle both concurrency and failures. Programming these systems is challenging because of the need to reason about numerous control paths resulting from the myriad interleaving of event handlers and unexpected failures. Unsurprisingly, it is easy to introduce subtle errors while improvising to fill gaps between abstract protocol descriptions and their concrete implementations. These problems have been highlighted by creators of large-scale distributed systems (e.g., [15]).

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 [16, 37, 45, 52, 69, 76, 80] and distributed systems [36, 60, 77]. The central correctness argument for proving a safety property on a distributed system is 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 [34, 44, 79]. 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.

In this paper, we present ModP, a framework for programming and testing dynamic distributed systems. ModP introduces a module system that provides constructs to build the distributed system modularly while providing abstractions for decomposition. ModP then validates each decomposed problem automatically in isolation using systematic testing. When the whole-system testing problem is decomposed into a collection of simpler testing problems, one can scale to large, industrial-scale implementations. Moreover, the results of localized testing can be lifted to the whole system level by leveraging the theory of assume-guarantee (AG) reasoning [1, 4, 55]. AG reasoning has been implemented in model checkers [5, 54, 56] and successfully used for hardware verification [24, 39, 55] and software testing [10, 73]. However, the present paper is the first to apply it to distributed systems of considerable complexity and dynamic behavior.

ModP occupies a spot between proofs and black-box monolithic testing in terms of the trade-off between validation coverage and programmer effort.

Our goal is to exploit assume-guarantee principles to decompose the testing of a distributed system’s software stack as shown in Figure 1. This figure 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 [33], and (ii) replicated data structures such as hashtables and lists. These services use State Machine Replication (SMR) for fault-tolerance. Protocols for SMR, such as Multi-Paxos [47] and Chain Replication [75], in turn use other protocols like leader election, failure detectors, and network channels.
The composition of these interacting complex protocols with failure scenarios leads to explosion of behaviors which makes systematic testing prohibitively expensive. The AG approach would be to test each of the sub-protocol (green block) 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.

Actors [2, 3, 7, 12, 63] and state machines [20, 35, 43] are a popular programming paradigm for building distributed systems. The state machine (or actor) based programming model is popular for implementing distributed services as they 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). These dynamic features, in turn, have an important impact on AG reasoning, which typically relies on having clear interfaces between components – e.g., wires between circuits or shared variables between programs [4, 53]. In dynamic distributed systems, the interfaces between modules can change dynamically as new state machines are instantiated or communication topology changes and this dynamic behavior depends on the context of a module. While there are a few formalisms for AG reasoning [8, 27] that 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 supports a module system that enables assume-guarantee reasoning of dynamic communicating state-machines (actors). A module in ModP is a collection of dynamically-created and concurrently-executing state machines with a semantics that 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 also provides operators for hiding events of a module, to construct a more abstract module. To ensure composition is intersection and that hiding events lead to sound abstractions, an especially challenging problem in a language where permission (machine-reference) to send events flows dynamically through, we use a methodology based on permission based capabilities control [38, 61, 65]. Finally, ModP introduces a notion of interfaces as a proxy for state machines. Instead of creating state machines directly, our programming methodology requires creating a machine indirectly as an instantiation of an interface, with the binding from interface to machines specified explicitly by the programmer. Separating the specification of the interface binding from the code that instantiates it allows great flexibility in specializing machines and substituting one machine for another.

We have implemented ModP on top of P [20], 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 [20] and for programming safe robotics systems [19, 22]; P# [18], a library extension to .NET based on P is being used to implement distributed services in Azure. The ModP compiler generates C# code for compositional systematic testing. The ModP systematic testing engine implements search prioritization techniques [21] for systematic testing of actor programs and also supports refinement checking based on trace containment.

To evaluate ModP, we implemented each sub-protocol (green blocks) in Figure 1 as a separate module and performed compositional reasoning at each layer of the protocol stack. Such a compositional approach enables clean separation of concerns and a modular development process. Using compositional systematic testing we were able to find several (70+) critical bugs in our implementation. We empirically demonstrate that using abstractions helps in amplifying test-coverage; in fact, monolithic testing based on scalable concurrency testing techniques [21] could not find more than half the bugs found using ModP within equivalent time budgets. Further, our approach for checking refinement through testing is effective in finding errors in module abstractions.

We conclude with a summary of our contributions:

1. We present a new theory of compositional refinement and a module system for 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; our approach amplifies test coverage and finds bugs faster than monolithic testing.

Figure 1. Fault-tolerant Distributed Services Software Stack

<table>
<thead>
<tr>
<th>Transaction Commit Service</th>
<th>Data Structure Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two Phase Commit</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>HashTable</td>
</tr>
<tr>
<td>Fault Tolerance using State Machine Replication (SMR)</td>
<td>Chain Replication based SMR</td>
</tr>
<tr>
<td>Multi Paxos</td>
<td>Chain Replication</td>
</tr>
<tr>
<td>Multi Paxos</td>
<td>Fault Detector</td>
</tr>
<tr>
<td>Leader Election</td>
<td>Master Protocol</td>
</tr>
<tr>
<td>(with view membership)</td>
<td>(with view membership)</td>
</tr>
<tr>
<td>Timer</td>
<td>Storage</td>
</tr>
<tr>
<td>Operating System and other services</td>
<td>Network</td>
</tr>
</tbody>
</table>
2 Model of Computation

ModP builds on top of P [20], an actor [2] oriented programming language with the modification that actors in P are implemented as state machines. A P 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 local store. The machines run concurrently, receiving and sending events, creating new machines, and updating their local store.

There are three key features that lead to dynamism in this model of computation, and makes compositional reasoning challenging: (1) new: This operation creates an instance of a machine from its behavior description and returns a reference to the newly-created machine. Machines can be created dynamically during the execution of the program. (2) send: This operation sends an event from one machine to another using the reference of the receiver. Sends are buffered, non-blocking, and directed. (3) machine-references: 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. A programming model used for building distributed systems must support feature (1) because processes may fail at runtime and new processes have to be created for maintaining service availability. Features (2) and (3) are required because broadcast communication can be prohibitively expensive in distributed services.

In this section, we describe the extensions ModP makes to P state machines in terms of syntactic constructs and semantics. We will refer to them as ModP state machines.

2.1 ModP State Machines

A module in ModP is a collection of dynamic instances of ModP state machines, the extensions to P state machines are required for defining the operational semantics of ModP modules (Section 3) and making them amenable to compositional reasoning.

interface I1 receives E1;
interface I2 receives E3;
interface I3 receives E4,E6;
interface I4 receives E1,E2;

machine M1 receives E1, E2;
sends E2, E3; creates I2;
{ /* body */ }

machine M2 receives E1, E3;
sends E4; creates I3;
{ /* body */ }

machine M3 receives E4, E6;
sends E1; creates I4;
{ /* body */ }

Figure 2. Declarations of Machines and Interfaces

First, we add interfaces that are symbolic names for machines. Each interface declaration (Figure 2) has an interface name and a set of received events. For example, interface I1 is willing to receive only event E1 but interface I4 is willing to receive E1 and E2. 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. The execution of the statement id := new I creates a fresh instance of machine M and stores an unique identifier representing the new machine instance in id. The link between interface I and machine M is provided separately by the programmer. This indirection and the use of interfaces is critical for enabling the key feature of substitution during compositional reasoning, the ability to replace the implementation machine with its abstraction. For example, linking I to M in implementation but to its abstraction MA during testing.

Second, we extend P machines with annotations declaring the set of receive, send and create actions the dynamic instance of that machine can perform. For example, an instance of a machine M1 (Figure 2) is willing to receive events E1 or E2, guarantees to send no event other than E2 or E3, and guarantees to create no interface other than I2. These annotations are used to statically infer the actions a module can perform based on the actions of its comprising machines.

Finally, in P, executing the statement send t, e, v adds event e with payload value v into the buffer of the target machine t. We extend the semantics of send to provide the guarantee that a ModP state machines can never receive an event (from any other machine) that is not listed in its receive set. This is achieved by extending machine-identifier with permissions, described in more details in Section 2.3.

2.2 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 ($\mathcal{E}^2$). 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-store of a machine could have during execution. The local-store 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 an event in a send action.

Let $\mathcal{Z}$ be the set of all the machine identifiers. For our formalization, we are interested in machine identifiers that are embedded inside the data structures in a machine local-store $s \in \mathcal{S}$ or a value $v \in \mathcal{V}$. Instead of formalizing all datatypes in ModP, we assume the existence of a function $\text{id}_s$
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 \( v \). We use \( dom(x) \) and \( codom(x) \) to refer to the domain and codomain of any map \( x \).

A state machine in ModP is a tuple \((MRecvs, MSends, MCreates, Rem, Enq, New, Local)\) where: (1) \( MRecvs \subseteq E \) is the nonempty set of events received by the machine. (2) \( MSends \subseteq E \) is the set of all events sent by the machine. (3) \( MCreates \subseteq I \) is the set of interfaces created by the machine. (4) \( Rem \subseteq S \times B \times \mathbb{N} \times S \) is the transition relation for removing a message from the input buffer. If \((s, b, n, s') \in Rem\), then the \( n \)-th entry in the input buffer \( b \) is removed and the local state moves from \( s \) to \( s' \). (5) \( Enq \subseteq S \times Z \times E \times V \times S \) is the transition relation for sending a message to a machine. If \((s, i, e, v, s') \in Enq\), then event \( e \) with payload \( v \) is sent to machine \( id_i \) and the local state of the sender moves from \( s \) to \( s' \). (6) \( New \subseteq S \times I \times S \) is the transition relation for creating an interface. If \((s, i, s') \in New\), then the machine linked against interface \( i \) is created and the machine moves from \( s \) to \( s' \). (7) \( Local \subseteq S \times Z \times S \times S \) is the transition relation for local computation in the machine. The state of a machine is a pair \((s, id) \in S \times Z\). The first component \( s \) is the machine local-store. 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.

**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 \( m \in M \), \( s, s' \in S \), \( id, id' \in Z \), \( e \in E \), \( v \in V \), \( i \in I \), \( b \in B \), and \( n \in \mathbb{N} \):

1. \((s, id, s', id') \in Local(m) \Rightarrow ids(s') \subseteq ids(s) \cup \{id\}
2. \((s, b, n, s') \in 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 Enq(m) \Rightarrow ids(v) \cup ids(s') \subseteq ids(s)
4. \((s, i, s') \in New(m) \Rightarrow ids(s') \subseteq ids(s)\)

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

### 2.3 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 compositional reasoning of communicating state machines using our module system: (1) *machines 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 4); (2) *the ability to control the flow of capability to send certain event to a machine*, this property is required to ensure that the capability to send a private (internal) event of a module does not leak outside the module (Section 5.3). 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 \( \pi \)-calculus [38, 61, 65] where permissions are associated with channels or locations and enforced using type-systems.

We concretize the set of machine identifiers \( Z \) as \( I \times \mathbb{N} \times K \). An identifier \((i, n, \alpha) \in Z \) refers to the \( n \)-th instance of interface represented by \( i \in I \). We refer to the final component \( \alpha \) of a machine identifier as its *permissions*. This set represents all events that may be sent via this machine identifier using the \texttt{send} operation. The creation of an interface \( I \) returns a machine identifier \((I, n, \beta) \in Z \) referencing the \( n \)-th instance of interface \( I \) where \( \beta \) represents the receive set associated with the interface \( I \) \((\beta = IRecvs(I))\). The ModP compiler checks that if an interface \( I \) is bound to \( M \), the received events of \( I \) are contained in the received events of \( M \) \((IRecvs(I) \subseteq MRecvs(M))\). 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 (we refer to this property as \texttt{ReceiveOk}). Note that the permissions embedded in a machine identifier controls the capabilities associated with that identifier.

In order to control the flow of these capabilities, ModP require the programmer to annotate each event with a set \( A \in \mathcal{K} \) of allowed permissions. For an event \( e \), the set \( A(e) \) represents any permission that the programmer can put in inside the payload accompanying \( e \) i.e., if \( v \) represents any legal payload value with \( e \) then \( \forall (e, \alpha) \in A(e), \alpha \in A(e) \). 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 \texttt{send} \( t, e, v \) succeeds only if \( e \) is in the permissions of machine identifier \( t \), and all permissions embedded in \( v \) are in \( A(e) \), the send fails otherwise (we refer to this specification as \texttt{SendOk}). This changed semantics of \texttt{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 5.3).

### 3 Modules

ModP seeks to manage the complexity of a distributed system by designing it in a structured way, at different levels of abstractions and modularly as the composition of interacting simpler modules. Figure 3 presents the expression language supported by ModP module system for module construction.

The \texttt{bind} constructor creates a primitive module as a collection of machines \( m_1, \ldots, m_k \) bound to interfaces \( l_1, \ldots, l_k \)
respectively. 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 resolve 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 when writing proof obligations for compositional reasoning (Section 6).

The module constructors and static typing rules for well-formedness of the constructed module are described in Section 5. In ModP module system, a module P is a syntactic expression and its well-formedness is checked using the judgment P ⥵ EFP, IIP, IP, LP, ERP, ESP, ICP. If module P satisfies the judgment then we read it as: Module P is well-formed with private events EFP, private interfaces IIP, interface definition map IP, interface link map LP, events received ESP, events sent ESP, and interfaces created ICP. The judgment derives the components on the right hand side which are used for defining the operational semantics of a well-formed module (Section 4). We next describe the components on the right hand side of the judgment:

1. \( EFP \in 2^I \) 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 ∈ EFP, then the target must be some machine in P and, if a machine in P receives e ∈ EFP, the sender must be some machine in P. The sending of a private event is an internal (invisible) action of a module.

2. \( IIP \in 2^I \) represent the interfaces that are declared private in P; the creation of any interface in IIP is an internal (invisible) action of P.

3. \( IP \in I \to M \) is the interface definition map that binds an interface name i to a machine name \( LP[i] \). Recollect that in ModP model of computation dynamic instances of machines are created indirectly using interfaces. An interface definition map \( (IP) \) is a collection of bindings from interface names to machine names. These bindings are initialized using the bind operation, so that if \( (i,m) \in IP \) then creation of an interface i in module P leads to creation of an instance of machine m.

4. \( LP \in I \to I \to I \) is the interface link map that maps each interface i ∈ dom(IP) to a machine link map that binds interfaces created by the code of machine \( LP[i] \) to an interface name. If the statement \{new x\} is executed by an instance of machine \( LP[i] \), an interface actually created in lieu of the interface name x is provided by the machine specific link map \( LP[i] \). If \( (x,x') \in LP[i] \), then the compiler interprets x in statement \{new x\} in the code of machine \( LP[i] \) as creation of interface x', creating an instance of machine \( LP[x'] \).

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

5. \( ERP \in 2^C \) represent the set of events received by module P. It is inferred as the set of non-private events received by machines in P, ERP = ∪\( m ∈ \text{codom}(IP) \) MRecvs(m) \( \setminus \) EFP.

6. \( ESP \in 2^C \) represent the set of events sent by module P. It is inferred as the set of non-private events sent by machines in P, ESP = ∪\( m ∈ \text{codom}(IP) \) MSends(m) \( \setminus \) EFP.

7. \( ICP \in 2^C \) represent the set of interfaces created by module P. It is the set of interfaces created by machines in P (interpreted based on the link map), ICP = ∪\( i,m ∈ LP, x ∈ MC\text{reated}(m) \) \{LP[i][x]\}.

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

4 Operational Semantics

A key requirement for assume-guarantee reasoning [5, 53] is to consider each component as an open system that continuously reacts to input that arrive from its environment and generates outputs. 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.

In this section, we present the open system semantics of a well-formed module P as a labeled transition system. The input events of module P are the events that are received but not sent by P i.e. ERP \( \setminus \) ESP. The input interfaces of P are the set of interfaces that are exported but not created by P i.e. dom(IP) \( \setminus \) (IIP \( \cup \) ICP). The output events of P are the sent events i.e. ESP and the output interfaces are the created non-private interfaces of P i.e. ICP \( \cup \) IIP. We formally define the executions, traces, and visible input-output signature of a module. The refinement relation between modules is based on trace containment. We will refer to derived components on the right hand side of the judgment when defining the operational semantics of a well-formed module P.

Configuration. A configuration of a module is a tuple \((S, B, C)\): (1) The first component S is a partial map from \( I \times N \) to \( S \times Z \). If \((i,n) \in \text{dom}(S)\), then \( S[i,n] \) is the state of the n-th instance of machine \( LP[i] \). The state \( S[i,n] \) has two components, local store \( s ∈ S \) and a machine identifier \( id ∈ Z \) (as described in Section 2.2). (2) The second component B is a partial map from \( I \times N \) to \( B \). If \((i,n) \in \text{dom}(B)\), then \( B[i,n] \) is the input buffer of the n-th instance of the machine \( LP[i] \). (3) The third component C is a map from I
to \( \mathbb{N} \). \( C[i] = n \) means that there are \( n \) dynamically created instances of interface \( i \).

Let \( (S_P, B_P, C_P) \) represent the configuration for a module \( P \). The initial configuration of any module \( P \) is defined as \( (S_P^0, B_P^0, C_P^0) \) where \( S_P^0 \) and \( B_P^0 \) are empty maps, and \( C_P^0 \) maps each element in its domain \( (I) \) to 0.

Figure 4 presents the operational semantics of a well-formed \( P \) as a transition relation over its configurations. A transition is represented as \( (S, B, C) \xrightarrow{a} (S', B', C') \cup \{ \text{error} \} \) where \( a \) is a label on the transition indicating the type of step being taken.

**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 \( (s, b, \text{pos}, s') \in \text{Rem}(I_P[i]) \), the entry in \( \text{pos}-\text{th} \) 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 these rules, \( I_P \) is used to look-up the concrete 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 \( (s_0, \text{id}) \) where \( \text{id} \) 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 safe since rules of module composition forbid composing \( P \) with another module that creates an interface in common with \( P \).

The rule Create-Bad creates a transition into \( \text{error} \) if the interface \( i'' \) being created by machine \( (i, n) \) violates the predicate \( CreateOk(m, x) = x \in MCreate(m) \). Thus, machine \( (i, n) \) may only create machines in \( MCreate(I_P[i]) \). The rule Output-Create-Outside allows machine \( (i, n) \) to create an interface \( i'' \) that is not implemented inside \( P \), indicated by \( i'' \notin \text{dom}(I_P) \). Create of interface \( i'' \) will get bound to an appropriate machine when \( P \) is composed with another module \( Q \) that has binding for \( i'' \). 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 \( P \). A well-formed module satisfies the following property:

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

This property together with the property that machines cannot create identifiers out of thin air (Section 2.2) 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 Create-Private allows the creation of a private interface in \( P \) (checked by \( i'' \in I_P[i] \)); consequently, the transition is labeled with \( e \) and is an internal transition.

In the rules Create-Bad, Output-Create-Outside, Output-Create-Inside and Create-Private, the link map \( (L_P) \) is used to look up the interface \( i'' \) to be created corresponding to \( \text{new} \ i'' \). The following property holds for any well-formed module and guarantees that this look up always succeeds.

**WF2** Interface link map is consistent: The domains of \( I_P \) and \( L_P \) are identical and for all \( (i, m) \in I_P \) and \( x \in MCreate(m) \), we have \( x \in \text{dom}(L_P[i]) \).

**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 four are triggered by \( P \) itself. The rule Input-Send enqueues a pair \( (e, v) \) into \( n \)-th instance of interface \( i \) 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 permissions. An event that may be enqueued into a machine in \( P \) is restricted by several rules. First, an event that is sent by \( P \) cannot be considered as an input event, which is safe since rules of module composition forbid composing \( P \) with another module that sends an event in common with \( P \). Second, only an event in the receive set of a machine is considered as an input event. As we discussed earlier, the environment 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 receive set of the machine referenced by it. Finally, private events or payload values with private permissions in them are not considered as input because permission to send a private event cannot leak out of a well-formed module, a property formally stated as follows.

**WF3** Permissions to send private events does not leak:

For all \( e \in ER_P \cup ES_P \) and \( \alpha \in A(e) \), we have \( \alpha \cap EP_P = \emptyset \).

Before executing a send statement the target machine identifier is loaded in local-store represented by \( id_t \) using
the Internal transition. The rule Send-Bad creates a transition into error if the sender module, m, the permission in the target machine identifier α, event e and payload v violates the predicate SendOk(m, α, e, v) = e ∈ M Sends(m) \α \land e ∈ \alpha ∧ ∀(α, β) ∈ ids(v). The machine (i, n) may only send events declared by it in M Sends(m) and allowed by the permission α of the target machine and should not embed machine identifiers with private permissions in the payload v. Note that the dynamic check SendOk helps guarantee the wellformedness rule WF3 and also ensure that a module receives only those events from other modules that are its input events (and is expected to be receptive against). The rule Output-Send-Outside sends an event to machine outside P whereas rules Output-Send-Inside and Send-Private send an event to some machine inside P. In the former, the target machine m, is not in the domain of IP, whereas in the later cases the target machine is inside the module and hence present in domain of IP. For Send-Private, the label on the transition is e as a private event is sent.

Using the transition relation of a module, we now define executions, traces, and refinement. For brevity, we will refer to a configuration (Sk, Bk, Ck) as Gk.

**Definition 4.1 (Execution).** An execution of P is a finite sequence Go α0 −→ G1 α1 −→ ... αn−1 −→ Gn for some n ∈ N such that Gk αi −→ Gk+1 for each i ∈ {0, n}.

Let \text{exec}(P) represents the set of all possible executions of the module P. An execution is unsafe if Gn αn −→ error; otherwise, it is safe. The module P is safe, if for all τ ∈ \text{exec}(P), τ is a safe execution. The module P is unsafe if it is not safe.

**Definition 4.2 (Safe Module).** The module P is safe, if for all τ ∈ \text{exec}(P), τ is a safe execution. The module P is unsafe if it is not safe.

**Theorem 4.1 (Invariants for Executions of a Module).** Let P be a well formed module. For any execution τ ∈ \text{exec}(P) where τ is a sequence of global configurations G0 α0 −→ G1 α1 −→ ... −→ αn−1 −→ Gn, all global configurations Gτ satisfy the invariants:

11. \text{dom}(Sk) = \text{dom}(Bk)
12. ∀(i, n) ∈ \text{dom}(Sk), i ∈ \text{dom}(lp\tau) ∧ n < C[i]
13. ∀i ∈ \text{dom}(lp\tau). C[i] = \text{card}(n \{ (i, n) \in \text{dom}(Bk) \})
14. ∀(x, n, α) ∈ ids(Sk) ∪ ids(Bk), x ∈ \text{dom}(lp\tau) ⇒ (x, n) ∈ \text{dom}(Bk)
15. ∀(x, n, α) ∈ ids(Sk) ∪ ids(Bk), n ∈ Ck[x]
Definition 4.3 (Traces). Given an execution $\tau = G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n$ of $P$, the trace of $\tau$ is the sequence $\sigma$ obtained by removing occurrences of $\epsilon$ from the sequence $a_0, a_1, \ldots, a_{n-1}$.

Let $\text{traces}(P)$ represents the set of all possible traces of $P$. The set of traces of $P$ is the set $\{\beta \mid \exists \alpha \in \text{execs}(P) \land \beta = \text{trace}(\alpha)\}$. The transition with label not equal to $\epsilon$ is an externally visible or observable transition of the module. The signature of a module $P$ is the set of labels corresponding to all externally visible transitions in executions of $P$.

Definition 4.4 (Module-Signature). The signature of a module $P$ is the set $\Sigma_P = (I \setminus I_P) \cup ((I \times N) \times (E_S \cup E_R) \times V)$. The signature is partitioned into output signature $(IC_P \setminus I_P) \cup ((I \times N) \times E_S \times V)$ and input signature $(I \setminus IC_P) \cup ((I \times N) \times (E_R \setminus E_S) \times V)$.

The transitions in an execution labeled by elements of the output signature are driven by $P$ whereas transitions labeled by elements of input signature are driven by the environment of $P$.

Definition 4.5 (Trace Projection). If $\sigma \in \text{traces}(P)$ then $\sigma[\Sigma]$ represents the projection of trace $\sigma$ over the alphabet set $\Sigma$ where if $\sigma = a_0, a_1, \ldots, a_n$, then $\sigma[\Sigma]$ is the sequence obtained after removing all $a_i$ such that $a_i \notin \Sigma$.

Definition 4.6 (Refinement). The module $P$ refines the module $Q$, written $P \preceq Q$, if the following conditions hold:

1. $IC_Q \setminus I_P \subseteq IC_P \setminus I_P$,
2. $dom(I_Q) \setminus I_P \subseteq (dom(I_P) \cup IC_P) \setminus I_P$,
3. $ES_Q \subseteq ES_P$,
4. $ER_Q \subseteq ER_P \cup ES_P$.

Together imply that $\Sigma_Q \subseteq \Sigma_P$ (5) and for every trace $\sigma$ of $P$ the projection $\sigma[\Sigma_Q]$ is a trace of $Q$.

It is easy to check that every module $P$ refines itself, and that if $P \preceq Q$ and $Q \preceq R$, then $P \preceq R$.

5 Verifying Modules

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. For any $\pi \in \Sigma^*_P||Q$, $\pi \in \text{traces}(P||Q)$ if and only if $\pi[\Sigma_P] \in \text{traces}(P)$ and $\pi[\Sigma_Q] \in \text{traces}(Q)$.

Theorem 5.2 (Composition Preserves Refinement). Let $P$, $Q$, and $R$ be well-formed modules such that $P||Q$ and $P||R$ are well-formed. Then $R \preceq Q$ implies that $P||R \preceq P||Q$.

Theorem 5.2 states that parallel composition of modules 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 forms the basis of our theory of compositional refinement and is used for proving the theorems underlying our compositional testing methodology (Theorems 5.3 and 5.4).

In this section, we describe the various module constructors and present the static rules to ensure that the constructed modules satisfies: (1) well-formedness rules $WF_1-WF_3$ assumed when defining the semantics of a module, and (2) the compositionality Theorems 5.1-5.2. When describing the well-formedness rules, we do not present the derivation for the last three components of the judgment as they can be inferred but we use these components above the line.

5.1 Primitive Module

In ModP, a primitive module is constructed using the $\text{bind}$ operation. We use the machine and interface declarations in Figure 2 for our example. The following code creates a module $P$ that binds interfaces $I_1$ and $I_2$ to machines $M_1$ and $M_2$, respectively.

```
module P = bind I1 \rightarrow M1, I2 \rightarrow M2;
```

The ModP compiler checks that the set of received events of $I_1$ and $I_2$ are contained in the received events of $M_1$ and $M_2$, respectively (to ensure $WF_2$). In module $P$, the creation of interface $I_2$ in machine $M_1$ is bound to machine $M_2$. Programmatically initializing $I_P$ enables binding 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.

Rule $\text{Bind}$ presents the rule for $\text{bind}$ $i_1 \rightarrow m_1, \ldots, i_k \rightarrow m_k$ that constructs a primitive module by binding each interface $i_k$ to machine $m_k$ for $k \in [1, n]$.

```
(\text{Bind})

\begin{align*}
    f &= \{(i_1, m_1), \ldots, (i_n, m_n)\} \subset I \rightarrow M^{(n)} \\
    \forall (i, m) \in f. \text{IRecv}(i) \subseteq \text{MRecv}(m)^{(2)}
\end{align*}
```

These bindings are captured in $f$; condition (b) checks that $f$ is a function. Condition (c) checks that the received events of an interface are contained in the received events of the machine bound to it (ensure $WF_2$). 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 $f(i)$ (ensure $WF_1$). The first entry for name $x$ ever added to $I_P[i]$ is the identity map $(x, x)$; subsequently, if interface $x$ is renamed to $x'$ (using $\text{rename}$ constructor), this entry is updated to $(x, x')$. 

5.2 Module Composition

A large system is built by composing multiple modules together. Consider the following module definitions:

module $P = \text{bind} \ I_1 \rightarrow M_1, \ I_2 \rightarrow M_2$

module $Q = \text{bind} \ I_3 \rightarrow M_3$

module $R = P \parallel Q$

Here, module $P$ is borrowed from Section 5.1 and the definition of machines $M_1, M_2, \text{and} \ M_3$ comes from Figure 2. Module $R$ composes $P$ and $Q$. The composition of $P$ and $Q$ is allowed only if their exported interfaces are disjoint, which holds because the exported interfaces are $\{I_1, I_2\}$ and $\{I_3\}$, respectively. The interface definition map of $R$ is the union of the corresponding maps for $P$ and $Q$. The sent (received) events of $R$ is the union of sent (received) events of $P$ and $Q$.

Substitution. Note that the interface $I_3$ created by $P$ but not bound to any machine in $P$ is bound to machine $M3$ (from $Q$) in $R$. In $R$, creation of $I_3$ will lead to the creation of an instance of machine $M3$. The combination of interfaces (indirect machine creation) and the bind operation introduced in Section 5.1 helps in enabling substitution of a module with its abstraction. The module $\text{module} \ Q' = \text{bind} \ I_3 \rightarrow M_3'$ binds creation of $I_3$ to $M3'$. Let $Q'$ be abstraction of module $Q$, then substituting $Q$ with $Q'$, $R' = P \parallel Q'$, results in a module $R'$ where creation of $I_3$ leads to the creation of an instance of the abstract machine $M3'$.

Output disjointness. 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 (explained in [8]). Recall that when modeling $P$ as an open system (Section 4) we assume $P$ 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 any interface that is created by $P$ itself. Any input action of $P$ that is an output action of $Q$ is an output action of $R$ and hence not an input action of $R$. 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, composition is the mechanism used to introduce details about the environment of a component, which constrains its behaviors (composition is intersection), and ultimately allows us to establish safety properties of the component.

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

\[
\text{(Composition)} \quad (A \oplus B) \cup (B \oplus A) \quad P + EP, \ I_P, \ I_P, \ EP, \ IC, \ EQ, \ ICQ, \ \text{dom} (I_P) \cap \text{dom} (I_Q) = \emptyset^{(1)} \quad (EP \cup EQ) \cap (ICP \cup ICQ) \cap (ICP \cup ICQ) = \emptyset^{(2)}
\]

\[
\forall x \in (\text{dom}(I_P)) \cap \text{dom}(I_Q), \ \text{Rec}(x) \cap (EP \cup EQ) = \emptyset^{(3)} \quad I_P \cap IP = \emptyset^{(4)} \quad I_P \cup ICQ = \emptyset^{(5)}
\]

\[
P \parallel Q + EP \cup EQ, \ IP \cup IQ, \ IP \cup IQ, \ LP \cup LQ
\]

Rule Composition handles the composition of $P$ and $Q$. Condition (c) enforces that the domains of $I_P$ and $I_Q$ are disjoint, thus preventing conflicting interface bindings. Conditions (c2) ensures that input and output actions of $P$ are not hidden by private events of $Q$ and vice-versa. Conditions (c3) and (c4) together check that private permissions of $P \cup Q$ 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 (ensures WF3). The hide constructor introduces private events (and interfaces) in a module, we postpone further discussion about (c3) and (c4) to Section 5.3. Condition (c5) checks that created interfaces are disjoint; condition (c6) checks that sent events are disjoint. Composition is associative and commutative.

5.3 Hiding Events and Interfaces

Hiding events and interfaces in a module allows us to construct a more abstract module [8, 53]. To illustrate hiding of an event, consider the module definition below.

$H = \text{hide} \ E2 \text{ in } P$

To legally hide an event in module $P$, it must be both a send and received event of $P$. Module $H$ is well-formed and $E2$ becomes a private interface in $P$. A send of event $E2$ is a visible action in $P$ but a private action in $H$. To illustrate hiding of an interface, consider the module definition below.

$K = \text{hide} \ I2 \text{ in } P$

To hide an interface in module $P$, it must be both an exported and created interface of $P$. Module $K$ is well-formed and interface $I2$ becomes a private interface in it. Creation of interface $I2$ is a visible action in $P$ but a private action in $K$. Hiding makes events and interfaces private to a module and converts output actions into internal actions.

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 $R$ refines $P$. But event $E2$ is used for internal interaction among machines, for completely different purposes, in both $P$ and $R$. Then, the check that $R$ refines $H$ is more likely to hold since sending of $E2$ is not a visible action of $H$. Second, hiding helps make a module more composable with other modules. Recall that to compose two modules, their sent events and created interfaces must be disjoint. 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.

**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 2.3). Suppose a machine $M$ in module $P$ has a private event $e$ in its set of received events. Any machine that possesses a reference to an instance of $M$ could send $e$ to this instance. If such a reference were to leak outside the module $P$ to a machine in a different module, it would create an obstacle to reasoning about $P$ separately (and proving the compositionality theorems for module with private events), since private events targeted at a machine inside $P$ may now be sent by the environment. ModP ensures that such leakage of a machine reference with permissions containing a private event cannot happen. In ModP, there are two ways for permissions to become available to a machine: (1) by creating an interface, or (2) by sending permissions to the machine in the payload accompanying some event. To tackle private permission leakage through (1), ModP require 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 (e3) in COMPOSITION rule and (e2) below. To tackle private permission leakage through (2), using a combination of static and dynamic checking. ModP enforces that (1) each send of event $e$ adheres to the specification $\text{SendOk}$ in Section 2 (dynamic check), and (2) the set of private events is disjoint from any permission in $\mathcal{A}(e)$ for any non-private event $e$ (condition (e4), WF3, static check). Together, these two checks ensure that a permission containing a private event does not leak outside the module through sends.

**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. Rule HideInterface handles the hiding of a set of interfaces $\beta$ in module $P$. This rule adds $\beta$ to $IP_P$. Condition (hi4) is similar to the condition (he1) of rule HideEvent; this condition ensures that the resulting module is an abstraction of $P$ (Lemma E.7 and E.8).

5.4 Renaming Interfaces

Consider the following code where interfaces I1 and I2 and machines M1 and M2 are taken from Figure 2.

interface I receives E1;  
machine M receives E1, E3;  
creates I;  
{ /* body */}

module A = bind I1 -> M1, I -> M1;
module B = bind I2 -> M, I -> M2;

In module A, the creates of interface I in machine M is bound to machine M1. But in module B which binds I2 to M, the creates of I in M is bound to machine M2. Thus, it is no longer possible to take the union of modules A and B because of conflicting bindings of interface I (rule UNION). Interface renaming comes to the rescue in such a situation.

interface J receives E1;  
module C = rename I -> J in B;

In module C, the interface name I is renamed to J. The binding I -> M2 is converted to J -> M2 and any creation of interface I in M is automatically interpreted as creation of J. As a result, the union of modules A and C is now possible. Thus, interface renaming increases code reuse by allowing machine M to be reused and specialized for different module contexts without sacrificing compositability.

**Rule Rename**


$$i' \in I \setminus (\text{dom}(I_P) \cup IC_P)^{\epsilon}$$

$$A = \{x \mid (x', y) \in EP_P \wedge x = i$$

$$B = \{x \mid (x', y) \in EP_P \wedge x$$

$$C = \{x \mid (x', y, z) \in EP_P \wedge x$$

$$\text{rename } i \to i' \text{ in } P \vdash EP_P, A, B, C$$

Rule Rename handles the renaming of interface $i$ to $i'$ in module $P$. Condition (r5) checks that $i'$ is well-scoped; the set of $\text{dom}(I_P) \cup IC_P$ is the universe of all interfaces relevant to $P$. Condition (r6) checks that $i'$ is a new name different from the current set of interfaces relevant to $P$. Condition (r7) checks that the set of received events of $i$ and $i'$ are the same. Together with condition (b2) in rule BND, 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 (r8) 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.

5.5 Compositional Reasoning

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

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

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

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

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

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 \texttt{test tname: P introduces a safety test obligation that the executions of module P do not result in a failure (module P is safe). The declaration test tname: P refines Q introduces a test obligation that module P refines module Q. For each test declaration, the programmer provides a test harness 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 for each test declaration to cover various testing scenarios.

ModP automatically discharges the test obligations using systematic testing. The ModP compiler performs static analysis of the modules and generates C# code corresponding to each test declaration. The ModP systematic testing engine efficiently enumerates executions resulting from scheduling and explicit nondeterministic choices (similar to [21]). The engine tests \( P \leq Q \) in two phases, in the first phase, it enumerates all traces in the abstract module \( Q \) and caches it. Then in the second phase, it samples executions from the implementation module \( P \) to assert that the corresponding trace exists in the set of traces of module \( Q \). 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 (more details in Appendix C).

We illustrate using the protocol stack in Figure 1 (details in Section 7) how ModP can be used to test the transaction commit service compositionally. In ModP, both the specification and implementation are modeled as modules at different levels of abstraction. Let module TPCommit represent the module implementing the two-phase commit protocol, and TPCommitAbs be its abstraction. Module MPaxosSMR and ChainRepSMR represents the implementation of the SMR service using the Multi-Paxos protocol suite and Chain Replication protocol suite, respectively. Let LinearAbs represent the linearizability abstraction exposed by both the SMR services, based on Multi-Paxos and Chain Replication.

```
module TPCommit' = (hide X in TPCommit);
module MPaxosSMR' = (hide X in MPaxosSMR);
module TransCommitImpl = TPCommit' || MPaxosSMR';
// compositional safety testing */
test S1: TPCommitAbs || MPaxosSMR';
test S2: TPCommit' || LinearAbs;
// circular assume guarantee */
test R1: TPCommit' || LinearAbs refines TPCommitAbs;
test R2: TPCommitAbs || MPaxosSMR' refines LinearAbs;
// extra check */
test S3: TPCommitAbs || ChainRepSMR;
test R3: TPCommitAbs || ChainRepSMR refines LinearAbs;
```

ModP enables programmers to test their implementation compositionally. Let say the programmer implemented TPCommit and MPaxosSMR independently in isolation, it is possible that these modules use same action names or reuse some module (like OS Services) that make them incompatible. The hide operation comes to the rescue and can be used to create a more abstract module by hiding all the conflicting actions represented by \( X \). The resulting modules TPCommit' and MPaxosSMR' are compatible. TransCommitImpl represents the implementation module. The whole system testing problem is to check the safety of TransCommitImpl module.
Using Theorem 5.3, we can decompose the problem into safety tests S1 and S2 under the assumption that TransCommitImpl refines each module in \{ TPCommItAbs, LinearAbs \}. This assumption is then validated using the Theorem 5.4 and tests R1-R2. The power of compositional reasoning is substitutability, if the 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 S3-R3.

7 Evaluation
To evaluate the efficacy of ModP approach we implemented and tested two practical distributed services: (1) a fault-tolerant transaction-commit service, and (2) distributed data structures such as hashtables and lists. We implement the transaction commit service using the two-phase commit protocol [33], which uses a single coordinator state-machine to atomically commit updates across multiple participant state-machines. Hashable and list are implemented as deterministic state-machines with PUT and GET operations. These services by themselves are not tolerant to node failures. A common approach for achieving fault tolerance is to use state-machine replication (SMR), with protocols such as Multi-Paxos [47] and Chain Replication [75], where a consistent sequence of events is fed to deterministic state machines running on multiple nodes. We use this generic approach to make two-phase commit and the data structures fault-tolerant by replicating the deterministic coordinator, participant and hashable (list) state-machines across multiple nodes. Multi-Paxos and Chain Replication in turn use different sub-protocols but they both provide linearizability guarantees to the services being replicated. The ModP compiler also generates executable C code for the implementation modules. We deployed the generated code on a production cluster. Specs. column represents the component-level temporal properties (Appendix A). 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 test declarations (last column) across protocols together ensure that “whole-system” satisfies the system-level properties based on the compositional refinement theorems discussed in Section 5.5

<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>38</td>
</tr>
<tr>
<td>Chain Rep.</td>
<td>577</td>
<td>220</td>
<td>83</td>
<td>130</td>
<td>35</td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>487</td>
<td>101</td>
<td>81</td>
<td>92</td>
<td>30</td>
</tr>
<tr>
<td>Data structures</td>
<td>275</td>
<td>25</td>
<td>-</td>
<td>89</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>1779</td>
<td>Others</td>
<td>1086</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. ModP source lines of code for different protocols

7.2 Compositional Testing
Compositional reasoning led to state-space reduction and hence amplification of the test-coverage, uncovering many critical bugs in our implementation of the software stack. Some of these bugs require subtle interleaving which were hard to find using the monolithic testing approach. Overall, we found around 70+ critical bugs during the development process and most of these bugs were not found using monolithic testing with prioritized searching.

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, and (2) this test-coverage amplification results in finding bugs faster than the monolithic approach.

**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 reduction obtained in our case study, we conducted an experiment to count the number of unique executions in the implementation of a protocol that map to a trace in its abstraction. Figure 5-6 present the corresponding graphs where x-axis represents the traces in the abstraction sorted based on the number of implementation executions that map onto the abstraction trace and y-axis represent the number of executions in the implementation that maps (projects) to the trace in abstraction. Figure 5 shows the graph for the leader election (LE) protocol with 3 nodes and max length of a trace as 100, there were in total 187 traces in the LE abstraction and approx. $10^4$ executions in the implementation. It can be seen that the executions in the implementation map almost uniformly onto the traces in the abstraction with a maximum of 271 unique executions mapping a single trace in abstraction. LE abstraction helps in increasing the test-coverage as
exploring one execution in the abstraction is equivalent to exploring approx. 271 executions in the implementation.

We conducted a similar experiment for chain replication (CR) protocol with a finite test-harness that randomly pumps in 5 update operations. Figure 6 shows that the graph is highly skewed for more complex protocols like chain replication. The linearizability abstraction (guaranteed by chain replication protocol) has 1931 traces for the finite test-harness and there were exponentially many executions in the CR implementation. We sampled $10^6$ unique executions in the CR implementation for this experiment.

The graph in Figure 6 can be divided into three regions of interest: region (A) correspond to those traces in the abstraction to which no execution mapped from the sample 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.

Next, we compare ModP-based compositional systematic testing (CST) against monolithic testing. We randomly chose 8 bugs (out of 70) in the implementation of different protocols, these bugs are inspired from the ones that we found during the development process. We compared the performance of compositional testing against 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. Table 7 shows that ModP-based compositional approach finds bugs faster that the monolithic approach and in most cases the monolithic approach fails to find the bug even after exploring $10^6$ different schedules.

### 7.3 Refinement Checking

Meaningful testing requires that the abstractions used during compositional reasoning are sound abstractions of the components being replaced. We were able to uncover several scenarios where bugs were missed during testing because of an unsound (under-approximating) abstraction.

As an example, the linearizability abstraction was used when testing the distributed services built on top of SMR. Our initial implementation of the abstraction guaranteed that for every request there is a single response. For chain replication protocol (as described in [75]), 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 with duplicate responses. Our refinement checker was able to find this unsound assumption which lead to modifying our chain-replication implementation. This bug could have caused an error in the client of the chain replication protocol (like two phase commit) as it was tested against the linearizability abstraction. We found several such assumption violations, some related to mismatch in how machine identifiers flow (captured in trace signature) in the abstraction and in the implementation which could have led to missed bugs. Refinement checking helped increase our confidence about the “whole-system” correctness. The number of false positives uncovered in our development process was low ($<5$) compared to the real bugs that we found (70), this could be because the protocols that we considered in this paper have well-understood abstractions.
8 Related work

We have discussed related work throughout the paper. We next situate ModP with related techniques for modeling, analysis, and testing 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 [4], and I/O automata [53].

Process algebra. In the process algebra approach deriving from Hoare’s CSP [41] and Milner’s CCS [57], the π-calculus [58, 62] has become the de facto standard in modeling mobility and reconfigurability for applications with message-based communication. Extensions of π-calculus such as asynchronous π-calculus, distributed join calculus [29, 30], Dπ-calculus [65] deal with distributed systems challenges like asynchrony and failures respectively. The popular approach for reasoning about behaviour 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 π-calculus based on trace inclusion [17].

ModP chooses actors [2] 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.

More recent work like session-types [13, 23, 42] and behavioral-types [6] that have their roots in process calculi can encode abstractions in the type language. Stronger types can handle arbitrary state transition system [11]. Our abstractions (modules) are state machines capable of expressing arbitrary trace properties.

Reactive modules and I/O automata. Reactive modules [4] 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 lock step. Dynamic Reactive Modules [27] is a dynamic extension of Reactive Modules with support for dynamic creation of modules and dynamic topology. The semantics of dynamic reactive modules are given by dynamic discrete systems [27] 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 compositional theorem for a hide operation. Also, our module system is novel compared to DRM because of the fundamental differences in the supported programming model. Dynamic I/O automata (DIOA) [8] is a compositional model of dynamic systems, based on I/O automata [53]. 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 used for building industrial scale systems.

Verification and testing. There has been a lot of work towards reasoning about concurrent systems using program logics deriving from Hoare logic [28, 40] – which includes rely-guarantee reasoning [31, 72, 78] and concurrent separation logic [26, 51, 59]. Actor services [71] is a more recent work that proposes program logic for proofs of actor programs. DISEL [68] is a framework for implementation and compositional verification of distributed systems using proof assistants. The focus of all of these techniques is formal proof; they decompose reasoning along the syntactic structure of the program and emphasize modularity principles that allow proofs to be easily constructed and maintained. On the other hand, the focus of our work is not formal proof but rather a method to decompose a large testing problem into a collection of smaller testing problems and enabling it for any actor based programming language.

Researchers have built testing tools [50, 67] for automated unit testing of Java actor programs. Mace [43], Teapot [14] and P [20] provide language support for implementation, specification and systematic testing of asynchronous systems. Rebeca [70] is an actor oriented programming language with support for model-checking based verification. The theory of compositional verification proposed by Rebeca for actor programs does not consider dynamic machine creation.

MaceMC [44] and MoDist [79] operate directly on the implementation of a distributed system and explore the space of executions to detect bugs in distributed systems. The conclusion of researchers who developed these systems is similar to ours: monolithic testing of distributed systems does not scale [34].

9 Conclusion

ModP is a new programming framework that makes it easier to build, specify, and systematically 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’s compositional testing has the power to generate and reproduce within minutes, executions that could take months or even years to manifest in a live distributed system.

Acknowledgment

The authors would like to thank Daniel Fremont, Jay Lorch, Nikhil Swamy and Damien Zufferey for their valuable feedback and suggestions. The first and last author were supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus
Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

**Appendix Summary**

**Formalism and Proofs:**
1. Section A presents how temporal specifications can be attached to a module.
2. Section B presents all the theorems and proofs for the ModP module system.

**Implementation and Case Study Details:**
1. Section C presents the ModP tool chain.
2. Section D presents details about the distributed service software stack implemented along with the specification checked.

### A Attaching Specifications to Modules

Figure 8 presents the syntax for attaching specifications to modules.

\[
s \in \text{SpecName} \\
P \in \text{ModuleExpr} ::= \ldots \\
\mid \text{assert } s \text{ in } P
\]

*Figure 8. Attaching Specification s to Module P*

In ModP, a programmer can specify temporal properties via specification machines.

```
spec s observes E1, E5, E6 { .. }
```

Machine \( s \) is a specification machine that observes events \( E1 \), \( E5 \), and \( E6 \). 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 \( e \). Thus, specification machines introduce a publish-subscribe mechanism for monitoring events to check temporal specifications while testing a ModP program. The declaration `assert s in P` attaches specification machine \( s \) to module \( P \).
B Theorems and Proofs

Theorems for ModP Module System

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.2, 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 construction of two modules creates a new module which is equally or more detailed than its components. This is captured by the Lemma B.5.

3. Circular Assume-Guarantee: Lemma B.6 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 \( Q \) to prove \( Q_2 \) and \( Q_2 \) to prove \( Q_1 \).

4. Compositional Safety Analysis: Lemma B.7 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.8 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.9 states that the hide interface operation preserves refinement, is compositional and create a sound abstraction of the module.

\[ \forall (i, n) \in (\text{dom}(G_S) \cap \text{dom}(G'_S)), G_S[i, n] = G'_S[i, n], \]

\[ \forall i \in (\text{dom}(G_C) \cap \text{dom}(G'_C)), G_C[i] = G'_C[i] \]

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 \( \langle G \times G \rangle \) to \( G \) satisfying the following properties:

1. \( (G, G') \in \text{dom}(\text{union}) \) iff \( G \) and \( G' \) are compatible.

2. \( (G \circ \circ G', G^\circ) \) in union iff \( G^\circ = (G^\circ_S) \cup (G^\circ_B) \cup (G^\circ_C) \).

---

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 \( \tau = G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_n} G^n \) then \( \text{last}(\tau) = G^n \)

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

4. Two configurations \( G, G' \in G \) 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. \( \forall i \in (\text{dom}(G_C) \cap \text{dom}(G'_C)), G_C[i] = G'_C[i] \)

---

Lemma B.1: Invariants for Executions of a Module

Let \( P \) be a well formed module. For any execution \( \tau \in \text{exec}(P) \) where \( \tau \) is a sequence of global configurations \( G_0, G_1, \ldots, G_n \), all global configurations \( G_i \) satisfy the invariants:

- \( I_1 \) \( \text{dom}(S_P) = \text{dom}(B_P) \)
- \( I_2 \) \( \forall (i, n) \in \text{dom}(B_P), i \in \text{dom}(l_P) \land n < C[i] \)
- \( I_3 \) \( \forall i \in \text{dom}(l_P), C[i] = \text{card}(\{n \mid (i, n) \in \text{dom}(B_P)) \}
- \( I_4 \) \( \forall (x, n, \alpha) \in \text{ids}(S_P) \cup \text{ids}(B_P), x \in \text{dom}(l_P) \Rightarrow (x, n, \alpha) \in \text{dom}(B_P) \)
- \( I_5 \) \( \forall (x, n, \alpha) \in \text{ids}(S_P) \cup \text{ids}(B_P), n < C_P[x] \)

Proof. The invariants \( I_1 \) - \( I_5 \) 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.

Lemma B.2: Compositional Is Intersection

Let \( P, Q \) and \( P||Q \) be well-formed. For any \( \pi \in \Sigma^*_P||Q \), \( \pi \in \text{traces}(P||Q) \) 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.3 and Lemma B.4.

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.3.

**Backward Implication for traces:**
If there exists a sequence \( \sigma \in \Sigma^*_P||Q \) 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.4.
Lemma B.3

For any execution \( \tau_c \in \text{exec}(P|Q) \), there exists an execution \( \tau_p \in \text{exec}(P) \) such that \( \text{trace}(\tau_p)[\Sigma_P] = \text{trace}(\tau_c)[\Sigma_P] \) and there exists an execution \( \tau_q \in \text{exec}(Q) \) such that \( \text{trace}(\tau_q)[\Sigma_Q] = \text{trace}(\tau_c)[\Sigma_Q] \).

Proof: We perform induction over the length of execution \( \tau_c \) of the composed module \( P|Q \).

**Inductive Hypothesis:** For every execution \( \tau_c \in \text{exec}(P|Q) \), there exists an execution \( \tau_p \in \text{exec}(P) \) such that \( \text{trace}(\tau_p)[\Sigma_P] = \text{trace}(\tau_c)[\Sigma_P] \), there exists an execution \( \tau_q \in \text{exec}(Q) \) 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, \( \tau_c \in \text{exec}(P|Q) \). 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 \( \tau_p \in \text{exec}(P) \) and \( \tau_q \in \text{exec}(Q) \) 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 \( \tau_c \in \text{exec}(P|Q) \). 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 \( \tau'_c \in \text{exec}(P|Q) \) where \( \tau'_c = \tau_c \rightarrow \delta G \) 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. Let's 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, Create-Private, Send-Bad, and Send-Private. Consider the case when \( i \in \text{dom}(\tau_p) \) 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, \( \text{trace}(\tau_p'[i, n]) = \text{trace}(\tau_p'[i, n]) \). 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 \( m \in \text{dom}(\tau) \).

- \( a = i \) 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 \( i \) 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 \( \text{dom}(\tau) \).

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

2. **Input-CREATE**

Our definition of composition and compatibility guarantees that if \( P|Q \) is well-formed then:

1. \( \text{dom}(\tau_p) \cup \text{dom}(\tau_q) \)
2. \( \text{dom}(\tau_p) \cap \text{dom}(\tau_q) = \emptyset \)

Hence, if the composed module \( P|Q \) receives an input create request for \( i \in \text{dom}(\tau) \) \( \in \text{IC}_P \) from the environment, then either \( i \in \text{dom}(\tau_p) \), or \( i \in \text{dom}(\tau_q) \). 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 \( i \in \text{dom}(\tau) \). 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 \( i \notin \text{dom}(\tau) \cup \text{IC}_Q \)), hence \( Q \) takes the Environment-Create transition. The resultant executions \( \tau'_p, \tau'_q \) and \( \tau'_p, \tau'_q \) satisfy the condition that \( \text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau'_q)) \).

Also, \( \text{trace}(\tau'_p)[\Sigma_P] = \text{trace}(\tau'_p)[\Sigma_P] \) and \( \text{trace}(\tau'_q)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q] \) since all modules took the same labeled transition.
The same analysis can be applied to the case when $i \in dom(I_Q)$.

3. **Output-Create-Outside**

This is the case when a machine instance $(i', n) \in dom(last(\tau_c)_S)$ creates an interface $i$ and $i \notin dom(I_P||Q)$ which means that interface $i$ is implemented by some machine in the environment of $P||Q$.

Consider the case when $i' \in dom(I_P)$ (which implies that $i' \notin dom(I_Q)$), since $i \notin dom(I_P||Q)$ we know that $i \notin dom(I_P)$.

Based on the assumption that $last(\tau_c) = union(last(\tau_p), last(\tau_q))$ we know that $S[i, n] = S[\tau_p][i, n]$ and $B[\tau_c][i, n] = B[\tau_p][i, n]$ and hence if $P||Q$ takes the visible Output-Create-Outside transition when in global configuration $last(\tau_c)$, the same transition can be taken by module $P$ in configuration $last(\tau_p)$.

$i \in \Sigma_Q$ and hence the environment of module $Q$ creates an interface $i$ (Environment-Create) and the resultant executions satisfy the condition that $last(\tau_c') = union(last(\tau_p'), last(\tau_q'))$.

4. **Output-Create-Inside**

Similar analysis can be applied to prove that our inductive hypothesis holds when the composed module $P||Q$ takes an Output-Create-Inside transition.

---

**Lemma B.4**

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

**Proof.** Given a pair of executions $(p, q)$ and $(p', q')$, we define a partial order over pair of executions as $(p, q) \leq (p', q')$ 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 $\sigma[\Sigma_P] = trace(\tau_p)[\Sigma_P]$ and $\sigma[\Sigma_Q] = trace(\tau_q)[\Sigma_Q]$ then there exists an execution $\tau_e \in execs(P||Q)$ such that $trace(\tau_e)[\Sigma_P][\Sigma_Q] = \sigma$ and $last(\tau_e) = union(last(\tau_p), last(\tau_q))$.

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

$\tau(\tau_p)[\Sigma_P] = \tau(\tau_q)[\Sigma_P] = \epsilon$ ($\epsilon \in \Sigma^{*}_{P||Q}$).

---

we know that: there exists $\tau_p = (S_p^0, B_p^0, C_p^0) \in execs(P)$, there exists $\tau_q = (S_q^0, B_q^0, C_q^0) \in execs(Q)$ and there exists $\tau_e = (S_e^0, B_e^0, C_e^0) \in execs(P||Q)$.

Hence, there exists an execution $\tau_e \in execs(P||Q)$ such that $\tau(\tau_e)[\Sigma_P][\Sigma_Q] = \epsilon$.

Finally, we have $last(\tau_e) = union(last(\tau_p), last(\tau_q))$: as:

- $S_e^0 = S_p^0 = S_q^0 = S_0$ (empty map)
- $B_e^0 = B_p^0 = B_q^0 = B_0$ (empty map)
- $C_e^0 = C_p^0 = C_q^0 = C_0$ (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 $(\tau_p', \tau_q')$.

Let us consider the case when $\tau_p'$ 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-Outside with label $a = ((i_t, n_t), e, v)$. Let $(i, n) \in dom(last(\tau_s)_S)$ be the machine that takes this transition. Hence, $\sigma' = \sigma.a$ and $trace(\tau_p')[\Sigma_P] = \sigma'[\Sigma_P]$.

Let us consider the case when $i_t \in dom(I_Q)$, and $e \in MReceivs(i_t)$ \ $(EP_Q \cup ES_Q)$ (input event of $Q$). Based on the assumption that $last(\tau_c) = union(last(\tau_p), last(\tau_q))$ and the invariants 11-16 about the state configurations, we know that $(i_t, n_t) \in dom(last(\tau_q)_b)$.

Hence, $Q$ can take a Input-Send transition with label $a = ((i_t, n_t), e, v)$ and therefore $trace(\tau_q')[\Sigma_Q] = \sigma'[\Sigma_Q]$. Finally, using the same assumption $last(\tau_e) = union(last(\tau_p), last(\tau_q))$ and the invariants 11-16, the composed module $P||Q$ can take the transition Output-Send-Inside with the same label $a = ((i_t, n_t), e, v)$. Hence, $trace(\tau_e')[\Sigma_P][\Sigma_Q] = \sigma'$.

The resultant executions still satisfy the condition that $last(\tau_e) = union(last(\tau_p'), last(\tau_q'))$.

Note: Proving that executions of modules satisfy the property $last(\tau_e) = union(last(\tau_p), last(\tau_q))$ 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.5: Composition preserves refinement

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

Proof: (1) follows directly from the Lemma B.2. 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, $P \leq Q$ therefore $\sigma[Q]$ is a trace of $Q$ and using the Lemma B.2 $\sigma[\Sigma_Q||R]$ is a trace of $Q||R$.

Lemma B.6: Circular Assume-Guarantee

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

Proof: Definitions:
- Let $Q$ be a collection of $(n > 1)$ composable modules represented by the set $\{Q_1, Q_2, \ldots, Q_n\}$.
- Let $P$ be a collection of $(n' > 1)$ composable modules represented by the set $\{P_1, P_2, \ldots, P_{n'}\}$. In this proof, we refer to $\parallel P \parallel$ (composition of all modules in $P$) as module $P$.
- Let $\forall X_k$ be a subset of $P @ Q$.

Let $Q_k$ exist such that $X_k \leq Q_k$.

**Inductive Hypothesis:** Our inductive hypothesis is that for every execution $\tau_P \in \text{exec}(P)$ and for all $Q_k \in Q$, there exists an execution $\tau_{Q_k} \in \text{exec}(Q_k)$ such that $\text{trace}(\tau_P)[\Sigma_{Q_k}] = \text{trace}(\tau_{Q_k})[\Sigma_{Q_k}]$.

Note that the inductive hypothesis is the traces of $P$ but it implies that, for all $Q_k \in Q$, there exists a $X_k$ such that $X_k \leq Q_k$ then for all traces $\sigma_P \in \text{traces}(P)$ and for all $Q_k \in Q$ we have $\sigma_P[\Sigma_{Q_k}] \in \text{traces}(Q_k)$.

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 $\tau_p \in \text{exec}(P)$ of length $k$.

To prove that the hypothesis is inductive, we show that the hypothesis also holds for any execution $\tau'_P$ such that $\tau'_P \xrightarrow{a} G$.

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: $\text{Output-Create-Outside}$, $\text{Output-Create-Inside}$ or (2) the environment creates it using the transitions: $\text{Environment-Create}$, $\text{Input-Create}$.

Let us consider the case when $a = i$ because $P$ executes the $\text{Output-Create-Outside}$ transition.

Recall that $P$ is a composition of modules $P_1, P_2, \ldots, P_{n'}$. Using Lemma B.3, we can decompose the execution $\tau_P$ of module $P$ ($\tau_P \in \text{exec}(P)$) into the executions $\tau_{P_1}, \tau_{P_2}, \ldots$ 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 $\text{Output-Create-Outside}$, we know that $i \in IC_P$ and $i \notin \text{dom}(IC_P)$.

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 $\forall j, k \neq i, i \notin IC_{P_j}$.

If $\exists j, \text{s.t. } j \neq k \land i \notin IC_{P_j}$, then $P_j$ can take the $\text{Environment-Create}$ 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 $\forall Q_k \in Q_k, i \notin IC_{Q_k}$, then all $Q_k$ can take the $\text{Environment-Create}$ 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 $i \in IC_{Q_k}$.

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

Using Lemma B.4, and the fact that $X_k \leq Q_k$, we know that for any given $\tau'_{P_k} \in \text{exec}(P_k)$ 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 $\tau'_P = \tau_P \xrightarrow{a} G$ (Output-Create-Outside)
2. $i \in IC_{Q_k}$ and $i \in IC_{P_k}$.
3. $\forall j \neq k \neq i \notin IC_{P_j}$ and $\forall j \neq k \neq i \notin IC_{Q_j}$.
4. there exists an execution $\tau'_{P_k} \in \text{exec}(P_k)$ such that $\text{trace}(\tau'_{P_k})[\Sigma_{P_k}] = \text{trace}(\tau_{P_k})[\Sigma_{P_k}]$.
5. there exists an execution $\tau'_{Q_k} \in \text{exec}(Q_k)$ such that $\text{trace}(\tau'_{P_k})[\Sigma_{Q_k}] = \text{trace}(\tau_{P_k})[\Sigma_{Q_k}]$.

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

19
And using (3), we also know that for all \( Q_j \in Q_k \),
\( \text{trace}(r'_p) \parallel Q_j = \text{trace}(r_0) \parallel Q_j \)
Hence, the inductive hypothesis holds for the execution \( r'_p \).

We do similar analysis to prove the other cases.

\[\square\]

**Lemma B.7: Compositional Safety Analysis**

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 \( P \oplus Q \) such that \( P \in X \), \( \|X\) is well-formed, and \( \|X\) is safe. Then \( \|P\) 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 us assume that there exists an error execution in \( r^e \) in \( P_1 \parallel P_2 \). Using the compositional refinement Lemma, we can decompose the execution \( r^e \) into \( r^e_1 \) of \( P_1 \) and \( r^e_2 \) of \( P_2 \). Let us say the error was because of module \( P_1 \) taking a transition and hence \( r^e_1 \) 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 \( r^e_1 \) 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 \( r^e_1 \). 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.8: 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 \( r_{hP} \) of module \( hP \).

**Inductive Hypothesis:** For every execution \( r_P \in \text{execs}(hP) \), there exists an execution \( r_P \in \text{execs}(hQ) \) such that \( \text{trace}(r_P) \parallel \Sigma_{hQ} = \text{trace}(r_Q) \parallel \Sigma_{hQ} \)

We prove our inductive hypothesis by performing induction over the length of execution \( r_P \).

- **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 \( r_{hP} \in \text{execs}(hP) \) and the corresponding execution of module \( hQ \) be \( r_{hQ} \in \text{execs}(hQ) \).

To prove that the hypothesis is inductive we show that it also holds for the execution \( r'_{hP} \in \text{execs}(hP) \) where \( r'_{hP} = r_{hP} \rightarrow G \) and \( r'_{hQ} \) 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.

\[\square\]

**Lemma B.9: 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.8. 

\[\square\]
C ModP Tool Chain

![Diagram of ModP tool chain](image)

Figure 9. Overview of ModP programming framework

In this section, we describe implementation of the ModP tool chain consisting of the compiler, systematic testing, and runtime. Figure 9 shows an overview of the entire ModP tool chain. On the left is the ModP program comprising four blocks of declarations—machines, modules, specifications, 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 module, specification, and test declarations are well-formed. The compiler generates C# code for each test declaration; this generated code makes all sources of nondeterminism explicit and controllable by the systematic testing engine, which generates executions in the test program checking each execution against implicit and explicit specifications. The compiler also generates C code which is compiled and linked against the runtime to generate application executables (Section D).

C.1 Systematic Testing Engine

The systematic testing engine efficiently enumerates all possible executions of the program using delay-bounded search prioritization [21, 25]. The systematic testing engine performs refinement checking of ModP programs based on trace containment. Our algorithm for checking $P \preceq Q$ (where $P$ and $Q$ are closed modules) 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 specification 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 cache (traces of $Q$).

The ModP framework automatically tests each test declaration independently and produces a counter example if a bug is found. The counter example is 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.

C.2 ModP’s distributed runtime

![Diagram of ModP runtime](image)

Figure 10. Components of the ModP runtime with a ModP application split across two physical nodes.

Figure 10 shows the components of the ModP runtime on two nodes in a cluster. Each node hosts a collection of Container processes. A Container is a 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 that 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.

Any remote procedure call to a Container results in a thread enqueuing an event in the appropriate state machine running inside the Container and subsequently executing the event handling loop of the target state-machine. In this loop, events are dequeued and their handlers are executed to completion. If the handler enqueues an event into another machine in the same Container, the event handling loop of that machine is executed, and so on. If the handler tries to send an event to a machine in a different Container, the message is serialized and an RPC call is performed. The runtime uses efficient fine grain locking to guard against race conditions where more than one thread tries to execute a handler for a machine; such races, if allowed, would violate the operational semantics of ModP.

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 Node-Manager process and one Container process executing on it, but ModP also supports a collection of Containers per node enabling emulation of large-scale services running on only a handful of nodes. New Containers are created by invoking the runtime’s CreateContainer function. A new local or remote state machine can be created by specifying the hosting container’s ID. Hence, the ModP runtime enables the programmer to distribute state-machines across nodes in
a cluster and also group them within containers to improve performance.

In summary, the runtime executes the generated C representation of the ModP program, ensuring its operational semantics. It has the capability to (1) create, destroy and execute state machines, (2) create and destroy runtime representations of ModP types and values, (3) efficiently communicate among state machines that can be distributed across physical nodes, (4) serialize data values before sends and deserialize them after receives.

D Distributed Services Software Stack

We use the modularity features of ModP to implement and test the software stack for two distributed services: (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit [9, 32, 49, 64], and (ii) distributed data structures such as hashtables and lists. These distributed services use State Machine Replication (SMR) for fault-tolerance [33, 46, 66].

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. Hashable and list are implemented as deterministic state-machines with PUT and GET operations. These services by themselves are not tolerant to node failures. A common approach for achieving fault tolerance is to use state-machine replication (SMR), with protocols such as Multi-Paxos [47] and Chain Replication [75], where a consistent sequence of events is fed to deterministic state machines running on multiple nodes. These events could be operations on a data-structure or operations for two-phase-commit. We use this generic approach to make two-phase-commit and the data structures fault-tolerant by replicating the deterministic coordinator, participant and hashable (list) state-machines across multiple nodes. Multi-Paxos and Chain Replication in turn use different sub-protocols. Though both these protocols provide linearizability guarantees their implementations are very different and have different fault models. For example, Multi-Paxos uses $2n+1$ replicas to tolerate $n$ failures whereas Chain Replication exploits a reliable failure detector to use only $n+1$ replicas for tolerating $n$ failures. Despite their differences, both Multi-Paxos and Chain Replication expose the same abstraction and hence acts as a good case study for protocol substitution.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>2PC</td>
<td>Transactions are atomic [32]</td>
</tr>
<tr>
<td>Chain Repl.</td>
<td>All invariants in [75], cmd-log consistency</td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>Consensus requirements [48], log consistency [74]</td>
</tr>
</tbody>
</table>

Figure 11. Specifications checked for each protocol

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 safety and liveness specifications (as spec-machines) of all the protocols as described in their respective papers [32, 47, 49, 75]. Figure 11 shows examples of specifications checked for some of the distributed protocols. The ModP compiler also generates executable C code for the implementation modules. We deployed the generated code on a production cluster.

References


