Compositional Reasoning for 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-2017-154
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-154.html

September 10, 2017
Fault-tolerant distributed systems are difficult to get right because they must deal with concurrency and failures. Despite decades of research, current approaches for a more rigorous way of building distributed systems are unsatisfactory. This paper proposes new 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 the 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) and hierarchical (refinement) 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. Each system uses state machine replication (SMR) protocols for fault-tolerance. Our framework helps build these systems modularly and validates them via compositional systematic testing. We empirically show that using abstractions based compositional reasoning helps amplify the coverage during testing. We find that monolithic testing based on existing search prioritization techniques cannot find more than half the bugs found using ModP within equivalent time budgets.

1 INTRODUCTION

Fault-tolerant distributed systems must reliably handle concurrency and failures. Programming with concurrency and failures is challenging because of the need to reason about numerous program control paths. These control paths result from two main sources of nondeterminism—interleaving of event handlers and unexpected failures. Not surprisingly, programmers find it difficult to reason about the correctness of their implementations. Even worse, it is extremely difficult to test distributed systems; unlike sequential programs whose execution can be controlled via the input, controlling the execution of a distributed program requires fine-grained control over the timing of the execution of event handlers and fault injection. In the absence of such control, most execution paths remain untested and serious bugs lie dormant for months or even years subsequent to deployment. Finally, bugs that occur during testing or after deployment tend to be Heisenbugs; they are difficult to reproduce because their manifestation requires timing requirements that might not hold from one execution to another. These problems are well-known and have been highlighted by creators of large-scale distributed systems [14].

Existing validation methods for distributed systems fall into two broad categories: proof-based verification and black-box systematic testing. While researchers have used theorem provers to validate different parts of a single-node systems stack [15, 33, 41, 50, 65, 72, 75], doing the same for a distributed system requires a herculean effort. One of the rare example is IronFleet [32]. IronFleet models a distributed system as a collection of concurrently-executing actions in a TLA+ specification [76], proves important safety and liveness properties using manually-specified inductive invariants and interactive theorem proving [48], and finally refines each action into an executable procedure in a classical imperative language. This method, based on formal proofs, offers strong coverage guarantees. However, writing down inductive invariants and interactively proving them is very challenging and time-consuming, making it difficult to maintain the proof as the system implementation
inevitably evolves; such concerns have also been highlighted in recent single-node verification research [65]. In contrast, systematic testing techniques operate directly on the implementation of a distributed system and provide best-effort testing guarantees [30, 40, 74]. They eliminate manual specification of inductive invariants and the need for interactive theorem proving at the cost of validation coverage. Unfortunately, even with state-of-the-art systematic testing techniques, coverage scales poorly and decreases dramatically with increasing complexity of realistic distributed systems. The authors of SAMC [47] highlight the fact that these techniques hit a wall when no semantic knowledge about the program is provided.

In this paper, we introduce ModP, a module system for compositionally validating implementations of distributed systems using assume-guarantee reasoning. ModP occupies a spot between the two aforementioned techniques in terms of the trade-off between validation coverage and programmer effort. We exploit best-effort systematic testing but instead of testing the entire system monolithically, we decompose the “whole-system” testing problem into a collection of testing problems, one for each component of the system.

There are three key ideas to enabling such sound decomposition using ModP: first, is the principled use of abstractions of system components. ModP allows the programmer to specify an abstraction of a component and replace the component with its abstraction when testing the rest of the system. Second, ModP critically performs refinement checks that ensure that a component behaves like its abstraction thus validating that the use of the abstraction is sound. We observe that large distributed systems can be easily decomposed into interacting protocols, each of which can be specified separately as an abstraction that is much more compact than the implementation, leading to vastly improved test scalability. Third, ModP provides a module system that helps programmer perform sound compositional reasoning and achieve meaningful system-level guarantees by performing component-level testing. The central challenge that ModP has to overcome for assume guarantee reasoning of concurrent and distributed systems is the inherent dynamism in its programming model.

ModP builds on the theory of assume-guarantee reasoning [1, 4, 53]. Although the basic mathematical principles of assume-guarantee reasoning are well-known, incorporating those principles in a programming language suitable for implementing distributed systems is difficult. Most fault-tolerant distributed systems are dynamic: they require a programming model in which processes can be created and destroyed dynamically, communication topology of existing processes can change over time (either by creation or failures of node and network), and process identifiers are first-class values that can be sent from one process to another via messages. An important theoretical contribution of our work is extending assume-guarantee reasoning to incorporate such dynamic features.

To enable compositional reasoning of fault-tolerant distributed systems, we develop a novel theory of modules suitable for modeling assume-guarantee components in a dynamic message-passing system. A module is a repository of concurrently-executing state machines that interact with each other and with the environment of the module by creating machines, sending messages to other machines, and receiving messages from them. Our theory provides trace semantics to each module, defines when a module refines another, and permits composing modules together to build larger modules. Our module system provides a framework that supports both compositional (assume-guarantee) and hierarchical (refinement) design and analysis of distributed systems. We summarize the novel contributions of our module system in more details in Section 2.1

ModP programming framework. ModP builds on top of P [17], a state machine based language for safe event-driven programming. P is based on the computation model of asynchronously-communicating state machines (actors), and supports the dynamic programming features required for building real-world asynchronous systems. Based on our theory, we implemented ModP as a module system for P to enable compositional design and analysis of distributed systems. Our theory of compositional refinement and module system can be applied to any actor [2, 7, 11, 38] or state machines [17, 31, 39] based programming systems. The ModP compiler generates C# code for compositional systematic testing. The ModP systematic testing engine implements search prioritization techniques [19] for scalable testing of asynchronous programs and also supports refinement checking based on trace containment.
We use ModP to implement two distributed services (Figure 1): (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit [29], and (ii) distributed data structures such as hashtables and lists. These services use State Machine Replication (SMR) for fault-tolerance. Protocols for SMR, like Multi-Paxos [43] and Chain Replication [70], in turn use other protocols like leader election, failure detectors, and network channels.

ModP has the power to generate and reproduce within minutes, executions that could take months or even years to manifest in a live distributed system. Using compositional systematic testing we were able to find several critical bugs in our implementation. We empirically demonstrate that using abstractions during testing helps in amplifying validation coverage; in fact, monolithic testing based on scalable concurrency testing techniques [19] could not find more than half the bugs found using ModP within equivalent time budgets. We found that a principled approach to test decomposition based on compositional refinement is invaluable in catching errors in the specification of abstractions. These errors are caught by our refinement checker. We made several such errors during the design of the distributed services in Figure 1; these errors were caught and fixed within minutes.

In summary, the contributions of this paper are as follows:

• We present a new theory of compositional refinement and a module system for assume-guarantee reasoning of dynamic distributed systems.
• We implement a programming system, ModP, that leverages this theory to enable compositional systematic testing of complex distributed systems.
• Using ModP, we build real-world distributed services as a case study for showing the applicability of compositional systematic testing; our approach amplifies validation coverage and finds bugs faster than monolithic testing.

2 OVERVIEW
In this section, we motivate the need for compositional reasoning and present our approach to it using an example from the domain of distributed systems. Figure 1 shows the protocol stack of our distributed systems case study, a fault-tolerant transaction-commit service. It uses the two-phase commit protocol for implementing distributed atomic transactions. But this service can be unavailable in the face of node failures. The failure of any one of the participant nodes can lead to transactions being rejected indefinitely until the node is repaired. To make the service fault-tolerant, the stack in Figure 1 uses state-machine replication (SMR), implemented either via the Multi-Paxos [43] or the Chain Replication [70] protocol suite. An implementation of SMR guarantees that the
replicated state machines behave logically identically to a single state machine that never crashes and linearizes all operations sent to it.

Systematic testing of such large asynchronous distributed system is prohibitively expensive due to an explosion of behaviors caused by concurrency and failures. ModP allows programmers to use compositional reasoning to address this problem.

We next highlight the advantages of using abstraction for test amplification and present how compositional reasoning can be used for “whole-system” validation.

**Test amplification via abstractions.** Figure 2 presents the linearizability abstraction for SMR protocols like Multi-Paxos and Chain Replication using ModP pseudo-code.

The LinearAbs state machine maintains a bag of pending requests and the data they access or modify (smstate) in the state machine being replicated. Requests from clients are either added to the bag or dropped; nondeterministic choice is represented by $ (line 25). This is followed by Init where a request is removed from the bag, processed (DoOp), and a response is sent back to the client. This processing is repeated a non-deterministic number of times (line 15) before waiting for requests again (line 23). LinearAbs is considerably simpler than an actual implementation of SMR. For example, an open source implementation of the Multi-Paxos protocol suite has 1169 LOC [57]; our own implementation of Multi-Paxos in ModP is 638 LOC. More importantly, these
implementations are highly nondeterministic with multiple concurrently executing processes. For example, in a system that can tolerate \( n \) faults (node failures), there are \( 2n + 1 \) Paxos processes. In contrast, \( \text{LinearAbs} \) is a few dozen LOC and comprises a single process regardless of the value of \( n \). As a result, a huge number of executions in the Multi-Paxos implementation map to a single execution of the \( \text{LinearAbs} \) abstraction.

We exploit this property of implementations and their abstractions, common in real-world systems, to scale up systematic testing. For example, in Figure 1, we can test the implementation of two-phase commit using \( \text{LinearAbs} \) rather than the concrete Multi-Paxos implementation. If two-phase commit is tested against \( \text{LinearAbs} \), then it does not need to be tested separately against the Multi-Paxos protocol suite and the Chain Replication protocol suite. Each of those protocol suites may be checked separately against \( \text{LinearAbs} \) and composed with the implementation of two-phase commit to assemble the full system.

To enable such scenarios, backed by a new theory of compositional reasoning, \( \text{ModP} \) provides a module system where modules can be composed into larger ones while being tested independently using abstractions of other modules they interact with. We formalize a module as an open system whose semantics is a set of traces over its visible actions. To enable principled use of abstractions, \( \text{ModP} \) supports refinement between an implementation and its abstraction. A module \( P \) refines a module \( P' \), denoted \( P \preceq P' \), if every trace of \( P \) projected onto visible actions in \( P' \) is also a trace of \( P' \). Complex modules can be constructed from primitive ones using composition. The composition of \( P \) and \( Q \), denoted \( P || Q \), is a module that intuitively behaves like the union of \( P \) and \( Q \) and is the mechanism for building a large system from smaller sub-systems. Finally, refinement checking between the composition of a collection of implementation and specification modules can be decomposed, allowing each implementation module to be checked separately. This property is the key to enabling the benefits of scalability and reuse of systematic testing.

**Compositional reasoning.** Revisiting Figure 1, the system-level testing problem is to check that the implementation of the fault-tolerant transaction commit service satisfies the \( \text{TransCommitSpec} \) specification.

\[
\text{TwoPhaseCommit} \parallel \text{MultiPaxosSMR} \parallel \text{OSServ} \models \text{TransCommitSpec}
\]  

The refinement checking problem in (1) can be decomposed into the following simpler rules:

\[
\text{TwoPhaseCommit} \parallel \text{LinearAbs} \models \text{TransCommitSpec} \tag{2}
\]

\[
\text{ClientAbs} \parallel \text{MultiPaxosSMR} \parallel \text{OSServAbs} \preceq \text{LinearAbs} \tag{3}
\]

\[
\text{TwoPhaseCommit} \parallel \text{LinearAbs} \parallel \text{OSServAbs} \preceq \text{ClientAbs} \quad \text{and} \quad \text{OSServ} \preceq \text{OSServAbs} \tag{4}
\]

Rule (2) checks that the two phase commit (\( \text{TwoPhaseCommit} \)) protocol composed with \( \text{LinearAbs} \) satisfies \( \text{TransCommitSpec} \). Rule (3) checks that the Multi-Paxos based SMR (\( \text{MultiPaxosSMR} \)) implementation when composed with the client (\( \text{ClientAbs} \)) and OS services (\( \text{OSServAbs} \)) abstractions refine \( \text{LinearAbs} \). Rule (4) check that \( \text{ClientAbs} \) and \( \text{OSServAbs} \) are sound abstractions of the corresponding \( \text{TwoPhaseCommit} \) and \( \text{OSServ} \) implementations. Since the abstractions of the components are simpler than actual implementation, each of the Rules (2-4) are exponentially less expensive to check than the Rule (1). Note, that Rules (3-4) are circular as the abstraction of one module is used for testing the correctness of other modules. Such circular reasoning is enabled by \( \text{ModP} \)'s compositional refinement; when allowed by \( \text{ModP} \) the reasoning is sound.

To substitute Multi-Paxos with Chain Replication, we replace Rule (3) with the following Rule (5) to check that the chain replication based SMR (\( \text{ChainRepSMR} \)) implementation when composed with the client (\( \text{ClientAbs} \)) and OS services (\( \text{OSServAbs} \)) abstractions refine \( \text{LinearAbs} \).

\[
\text{ClientAbs} \parallel \text{ChainRepSMR} \parallel \text{OSServAbs} \preceq \text{LinearAbs} \tag{5}
\]
We can then use (5) and reuse (2), (4) to conclude:

\[ \text{TwoPhaseCommit} \parallel \text{ChainRepSMR} \parallel \text{OSServ} \models \text{TransCommitSpec} \]

### 2.1 Our contribution

The principles of decomposition described earlier to amplify systematic testing for distributed systems is well-known under the rubric of assume-guarantee reasoning \[1, 4, 53\], a style of reasoning that pioneered the use of stateful abstractions and refinement checking as a means to scalable verification.

Actors \[2\] and state machines \[17, 31, 39\] are a popular programming paradigm for building asynchronous systems; they are being used to build large applications of significant commercial interest \[11, 17\]. The contribution of this paper is to incorporate assume-guarantee reasoning principles in an actor-oriented programming language \[3, 11, 16, 17, 23, 61\] that supports dynamic asynchrony and concurrency via the following key features:

1. Dynamic creation of processes that are instances of named actor types (similar to dynamic creation of objects that are instances of named classes in an object-oriented language);
2. Directed message passing (as opposed to broadcast) to a specific actor instance;
3. Dynamic evolution of communication topology among actor instances as references to actor references are sent in messages.

A programming model used for building fault-tolerant distributed systems must support feature (1) because processes (nodes) may fail at runtime and new processes have to be created for maintaining service availability. Features (2) and (3) are required because broadcast-based communication can be prohibitively expensive in distributed services. Instead, the desired model is similar to client-server; the client sends a reference to itself to the server which may subsequently use the reference for sending responses. For example, the client of linearizability abstraction in Figure 2 sends its reference as a payload in the \( \text{REQ} \) message.

An important consequence of the programming model obtained by combining these dynamic features is that instance \( i \) of some actor type \( A \) is a first-class reference in the program conceptually represented by the value \((A, i)\) which can be passed from one actor instance to another via a message. An actor instance \((A, i)\) becomes aware of another actor instance \((B, j)\) either if it created \((B, j)\) or \((B, j)\) was embedded in some message received by it. This property must be exploited for compositional assume-guarantee reasoning about such systems; otherwise, the loss of precision and false error reports will make the reasoning system unusable in practice. Thus, incorporating assume-guarantee reasoning in such a dynamic programming framework is difficult and has not been done in any prior work. Our paper is the first to accomplish this task by designing, formalizing, and implementing assume-guarantee reasoning atop the actor-oriented programming language \( P \).

### 2.2 Our approach

There are three technical mechanisms required for introducing assume-guarantee reasoning into a programming framework: (1) a notion of a module that can be composed with other modules, (2) the ability to describe a module at multiple levels of abstraction and check that a concrete description refines an abstract description, (3) the capability to seamlessly substitute, in the environment of a module, any concrete module with its abstraction. Each of these mechanisms is challenging to design in a language with aforementioned dynamic features. We provide these capabilities in a system called ModP built atop the \( P \) programming language.

We attack challenge (1) by formalizing a collection of dynamically-created and concurrently-executing machines as a module with a semantics that is a collection of traces over visible actions. Our formalization is set up so that syntactic composition of modules \( P \) and \( Q \) leads to semantic intersection over the traces of \( P \) and \( Q \). We attack
challenge (2) by formalizing refinement as trace containment and providing operators for hiding private details of a module. To ensure that our hiding operators lead to sound abstractions, an especially challenging problem in a language where permission to send events flows dynamically through, we invent a programming methodology that is enforced via a combination of static and dynamic checks. We attack challenge (3) by introducing interfaces as a proxy for machines. Instead of creating machines directly, our programming methodology requires creating a machine indirectly via 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 creates interfaces allows great flexibility in specializing machines and substituting one machine for another. ModP provides interface binding and renaming operators to support these activities. Ultimately, ModP offers to the programmer a building-block approach to assembling concurrent and dynamic software components to yield a system that is both easy to understand and amenable to comprehensive testing and verification.

3 MODEL OF COMPUTATION IN MODP

A ModP program comprises state machines communicating asynchronously with each other using events accompanied by typed data values. Each machine has an input buffer, event handlers, and machine-local store. The machines run concurrently with each other, each executing an event handling loop that removes a message from the input buffer and executes the attached message handler. The model of computation is similar to that of Gul Agha’s actor model [2]. The syntax of ModP state machines is derived from P [17], we refer the reader to P manual [18], and the implementation of our case study provided in supplementary material. We adapted the semantics of new and send operation, and added annotations on P state machines as described below.

In the companion Appendix, we present the semantics of ModP state machines in depth. Here we focus on the two key operations that support dynamism:

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. *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 ModP program contains interface and machine declarations. Figure 3 shows the declarations that will be used through the rest of this section to illustrate key ModP features. Each interface declaration 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, E2, or E5. In ModP, the execution of the statement \( \text{id} := \text{new I} \) creates a fresh instance of machine M and stores the unique identifier representing the new machine instance in id. The binding between interface I and machine M is provided separately by the programmer (explained later). The machine reference id also contains the set of events I; we refer to this set as the *permission* contained in id. The machine reference in id is initially known only to the creator but can be communicated to other machines by sending it as a payload value attached to events.

A machine declaration has a machine name and received events but in addition also has sent events and created interfaces. For example, machine M1 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. The ModP compiler checks that any interface created by M1 is mentioned in the creates set of M1. Since events are first-class values that can be stored in variables inside a machine, the correctness of the sends annotation is checked via a combination of static analysis in the ModP compiler and dynamic analysis in the ModP tester. Finally, the ModP compiler checks that if an interface I is bound to M1 in a module, the received events of I are contained in the received events of M1.

Executing the statement \( \text{send t, e, v} \) adds event e with payload value v into the buffer of the target machine t. The send operation is performed only if e is in the permission of machine identifier t and fails otherwise.
Fig. 3. ModP machines

This mechanism ensures that (1) a machine never receives an event that it has not declared in its receive set, and (2) only those events can be sent to a machine using its reference that are contained in its permission set, which is same as the permission with which the machine instance was created. Permissions play a key role in the mechanism used by ModP to ensure correctness of the hide operation that adds private events to a module (Section 4.3).

In ModP, a programmer can specify temporal properties via specification machines. 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. For examples of such machines, we refer the reader to the Appendix.

4 MODP MODULES

Figure 4 presents the various operators supported by ModP for module construction. These operations are essential for building complex real-world systems modularly and hierarchically. The sets InterfaceName and...
MachineName are the sets of names of all interfaces and machines, respectively; these sets are disjoint from each other. The map MRecvs is defined over InterfaceName $\cup$ MachineName; it associates a set of received events with each interface and machine name. The maps MSends and MCreates are defined over MachineName; they associate each machine with its sent events and created interfaces, respectively. A module is constructed using one of the following constructors:

1. **bind** $i_1 \to m_1, \ldots, i_n \to m_n$. This declaration constructs a module with a each interface $i_k$ bound to machine $m_k$.

2. $P \cup Q$. This declaration creates the union of modules $P$ and $Q$.

3. $P \mid Q$. This declaration creates the composition of modules $P$ and $Q$.

4. **hide** $e_1, \ldots, e_k$ in $P$. This declaration hides events $e_1, \ldots, e_k$ in $P$; the events $e_1, \ldots, e_k$ become private events of the resulting module.

5. **hide** $i_1, \ldots, i_k$ in $P$. This declaration hides interfaces $i_1, \ldots, i_k \in \text{InterfaceName}$ in $P$; the machines $m_1, \ldots, m_k$ become private machines of the resulting module.

6. **rename** $i \to i'$ in $P$. This declaration renames interface $i \in \text{InterfaceName}$ to $i' \in \text{InterfaceName}$ in the module $P$.

In the rest of this section, we explain the expressive capability and usefulness of each of these operators. We will use the code in Figure 3 for our examples.

### 4.1 From machines to modules

The unit of compositional reasoning in ModP is a module comprising a collection of machines. A key feature required for compositional reasoning is substitution, the ability to seamlessly bind the interface creation statement `new I` in machine $M$ to either a concrete machine $\text{Impl}$ for execution or an abstract machine $\text{Abs}$ for testing.

To solve this problem, we enhance a module to be a collection of bindings from interface names to machine names. The following code creates a module $P$ that binds interfaces $I_1$ and $I_2$ to machines $M_1$ and $M_2$ (Figure 3), respectively.

```plaintext
module P = bind I1 -> M1, I2 -> 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. In the module $P$, the construction of interface $I_2$ in machine $M_1$ is bound to machine $M_2$. Using the `bind` operator, we can solve the substitution problem easily. The module `bind X -> M, I -> Impl` binds creation of $I$ in $M$ to $\text{Impl}$. But, the module `bind X -> M, I -> Abs` binds creation of $I$ in $M$ to $\text{Abs}$.

The map from interfaces to machines in a module is called its interface definition map. The domain of the interface definition map is the set of exported interfaces for module $P$; these interfaces can be created either by $P$ or its environment. The events received ($ER_P$) by module $P$ is the union of the set of events received by machines $M_1$ and $M_2$. Similarly, the events sent ($ES_P$) and the interfaces created ($IC_P$) by $P$ is the union of events sent and interfaces created by the machines $M_1$ and $M_2$, respectively.

- $IC_P = \{I2, I3\}$
- $ER_P = \{E1, E2, E3\}$
- $ES_P = \{E2, E3, E4, E5\}$

A module in ModP is an open system. The input events of module $P$ are the events in $\{E1\}$ that are received but not sent by $P$. The input interfaces of $P$ are the set of interfaces in $\{I1\}$ that are exported but not created by $P$. The input actions that can be performed by the environment of $P$ is the union of input events and interfaces of $P$. The output actions of $P$ is the union of sent events and created interfaces of $P$. The interface $I3$ created by $P$
is not implemented by any machine in $P$ and can be bound to any machine in the environment of $P$. Similarly, event $E5$ is sent to a machine in the environment of $P$.

A module in ModP is receptive to all its input actions. Receptiveness is guaranteed since the ModP programming model allows unbounded buffers and unbounded machine creation. Hence, a sent event can be enqueued into the target machine’s input buffer and a create command can be satisfied by adding another machine instance to the pool of machines executing inside the module.

An execution of $P$ is a sequence of states and transitions representing an interleaved execution of machines inside $P$ and its environment. Each transition in the execution is either unlabeled or labeled with a visible action which can be a create action or a send action. A create action, corresponding to the execution of a create command $id := \text{new} \ I$, is represented by the label $I$. A send action, corresponding to the execution of a send command $\text{send} \ t,e,v$, is represented by the label $(t,e,v)$. The trace corresponding to an execution is obtained by removing the states and unlabeled transitions from the execution, thereby resulting in a sequence of create and send actions in that execution. Our definition of a trace captures those operations that add dynamism in the system like machine creation and sends with payload values that can have machine-references embedded in them.

Abstraction and refinement of modules play an important role in assume-guarantee reasoning. A module $P$ refines a module $Q$ if (1) every visible label of $Q$ is a visible label of $P$, and (2) the projection of every trace of $P$ onto the visible labels of $Q$ is a trace of $Q$.

### 4.2 Module union and composition

A large system is built by unioning multiple modules together. Intuitively, the union of two modules means that their machines run concurrently. Consider the following module definitions:

```plaintext
module $P =$ bind $I1 \rightarrow M1, I2 \rightarrow M2$;
module $Q =$ bind $I3 \rightarrow M3$;
module $R = P \cup Q$;
```

Here, module $P$ is borrowed from Section 4.1 and the definition of machines $M1$, $M2$, and $M3$ comes from Figure 3. Module $R$ takes the union of $P$ with $Q$. The union of $P$ and $Q$ is allowed only if their exported interfaces are disjoint, which holds because the exported interfaces are $\{I1, I2\}$ and $\{I3\}$, 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 $Q$. The input and output actions of $R$ are computed similarly to the description for module $P$ in Section 4.1.

There are two important points to notice about $R$. First, event $E1$ is a sent event of $Q$ but a received event for $P$. Second, interface $I3$ created by $P$ but not bound to any machine in $P$ is bound to machine $M3$ (from $Q$) in $R$. In $R$, any occurrence of $\text{new} \ I3$ will lead to the creation of an instance of machine $M3$.

Module composition is module union with the extra constraint that the output actions of the modules being composed are disjoint. Since the send events and created interfaces, respectively, of $P$ and $Q$ are disjoint the module $X$ defined below is legal.

```plaintext
module $X = P \parallel Q$;
```

Notice that an output action of $Q$ is allowed to be the same as an input action of $P$ and vice-versa. We now explain the advantage of this extra constraint in module composition.

**Output disjointness.** The requirement that output actions of $P$ and $Q$ be disjoint in order to compose them is important for compositional reasoning. Recall that modeling $P$ as an open system requires $P$ to be receptive only to its input actions (sent by its environment). In other words, for the output send 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.

10
With the ability to dynamically pass machine references as payloads in events, without output disjointness, we might end up in a situation where \( P \) sends \( Q \) a reference to a machine within \( P \), giving \( Q \) the permission (as part of the reference) to send \( P \) an event which it never expected its environment to send it. The requirement that output actions of \( P \) and \( Q \) are disjoint ensures that the composition \( P \parallel Q \) does not violate the assumption made by \( P \) on its environment, thus allowing us to carry over to \( X \) any properties established via local reasoning on \( P \).

**Composition constrains behavior.** Any input action of \( P \) that is an output action of \( Q \) is an output action of \( X \) and hence not a input action of \( X \). 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, 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. In other words, trace containment is monotonic with respect to composition. This permits refinement of components in isolation.

4.3 Hiding events and interfaces

Hiding events and interfaces in a module allows us to construct a more abstract module. 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 legal and \( E2 \) becomes a private event in it. 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 legally hide an interface in module \( P \), it must be both an exported and created interface of \( P \). Module \( K \) is legal 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 legally 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 suitably 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 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, 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 machines with private permissions cannot be created from outside the module. To tackle private permission leakage through (2), ModP require the programmer to annotate each event with a set $A$ of allowed permissions. For an event $e$, the set $A(e)$ must contain any permission that the programmer can put inside the payload accompanying $e$. Through a combination of static and dynamic checking, ModP enforces that (1) each send of event $e$ adheres to this specification, and (2) the set of private events is disjoint from any permission in $A(e)$ for any non-private event $e$ (we refer the reader to the operational semantics of send in Appendix for more details). Together, these two checks ensure that a permission containing a private event does not leak outside the module through sends.

4.4 Renaming interfaces

Consider the following code where interfaces $I_1$ and $I_2$ and machines $M_1$ and $M_2$ are taken from Figure 3.

```plaintext
interface I receives $E_1$;
machine M receives $E_1$, $E_3$;
creates I;
{ /* body */ }

module A = bind $I_1$ → M, $I$ → $M_1$;
module B = bind $I_2$ → M, $I$ → $M_2$;
```

In module $A$, the creates of interface $I$ in machine $M$ is bound to machine $M_1$. But in module $B$ which binds $I_2$ to $M$, the creates $I$ in $M$ is bound to machine $M_2$. Thus, it is no longer possible to take the union of modules $A$ and $B$ because of conflicting bindings of interface $I$. Interface renaming comes to the rescue in such a situation.

```plaintext
interface J receives $E_1$;
module C = rename $I$ → J in B;
```

In module $C$, the interface name $I$ is renamed to $J$. The binding $I$ → $M_2$ is converted to $J$ → $M_2$ and any occurrence of new $I$ in the code of $M$ is automatically interpreted as new $J$. 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 composability.

5 WELL-FORMED MODULES

In this section, we present the rules for checking that a module expression is well-formed. As a side effect of executing these rules, information linking creation of interfaces to concrete machine names is also computed. The judgment $P : EP_P, IP_P, I_P, L_P$ asserts that the module $P \in ModuleExpr$ is well-formed. We read this judgment as: Module $P$ is well-formed with private events $EP_P$, private interfaces $IP_P$, interface definition map $I_P$, and interface link map $L_P$. The elements on the right side of this judgment are explained below.

1. **Private events.** $EP_P \in 2^{EventName}$ is the set of private events in $P$. These events must not cross the boundary of module $P$. If a machine $m$ in $P$ sends event $e \in EP_P$, the target must be some machine $m'$ in $P$. Similarly, if $m$ receives $e \in EP_P$, the sender must be some $m'$ in $P$. 

12
2. **Private interfaces.** $IP_P \in 2^{\text{InterfaceName}}$ is the set of private interfaces in $P$.

3. **Interface definition map.** $IP_i \in \text{InterfaceName} \rightarrow \text{MachineName}$ maps an interface name $i$ to a machine name $IP_i[i]$. Dynamically-created instances of interfaces in $P$ are given identifiers from $(i, n)$ where $i \in \text{dom}(IP_i)$ and $n$ is a non-negative number and $\alpha \subseteq \text{MRecvs}(IP_i[i])$ represents the permissions associated with identifier; the identifier $(i, n, \alpha)$ represents the $n$-th instance of interface $i$ (also the $n$-th instance of concrete machine $IP_i[i]$).

To execute the machine with identifier $(i, n)$ in $P$, the code for concrete machine $IP_i[i]$ is looked up. If the statement `new x` is being executed, the concrete machine actually created in lieu of the interface name $x$ is provided by the machine link map $LP_i[i]$ discussed next.

4. **Interface link map.** $LP_i \in \text{InterfaceName} \rightarrow \text{InterfaceName} \rightarrow \text{InterfaceName}$ maps each interface $i \in \text{dom}(IP_i)$ to a link map that binds machines created by the code of machine $IP_i[i]$ to an interface name. If $(x, x') \in LP_i[i]$, then the compiler interprets name $x$ in statement `new x` in the code of machine $IP_i[i]$ as $x'$ while maintaining the invariant that $x' \in \text{dom}(IP_i)$ and $\text{MRecvs}(x) \subseteq \text{MRecvs}(LP_i[i][x'])$. The first entry for name $x$ ever added to $LP_i[i]$ is the identity map $(x, x)$; subsequently, if interface $x$ is renamed to $x'$, this entry is updated to $(x, x')$. Different machines in $P$ may not agree on the binding for a particular machine name $x$. It is also possible that the code of machine $IP_i[i]$ has an occurrence of `new x` and $x \notin \text{dom}(LP_i[i])$. In this case, the binding for interface $x$ must be provided by another module in the environment of $P$. We refer the reader to the operational semantics of create operation in Appendix for more details about how link maps are used for creating machines.

Figure 5 shows the rules for deriving the judgment $P \vdash EP_P, IP_P, IP, LP$. These rules use the auxiliary judgment $EP_P, IP_P, IP, LP \vdash ER_P, ES_P, IC_P$ for inferring other useful attributes of the module $P$ (rule $\text{InferredAttributes}$).

1. $ER_P$ is the set of non-private events received by machines in $P$.
2. $ES_P$ is the set of non-private events sent by the machines in $P$. This set may have a nonempty intersection with $ER_P$. The send of any event in $ER_P \cup ES_P$ is a visible action.
3. $IC_P \subseteq \text{dom}(IP_i)$ is the set of interfaces created by $P$.


- **Input and output actions are disjoint from private actions:** The set of received events $ER_P$ and sent events $ES_P$ are disjoint from the set of private events $EP_P$.
- **Permissions to send private events does not leak:** A permission to send a private event in $EP_P$ does not leak out of $P$, an important challenge discussed in Section 4.3. To achieve this goal, the rules ensure that for all $e \in ER_P \cup ES_P$ and $\alpha \in \mathcal{A}(e)$, we have $\alpha \cap EP_P = \emptyset$.
- **Interface definition map is consistent:** For each $(i, m) \in IP_i$, we have $\text{MRecvs}(i) \subseteq \text{MRecvs}(m)$.
- **Interface link map is consistent:** The domains of $IP_i$ and $LP_i$ are identical and for all $(i, m) \in IP_i$ and $x \in \text{MCreates}(m)$, we have $x \in \text{dom}(LP_i[i])$.

Next, we describe the rules in Figure 5 in more detail.

Rule $\text{Bind}$ handles the operator bind $i_1 \rightarrow m_1, \ldots, i_k \rightarrow m_k$ that constructs a primitive module by binding each interface $i_k$ to machine $m_k$ for $k \in [1, n]$. These bindings are captured in $f$; condition $(b1)$ checks that $f$ is a function. Condition $(b2)$ checks that the received events of an interface are contained in the received events of the machine bound to it. 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 $f(i)$.

Rule $\text{Union}$ handles the union of modules $P$ and $Q$. Condition $(u1)$ enforces that the domains of $IP_i$ and $IQ_i$ are disjoint, thus preventing conflicting interface bindings. Conditions $(u2)$ ensures that input and output actions
\[ \text{\textsc{InferredAttributes}} \]
\[
\begin{align*}
EP &= \bigcup_{\text{recv}(Ip)} M\text{Recvs}(m) \setminus EP \\
\text{ES} &= \bigcup_{\text{recv}(Ip)} M\text{Sends}(m) \setminus EP \\
IC &= \bigcup_{(i, m) \in EP, x \in M\text{Create}(m)} \{L[i][x]\}
\end{align*}
\]
\[
\text{EP}, IP, LP \vdash \text{EP, IS, IC}
\]
\[
(b) \quad f = \{(i_1, m_1), \ldots, (i_n, m_n)\} \quad f \subseteq \text{InterfaceName} \rightarrow \text{MachineName}^{(b)} \\
\quad \forall (i, m) \in f, \text{MRecvs}(i) \subseteq \text{MRecvs}(m)^{(b)} \\
\quad \text{bind } i_1 \rightarrow m_1, \ldots, i_n \rightarrow m_n \vdash \{\}, \{\}, f, \{(i, x, x) \mid (i, m) \in f \land x \in M\text{Create}(m)\}
\]
\[
\text{\textsc{Union}}
\]
\[
\begin{align*}
P \vdash \text{EP, IP, LP, ER, ES, IC} \\
Q \vdash \text{EP, IP, IQ, LQ, EQ, ICQ} \\
\quad \text{dom}(Ip) \cap \text{dom}(Iq) = \emptyset^{(a)} \\
\quad \forall x \in (\text{dom}(Ip) \land ICp) \cup (\text{dom}(Iq) \land ICQ), \text{MRecvs}(x) \cap (EP \cup EQ) = \emptyset^{(a)} \\
\quad \forall e \in EP \cup EQ \cup ES \cup EQ, \forall \alpha \in \mathcal{A}(e), \alpha \cap (EP \cup EQ) = \emptyset^{(a)} \\
P \cup Q \vdash \text{EP, IP, IQ, LQ, EQ, IC}
\end{align*}
\]
\[
\text{\textsc{Composition}}
\]
\[
\begin{align*}
P \vdash \text{EP, IP, LP, ER, ES, ICp} \\
Q \vdash \text{EP, IP, IQ, LQ, EQ, ICQ} \\
\quad P \cup Q \vdash A, B, C, D \\
\quad ICp \cap ICQ = \emptyset^{(c)} \\
\quad ESp \cap ESQ = \emptyset^{(d)} \\
\quad P \parallel Q \vdash A, B, C, D
\end{align*}
\]
\[
\text{\textsc{HideEvent}}
\]
\[
\begin{align*}
P \vdash \text{EP, IP, LP, ER, ES, ICp} \\
\quad \beta = \{e_1, \ldots, e_n\} \\
\quad \beta \subseteq \text{EP} \cap \text{ES}^{(h)} \\
\quad \forall x \in ICp \land \text{dom}(Ip), \text{MRecvs}(x) \cap \beta = \emptyset^{(h)} \\
\quad \forall e \in (EP \cup ES) \setminus \{\}, \forall \alpha \in \mathcal{A}(e), \alpha \land \beta = \emptyset^{(h)}
\end{align*}
\]
\[
\text{\textsc{HideInterface}}
\]
\[
\begin{align*}
P \vdash \text{EP, IP, LP, ER, ES, ICp} \\
\quad \beta = \{i_1, \ldots, i_n\} \\
\quad \beta \subseteq \text{dom}(Ip) \cap ICp^{(h)} \\
\quad \text{hide } i_1, \ldots, i_n \text{ in } P \vdash \text{EP, IP, LP, ER, ES, ICp}
\end{align*}
\]
\[
\text{\textsc{Rename}}
\]
\[
\begin{align*}
P \vdash \text{EP, IP, LP, ER, ES, ICp} \\
\quad i \in \text{dom}(Ip) \land ICp^{(r)} \\
\quad i' \in \text{InterfaceName} \setminus (\text{dom}(Ip) \cup ICp)^{(r)} \\
\quad M\text{Recvs}(i) = M\text{Recvs}(i')^{(r)} \\
\quad A = \{x \mid x' \in IP, \land x = it(x' = i, i', x')\}^{(r)} \\
\quad B = \{(x, y) \mid (x', y) \in IP \land x = it(x' = i, i', x')\}^{(r)} \\
\quad C = \{(x, y, z) \mid (x', y, z') \in LP \land x = it(x' = i, i', x') \land z = it(x' = i, i', z')\}^{(r)} \\
\quad \text{rename } i \rightarrow i' \text{ in } P \vdash \text{EP, A, B, C}
\end{align*}
\]

Fig. 5. Rules for well-formed modules (A∩B = (A \ B) \cup (B \ A))

of P are not hidden by private events of Q and vice-versa. Conditions (u3) and (u4) together check that private permissions of $P \cup Q$ do not leak out. Condition (u3) checks that creation of an input interface of P does not leak a permission containing a private event of $P \cup Q$ and vice-versa. Condition (u4) checks that non-private events sent or received by P do not leak a permission containing a private event of $P \cup Q$ and vice-versa. Rule COMPOSITION handles the composition of P and Q. The composition of modules P and Q is essentially their union with the added requirement that their output actions are disjoint. Condition (c1) checks that created interfaces are disjoint; condition (c2) checks that sent events are disjoint. Both union and composition are associative and commutative. Hence, these rules can be extended to an arbitrary collection of modules; the order is immaterial.

Rule HideEvent handles the hiding of events $e_1, \ldots, e_n$ in module P. This rule adds $\beta$, where $\beta = \{e_1, \ldots, e_n\}$, to EP. Condition (he1) checks that 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 \( \text{HideInterface} \) handles the hiding of interfaces \( i_1, \ldots, i_n \) in module \( P \). This rule adds \( \{i_1, \ldots, i_n\} \) to \( IP_P \).

Condition (hi1) is similar to the condition (he1) of rule \( \text{HideEvent} \); this condition ensures that the resulting module is an abstraction of \( P \).

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

6 COMPOSITIONAL REASONING USING MODP MODULES

In this section, we introduce the principles of compositional reasoning supported by the ModP module system. Using the distributed services example, we describe how ModP enables compositional testing using test-declarations. Finally, we provide a brief overview of the implementation of the ModP systematic testing engine that performs both safety and refinement checking.

6.1 Principles of Compositional Reasoning

For two compatible modules \( P \) and \( Q \), ModP compiler ensures that the composition \( P || Q \) is again a module (Section 4.2). Composition behaves like language intersection. This is captured by Theorem 6.1, which asserts that traces of a composed module are completely determined by the traces of the component modules. This theorem forms the foundational basis of our theory of compositional refinement and is used for proving the theorems underlying our compositional testing methodology (Theorems 6.4 and 6.5).

**Theorem 6.1 (Composition Is Intersection).** Let \( P \) and \( Q \) be two compatible and well-formed modules. For any trace \( \pi \), we have \( \pi \) is a trace of \( P || Q \) iff the projection of \( \pi \) over visible actions of \( P \) is a trace of \( P \) and the projection of \( \pi \) over visible actions of \( Q \) is a trace of \( Q \).

(Proof is in the appendix)

ModP compiler ensures that legal modules constructed using operators like hiding events, renaming interfaces and others are well-formed.

Theorem 6.2 states that the hide operation is hierarchical and compositional in nature.

**Theorem 6.2 (Hide).** For well-formed modules \( P \) and \( Q \) and a set of events or machines \( \alpha \), 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 is in the appendix)

Theorem 6.3 states that the renaming interfaces operation is compositional in nature.

**Theorem 6.3 (Rename).** For well-formed modules \( P \) and \( Q \), if \( (\text{rename } i \to i' \text{ in } P) \) and \( (\text{rename } i \to i' \text{ in } Q) \) are well-formed, and \( P \leq Q \), then \( (\text{rename } i \to i' \text{ in } P) \leq (\text{rename } i \to i' \text{ in } Q) \).

The assume-guarantee principle allows reasoning about components \( P \) and \( Q \) of a system separately using their respective abstractions. For example, if module \( Q \) refines module \( QA \) and a safety property \( \phi \) holds for \( P || QA \), then \( \phi \) must hold for \( P || Q \) also. Often, a more flexible, circular form of reasoning is also permitted: if \( P || QA \) refines module \( PA \), \( PA || Q \) refines \( QA \), and a safety property \( \phi \) holds for \( PA || QA \), then \( \phi \) holds for \( P || Q \) also.

We now present the two key theorems that describe this principle of circular assume guarantee reasoning for ModP systems. First, we introduce a generalized composition operation \( \| \mathcal{P} \), where \( \mathcal{P} \) is a nonempty list of modules. This operator represents the composition of all modules in \( \mathcal{P} \). The binary parallel composition operator
is both commutative and associative. Thus, it is also possible to think of \( \| P \) as the module obtained by composing the modules in \( P \) in some arbitrary order; the result is independent of the order chosen.

We use the symbol \( \oplus \) to represent concatenation operation for list of modules. Let \( P \) and \( Q \) be lists of modules, we say that \( P \) is a sublist of \( Q \), if \( P \) can be obtained by dropping some of the modules in \( Q \). A module is safe if none of its executions lead to an error state (definition in Appendix).

**Theorem 6.4 (Safety).** Let \( \| P \) and \( \| Q \) be well-formed. Let \( \| P \) refine each module \( Q \in Q \). Suppose for each \( P \in P \), there is a sublist \( X \) of \( P \oplus Q \) such that \( P \in X \), \( \| X \) is well-formed, and \( \| X \) is safe. Then \( \| P \) is safe. (Proof is in the appendix)

When using Theorem 6.4 in practice, modules in \( P \) and \( Q \) typically consists of implementation and abstraction modules respectively. When proving safety of any module \( P \in P \), it is allowed to pick any modules in \( Q \) for constraining the environment of \( P \).

To use Theorem 6.4, we need to show that \( \| P \) refines each module \( Q \in Q \) which requires reasoning about all modules in \( P \) together. The following theorem shows that the refinement between \( \| P \) and \( Q \) can also be checked compositionally.

**Theorem 6.5 (Circular Assume-Guarantee).** Let \( \| P \) and \( \| Q \) be well-formed. Suppose for each module \( Q \in Q \) there is a sublist \( X \) of \( P \oplus Q \) such that \( Q \not\in X \), \( \| X \) is well-formed, and \( \| X \) refines \( Q \). Then \( \| P \) refines each module \( Q \in Q \).

(Proof is in the appendix)

Theorem 6.5 states that to show that \( \| P \) refines \( Q \in Q \), any subset of modules in \( P \) and \( Q \) can be picked as long as \( Q \) is not picked. Therefore, it is possible to perform sound circular reasoning, i.e., use \( Q_2 \) to prove refinement of \( Q_2 \) and \( Q_3 \) to prove refinement of \( Q_1 \). This capability of circular reasoning is essential for compositional testing of the distributed systems we have implemented.

### 6.2 From theory of compositional reasoning to compositional testing

Theorems 6.4 and 6.5 show that there are two kinds of proof obligations that result from assume-guarantee reasoning—safety and refinement. ModP using systematic testing for validating the proof obligations. ModP allows the programmer to write each of these obligations as test declarations. Discharging these test obligation depends on a systematic testing engine with capabilities for performing refinement checking, described in Section 6.3.

**Test declarations.** Figure 6 presents the syntax for test declarations. The declaration \texttt{test t0 : P} introduces a safety test obligation that the executions of module \( P \) do not result in a failure (\( P \) is safe). The declaration \texttt{test t1 : P refines Q} introduces a test obligation that module \( P \) refines module \( Q \). \texttt{assert s in P} declaration attaches specification machine \( s \) to module \( P \) (Section 3).

**Test drivers.** For systematic testing of each test declaration, we require that the modules \( P, Q \) in \texttt{test t0 : P} or \texttt{test t1 : P refines Q} are closed using a test-harness module. The test-harness or test-driver module consists of non-deterministic machines that close the module under test by either supplying inputs or injecting failures. The programmer can write a collection of test-drivers for each module under test to cover various test-scenarios.
To summarize, (test $T_0 : \text{TestDriv} \parallel P$) tests that all executions in the closed module $\text{TestDriv} \parallel P$ are safe and (test $T_1 : \text{TestDriv} \parallel P \text{ refines} \text{TestDriv} \parallel Q$) tests that all traces of the closed module $\text{TestDriv} \parallel P$ are contained in traces of $\text{TestDriv} \parallel Q$

**Example.** We revisit the example in Section 2 and show how we tested the fault tolerant transaction commit service compositionally using the principles of assume-guarantee reasoning discussed earlier.

The set of implementation modules for the fault tolerant transaction commit service are $P = \{\text{MultiPaxosSMR}, \text{TwoPhaseCommit}, \text{OSServ}\}$. We used the abstractions modules $Q = \{\text{LinearAbs}, \text{ClientAbs}\}$ for compositionally testing the implementation.

The testing problem is to validate that $\parallel P$ satisfy the safety property $\text{TransCommitSpec}$ (implemented as spec-machine). This problem is same as testing that the following module is safe:

$$ (\text{assert TransCommitSpec in TwoPhaseCommit}) \parallel \text{MultiPaxosSMR} \parallel \text{OSServ} \tag{6} $$

To test the safety of module in 6, we use the Safety Theorem 6.4 and assume that:

$$ \parallel P \leq \text{LinearAbs, ClientAbs} \tag{7} $$

We perform the following tests:

- test $T_0 : \text{TestDriv1} \parallel \text{MultiPaxosSMR} \parallel \text{OSServ} \parallel \text{ClientAbs}$
- test $T_1 : \text{TestDriv2} \parallel \text{LinearAbs} \parallel (\text{assert TransCommitSpec in TwoPhaseCommit}) \parallel \text{OSServ} \tag{8}$

Using assumption (7) and the test declarations $T_0, T_1$, we conclude that the module in 6 is tested to be safe. We perform the following checks to test that the assumption (7) holds:

- test $T_2 : \text{TestDriv3} \parallel \text{MultiPaxosSMR} \parallel \text{OSServ} \parallel \text{ClientAbs}$
- refines $\text{TestDriv3} \parallel \text{LinearAbs} \parallel \text{OSServ} \parallel \text{ClientAbs}$
- test $T_3 : \text{TestDriv4} \parallel \text{LinearAbs} \parallel (\text{assert TransCommitSpec in TwoPhaseCommit}) \parallel \text{OSServ}$
- refines $\text{TestDriv4} \parallel \text{LinearAbs} \parallel \text{ClientAbs} \parallel \text{OSServ} \tag{9}$

Note that the test cases $T_2$ and $T_3$ are circular and they together imply that the assumption (7) holds (using Theorem 6.5).

**Remark:** Although the focus of this paper is on scalable testing with guarantees, other researchers could exploit our theory to do proofs.

### 6.3 Implementation of ModP’s systematic testing engine

The ModP compiler performs static analysis of the source code and generates executable C and C# code. Parts of the compiler corresponding to machines and specifications are borrowed from P [17], a programming language based on states, transitions, structured event handlers, and functional datatypes. The ModP compiler generates C# code corresponding to each test declaration. The ModP systematic testing engine takes as input the generated C# code and systematically enumerates (exhaustive or sampling) executions resulting from scheduling and explicit nondeterministic choices. ModP uses stratified search priorization techniques based on delay bounding [19, 22] for finding bugs faster.

The systematic testing engine performs refinement checking of ModP programs based on trace containment. Our algorithm for checking $P \leq Q$ (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.
### 7 EVALUATION

We evaluate ModP along three dimensions:

1. We demonstrate that compositional reasoning significantly reduces the state space for systematic testing of distributed systems, and finds more critical bugs (also faster) than the monolithic approach (Section 7.2).
2. We present programmer effort required (in terms of source lines of code) for building and compositionally testing a real-world distributed system using ModP (Section 7.3).
3. We present anecdotal evidence of the benefits of using a refinement based approach to validate the soundness of abstractions used during testing (Section 7.4).

**Experimental setup:** We conducted all the systematic testing experiments on a server with Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), and 160GB RAM.

### 7.1 Case Study: Distributed services software stack

We use the modularity features of ModP to implement and test the software stack for two distributed services shown in Figure 1: (i) distributed atomic commit of updates to decentralized, partitioned data using two-phase commit \([9, 28, 45, 62]\), and (ii) distributed data structures such as hashtables and lists. These distributed services use State Machine Replication (SMR) for fault-tolerance \([29, 42, 63]\).

We implement distributed transaction commit using the two-phase commit protocol, which uses a single coordinator state-machine to atomically commit updates across multiple participant state-machines. Hashtable and list are implemented as deterministic state-machines with PUT and GET operations. These services by themselves are not tolerant to node failures. A common approach for achieving fault tolerance is to use state-machine replication (SMR), with protocols such as Multi-Paxos \([43]\) and Chain Replication \([70]\), 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 hashtable (list) state-machines across multiple nodes. Multi-Paxos and Chain Replication in turn use different sub-protocols. As mentioned in Section 2, though 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.
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 [28, 43, 45, 70]. Figure 7 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. The details about the framework and runtime are provided in Appendix but is not the focus of this paper.

(The ModP source code for the case study is available as an anonymous supplementary material and we refer the reader to Appendix for more details about the code.)

7.2 Compositional testing

We implemented the fault-tolerant distributed services as a composition of ModP modules. The distributed protocols were implemented from the high-level specifications; however as highlighted in [14], there is a big gap between the specification and the actual implementation of a distributed protocol. In our experience, ModP helps bridge this gap finding bugs both at the specification and implementation level.

We used the principles of assume-guarantee reasoning described in Section 6.1 to compositionally test the entire software stack. The system-level testing problem was decomposed into smaller sub-problems of testing each protocol in isolation, each of the sub-problem was discharged by writing a combination of both safety and refinement test-declarations. We used abstractions of protocols during compositional reasoning which led to state-space reduction and hence amplification of the test-coverage leading to uncovering many critical bugs in our implementation. Some of these bugs require subtle interleaving of the failure injector (multiple failures) which were hard to find using monolithic testing approach because of state-space explosion. We also validated that the abstractions used during compositional reasoning are sound, which helped us not miss bugs because of unsound abstractions (Section 7.4). Overall, we found around 70+ bugs during the development process and most of these bugs were not found using monolithic testing with prioritized searching.

To demonstrate 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: It is well-known that in most cases abstractions help in simplifying the testing problem by reducing the state-space to be explored. 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 (Figure 8) to count the number of unique executions in the implementation of a protocol that map to a trace in its abstraction. Figure 8a 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 8b shows that the graph is highly skewed for more complex protocols like chain replication. The linearizability abstraction has 1931 traces for the finite test-harness and there were exponentially many executions in the chain replication implementation. We sampled \(10^6\) unique executions in the CR implementation for this experiment. The graph in Figure 8b 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
(a) Leader Election Protocol
(A) Hard to find executions or false positives, (B) Low probability executions, (C) High probability executions

(b) Chain Replication Protocol

Fig. 8. **Test-amplification via abstractions:** X-axis = Traces in the abstraction sorted based on the number of implementation executions that map onto the abstraction trace. Y-axis = Number of executions in the implementation that maps (projects) to the trace in abstraction.

<table>
<thead>
<tr>
<th>Bug in Protocol</th>
<th>No. of Schedules explored</th>
<th>Monolithic</th>
<th>CST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Paxos (bug1)</td>
<td>13</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Two-Phase-Commit (bug2)</td>
<td>1944</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>Chain-Repl. (bug3)</td>
<td>2018</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Multi-Paxos (bug4)</td>
<td>NF</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Two-Phase-Commit (bug5)</td>
<td>NF</td>
<td>112</td>
<td></td>
</tr>
<tr>
<td>Chain-Repl. (bug6)</td>
<td>NF</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>Chain-Repl. (bug7)</td>
<td>NF</td>
<td>782</td>
<td></td>
</tr>
<tr>
<td>Multi-Paxos (bug8)</td>
<td>NF</td>
<td>2176</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Comparing compositional systematic testing (CST) with monolithic testing. (NF: Bug not found)

$10^6$ 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 testing against monolithic testing. As described in Section 6.3, the ModP uses prioritized searching techniques for systematic testing each test-declaration. We introduced 8 bugs 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 (only implementation modules) 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 1 shows that ModP-based compositional approach finds bugs faster that the monolithic approach and in some cases the monolithic approach fails to find the bug even after exploring $10^6$ different schedules.
7.3 Programmer effort

Table 2 shows a five-part breakdown, in source lines of code, of our ModP implementation of the distributed service. The Impl. column represents the detailed implementation of each module whose – generated C code can be deployed on the production cluster. Specs. column represents the component-level temporal properties of each protocol implemented as spec-machines. Abst. column represents sound 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 6.1

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Commit</td>
<td>Two Phase Commit</td>
<td>441</td>
<td>61</td>
<td>41</td>
<td>35</td>
<td>38</td>
</tr>
<tr>
<td>Chain Replication</td>
<td>Chain Repl. server</td>
<td>401</td>
<td>220</td>
<td>83</td>
<td>130</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>Fault Detector</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Master Protocol</td>
<td>126</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-Paxos</td>
<td>Multi-Paxos server</td>
<td>421</td>
<td>101</td>
<td>81</td>
<td>92</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Leader Election</td>
<td>65</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data structures</td>
<td>List, Hashset</td>
<td>275</td>
<td>25</td>
<td>-</td>
<td>89</td>
<td>25</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1779</strong></td>
<td></td>
<td></td>
<td><strong>1086</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 2. ModP source lines of code for different protocols

7.4 Refinement checking

Theory of compositional refinement helps conclude that the “whole-system” satisfies a property if each of the components satisfy the property in isolation. For mitigating the state-space explosion problem, the components in the environment are replaced by their abstractions. Meaningful testing requires that these abstractions are sound abstractions of the components being replaced. ModP uses trace containment based approach to check refinement (Section 6.3). We were able to uncover several scenarios where bugs were missed during testing because of an unsound (under-approximating) abstraction. Our refinement checking algorithm generates counter examples represented by traces in implementation that are not contained in the abstraction along with the first point of deviation.

As an example, the linearizability abstraction presented in Figure 2 is used when testing the distributed services built on top of SMR. The abstraction guarantees that for every request there is atmost one response. For chain replication protocol (as described in [70]), 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 during testing which could have led to missed bugs and hence helped increase our confidence about the “whole-system” correctness.

In our experience based on the case-study considered in this paper and other systems built using ModP, most of the modules have simple and well-understood abstractions; examples include leader election, consistency, and linearizability abstractions. The number of false positives uncovered in our development process was low (< 5) compared to the real bugs that we found (70).
Our work on compositional systematic testing is inspired by the theory of assume-guarantee reasoning, introduced by Abadi and Lamport [1], implemented in model checkers [5, 52, 54], and successfully used for hardware verification [21, 34, 53] and software testing [10, 68]. The I/O automata [51] model uses transition labels for synchronization between processes; in that respect, it is similar to our formalization of module composition. Dynamic I/O automata [8] allows dynamic creation and deletion of processes as well. However, the I/O automaton model does not have a notion of process identifiers and hence does not support dynamic communication topology. Reactive modules [4] is a modeling language for message-based systems. Communication between modules is done via single-writer multiple-reader shared variables and a shared global clock that drives each module in lock step. Module constructors like composition, hiding and renaming are well-studied in the context of other formalisms like Reactive modules and I/O automaton, our treatment of these operators is novel because of fundamental differences in the supported programming model. Our work improves upon all of this prior work by targeting a more general and dynamic model of computation and by incorporating these ideas into a practical programming framework.

There are two great traditions for reasoning about concurrent systems—program logics deriving from Hoare logic [25, 35] and process calculi deriving from Hoare’s CSP [36] and Milner’s CCS [55]. The former include rely-guarantee reasoning [27, 67, 73] and concurrent separation logic [24, 49, 59], while the latter include π-calculus [56, 60], join calculus [26], and more recent work like session-types [12, 20, 37] and behavioral-types [6]. Session types encode abstractions in the type language whereas our abstractions are distinct machines capable of expressing arbitrary safety properties. Actor services [66] is a more recent work that proposes program logic for proofs of actor programs. It can be used to prove liveness properties. It requires fine grained specifications at the level of event-handler, in our case programmer writes specifications for components as abstractions. 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 and localized testing problems. The focus on localized testing instead of proof allows us to attach an abstraction to an entire protocol rather than individual actions within that protocol, thereby reducing the extra annotations required for validation.

Researchers have built testing tools for message-passing systems. JCute [64] and Basset [46] are tools for automated unit testing of Java actor programs. The latter uses Java Pathfinder (JPF) [71] to explore concurrent executions. Mace [39], TeaPot [13] and P [17] provide abstractions to support implementation and specification of asynchronous systems. These language mechanisms are backed by techniques for systematic testing of the nondeterministic program semantics. MaceMC [40] and MoDist [74] 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. Demeter [30] improves the scalability of MoDist by exploiting the structure of a large distributed system as a collection of protocols. In our work, we provide linguistic support for programmers to write down these abstractions, not just for distributed protocols but for general programs written in the actor model.

**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 Damien Zufferey and Daniel Fremont 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.
A FORMALISM OF MODP MODULES

In this section, we first present the model of computation for ModP programs and formalize ModP state machines (Section A.2). We present the operational semantics for modules (Section A.3) and definitions for executions, traces, and refinement relation between modules (Section A.4).

A.1 Definitions

In this section, we present the definitions needed for the rest of the formalism.

1. Events: Let $E$ represent the set of names of all the events.
2. Permissions: An permission is a nonempty subset of $E$; Let $I$ represent the set of all permissions ($2^E$).
3. Machines Names: Let MachineName represent the set of names of all the machines.
4. Interface Names: Let InterfaceName represent the set of names of all the interfaces.
5. Machine-State: Let $S$ represent the set of all possible states a machine could have during execution.
6. Payload-Values: Let $V$ represent the set of all possible payloads that may accompany an event in a send action.
7. Machine-Buffer: The input buffer of a machine is a sequence of $(e, v) \in E \times V$ pairs; Let $B$ represent the set of all possible buffer values.
8. Machine-Identifier: The set $Z$ is the set of all the machine identifiers. We concretize the set of machine identifiers $Z$ as $InterfaceName \times I \times S$. An identifier $(i, n, \alpha)$ inside $P$ refers to the $n$-th instance of interface represented by $i \in InterfaceName$. We refer to the final component $\alpha$ of a machine identifier as its permission. This set represents all events that may be sent via this machine identifier using the Enq transition relation.
9. Embedded Machine-Identifiers: For our formalization, we are interested in machine identifiers that are embedded inside the data structures in a machine local-store $s \in S$ or a value $v \in V$. Instead of formalizing all datatypes in ModP, we assume the existence of a function $ids$ such that $ids(s)$ is the set containing all machine identifiers embedded in $s$ and $ids(v)$ is the set containing all machine identifiers embedded in $v$.

A.2 State Machines in ModP

A state-machine in ModP is a tuple $(MRecvs, MSends, MCreates, Local, Rem, Enq, New)$ with the following components:

1. $MRecvs \subseteq E$ is the a nonempty set of events received by the machine.
2. $MSends \subseteq E$ is the set of all events sent by the machine.
3. $MCreates \subseteq InterfaceName$ is the set of interfaces created by the machine.
4. $Local \subseteq S \times Z \times S \times Z$ is the transition relation for local computation in the machine. The local store of a machine is a pair $(s, id) \in S \times Z$. The first component $s$ is the local-store containing everything that can influence the execution of the machine, including control stack and data structures. The second component $id$ is used specifically to define the operational semantics of ModP operations like new and send. It is a placeholder used to receive the identifier of a freshly-created machine or to indicate the target of a send operation. If $(s, id, s', id') \in Local$, then the local store can move from $(s, id)$ to $(s, id')$, which allows us to model the movement of machine identifiers from $s$ to $id$ and vice-versa.
5. $Rem \subseteq S \times B \times N \times S$ 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 $s'$.
6. $Enq \subseteq S \times Z \times E \times V \times S$ is the transition relation for sending a message to a machine. If $(s, id, e, v, s') \in Enq$, then event $e$ with payload $v$ is sent to machine $id$ and the local state of the sender moves from $s$ to $s'$.
7. $New \subseteq S \times InterfaceName \times S$ is the transition relation for creating an interface. If $(s, i, s') \in New$, then the interface $\alpha$ is created and the machine moves from $s$ to $s'$.
We refer to components of machine \( m \in \text{MachineName} \) as \( M\text{Recvs}(m) \), \( M\text{Sends}(m) \), \( M\text{Creates}(m) \), \( \text{Local}(m) \), \( \text{Rem}(m) \), \( \text{Enq}(m) \), and \( \text{New}(m) \), respectively.

A.3 Operational Semantics of ModP Modules

As described in Section 4, a well-formed module \( P \) can be represented by the tuple \((EP_P, IP_P, LP_P, LP_P)\). For defining the operational semantics of ModP modules, we reuse the definitions of inferred attributes for a module \( P \) described in Section 4, the set of \( \text{events received} \) \((ER_P)\), the set of \( \text{events sent} \) \((ES_P)\), the set of \( \text{interfaces created} \) \((IC_P)\).

**Configuration:** A configuration of a module is a tuple \((S, B, C)\):

1. The first component \( S \) is a partial map from \( \text{InterfaceName} \times \mathbb{N} \) to \( S \times \mathbb{Z} \). If \((i, n) \in \text{dom}(S)\), then \( S[i, n] \) is the local state of the \( n\)-th instance of interface represented by \( i \). The local state \( S[i, n] \) has two components, local store \( s \in S \) and a machine identifier \( id \in \mathbb{Z} \).
2. The second component \( B \) is a partial map from \( \text{InterfaceName} \times \mathbb{N} \) to \( B \). If \((i, n) \in \text{dom}(B)\), then \( B[i, n] \) is the input buffer of the \( n\)-th instance of the interface represented by \( i \).
3. The third component \( C \) is a partial map from \( \text{InterfaceName} \) 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 \((S^0_P, B^0_P, C^0_P)\) of any module \( P \) is defined as follows:

1. \( S^0_P \) are empty maps.
2. The domain of \( C^0_P \) is \( \text{InterfaceName} \). \( C^0_P \) maps each element in its domain to 0.

We formalize 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.

\[
\begin{align*}
(S_P[i, n] = (s, id), \quad & (s, id, s', id') \in \text{Local}(LP_P[i]) & S_P[i, n] = (s, id) & B_P[i, n] = b \\
(S_P, B_P, C_P) \xrightarrow{\text{INTERNAL}} (S_P [(i, n) \mapsto (s', id')], B_P, C_P) \quad & (s, b, pos, s') \in \text{Rem}(LP_P[i]) \quad b' = \text{rem}(b, pos) \\
(S_P, B_P, C_P) \xrightarrow{\text{REMOVE-EVENT}} (S_P [(i, n) \mapsto (s', id')], B_P [(i, n) \mapsto b'], C_P)
\end{align*}
\]

Fig. 9. Rules for local computation

**Rules for local computation:** Figure 9 contains the rules for local computation. Rule \text{INTERNAL} picks an interface \( i \) and instance number \( n \) and updates \( S[i, n] \) according to the transition relation \( \text{Local} \), leaving \( B \) and \( C \) unchanged. The map \( LP_P \) is used to obtain the concrete machine corresponding to the interface \( i \). Rule \text{REMOVE-EVENT} updates \( S[i, n] \) and \( B[i, n] \) according to the transition relation \((s, b, pos, s') \in \text{Rem}(LP_P[i])\), the entry in \( pos\)-th position of \( B[i, n] \) is removed and the local state in \( S[i, n] \) is updated to \( s' \) leaving the machine identifier \( id \) unchanged. The transition for both these rules is labeled with \( e \) to indicate that the computation is local and is an internal transition of the module \( P \).

**Rules for creating interfaces:** Let \( s_0 \in S \) represent a state such that \( \text{ids}(s_0) = \emptyset \). Let \( b_0 \in B \) be the empty sequence over \( E \times V \). Figure 10 contains the rules for interface creation. In all these rules, \( LP_P \) 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 \text{ENVIRONMENT-CREATE} creates an interface that is neither create nor exported by \( P \); consequently, it updates \( C \) by incrementing the number of instances of \( i \) but leaves \( S \) and \( B \) unchanged. The rule \text{INPUT-CREATE} creates an interface \( i \) exported by \( P \) that is
(ENVIRONMENT-CREATE)
\[ i \in \text{InterfaceName} \setminus \{IC_p \cup dom(I_p)\} \quad n = C_p[i] \]
\[ (S_p, B_p, C_p) \xrightarrow{i} (S_p, B_p, C_p[i \mapsto n + 1]) \]

(INPUT-CREATE)
\[ i \in \text{dom}(I_p) \setminus IC_p \quad n = C_p[i] \quad id = (i, n, MRecvs(i)) \]
\[ (S_p, B_p, C_p) \xrightarrow{i} (S_p[i, n] \mapsto (s_0, id)), B_p[(i, n) \mapsto b_0], C_p[i \mapsto n + 1]) \]

(CREATE-BAD)
\[ S_p[i, n] = (s, \_\_i, \_\_i) \in \text{New}(I_p[i]) \quad \neg \text{CreateOk}(I_p[i], i') \]
\[ (S_p, B_p, C_p) \xrightarrow{i''} \text{error} \]

(OUTPUT-CREATE-1)
\[ S_p[i, n] = (s, \_, \_) \quad (s, i', s') \in \text{New}(I_p[i]) \]
\[ \text{CreateOk}(I_p[i], i') \quad i'' = L_p[i][i'] \quad n' = C_p[i''] \quad \_\_i' \notin \text{dom}(I_p) \quad id' = (i'', n', MRecvs(i'')) \]
\[ (S_p, B_p, C_p) \xrightarrow{i''} (S_p[(i, n) \mapsto (s', id')], B_p, C_p[i'' \mapsto n' + 1]) \]

(OUTPUT-CREATE-2)
\[ S_p[i, n] = (s, \_, \_) \quad (s, i', s') \in \text{New}(I_p[i]) \quad \text{CreateOk}(I_p[i], i') \quad i'' = L_p[i][i'] \]
\[ i'' \in \text{dom}(I_p) \setminus IP_p \quad n' = C_p[i''] \quad \_\_i' \notin \text{dom}(I_p) \quad id' = (i'', n', MRecvs(i'')) \quad id'' = (i'', n', MRecvs(I_p[i''])) \]
\[ (S_p, B_p, C_p) \xrightarrow{i''} (S_p[(i, n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], B_p[(i'', n') \mapsto b_0], C_p[i'' \mapsto n' + 1]) \]

(OUTPUT-CREATE-3)
\[ S_p[i, n] = (s, \_, \_) \quad (s, i', s') \in \text{New}(I_p[i]) \quad \text{CreateOk}(I_p[i], i') \]
\[ i'' = L_p[i][i'] \quad i'' \in IP_p \quad n' = C_p[i''] \quad \_\_i' \notin \text{dom}(I_p) \quad id' = (i'', n', MRecvs(i'')) \quad id'' = (i'', n', MRecvs(I_p[i''])) \]
\[ (S_p, B_p, C_p) \xrightarrow{i''} (S_p[(i, n) \mapsto (s', id'), (i'', n') \mapsto (s_0, id'')], B_p[(i'', n') \mapsto b_0], C_p[i'' \mapsto n' + 1]) \]

Fig. 10. Rules for creating interfaces

not created by \( P \). The instance number of the new interface is \( C[i] \); its local-store is initialized to \( (s_0, id) \) where \( id \) in this case stores the “self” identifier. Self-identifier is an identifier that references the machine itself. Note that the environment cannot create an interface that is also created by \( P \), this is to ensure that the output actions of modules are disjoint.

The rule CREATE-BAD creates a transition into error if the interface \( i' \) being created by machine \( (i, n) \) violates the predicate CreateOk(m, x) = x \in MCreates(m). Thus, machine \( (i, n) \) may only create machines in MCreates(I_p[i]). The rule OUTPUT-CREATE-1 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-2 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 MRecvs(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 MRecvs(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 $P$. The rule OUTPUT-CREATE-3 allows the creation of a private interface in $P$ (checked by $i^\prime \in Ip_P$); consequently, the transition is labeled with $e$ and is an internal transition. Notice that in these rules the link map $(L_P)$ is used to look-up the interfaces $i'$ to be created corresponding to (new $i'$).

**Rules for sending events:** Figure 11 contains 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 $e \in MRecvs(Ip[i])$ and $e$ is neither private in $P$ nor sent by $P$ and $v$ does not contain any machine identifiers with private permissions. Note that the environment cannot enqueue an event that is sent by $P$. This restriction is required so that the output actions are disjoint.

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 machine $m$, the permission in the target machine identifier $\alpha$, event $e$ and payload $v$ violates the predicate $SendOk(m, \alpha, e, v) = e \in MSends(m) \land e \in \alpha \land \forall(\_, \_, \beta) \in ids(v). \beta \in \mathcal{A}(e)$. Thus, machine $(i, n)$ may only send events declared by it in $MRecvs(Ip[i])$ and allowed by the permission $\alpha$ of the target machine and should not embed machine identifiers with private permissions in the payload $v$. The rule OUTPUT-SEND-1 sends an event to machine outside $P$ whereas rules OUTPUT-SEND-2 and OUTPUT-SEND-3 send an event to some machine inside $P$. In the former, the target
We present the definitions needed for the formalisms and proofs in this section. 

A.4 Execution, Trace and Refinement
For brevity, we will refer to a configuration \((S^k, B^k, C^k)\) as \(G^k\).

**Definition A.1 (Execution).** An execution of \(P\) is a finite sequence \(G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n\) for some \(n \in \mathbb{N}\) such that \(G^i \xrightarrow{a_i} G^{i+1}\) for each \(i \in [0, n)\).

Let \(\text{execs}(P)\) represents the set of all possible executions of the module \(P\). An execution is *unsafe* if \(G^n \xrightarrow{\epsilon} \text{error}\); otherwise, it is *safe*.

**Definition A.2 (Safe Module).** The module \(P\) is *safe*, if for all \(\tau \in \text{execs}(P)\), \(\tau\) is a safe execution. The module \(P\) is *unsafe* if it is not safe.

**Definition A.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 A.4 (Module-Signature).** The signature of a module \(P\) is the set \(\Sigma_P = (\text{InterfaceName} \setminus IP_P) \cup ((\text{InterfaceName} \times \mathbb{N}) \times (ES_P \cup ER_P) \times \mathcal{V})\). The signature is partitioned into output signature \((\text{IC}_P \setminus IP_P) \cup ((\text{InterfaceName} \times \mathbb{N}) \times ES_P \times \mathcal{V})\) and input signature \((\text{InterfaceName} \setminus IC_P) \cup ((\text{InterfaceName} \times \mathbb{N}) \times (ER_P \setminus ES_P) \times \mathcal{V})\).

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

**Definition A.5 (Projection of Trace).** 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 A.6 (Refinement).** The module \(P\) refines the module \(Q\), written \(P \preceq Q\), if the following conditions hold: (1) \(\text{IC}_Q \setminus IP_Q \subseteq \text{IC}_P \setminus IP_P\), (2) \(\text{dom}(IC_Q) \setminus IP_Q \subseteq (\text{dom}(IP_P) \cup IC_P) \setminus IP_P\), (3) \(ES_Q \subseteq ES_P\), (4) \(ER_Q \subseteq ER_P \cup ES_P\), ((1)-(4) 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 \(P \preceq R\).

B APPENDIX: CORRECTNESS PROOFS

B.1 Definitions
We present the definitions needed for the formalisms and proofs in this section.

1. **Configuration:** 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 \(last\) be a function that given an execution which is a sequence of alternating global configuration and transition labels returns the last global configuration state. If \(\tau = G^0 \xrightarrow{a_0} G^1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} G^n\) then \(last(\tau) = G^n\)
3. Let \(\text{trace}(\tau_P)\) represent the trace corresponding to execution \(\tau_P \in \text{execs}(P)\).
When defining the operational semantics of \( \text{ModP} \),

**B.3 Invariants for Executions**

- \( \text{ModP} \) configurations \( L_P \)

- The machine identifiers cannot appear “out-of-thin-air” is formalized as follows. For all identifier as a payload) or through \( \text{remove} \) \( \text{ModP} \) in \( s \)

- Let \( \text{new} \) \( \text{ModP} \) out of thin air

- The machine identifiers that a state-machine has access to cannot be created “out-of-thin-air” is formalized as follows. For all \( m \in \text{MachineName} \), \( s, s' \in S \), \( id, id' \in Z \), \( e \in E \), \( v \in V' \), \( i \in \text{InterfaceName} \), \( b \in B \), and \( n \in N \):
  1. \( s, id, s', id' \in \text{Local}(m) \Rightarrow \text{ids}(s') \cup \{id'\} \subseteq \text{ids}(s) \cup \{id\} \).
  2. \( s, b, n, s' \in \text{Rem}(m) \Rightarrow \text{ids}(s') \subseteq \text{ids}(s) \cup \{\text{ids}(e) | \exists e. b[n] = (e, v)\} \).
  3. \( s, id, e, v, s' \in \text{Enq}(m) \Rightarrow \text{ids}(e) \cup \text{ids}(s') \subseteq \text{ids}(s) \).
  4. \( s, i, s' \in \text{New}(m) \Rightarrow \text{ids}(s') \subseteq \text{ids}(s) \).

**ModP Modules:** Recollect that a module \( P \) can be represented as a tuple \( (E_P, I_P, I_P, L_P) \). The judgment \( P \vdash E_P, I_P, I_P, L_P \) described in Section 5 asserts that the module \( P \) is well-formed. We read this judgment as: Module \( P \) is well-formed with private events \( E_P \), private interfaces \( I_P \), interface definition map \( I_P \), and interface link map \( L_P \). We say that a module \( P \) and \( Q \) are composable if \( P \upharpoonright Q \) satisfies the \text{Composition} rule in Figure 5.

**B.3 Invariants for Executions**

During the execution of a \( \text{ModP} \) module \( P \), all reachable configurations \( (S_P, B_P, C_P) \) satisfy the following invariants:

1. \( \text{dom}(S_P) = \text{dom}(B_P) \)
2. \( \forall (x, n, a) \in \text{ids}(S_P) \cup \text{ids}(B_P). x \in \text{InterfaceName} \setminus \text{dom}(I_P) \lor (x \in \text{dom}(I_P) \land (x, n) \in \text{dom}(B_P)) \)
3. \( \forall (x, n, a) \in \text{ids}(S_P) \cup \text{ids}(B_P). x \in \text{InterfaceName} \Rightarrow n < C_P[x] \)
4. \( \forall i \in \text{dom}(I_P), \forall n \in N. n < \text{card} \{ (i, x) | (i, x) \in \text{dom}(B_P) \} \Rightarrow (i, n) \in \text{dom}(B_P) \)
5. \( \forall x \in \text{dom}(I_P). C[x] = \text{card} \{ n | (i, x) \in \text{dom}(B_P) \} \)
6. \( \forall i, i \in \text{dom}(B_P). i \in I_P \lor i \in \text{dom}(I_P) \setminus I_P \)

**Lemma B.1.** Let \( P \) be a well formed module. For any execution \( \tau \in \text{execs}(P) \) where \( \tau \) is a sequence of global configurations \( G_0 \overset{a_0}{\rightarrow} G_1 \overset{a_1}{\rightarrow} \ldots \overset{a_{n-1}}{\rightarrow} G_n \), all global configurations \( G_i \) satisfy the invariants (11) - (16).

**Proof.** The invariants (11) - (16) are inductive and can be proved by performing induction over the length of the execution \( \tau \) for all the transitions (rules) defined in \( \text{ModP} \) operations semantics. \( \square \)
B.4 Theorems and Proofs

Summary: 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 Operation is Compositional: The traces of a composed module is a subset of the traces of each component module. Hence, the composition of two modules creates a new module which is equally or more detailed than its components. This is captured by the Lemma B.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 \( Q_1 \) 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 Operation: Lemma B.8 states that hide operation is hierarchical and compositional in nature.

\[ \text{Lemma B.2 (Compositional Is Intersection).} \quad \text{Let} \ P \ \text{and} \ Q \ \text{be any two well-formed modules such that} \ P\|Q \ \text{is also well-formed. For any} \ \sigma \in \mathcal{C}_{P\|Q}, \ \text{we have} \ \sigma \in \mathcal{T}(P\|Q) \iff \sigma|_{P} \in \mathcal{T}(P) \ \text{and} \ \sigma|_{Q} \in \mathcal{T}(Q). \]

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 \( \sigma \in \mathcal{T}(P\|Q) \) then the projection \( \sigma|_{P} \in \mathcal{T}(P) \) and the projection \( \sigma|_{Q} \in \mathcal{T}(Q) \). This follows from the Lemma B.3.

Backward Implication for traces:
If there exists a sequence \( \sigma \in \mathcal{C}_{P\|Q} \) such that \( \sigma|_{P} \in \mathcal{T}(P) \) and \( \sigma|_{Q} \in \mathcal{T}(Q) \), then \( \sigma \in \mathcal{T}(P\|Q) \). This follows from the Lemma B.4.

\[ \text{Lemma B.3.} \quad \text{For any execution} \ \tau_{c} \in \mathcal{E}(P\|Q), \ \text{there exists an execution} \ \tau_{p} \in \mathcal{E}(P) \ \text{such that} \ \mathcal{T}(\tau_{c}|_{P}) = \mathcal{T}(\tau_{p}|_{P}) \ \text{and there exists an execution} \ \tau_{q} \in \mathcal{E}(Q) \ \text{such that} \ \mathcal{T}(\tau_{c}|_{Q}) = \mathcal{T}(\tau_{q}|_{Q}). \]

Proof. We perform induction over the length of execution \( \tau_{c} \) of the composed module \( P\|Q \).

Inductive Hypothesis: For every execution \( \tau_{c} \in \mathcal{E}(P\|Q) \), there exists an execution \( \tau_{p} \in \mathcal{E}(P) \) such that \( \mathcal{T}(\tau_{c}|_{P}) = \mathcal{T}(\tau_{p}|_{P}) \), there exists an execution \( \tau_{q} \in \mathcal{E}(Q) \) such that \( \mathcal{T}(\tau_{c}|_{Q}) = \mathcal{T}(\tau_{q}|_{Q}) \), 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 \( \text{last}(\tau_{c})_{S}, \text{last}(\tau_{c})_{B}, \text{last}(\tau_{c})_{C}. \)

Base case: The base case for the inductive proof is for an execution \( \tau_{c} \) of length 0, \( \tau_{c} \in \mathcal{E}(P\|Q) \). 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 \( \tau_{p} \in \mathcal{E}(P) \) and \( \tau_{q} \in \mathcal{E}(Q) \) 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 \( \tau_{c} \in \mathcal{E}(P\|Q) \). Let \( \tau_{p} \) and \( \tau_{q} \) be the corresponding executions for module \( P \) and \( Q \) such that \( \mathcal{T}(\tau_{c}|_{P}) = \mathcal{T}(\tau_{p}|_{P}), \mathcal{T}(\tau_{c}|_{Q}) = \mathcal{T}(\tau_{q}|_{Q}) \) 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 \( \tau'_{c} \in \mathcal{E}(P\|Q) \) where \( \tau'_{c} = \tau_{c} \xrightarrow{a} G \) and \( \tau'_{p}, \tau'_{q} \) be the corresponding executions of \( P \) and \( Q \).
We perform case analysis for all possible transitions labels $a$.

- **$a = \epsilon$**
  This is the case when the composed module $P || Q$ takes an invisible transition. Lets say $n$-th instance of an interface $i$ identified by $(i, n) \in \text{dom}(\text{last} \tau_c)\text{s}$ made an invisible transition. This could be because the machine took any of the following transitions: \text{Internal, Remove-Event, Create-Bad,Output-Create-3, Send-Bad, and Output-Send-3.}
  Consider the case when $i \in \text{dom}(I_P)$ i.e. machine corresponding to interface $i$ is implemented in module $P$.
  Based on the assumption that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p),\text{last}(\tau_q))$, we know that $\text{last}(\tau_c)[i, n] = \text{last}(\tau_p)[i, n]$ and $\text{last}(\tau_c)[i, n] = \text{last}(\tau_q)[i, n]$. Hence, if machine instance $(i, n)$ in $P || Q$ can make an invisible transition $\tau$ 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, $\text{trace}(\tau'_p)[\Sigma_P] = \text{trace}(\tau'_q)[\Sigma_P]$ (where $\tau'_p = \tau_p \rightarrow G'$). Since $a = \epsilon$, $\text{trace}(\tau_q)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q]$.
  Note that the invisible transitions do not change the map $C$. Since, the module $P || Q$ and $P$ took the same transition $a$ and configuration of module $Q$ has not changed, the resultant configurations satisfy the property $\text{last}(\tau'_p) = \text{union}(\text{last}(\tau'_p),\text{last}(\tau_q))$.
  The same analysis can be applied to the case when $m \in \text{dom}(M_Q)$.

- **$a = i$ where $i \in \text{InterfaceName}$**
  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 composed 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 $IC_{P || Q}$ and $\text{dom}(I_P || Q)$.
   Hence the environment of both $P$ and $Q$ can take the transition and the resultant executions $\tau'_p, \tau'_q$ will satisfy the condition $\text{last}(\tau'_p) = \text{union}(\text{last}(\tau'_p),\text{last}(\tau'_q))$, $\text{trace}(\tau'_p)[\Sigma_P] = \text{trace}(\tau'_q)[\Sigma_P]$, $\text{trace}(\tau'_p)[\Sigma_Q] = \text{trace}(\tau'_q)[\Sigma_Q]$.

2. **Input-Create**
   Our definition of composition and compatibility guarantees that if $P || Q$ is well-formed then:
   (a) $\text{dom}(I_P || Q) = \text{dom}(I_P) \cup \text{dom}(I_Q)$
   (b) $\text{dom}(I_P) \cap \text{dom}(I_Q) = \emptyset$
   Hence, if the composed module $P || Q$ receives an input create request for $i \in \text{dom}(I_P) \in IC_P$ from the environment, then either $i \in \text{dom}(I_P)$, or $i \in \text{dom}(I_Q)$. Also, since $i \notin IC_{P || Q}$, it implies that $i \notin IC_P$ and $i \notin IC_P$.
   Consider the case when $i \in \text{dom}(I_P)$. Based on the assumption that $\text{last}(\tau_c) = \text{union}(\text{last}(\tau_p),\text{last}(\tau_q))$, we know that $\text{last}(\tau_c)[i, n] = \text{last}(\tau_p)[i, n]$ and $\text{last}(\tau_c)[i, n] = \text{last}(\tau_q)[i, n]$. Hence, if $P || Q$ takes the visible Input-Create transition $i$, when in global configuration $\text{last}(\tau_c)$, then the same transition can be taken by module $P$ in configuration $\text{last}(\tau_p)$, $i \in \Sigma_Q$ (we know that $i \notin \text{(dom}(I_P) \cup IC_Q)$), hence $Q$ takes the Environment-Create transition. The resultant executions $\tau'_c$, $\tau'_p$, and $\tau'_q$ satisfy the condition that $\text{last}(\tau'_c) = \text{union}(\text{last}(\tau'_p),\text{last}(\tau'_q))$. Also, $\text{trace}(\tau'_p)[\Sigma_P] = \text{trace}(\tau'_q)[\Sigma_P]$ and $\text{trace}(\tau'_c)[\Sigma_Q] = \text{trace}(\tau'_c)[\Sigma_Q]$ since all modules took the same labeled transition.
   The same analysis can be applied to the case when $i \in \text{dom}(I_Q)$.

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

31
Consider the case when \( i' \in \text{dom}(I_P) \) (which implies that \( i' \notin \text{dom}(I_Q) \)), since \( i \notin \text{dom}(I_P) \) and \( i \notin \text{dom}(I_Q) \) we know that \( i \notin \text{dom}(I_P) \) and \( i \notin \text{dom}(I_Q) \).

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

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

(4) \( \text{OUTPUT- CREATE-2} \)

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

\[ \square \]

**Lemma B.4.** For every pair of executions \( \tau_p \in \text{execs}(P) \) and \( \tau_q \in \text{execs}(Q) \), if there exists \( \sigma \in \Sigma^*_{P||Q} \) such that \( \sigma[\Sigma_P] = \text{trace}(\tau_p)[\Sigma_P] \) and \( \sigma[\Sigma_Q] = \text{trace}(\tau_q)[\Sigma_Q] \), then there exists an execution \( \tau_c \in \text{execs}(P||Q) \) such that \( \text{trace}(\tau_c)[\Sigma_{P||Q}] = \sigma \).

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

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

**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.

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

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

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

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

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

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

Just to provide an intuition, \( (\tau_p, \tau_q') \) represents the case when module \( P \) takes a transition with label \( a \) and \( a \notin \Sigma_Q \), similarly \( (\tau_p, \tau'_q) \) represents the case when module \( Q \) takes a transition with label \( a \) and \( a \notin \Sigma_P \). \( (\tau'_p, \tau_q') \) represents the case when module \( P \) and \( Q \) both take transition with label \( a \), as \( a \in \Sigma_P, a \in \Sigma_Q \).

We perform case analysis for all possible transitions taken by module \( P \) and module \( Q \). We provide a proof for one such case:

(1) Let us consider the case when module \( P \) takes a transition \( \text{OUTPUT-Send-1} \) with label \( a = ((i, n), e, v) \). Let \( (i, n) \in \text{dom}(\text{last}(\tau_p)_S) \) be the machine that takes this transition. Hence, \( \sigma' = \sigma.a \) and \( \text{trace}(\tau'_p)[\Sigma_P] = \sigma'[\Sigma_P] \).
Let us consider the case when \(i_\tau \in \text{dom}(l_\tau)\), and \(e \in \text{MReceiv}(i_\tau) \setminus (\text{EP}_Q \cup \text{ES}_Q)\) (input event of \(Q\)). Based on the assumption that \(\text{last}(\tau_v) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))\) and the invariants I1-I6 about the state configurations, we know that \((i_\tau, n_\tau) \in \text{dom}(\text{last}(\tau_q))\).

Hence, \(Q\) can take a \text{Input-Send} transition with label \(a = ((i_\tau, n_\tau), e, v)\) and therefore \(\text{trace}(\tau'_v)[\Sigma_Q] = \sigma'^{\Sigma_Q}\).

Finally, using same assumption \(\text{last}(\tau_v) = \text{union}(\text{last}(\tau_p), \text{last}(\tau_q))\) and the invariants I1-I6, the composed module \(P||Q\) can take the transition \text{Output-Send-2} with the same label \(a = ((i_\tau, n_\tau), e, v)\).

Hence, \(\text{trace}(\tau'_v)[\Sigma_{P||Q}] = \sigma'\). The resultant executions still satisfy the condition that \(\text{last}(\tau'_v) = \text{union}(\text{last}(\tau'_p), \text{last}(\tau'_q))\).

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

Similar analysis was performed for all possible transitions taken by modules \(P\) and \(Q\).

\[ \square \]

**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 \(\sigma[\Sigma_{P||Q}]\) is a trace of \(Q||R\).

\[ \square \]

**Lemma B.6 (Circular Assume-Guarantee).** Let \(\| P \|\) and \(\| Q \|\) be well-formed. Suppose for each module \(Q \in Q\), there is a sublist \(X\) of \(P \oplus Q\) such that \(Q \notin X\), \(\| X \|\) is well-formed, and \(\| X \|\) refines \(Q\). Then \(\| P \|\) refines each module \(Q \in Q\).

**Proof.** Definitions:

- Let \(Q\) be a collection of \((n > 1)\) composable modules represented by the set \(\{Q_1, Q_2, ... Q_n\}\).
- Let \(P\) be a collection of \((n' > 1)\) composable modules represented by the set \(\{P_1, P_2, ... P_{n'}\}\). In this proof, we refer to \(\| P \|\) (composition of all modules in \(P\)) as module \(P\).
- Let \(\forall i, X_i\) be a sublist of \(P \oplus Q\).

Let us assume that \(\forall Q_i, Q\) there exists a \(X_i\) such that \(X_i \leq Q_i\).

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

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

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

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

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

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

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

- \(a = \alpha\) where \(\alpha \in \text{InterfaceName}\) (creation of an interface)

  \(a\) can be equal to \(\alpha\) because of any of the following cases: (1) module \(P\) creates an interface using the
transitions: Output-Create-1, Output-Create-2 or (2) the environment creates it using the transitions: Environment-Create, Input-Create.

Let us consider the case when \( a = a \) because \( P \) executes the Output-Create-1 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 \( P \) (\( \tau_P \in \text{execs}(P) \)) into the executions \( \tau_{P_1}, \tau_{P_2}, \ldots \) of the component modules such that for all \( P_i \in P \), \( \text{trace}(\tau_P)[\Sigma_{P_i}] = \text{trace}(\tau_{P_i})[\Sigma_{P_i}] \).

From the operational semantics of Output-Create-1, we know that \( a \in IC_P \) and \( a \notin \text{dom}(I_P) \). Let us consider the case when there exists a module \( P_i \in P \) such that \( a \in IC_{P_i} \), and from the definition of composition we know that \( \forall j, j \neq i, a \notin IC_{P_j} \).

If \( \exists j, \text{ s.t. } j \neq i \land a \in \Sigma_{P_j} \) then \( P_j \) can take the Environment-Create transition to match the visible action \( a = a \).

If \( a \in IC_Q \), then for some \( Q_i \in Q, a \in IC_{Q_i} \), (1).

If \( \forall Q_i \in Q, a \notin IC_{Q_i} \), then all \( Q_i \) can take the Environment-Create transition to match the visible action \( a = a \).

Let us consider the case when only (1) is true. Since \( a \in IC_{Q_i} \) and \( X_i \leq Q_i \) we have \( a \in IC_{X_i} \).

Note that \( P \) and \( Q \) are well-formed modules. Since (1) \( Q_i \notin X_i \) (2) \( \forall j, j \neq i.a \notin IC_{P_j} \land a \notin IC_{Q_j} \), we know that \( P_i \in X_i \).

Using Lemma B.4, and the fact that \( X_i \leq Q_i \), we know that for any given \( \tau_{P_i} \in \text{execs}(P_i) \) there exist \( \tau'_{Q_i} \) such that \( \text{trace}(\tau'_{Q_i})[\Sigma_{Q_i}] = \text{trace}(\tau'_{Q_i})[\Sigma_{Q_i}] \).

Finally, we know that:
1. Inductive hypothesis holds for any execution \( \tau_P \) and \( \tau'_P = \tau_P \xrightarrow{a} G \) (Output-Create-1)
2. \( a \in IC_Q \), and \( a \in IC_{P_i} \).
3. \( \forall j, j \neq i.(m, a) \notin IC_{P_j} \) and \( \forall j, j \neq i.a \notin IC_{Q_j} \).
4. there exists an execution \( \tau'_{P_j} \in \text{execs}(P_j) \) such that \( \text{trace}(\tau'_{P_j})[\Sigma_{P_j}] = \text{trace}(\tau'_{Q_j})[\Sigma_{Q_j}] \).
5. there exists an execution \( \tau'_{Q_i} \in \text{execs}(Q_i) \) such that \( \text{trace}(\tau'_{Q_i})[\Sigma_{Q_i}] = \text{trace}(\tau'_{Q_i})[\Sigma_{Q_i}] \).

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

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

Hence, the inductive hypothesis holds for the execution \( \tau'_P \).

We do similar analysis to prove the other cases.

\[ \square \]

**Lemma B.7 (Compositional Safety Analysis).** Let \( \parallel P \parallel Q \) be well-formed. Let \( \parallel P \parallel \) refine each module \( Q \in Q \). Suppose for each \( P \in P \), there is a sublist \( X \) of \( P \oplus Q \) such that \( P \in X, \parallel X \parallel \) is well-formed, and \( \parallel X \parallel \) is safe. Then \( \parallel P \parallel \) 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 \( P_1 \parallel Q_2, P_1 \parallel P_2 \) are safe then \( P_1 \parallel P_2 \) is safe.

Let us consider the case when there exists an error execution \( \tau_e \) in \( P_1 \parallel P_2 \). Using the compositional refinement Lemma, we can decompose the execution \( \tau_e \) into \( \tau_e \) of \( P_1 \) and \( \tau_e \) of \( P_2 \). Let us say the error was because of module \( P_1 \) taking a transition and hence \( \tau_e \) is an error trace.

We know that \( P_1 \parallel Q_2 \) is safe which means that for all executions of module \( P_1 \parallel Q_2 \) there is no execution of \( P_1 \) that is equal to \( \tau_e \) after decomposition.

34
The above condition also implies that in the composed module \( P_1 \parallel P_2 \), module \( P_2 \) using an output action is triggering an execution in \( P_1 \) which results in execution \( \tau \). And this output action is not triggered by \( Q \) in the composition \( P_1 \parallel Q \).

The above condition implies that \( P_1 \parallel P_2 \leq Q \) does not hold which is a contradiction.

We generalized this proof strategy for proving the given lemma.

\[ \Box \]

**Lemma B.8 (Hide Operation).** For all well-formed modules \( P \) and \( Q \) and a set of events or machines \( \alpha \), if \((\text{hide } \alpha \text{ in } P)\) is well-formed, \((\text{hide } \alpha \text{ in } Q)\) is well-formed, and \( P \leq Q \), then \((\text{hide } \alpha \text{ in } P) \leq (\text{hide } \alpha \text{ in } Q)\).

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

We perform induction over the length of execution \( \tau_{hP} \) of module \( hP \)

<table>
<thead>
<tr>
<th>Inductive Hypothesis:</th>
<th>For every execution ( \tau_P \in \text{execs}(hP) ), there exists an execution ( \tau_Q \in \text{execs}(hQ) ) such that ( \text{trace}(\tau_P)[\Sigma_{hQ}] = \text{trace}(\tau_Q)[\Sigma_{hQ}] )</th>
</tr>
</thead>
</table>

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

- **Base case:** The base case is trivially satisfied by an execution of length zero.
- **Inductive case:** Let us assume that the hypothesis holds for any execution \( \tau_{hP} \in \text{execs}(hP) \) and the corresponding execution of module \( hQ \) be \( \tau_{hQ} \in \text{execs}(hQ) \).

To prove that the hypothesis is inductive we show that it also holds for the execution \( \tau'_{hP} \in \text{execs}(hP) \) where \( \tau'_{hP} = \tau_{hP} \overset{\alpha}{\rightarrow} G \) and \( \tau'_{hQ} \) be the resultant executions of \( hQ \).

Hide operation only converts visible actions into internal actions. Hence, it can be easily shown that any execution of \( hP \) is also an execution of \( P \), similarly for module \( hQ \) and \( Q \), every execution of \( hQ \) is an execution of \( Q \).

The above property, along with the fact that \( P \leq Q \) helps us conclude that the inductive hypothesis always holds.

\[ \Box \]

**References**


Networked Systems Design and Implementation (NSDI).
