Practical Software Model Checking via Dynamic Interface Reduction

Huayang Guo*† Ming Wu† Lidong Zhou† Gang Hu*† Junfeng Yang◦ Lintao Zhang†
* Tsinghua University † Microsoft Research Asia ◦ Columbia University
{huayang.guo,henry.hu.sh}@gmail.com {miw,lidongz,lintaoz}@microsoft.com
junfeng@cs.columbia.edu

ABSTRACT

Implementation-level software model checking explores the state space of a system implementation directly to find potential software defects without requiring any specification or modeling. Despite early successes, the effectiveness of this approach remains severely constrained due to poor scalability caused by state-space explosion. DEMETER makes software model checking more practical with the following contributions: (i) proposing *dynamic interface reduction*, a new state-space reduction technique, (ii) introducing a framework that enables dynamic interface reduction in an existing model checker with a reasonable amount of effort, and (iii) providing the framework with a distributed runtime engine that supports parallel distributed model checking.

We have integrated DEMETER into two existing model checkers, MACEMC and MODIST, each involving changes of around 1,000 lines of code. Compared to the original MACEMC and MODIST model checkers, our experiments have shown state-space reduction from a factor of five to up to five orders of magnitude in representative distributed applications such as PAXOS, Berkeley DB, CHORD, and PASTRY. As a result, when applied to a deployed PAXOS implementation, which has been running in production data centers for years to manage tens of thousands of machines, DEMETER manages to explore *completely* a logically meaningful state space that covers both phases of the PAXOS protocol, offering higher assurance of software reliability that was not possible before.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking, Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Algorithms, Reliability

Keywords

Software model checking, state space reduction, dynamic interface reduction

SOSP '11, October 23-26, 2011, Cascais, Portugal.

1. INTRODUCTION

Reliability has become an increasingly important attribute for computer systems, as we are witnessing growing dependencies on computer systems that run continuously on commodity hardware despite adversity in the environment. Complete verification of system implementations has been a daunting job, if not infeasible for complex real-world systems. Implementation-level software model checking [18, 36, 32, 41, 40, 33, 29, 38, 39] proves to be a viable approach for improving reliability. It has advanced to a stage where it can be applied directly to a system implementation and can find rare program bugs by exploring a system's state space systematically to detect system misbehavior such as crashes, exceptions, and assertion failures. Despite this success, these model checkers are often unable to explore *completely* any non-trivial logically bounded state space (e.g., a normal single execution of consensus), making it hard to provide any degree of assurance for reliability. State-space explosion is a major obstacle to their effectiveness.

In this paper, we introduce *dynamic interface reduction* (DIR), a new state-space reduction technique for software model checking. DIR is based on two principles.

First, *check components separately.* A common practice to manage software complexity is to encapsulate the complexity using well-defined interfaces. Leveraging this common practice, a model checker considers a target software system as consisting of a set of *components*, each with a well-defined *interface* to the rest of the system. For example, a typical distributed system is comprised of a set of processes interacting with each other through message exchanges. The set of message-exchange sequences, or *message traces*, between a component and the rest of the system defines the *interface behavior* for that component. In general, all behavior such as shared memory, failure correlations, or other implicit channels that cause one component to affect another is captured by interface behavior. Any behavior other than interface behavior is locally contained. Given the interface behavior of each component, DIR can check its local state-space separately, avoiding unnecessary (and expensive) exploration of the global state-space when possible.

Second, *discover interface behavior dynamically.* Model checking each component separately requires knowing the interface behavior of the component. DIR discovers this behavior dynamically during its state-space exploration, by running the target components for real and combining their discovered interface behavior. This process is often efficient because it ignores intra-component complexity that does not propagate through interfaces. Moreover, this process is completely automated, so that developers do not have to specify interface behavior manually [22, 31], which may be tedious, error-prone, and inaccurate. A last benefit is that this process

discovers only the true interface behavior that may actually occur in practice, not made-up ones [23], thus avoiding difficult-to-diagnose false positives.

We incorporate the DIR technique into DEMETER, a model checking framework that includes an algorithm that progressively explores the local state-space of each component, while discovering interface behavior between components. DEMETER adopts a modular design as a framework to enable DIR in existing model checkers with a reasonably small amount of engineering effort. Its design can reuse the key modules for modeling a system and for state-space exploration in an existing model checker; DEMETER further defines a set of common data structures and APIs to encapsulate the implementation details of an existing model checker. The key DIR algorithm can then be implemented independently of any specific model checker and is accordingly reusable.

DEMETER implements a distributed runtime for DIR-enabled model checking that leverages the inherent parallelism of DIR, as local explorations for components with respect to given interface behavior are largely independent. As a result, DEMETER scales nicely when running on more machines, and is capable of tapping into any distributed system or cloud infrastructure that is becoming prevalent today to push model checking capabilities further.

To demonstrate the practicality of DEMETER, we have incorporated DIR into MACEMC and MODIST, two independently developed model checkers. Despite their fundamental differences in implementing model checking, each requires changes of only around 1,000 lines of code, thanks to the framework provided by DEMETER. The resulting model checkers take advantage of not only the new reduction technique, but also of the distributed runtime to run model checking in parallel on a cluster of machines.

The resulting checkers have been used to check representative applications, ranging from PASTRY and CHORD, two classic peer-to-peer protocols, to Berkeley DB (BDB), a widely used open source database, and to MPS, a deployed PAXOS implementation that has been running in production data centers for years to manage tens of thousands of machines. Our experiments show up to a 10^5 speedup in estimated state-space exploration, thanks to the effectiveness of interfaces in hiding local non-determinism related to thread interleaving and coordination. Furthermore, DEMETER's runtime shows nearly perfect scalability as we increase worker machines from 4 to 32. This significantly improved model-checking capability from both state-space reduction and parallelism translates directly to increased confidence in the reliability of systems that survive extensive checking: in our experiment with MPS, DEMETER was able to explore a complete sub-space, where three servers execute both phases in the PAXOS protocol. DEMETER was also able to explore a complete sub-space for CHORD on MACE with three servers until all have joined. To the best of our knowledge, neither would be possible for any published implementation-level model checker without DIR.

The rest of the paper is organized as follows. Section 2 presents an overview with an example system we use throughout the paper. Section 3 presents an overview of DIR and the algorithm. Section 4 outlines DEMETER's system architecture and how MACEMC and MODIST are integrated with DEMETER. Evaluations of and experiences with DEMETER are the subject of Section 5, followed by discussions in Section 6. We survey related work in Section 7 and conclude in Section 8.

```
Client                          Primary/Secondary
                                //Main thread              //Checkpoint thread
if (Choose(2)==0){              while (n=Recv()) {         Lock();
   Send(P,1);                     Lock();                  Log(sum);   <- Ckpt
   Send(P,2);                     sum+=n;      <- Sum      Unlock();
} else {                          Unlock();
   Send(P,1);                     if (isPrimary)
   Send(P,3);                        Send(S,n);
}                               }
```

Figure 1: Code example for a contrived distributed accumulator composed of a client `C`, a primary server `P`, and a secondary server `S`.

2. OVERVIEW AND AN EXAMPLE

Dynamic interface reduction in DEMETER considers a system consisting of a set of components, each with a well-defined interface to interact with the rest of the system. For example, a distributed system can have processes running on each machine as a component, with a sequence of message exchanges between components forming a *message trace* as interface behavior. (We assume no interactions occur via any means other than messages.) State-space exploration is then divided into a set of local explorations, one for each component, and a global exploration that explores the interactions between components; e.g., in the form of message traces. During the exploration, DEMETER tracks and builds up the interface behavior (e.g., message traces) between each component and the rest of the system. By dynamically discovering interface behavior, DEMETER removes the need for users to model interactions beforehand through manual or static-analysis methods, and follows closely the philosophy of implementation-level software model checking with no specification or modeling.

Before presenting the details of the system model, the DIR algorithm, the architecture, and the implementation of DEMETER, in this section, we use a simple code example to describe at a high level the work flow of DEMETER with DIR and what kind of reduction it can achieve. For simplicity, we focus on distributed systems where an execution *trace* captures the non-deterministic events such as thread interleaving, message send, and message receive operations in an execution, while a *message trace*, which includes only the message send and receive operations in an execution, captures the interface behavior across components.

2.1 An Example

Figure 1 shows the pseudo code of a contrived distributed accumulator composed of three components: a client, a primary server, and a secondary server. The client (left of Figure 1) calls function `Choose(2)` [18, 39, 40, 29], which non-deterministically returns 0 or 1. In practice, this can be used to imitate the effect of timeout, failure, or a random function. Depending on the returned value of the `Choose` function, the client code sends two different sequences of numbers to the primary, which then sums them up and forwards them to the secondary. A checkpoint thread writes the sum to disk. We label the critical sections in these two threads as `Sum` and `Ckpt`, respectively. Both the primary and the secondary run the same code (right of Figure 1), except that the secondary has `isPrimary` set to false. As a result, the secondary receives the numbers from the primary, but does not forward the numbers further.

Our example does only simple summation for clarity. However,

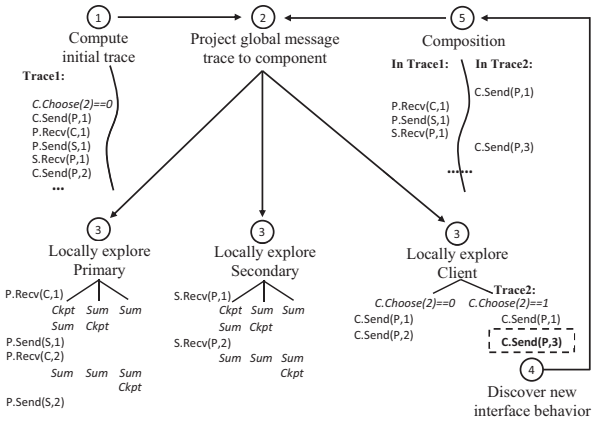


Figure 2: Work flow of DEMETER with DIR on the example in Figure 1. The work flow has five key steps, as explained in §2.2.

it still mimics real distributed systems in many aspects. For instance, it is built on top of common techniques that real distributed systems use, such as replication, message passing, multi-threading, and checkpointing. Moreover, it has a well-defined component interface that hides the implementation details (e.g., when the checkpoint thread of the server interleaves with the main thread) within a component. Because these local choices do not propagate outside of component interfaces, we can check them locally without resorting to expensive global exploration of all components.

2.2 DIR Work Flow

At a high level, the work flow of DEMETER with DIR alternates between a *global explorer* enumerating the global message traces across components and a set of *local explorers*, one per component, enumerating the local execution traces within each component. Figure 2 illustrates this work flow using the example in Figure 1. The DIR work flow has five key steps:

1. To bootstrap the checking process, the global explorer first performs a global execution including all components to discover an initial global execution trace, and the corresponding global message trace that keeps only the message send and receive operations. As shown in the figure, the global explorer first explores the choice of `Choose(2)` returning 0 in the client. The client then sends the sequence 1 and 2 to the primary, which forwards it to the secondary, resulting in the global trace `Trace1`. A corresponding global message trace can be obtained by removing all intra-component events from `Trace1`. The goal of the global explorer is to discover all global message traces.
2. The global explorer *projects* a newly discovered global message trace down to each component's local message trace by keeping only the message exchanges that are either sent or received by that component. It then sends to each component the corresponding projected message trace. Step 3 in Figure 2 shows the results of this projection for each component. As the global explorer discovers more and more global message traces, it keeps generating such projections, increasingly capturing the interface behavior of each component.
3. Checking now shifts to local explorers. A local explorer enumerates non-deterministic choices within the corresponding component. Because the local explorer does not control the execution of other components, whenever the component attempts to interact with other components, the local explorer will match any outgoing messages with those in the local message trace and replay any incoming messages according to the local message trace. As shown in step 3 of Figure 2, the local explorer for the primary (similarly for the secondary) explores the different interleavings of the `Sum` and `Ckpt` operations while matching the `Send` operations and replaying the `Recv`.
4. If a local explorer causes the component to send a new message that deviates from the local message trace, it can no longer follow the message traces it already knows, and has to report this deviation to the global explorer. For instance, as shown in Figure 2, when the local explorer of the client explores the choice of `Choose(2)` returning 1, it encounters a new interface operation `Send(P,3)` (boxed) deviating from the known message traces of the client. We label the new trace `Trace2`.
5. The global explorer then composes the new message trace with existing global message traces to construct new global message traces. For instance, in Figure 2, the global explorer locates the deviating points in the global message trace derived from `Trace1` and stitches the unchanged portion together with `Trace2` to form a new global message trace. (For details of this composition process, see Section 3.3.) Then, the global explorer goes back to step 2 and repeats until no new message trace is discovered and all the local explorations against the known message traces have finished.

2.3 Reduction Analysis

For the example in Figure 1, each component has two different message traces (one for each value returned by `Choose(2)` at the client). The client has one local execution trace per message trace. The primary and the secondary each have three different local traces per message trace, because `Sum` and `Ckpt` can interleave differently and lead to different local states (see Figure 2), but the changed local state does not propagate across the component interfaces. Thus, DEMETER with DIR explores $2*(1+3+3)=14$ different executions.

In contrast, a model checker without DIR has to re-explore the entire system whenever the local state of a component changes. The reason is that, without dividing a whole system into components and monitoring the interface behavior, a model checker has to assume that a local change may affect the rest of the system. Thus, it must re-explore all non-deterministic choices in the rest of the system under this local change. For instance, when the primary's main thread interleaves differently with its checkpoint thread and results in a different local state, a model checker without DIR would have to re-explore unnecessarily the choices in both the client and the secondary. As a result, it would explore a total of $2*3*3=18$ executions.

Analytically, DIR achieves exponential state-space reduction. To illustrate, consider a modified example where the client sends one sequence of n numbers and the primary forwards the numbers to $(m-1)$ replicas. Each server (primary or replica) has exactly one message trace (since the client sends only one sequence of numbers). Under this message trace, each server has $(n+1)$ different thread interleavings. Therefore, DEMETER would explore $1+m*(n+1)$ executions, whereas a model checker without DIR would explore $(n+1)^m$ executions.

From a system perspective, the reduction of DIR can be intuitively viewed as a result of caching. Consider a system where a component has many local non-deterministic choices but always sends the same message to the other components. When exploring this com-

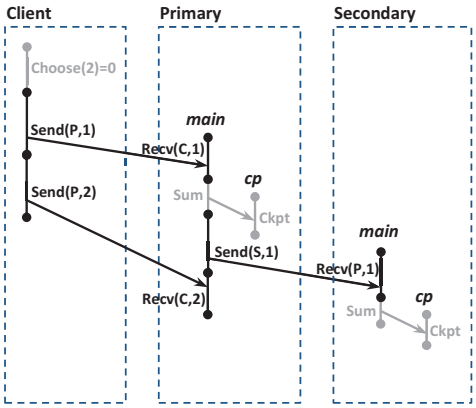


Figure 3: A trace τ of the example code in Figure 1. *main* and *cp* refer to the main and the checkpoint threads, respectively.

ponent, the first time we discover an outgoing message, we have to explore the effects of this message on the other components, which can be expensive. However, as we keep exploring this component, we discover that it sends the same message again in a different execution, and we can thus safely skip the expensive exploration of the other components under this same message. In other words, we effectively get a "cache hit." Following this intuition, we expect DIR to work well for any system where there are well-defined interfaces to hide implementation details. This is common for practically all real systems, especially loosely coupled distributed systems that are designed to reduce the amount of inter-process communication for performance reasons.

3. DYNAMIC INTERFACE REDUCTION

In this section, we present the system model we assume for DIR and the detailed algorithm.

3.1 System Model

DEMETER checks standard concurrent/distributed systems as defined previously in software model checking [16, 18]. Abstractly, a system starts from an initial state and at each step performs a *transition* into the next state. A transition is *enabled* if it is not blocked and can be scheduled to execute on the current state. The *environment* is used to model the *non-determinism* as different choices of enabled transitions at a state. Such non-determinism includes thread/process scheduling, message ordering, timers, failures, and other randomness in the system.

Implementation-level software model checkers work directly on actual implementations of target systems. They typically consist of two major pieces. The first is a *system wrapper* that exposes an underlying system and enables the control of non-determinism in the environment. The second is an *exploration mechanism* that builds on top of the system wrapper to explore the system state space by capturing and controlling non-determinism in order to find software defects such as unintended exceptions and crashes, assertion failures, and other safety violations.

In DEMETER, a system is divided into a static set $\mathscr{C}$ of components. Components interact with each other through *interface objects*, such as communication channels or shared objects. We classify transitions as *internal* transitions if they do not read or write interface objects, or *interface* transitions if they access and/or update interface objects. An interface transition is further an *output* transition if it updates an interface object (e.g., sending a message or updating a shared object); or an *input* transition if it reads an interface object (e.g., receiving a message or reading a shared object).

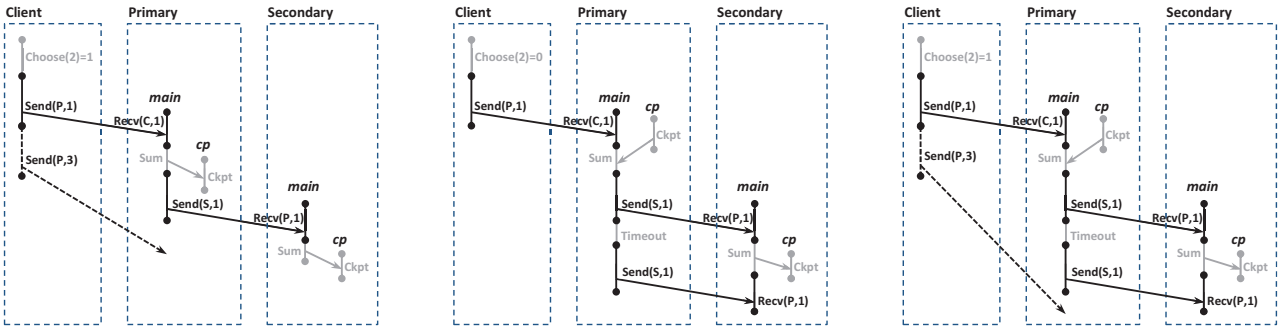
Two transitions are *dependent* if their executions interfere with each other: one could enable/disable the other, or executing them in a different order could change the final outcome. Examples are two lock operations on the same lock, a write operation and read/write operations on the same shared variable, and a message send operation and the corresponding receive are dependent. Starting from an initial state, a system execution is modeled as a *trace* that captures all transitions taken by the system and the partial order ($\preceq$) between those transitions based on transition dependencies. Partial-order equivalent traces are considered the same. Given a trace τ and an enabled transition t at the state after executing τ, we can extend τ to a new trace $\tau \circ t$ by carrying out transition t. We can further define a *prefix* relation between traces as follows. A trace τ_p is a prefix of τ if and only if any transition in τ_p is in τ and, for any transition t in τ_p and any $t_p \preceq t$ in τ, t_p must be in τ_p and $t_p \preceq t$ in τ_p holds.

Each transition belongs to a particular component. A global trace τ can be *projected* onto a component C to obtain a *local trace* by preserving only transitions that belong to component C (including output transitions from C to other components) and output transitions from other components to C, along with their partial order. The result is referred to as $proj_c(\tau)$. To capture interface behavior in a trace, we construct a *global skeleton* from a global trace τ by keeping only interface transitions and their partial order in the trace. We refer to the resulting skeleton as $skel(\tau)$. Similarly, a *local skeleton* $skel(proj_c(\tau))$ can be defined on local trace $proj_c(\tau)$ for component c. A local skeleton captures the interface behavior between c and the rest of the system. Two global traces τ and τ' are *interface-equivalent* with respect to component c if and only if their local skeletons on c are the same; that is, $skel(proj_c(\tau)) = skel(proj_c(\tau'))$ holds.

Figure 3 shows an example trace τ of the example code in Figure 1. Each segment corresponds to a transition, while arrows represent inter-thread/process communications, which also imply the happen-before relation between transitions. A partial order ($\preceq$) is defined between transitions in the same thread, between a send transition and its corresponding receive transition across threads and processes, and is transitive. Examples include `P.Recv(C,1)` $\preceq$ `P.Sum`, `P.Sum` $\preceq$ `P.Ckpt`, `P.Send(S,1)` $\preceq$ `S.Recv(P,1)`. All `Send` and `Recv` transitions (marked in bold) are interface transitions, while `Choose`, `Sum`, and `Ckpt` are internal transitions corresponding to local non-deterministic choices. The corresponding global skeleton of τ in Figure 3 contains the 6 interface transitions and their partial order as in the original trace. The local trace with a projection to the client contains `Choose(2)=0`, `C.Send(P,1)`, and `C.Send(P,2)`. The corresponding local skeleton contains only transitions `C.Send(P,1)` and `C.Send(P,2)`.

3.2 Partial-Replay Local System

The first core idea of DIR is to check each component separately. Checking a component c is possible with a local skeleton that specifies all the interface behavior between c and the rest of the system. This is done through a *partial-replay local system*. In theory, it is possible to replay just the interface transitions on a local skeleton (e.g., by supplying received messages recorded in the local skeleton). In reality, replaying only the interface transitions is difficult. For example, in order to replay message-exchange transitions, the underlying network channels (sockets) must be set up correctly.



(a) Branching trace τ_A with branching transition `C.Send(P,3)`.

(b) τ_B: A global trace with the same projected local skeleton on the client as τ_A, and with a message resend from the primary.

(c) Substitution: $subst_C(\tau_B, \tau_A)$.

Figure 4: Composition by Substitution: an Example.

This could involve earlier operations such as *bind*. Such internal dependencies might be hard to identify thoroughly; the process is often error-prone. Simulating network behavior for replaying is also a significant undertaking, as done in model checkers such as MODIST. Therefore, a partial-replay local system replays not only interface transitions, but also any other transitions in the rest of the system. This choice leads to a simple and modular design, albeit at the cost of running transitions in other components.

More precisely, given a local skeleton κ_c and a representative trace τ satisfying $proj_c(skel(\tau)) = \kappa_c$, a partial-replay local system tries to enumerate transitions in c, while *replaying* the behavior of the rest of the system (denoted as R) according to τ. Starting from the initial state, in each step the partial-replay local system either picks an enabled transition from component c or replays τ's transitions in R. A transition t made by R in $proj_R(\tau)$ can be replayed if and only if any transition in $proj_R(\tau)$ that t depends on has already been replayed.

A partial-replay local system could make an output transition in c that deviates from κ_c. Such a deviating output transition is called a *branching transition*. When a branching transition t_b is encountered, let τ_b be the trace explored right before taking the branching transition, the partial-replay local system reports $\langle t_b, \tau_b \rangle$ in order for DEMETER to discover new global and local skeletons through composition by substitution, which we describe next.

3.3 Composition by Substitution

The second core idea of DIR is to discover interface behavior dynamically. This is the responsibility of the *global explorer* through composition by substitution. The global explorer maintains the set G containing the pair $\langle \kappa, \tau \rangle$ for each discovered global skeleton κ and a corresponding global trace τ, where $\kappa = skel(\tau)$. The global explorer further maintains a set B of all discovered branching transition/trace pairs ($\langle t_b, \tau_b \rangle$) reported by partial-replay local systems.

Intuitively, the global explorer's process of discovering interface behavior can be thought of as a state-space exploration of a new transition system with only the interface transitions of the original system. The global explorer essentially builds up the transition system with the global skeletons captured in G, where the branching transitions captured in B are the transitions in that system. For a branching transition from component c, the local skeleton $\kappa_c = proj_c(skel(\tau_b))$ defines when that branching transition is enabled: for any global skeleton κ, the branching transition is enabled if and only if $proj_c(\kappa) = \kappa_c$ holds, in which case we can carry out that branching transition to extend κ to a new global skeleton.

This process is described more precisely through the following composition by substitution on traces, which uses the *subst* operation defined as follows. If two traces τ and τ' are interface-equivalent with respect to component c, $\tau_s = subst_c(\tau, \tau')$ defines a new trace by replacing all c's transitions in τ with c's transitions in τ' while preserving the partial order in the original traces; that is, for any transitions t and t' in τ_n, if t and t' are both in τ or both in τ' with $t \preceq t'$, then $t \preceq t'$ holds in τ_s. Such a substitution is possible because τ and τ' are interface-equivalent with respect to c: c's transitions in τ and τ' are indistinguishable to the rest of the system because they present the same interface behavior (i.e., local skeleton).

Given $\langle t_b, \tau_b \rangle \in B$ and $\langle \kappa_g, \tau_g \rangle \in G$, where τ_b and τ_g are interface-equivalent with respect to component c, we compose a new global trace $\tau_s = subst_c(\tau_g, \tau_b)$ through substitution, construct $\tau_n = \tau_s \circ t_b$ by taking the branching transition t_b, and add $\langle skel(\tau_n), \tau_n \rangle$ into G.

Figure 4 illustrates the process of composition by substitution. We enrich the example in Figure 1 slightly by enabling the primary to resend its message if a local timeout for that message is triggered. The secondary ignores the resent message if it has already received the previous one. The extension creates more variations in global skeletons and helps illustrate how composition by substitution creates new global skeletons. Figure 4(a) shows a global trace τ_A (containing all transitions in solid lines) with a branching transition $t_b =$ `C.Send(P, 3)` (in dotted lines), when the client has `Choose(2)` set to 1, rather than 0. Figure 4(b) shows another global trace τ_B that has the same local skeleton for the client as τ_A. It is a prefix of a complete trace when the client has `Choose(2)` set to 0. The local traces of τ_A and τ_B for the primary are different in the order between `Sum` and `Ckpt`. The global skeletons of the two are also different: in τ_B, the primary resends the message with value 1. The differences in τ_A and τ_B are however invisible to the client. Further assume that τ_B and its global skeleton have already been discovered in G. When the branching transition in Figure 4(a) is reported, a composition is performed to yield a new trace $subst_C(\tau_B, \tau_A)$, where the branching transition is also enabled, as shown in Figure 4(c). $subst_C(\tau_B, \tau_A) \circ t_b$ is then a new global trace.

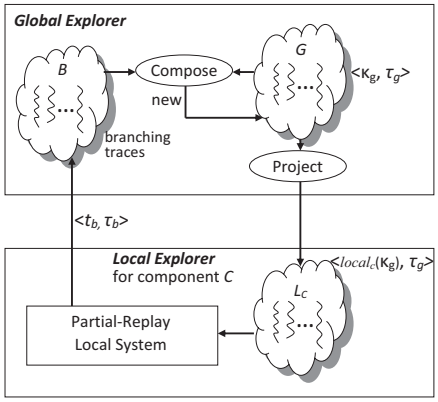


Figure 5: Interactions between global and local explorers.

3.4 Global and Local Explorers

The DIR algorithm consists of two types of cooperative progressive tasks that are running concurrently. Whereas the *global explorer* maintains a set G to track global skeletons and a set B to track branching transitions, a *local explorer* for component c maintains $L_c = \{\langle proj_c(\kappa), \tau\rangle \mid \langle \kappa, \tau\rangle \in G\}$ to track local skeletons for component c. Figure 5 illustrates the interactions between the global explorer and the local explorers. Local explorers use partial-replay local systems to explore each component separately and reports branching to the global explorer, while the global explorer uses composition by substitution to discover new global skeletons.

Local Explorer

1. Local explorer c initiates a partial-replay local system with respect to each $\langle \kappa_c, \tau\rangle \in L_c$.
2. When a partial-replay local system detects a branching transition t_b at trace τ_b, the local explorer backtracks and reports $\langle t_b, \tau_b\rangle$ to the global explorer to be added into B.

Global Explorer

1. Perform composition by substitution whenever B or G is updated until reaching a fixed point. For any $\langle \kappa_g, \tau_g\rangle \in G$, and $\langle t_b, \tau_b\rangle \in B$ satisfying $skel(proj_c(\tau_b)) = proj_c(\kappa_g)$, where t_b is a transition from component c, let $\tau_n = subst_c(\tau_g, \tau_b) \circ t_b$, add $\langle skel(\tau_n), \tau_n\rangle$ into G.
2. For each component c, update $L_c = \{\langle proj_c(\kappa), \tau\rangle \mid \langle \kappa, \tau\rangle \in G\}$ whenever G is updated.

Optimizations. It is worth noting that our presentation of the algorithm ignores certain obvious optimizations for simplicity and clarity. For example, any prefix of a global skeleton/trace can be subsumed because any prefix of a valid global skeleton/trace is a valid global skeleton/trace. We just need to record the longest ones. Also, to avoid an excessive number of branching transitions, when a new global skeleton is constructed, the global explorer will attempt to continue running the corresponding global trace to completion, including all system components. Similarly, the algorithm starts by having the global explorer perform a global execution including all system components to discover initial global traces, in order to initialize G with some global skeletons and associated global traces.

Correctness. A state-space reduction technique must be both sound and complete. In the context of DIR, soundness requires that every local trace explored by the algorithm is a projection of a valid global trace, while completeness states that, for any valid global trace τ, our algorithm discovers $skel(\tau)$ in the global explorer (G) and finds $proj_c(\tau)$ in the local explorer for every component c.

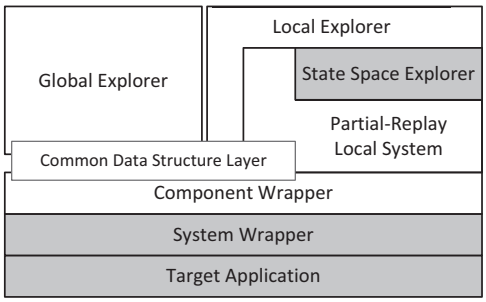


Figure 6: DEMETER Layering Architecture.

Intuitively, the soundness hinges on the following fundamental *substitution rule*: if two valid traces τ and τ' are interface-equivalent with respect to component c, $subst_c(\tau, \tau')$ is also a valid trace. The substitution rule derives directly from the notion of interface equivalence and reflects the following observation. A component's interface behavior, captured by its local skeletons, isolates a component from the rest of the system. For an execution of a single component, changes in the rest of the system are irrelevant as long as the behavior at the interface (as captured in local skeletons) remains the same. Conversely, if two executions of a component conform to the same local skeleton, they are indistinguishable from the rest of the system.

DIR upholds soundness because, during both the partial-replay local exploration and the composition in global exploration, each discovered local or global skeleton complies with a valid global trace due to the substitution rule. The completeness is guaranteed through the cooperation of local and global explorers, as the local exploration can find all the local states and discover all the possible branching transitions with respect to given local skeletons, while the global explorer can construct all new global skeletons through composition for given sets of global skeletons and branching transitions. A proof sketch for the soundness and completeness is described in Appendix A.

4. ARCHITECTURE AND IMPLEMENTATION

In this section, we present the layered architecture of DEMETER that is specifically designed to facilitate incorporation of DIR into an existing model checker, followed by notes on some implementation details and on how we retrofit MACEMC and MODIST to integrate DEMETER.

4.1 A Model Checking Framework

We design DEMETER as a model-checking framework, which can embed an existing software model-checker in order to enable DIR for it. We refer to the model checker embedded in DEMETER as *eMC*. This design significantly reduces the amount of work to build model checkers with DEMETER and avoids having DIR trapped in a particular model checking implementation.

Turning DEMETER into a model-checking framework requires a careful modular design. Figure 6 shows the layered architecture

of DEMETER, where the shaded rectangles correspond to the layering in *eMC*. These modules (system wrapper and state-space explorer) are unmodified when plugged into DEMETER. In particular, DEMETER is able to leverage *eMC*'s state-space explorer because it adds a partial-replay local system layer that gives the state-space explorer an illusion of a stand-alone complete system, similar to the original application. The partial-replay local system layer further uses a *component wrapper*, which defines component boundaries and interface transitions. To isolate the specific implementation details of *eMC*, DEMETER defines a common set of *eMC*-neutral data structures/API and implements a *Common Data Structure Layer* that converts between these common data structures and those used in a particular *eMC*. Consequently, the global explorer and the part of the local explorer built on top of the Common Data Structure Layer are reusable across different *eMC*s.

Partial-Replay Local System. As shown in Figure 6, DEMETER builds a partial-replay local system by reusing *eMC*'s state-space explorer and system wrapper. The partial-replay local system takes a component c, a local skeleton, and a corresponding global trace τ, and runs the entire system on the original system wrapper except that it checks whether a transition is local to c or not (provided by the component wrapper) and replays any transitions in the rest of the system R following τ. The replay of R's transitions in τ is done by instructing *eMC*'s state-space explorer to take the designated transitions, but all the choices within component c are left to *eMC*'s state-space explorer.

Common data structures and APIs. Conceptually, the global explorer can be regarded as performing model checking of components with only interface transitions. However, reusing either *eMC*'s system wrapper or state-space explorer is difficult partly because this higher-level system must be constructed with transitions not known beforehand.

We opt for simplicity and build the global explorer on a small set of common data structures and APIs. In particular, we model the basic concurrency unit of a system as a thread. A transition is represented by a simple data structure with the following core fields: (i) its thread identifier, (ii) its unique identifier, (iii) its vector clock, (iv) interface transition flag, and (v) additional information about the transition. The additional information is mainly for converting this data structure to any original trace representation in *eMC*. A trace is defined as a set of transitions organized by their partial order (according to vector clocks). A skeleton is defined as a kind of trace that contains only interface transitions. Common operations can be defined on those data structures, such as projection from global trace to local trace, extraction of interface skeleton from a trace, and composition of a branching trace and a global trace. All of these operations are independent from *eMC*.

We have further implemented the following core functions on top of *eMC*'s system wrapper for global explorer: (i) reset system to the initial state, (ii) execute a particular transition at the current trace prefix, and (iii) run a trace prefix to completion (after the prefix, any completion of a global trace is sufficient). For local explorer, the partial-replay local system also provides a simple API to set up and run a partial replay. Implementing the global explorer and part of the local explorer on this set of common data structures and APIs makes its core logic reusable as it is made independent from *eMC*. The common data structure layer in Figure 6 is responsible for providing the data-structures and the APIs.

4.2 Interface Equivalence and Vector Clock

Interface equivalence defined on the equality of two skeletons is a key concept in DEMETER and is widely used in the implementation of DIR. For example, the local explorer needs to check equality between branching traces and local skeletons so that it can decide whether an encountered branching trace or local skeleton is new or not; when performing composition, the global explorer also needs to check whether two traces are interface-equivalent.

Interface equivalence can be judged by comparing interface transitions in skeletons. An interface transition in a skeleton is identified through the following four properties: (i) the component it belongs to, (ii) the communication object it accesses, (iii) its operation and arguments (e.g., a *send* operation with its message content), and (iv) partial order information which can be expressed in vector clocks that capture the happen-before relation between transitions.

Special care must be taken when vector clocks are used for interface-equivalence checking. Using vector clocks on traces directly might be problematic because the vector clocks also take into account internal transitions that are not included in skeletons. DEMETER therefore recomputes a skeleton vector clock for each trace. It first extracts the interface transitions and their dependencies from the original trace to build a dependency graph of the interface transitions. Based on the dependency graph, DEMETER re-computes the vector clock for the skeleton.

To expedite frequently-used interface-equivalence checking, DEMETER first imposes the same canonical representation on partial-order equivalent skeletons and computes a *signature* for a skeleton by applying a hash on that canonical representation. The equality of two skeletons is then the same as the equality of their signatures.

4.3 Distributed Runtime

The architecture of DEMETER enables a fair degree of parallelism. Model checking in DEMETER involves a global explorer and a set of local explorers, one for each component. Each local explorer is responsible for one component of the model-checked system and has no direct interactions with others. For each local skeleton of the component, the local explorer starts an *MC Worker* that executes the partial-replay local system for that component with respect to that local skeleton.

In our current implementation, the global explorer is the only major centralized task in the whole execution flow of DEMETER. Its core task, composition by substitution, is independent for each matching pair from G and B and can be executed separately, with its complexity linear to the length of the input traces. The complexity of finding matching pairs in G and B is in the worst case quadratic to the number of elements in the sets, although better data structures can be used to speed up the process of finding matching pairs. The size of G could grow exponentially with the number of components. In our experiments, we have not observed the global explorer becoming a bottleneck for the scalability of the entire exploration of DEMETER (see Section 5.2), largely because there are only a small number of components. We do not focus on cases where there are a large number of components because, as will be discussed in Section 6.1, it is possible to keep the number of components (at each level) small by organizing a system into a hierarchy of components.

All state changes on the global explorer are logged and persisted so that it can be re-started after failures. No replication is enabled, although doing so is straightforward. Because the global explorer

always checks whether a reported branching trace is new, having duplicate branching traces sent to the global explorer is acceptable. As a result, any MC Worker can be re-started without causing any correctness problem. In the worst case, an MC Worker can be re-started (possibly on a different machine) and the previously explored local state space would be re-explored. Because it uses an existing model checker for local exploration, its ability to re-start from failure is determined by that underlying model checker. Ideally, each MC Worker leverages a checkpoint/recovery mechanism in the underlying model checker to avoid redundant exploration due to failures.

4.4 Integration with Existing Model Checkers

CompWrap DEMETER is designed to integrate with existing model checkers, and we have enabled DIR for MACEMC and MODIST using DEMETER. Table 1 shows line-number counts for the common parts of DEMETER, as well as those specifically for MACEMC and MODIST. The common DEMETER modules include the following: the global explorer, part of the local explorer that is responsible for coordinating with partial-replay local systems and with the global explorer, the common data structure and API, and other utilities, such as the network library, cross-OS utilities, and message-digest modules. For MACEMC and MODIST individually, we need to implement a partial-replay local system (PRLocal), a component wrapper (CompWrap), and a converter for the Common Data Structure Layer. The converters are simple in both: they take less than 100 lines of code and are integrated with other pieces.

	MACEMC	MODIST
PRLocal	1,006	574
CompWrap	108	183
Total	1,114	757
DEMETER Common	7,279	

Table 1: Development cost as lines of code for DEMETER, DEMETER-MACEMC, and DEMETER-MODIST.

MACEMC Integration. MACEMC is a software model checker for systems implemented using the MACE compiler and C++ language extensions. MACE models each node as a state machine with atomic event handlers for events such as message reception and timeouts. MACEMC treats a target application as a single program that composes every node with a simulated network environment for distributed applications. With such a system wrapper, at any time, MACEMC selects a node and one of its pending events to call the corresponding event handler to transition the system to the next state. This is considered one transition; each pending event therefore corresponds to an enabled transition. Control returns to MACEMC when a transition completes, while a transition could introduce new events to the system. MACEMC repeats this process as long as there are pending events.

For state-space exploration, MACEMC must control all sources of non-determinism, such as the scheduling of pending events, the use of a special Toss command in event handlers, or the use of timeouts in event handlers. In the implementation of MACEMC, the *RandomUtil* module in MACE controls such non-determinism in the system. Nodes in MACE interact with each other via TCP/UDP services. Each transition could trigger send operations that will enable corresponding receive events on receiving nodes. Transitions containing send or receive operations are candidates for interface transitions.

MACEMC's *system wrapper* therefore exposes and controls *RandomUtil*, as well as send and receive operations. Because the information associated with send and receive operations is insufficient (e.g., for identifying the destination of a send operation), a *component wrapper* has to trace it down in MACE to fill the needed information for interface transitions. In some cases, depending on how a component is defined, a send or receive operation might not be an interface transition. This happens when the receiving node is in the same component as the sender.

Data-structure conversion between MACEMC and DEMETER is relatively simple. Nodes in MACE are units of execution and we use node id as the thread identifier. Events in MACE have information about corresponding transitions. DEMETER does require recording any non-deterministic choices within an event handler. In fact, DEMETER enumerates all such choices to find out the set of possible transitions because different non-deterministic choices correspond to different transitions for the processing of an event. Each transition in MACE may contain multiple network operations that DEMETER must store to define interface transitions appropriately. MACEMC does not track partial-order dependencies. Without making any internal changes within MACEMC, DEMETER tracks dependencies for interface transitions conservatively where any two transitions from the same node are assumed to be dependent. As shown in Section 5, this conservative way of defining partial order has significant implications on the effectiveness of DIR.

MACEMC implements two search algorithms. The first is a depth-first search (DFS) that enumerates all possible execution paths with an execution depth bound and is used to verify safety properties in a limited state space. The other one is a random walk algorithm that is used to detect potential liveness bugs. We apply DEMETER only to improve the DFS part of MACEMC since its random exploration does not check whether a randomly executed transition introduces a redundant trace, and hence it gives up any hope of reducing redundancies or achieving any notion of completeness.

MODIST Integration. MODIST is a software model checker that detects bugs due to non-determinism in distributed applications. In MODIST, any concurrent program behavior can be modeled as different invocation orders of Win32 APIs, such as *EnterCriticalSection* and *WaitForSingleObject*. MODIST provides a module called *dist_sys* that maintains the application state and captures most Win32 API invocations of a target application, including synchronization, network, and file-system operations. This module constitutes the system wrapper for MODIST.

In MODIST, a transition is defined as an execution between two consecutive invocations of system APIs. There is a straightforward mapping between MODIST's data structures and DEMETER's. Process Id and thread Id are combined to identify a thread, while the operation of each transition can be identified by the MODIST id of the corresponding Win32 API. MODIST itself maintains traces as a partial order and its vector clock can be used directly in DEMETER's common data structure.

MODIST's state-space exploration uses DFS with dynamic partial-order reduction. This algorithm is designed for a general transition system and requires a partial-order dependency relation between transitions. Local explorers in DEMETER directly use this state-space exploration algorithm in their partial-replay local system.

5. EXPERIMENTS AND EVALUATIONS

In this section, we describe our experiments on DEMETER and report findings of our evaluation results on DEMETER-MACEMC and DEMETER-MODIST, two real model checkers that we have built in DEMETER by incorporating MACEMC and MODIST. We conduct all of our experiments on a cluster of machines (Intel Xeon x5550 2.67GHz CPU, 12GB main memory) on a 1Gb Ethernet.

Our experiments use representative applications for DEMETER-MACEMC and DEMETER-MODIST. For DEMETER-MACEMC, we check PASTRY and CHORD, two well-known peer-to-peer distributed hash-table implementations on MACE, as well as PAM, an unoptimized PAXOS implementation on MACE for a single consensus decision. PAM was independently developed by a student. For DEMETER-MODIST, we choose MPS, a production PAXOS implementation that has been running in Microsoft data centers for years and contains about 53K lines of code. We also check Berkeley DB (BDB), a widely used open-source transactional storage engine that supports replication for applications requiring high availability. We check its release version 4.7.25.NC as done with the original MODIST [39]. We use an example application `ex_rep_mgr` that comes with BDB as the test driver. This application manages its data using the replication manager of BDB. During the test, the multiple replicas first run an election. Once completed, the elected primary creates worker threads to modify the replicated database simultaneously. We have also implemented the standard *Dining Philosophers Problem* (DPhi) mostly for validation/debugging because we know the expected results in this case.

Our experiments are designed to evaluate the following three key aspects: (i) on effectiveness, how effective is DIR for reducing state spaces, and what factors could affect its effectiveness? (ii) on performance, cost, and parallelism, how much overhead does the extra complexity of DEMETER incur in model checking and how does the capability of state-space exploration increase with the use of more machines? (iii) on experience with verification and bug finding using DEMETER, does the state-space reduction translate into improved ability to cover a meaningful logical state space completely, and does it help find bugs more effectively?

5.1 Effectiveness

To estimate the effectiveness of DIR, we run DEMETER on target applications and record the number of local traces that have been explored by the local explorers. We then compute the number of global traces that are covered by those local traces. The computation is performed as follows: for each global skeleton κ, let n_c be the number of local traces in component c that are interface-equivalent with κ on c's interface. These local traces can compose across components to create global traces, whose number is then $\Pi_{c \in \mathscr{C}}(n_c)$. Let n_g be the sum on the number of global traces over all global skeletons. We then compute the *reduction ratio* as n_g divided by the number of explored local traces on all the local explorers.

Table 2 reports the reduction ratio (*Red-Ratio*), the actual number of global skeletons discovered, and the number of local traces explored. We run target applications in different settings in terms of the number of nodes (components) and perform each model checking for hours. App-*n* refers to the application running with *n* nodes (components), except that DPhi-*n* has *n* components with each containing 8 philosophers. Overall, we are seeing significant state-space reduction with the reduction ratio ranging from 5 to over 500,000. We see a significant increase when moving from a 2-node to a 3-node system due to the multiplicative factor. Notice that all the applications in Table 2 except MPS, BDB, and CHORD-3 can also be fully checked by the original model checker. For those applications, we have validated the calculated value of n_g used for reduction ratio with the number of traces explored by the original model checker. This confirms that DEMETER with DIR upholds completeness and provides the justification to use calculated n_g for reduction ratio when the state space is too large to be fully explored by the original model checker.

Application	Red-Ratio	Global Skel	Local Trace	RT-Ratio	Speed-up
DPhi-2	41.7	6	1,510	2.0	20.9
DPhi-3	7,098.0	25	2,236	1.2	5,915.0
MPS-2	487.9	5	5,599	3.2	152.5
MPS-3	542,944.0	457	377,965	2.5	217,177.6
BDB-2	277.2	527	25,113	5.6	49.5
BDB-3	278,481.2	664	50,592	6.3	44,203.4
Pam-2	5.4	39	856	2.3	2.3
Pam-3	97.8	65	6,081	5.2	18.9
Pastry-2	4.9	48	713	1.5	3.3
Pastry-3	132.4	2,220	7,360	9.7	13.6
Chord-2	19.0	48	3,282	2.7	7.0
Chord-3	1,587.0	1,326	17,384	2.9	547.2

Table 2: State-space reduction and cost reduction of DEMETER. The applications in top-half of the table are checked by DEMETER-MODIST, while the ones in bottom-half are checked by DEMETER-MACEMC.

The reduction ratios for MPS and BDB are particularly impressive. For MPS, each node is implemented with multiple threads that have to synchronize with others using *EnterCriticalSection*, e.g., to access a shared message queue. A significant portion of such different interleaving does not lead to changes in the interface, thereby resulting in state-space reduction. Most of such interleaving is in the underlying common network library, which is fairly complicated as it supports various forms of networking (e.g., AsyncIO with completion port). Similarly, BDB employs multiple threads to handle the delivered messages and update shared database or replication-related data structures. It also uses *WSAEventSelect* to process asynchronous network events. Again, most of the complex internal non-determinisms do not propagate across interfaces.

Although respectable, the reduction ratios for applications in MACE are relatively low. Our investigation shows that numbers of local traces for each global skeleton are relatively small in part because of our conservative partial-order tracking for DEMETER-MACEMC: two send transitions from the same node are always considered to have dependencies. Different orders of two inherently concurrent sends (on two separate threads, for example) would lead to different global skeletons. If their dependencies were accurately modeled, as MODIST does, they would be considered independent and their relative order due to intra-node non-determinism would not matter to other nodes, which would lead to a smaller number of global skeletons and better reduction.

5.2 Performance, Cost, and Parallelism

The reduction ratio tells only part of the story. The cost of exploring a trace in DEMETER can be noticeably higher due to the extra complexity related to DIR, which includes the extra cost in the partial-replay local system of the local explorer (e.g., computing the signature of the local skeleton for each local trace to check whether it is a branching trace), as well as the cost of composition and projection in the global explorer. In our experiment, we compute *RT-Ratio* as the relative cost of exploring a local trace in DEMETER

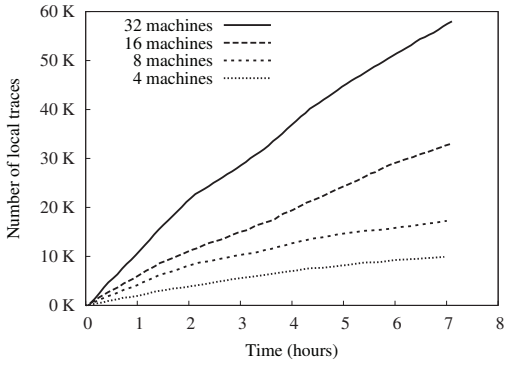


Figure 7: Numbers of explored local traces over time for MPS-3, with different numbers of worker machines.

with respect to the cost of exploring a global trace in the original model checker. The cost of exploring a local trace in DEMETER is the total amount of time spent on all the local explorers and the global explorer, amortized over the total number of explored local traces. As shown in Table 2, the *RT-Ratios* are significantly less than the *Red-Ratios*, which means that, although for the execution of a given trace, DEMETER is slower than original model checkers, it wins by exploring far fewer executions. We measure the effective speedup, without considering any potential parallelism in DEMETER, as *Red-Ratio* divided by *RT-Ratio*. These results are shown as *Speedup* in Table 2. For MPS-3, we are seeing an effective speedup of over 10^5, while for PAM-2 the speedup is only about 2.

While having a small number of nodes is sufficient to discover many protocol-level issues, in order to understand how the reduction effectiveness and the composition cost scale with the number of components, we did also run MPS with 5 nodes for 1.5 hours (without completely searching local state spaces for each global skeleton): the reduction ratio and speedup already reached 10^9, confirming the trend of increased effectiveness with increased number of components. We also noticed a significant increase in the cost of composition: an order of magnitude increase from MPS-3. We are likely to run into scalability issues at some point with the global explorer. Section 6.1 discusses how we might address those issues.

Scalability. *RT-Ratio* and *Red-Ratio* do not take into account the effect of distributed and parallel execution. We further evaluated the inherent parallelism in DEMETER by deploying it on a cluster of machines. The goal of the experiment is to understand the increased effectiveness in state-space exploration as it uses more machines. We use DEMETER-MODIST on MPS as the showcase and vary the number of machines running MC Workers. Separately, we have one machine running the global explorer and three more as the local explorers, one for each component, coordinating MC Workers.

We run each experiment for about 7 hours. Figure 7 shows the numbers of discovered local traces over time with different numbers of worker machines. In each case, DEMETER is able to explore new local traces linearly over time and we also see near-perfect scalability as the number of machines goes up. This demonstrates (i) partial-replay local systems are embarrassingly parallel and (ii) composition by the global explorer does not become a bottleneck and can always dispatch enough local skeletons to make each worker machine busy when the local workers can discover and report enough new branching traces for composition in a short period of initial time.

5.3 Experiences

It is natural to ask whether or not the observed significant state-space reduction translates into any tangible benefits for improving system reliability. In particular, we look at two aspects: (i) DEMETER's ability to explore completely a meaningful logically bounded state space of a system implementation for a higher degree of reliability assurance and (ii) how DEMETER improves our ability to find bugs.

Our experiment shows that DEMETER is capable of *completely* exploring a logically meaningful state space of a 3-node CHORD and MPS without any artificial bound on exploration depths. We do have to make the system finite: for CHORD, the system ends as soon as all three nodes join successfully with timeout fired at most once at each node. For MPS, we bound ballot numbers (to 2) and decree numbers (to 1). Such logical bounds still allow for a vast number of scenarios covering both phases of the PAXOS protocol. To see why previous model checkers do not come close to finishing the exploration, our CHORD exploration took 3 hours, exploring 17,384 local traces that correspond to 27,588,408 global traces, which would take more than 2 months for MACEMC to explore. Similarly, the exploration of DEMETER on MPS took 18 hours, exploring 182,689 local traces that correspond to 7,743,820,726 global traces, which would take about 34 years for MODIST to explore, even with its already significant partial-order methods for state-space reduction.

We believe the ability to explore thoroughly a meaningful logical state-space of a real implementation is significant. It offers a higher degree of assurance for system reliability as basic implementation-level protocol behaviors have now been "verified". Such kind of *coverage* statement for implementation was not possible before with the existing implementation-level model checkers and with the existing state-space exploration and reduction strategies on any non-trivial real production system.

Bug Finding. DEMETER naturally looks for safety bugs through state-space exploration. Finding liveness bugs often require a special set of strategies, as was done with MACEMC [29]. Those strategies are often incompatible with DIR, although they might still benefit from DIR. Our investigation focuses on safety bugs, while leaving liveness bugs to future work.

Our experiences with DEMETER on finding safety bugs are mixed, as significant state-space reduction does *not* translate automatically to proportional increases in bug-finding effectiveness. On the positive side, we have found two serious bugs in PAM: the depths at which those bugs were found are beyond the capability of the DFS search in MACEMC. The first bug arises due to loss of protocol state during replica recovery. In a 3-node replica system with nodes *a*, *b*, and *c*, replica *a* initially becomes a leader and passes a decree by getting the supporting vote from *b* only. Then *b* restarts from a failure and incorrectly votes with *c* to pass a different decree, because *b* has lost its state (related to *a*'s earlier actions) during failure/recovery. DEMETER found this bug in a trace with a total depth of 27. The second bug is due to an incorrect vote message. When a leader receives accepted values in phase 1, it must vote in phase 2 the accepted value with the highest ballot number. The initial implementation incorrectly chose the first received value instead. This bug appears only when two different values were accepted on two different nodes and in our experiment involves a trace with a total depth of 43.

On the negative side, we did not find any new bugs when running DEMETER on MPS, BDB, PASTRY, and CHORD through a simple brute-force search. We found only the first bug in PAM. Bug finding turns out to be significantly different from covering a state space. When a state space is large, it is more effective to cover as many interesting scenarios as possible. Bug finding is therefore best guided with application-specific knowledge and DEMETER offers a more powerful tool for this guided process. For example, rather than focusing on the initial phase and running a system for a long time, we periodically stop the system to get a checkpoint and start a new exploration from that checkpoint if we think that checkpoint state is "interesting" (e.g., having replicas with inconsistent states). We essentially do *vertical decomposition* of system execution and prune out "uninteresting" branches. This allows DEMETER to explore longer traces more effectively. The second PAM bug was found this way through 3 "inconsistent" checkpoints as stepping stones. The final buggy path is the result of concatenation of these sub-paths.

6. DISCUSSIONS

This section discusses three subtle issues that affect the effectiveness of DIR: how to define components, how to check global properties, and how to avoid branching redundancies.

6.1 Defining Components

The effectiveness of DIR depends on how a target system is partitioned into components. One natural way is to make each process a component. In our experience, this simple approach is effective because processes in a distributed system tend to communicate with each other through message passing, where the design tends to minimize communication between them for performance reasons. Application logic within a process is often implemented with multiple threads and asynchronous I/O for high performance, which introduces substantial sources of non-determinism in it. Therefore, interactions between processes can be significantly simpler than non-determinism within each process, leading to significant state-space reduction when explored with DIR.

It is also possible to group multiple processes together to form a component. Even with processes running the same code, different groupings often have different effects, due to different roles the processes play in an application. For example, in dining philosophers, it makes sense to group consecutive philosophers together because doing so will lead to an interface with only two forks no matter how many philosophers are included in that component. In the worst case, if philosophers are divided into two components in alternation, all forks will become interface objects. Even for the 3-node cases of PASTRY and CHORD, nodes 1 and 2 have more interactions between them. Our experiments show better reduction ratios when grouping those two into a group, compared to grouping nodes 2 and 3 in a component, although having three components yields the best reduction ratios.

The decision of whether certain processes should be grouped together as one component depends on a number of factors. Tightly coupled processes should ideally be grouped together, although this will increase the complexity of partial-replay local systems. When the number of components is high, the number of global skeletons goes up exponentially, which increases the overhead on the global explorer. We have developed an algorithm called *hierarchical dynamic interface reduction* to address this issue further. It reduces the overhead on the global explorer by recursively dividing a system into a small number of components at each level. We have shown the effectiveness of this method on dining philosophers, which leads to exponential state-space reduction in theory. We have yet to show that the added complexity brings significant practical benefits on real applications. Peer-to-peer protocols such as PASTRY and CHORD are ideal targets.

6.2 Global Property Checking

Not only can DEMETER discover local assertion failures and misbehavior during state-space exploration, but can also be used to check global safety properties. The ideal place to perform global property checking is at the global explorer as it has a global view on a system via global skeletons and global traces. To facilitate global property checking, each component has to expose not only interface transitions but also any local states that are referenced by the specified global property. Updates to local variables referenced will have to be reported. Those states are taken into account when assessing whether an execution creates a branching trace, although local skeletons for the local explorers do not have to contain such states because they are not used in partial-replay local systems. All such information is incorporated into global skeletons during the composition process. The global explorer can then enumerate all the consistent snapshots of those state variables on all the global skeletons to check global properties. From our experience, adding global property checking into DEMETER-MODIST is natural as MODIST has the mechanism to expose states. Adding the same functionality to DEMETER-MACEMC is harder because the state variables are not easily exposed in MACEMC.

6.3 Branching Redundancy

DEMETER builds a partial-replay local system for each local skeleton. A branching trace is not part of a local state space, but should be counted as overhead for local exploration. We have observed that some trace prefixes are explored in multiple partial-replay local systems for different skeletons, once as part of local traces in one, and again as part of branching traces in another. This leads to redundant state-space exploration by partial-replay local systems for different local skeletons. DEMETER could explore a branching trace multiple times since it does not know whether that branching trace is already explored in other partial-replay systems. One solution to this problem is to have DEMETER explore all partial-replay systems of a component on a single worker to avoid such redundancies as the redundancies are among MC Workers for the same component. As a result, DEMETER's parallel granularity is now limited to the number of components. However, we can accelerate the exploration by parallelizing the exploring algorithm itself. For example, it is possible to have a different worker exploring a sub-tree space of a particular local-state partial-replay system. One caveat is the potential interactions with the state-space exploration strategy in an *eMC*: for example, MODIST uses dynamic partial order reduction, where the exploration of a sub-tree space might need to add new transitions to the execution points above that subtree.

7. RELATED WORK

Model Checking. Model checkers have previously been used to find bugs in both the design and implementation of software. Traditional model checkers require that users transform a target system into an abstract model beforehand [11, 34, 2, 14, 26, 27]. This process is often expensive and error-prone, thereby limiting the use of these tools for large-scale software systems. Implementation-level software model checkers [18, 36, 32, 41, 40, 39, 33, 29, 38] can instead work directly on implementations of software systems by systematically controlling executions and exploring non-determinisms in a system implementation.

Both traditional model checkers and software model checkers have to face the problem of state-space explosion. Based on the observation that complex large-scale systems normally consist of loosely-coupled components, compositional reasoning techniques [6, 35, 12, 4, 22, 31] have been proposed and applied for effective state-space reduction. Such methods check each component of a system in isolation and infer global system properties appropriately. However, all of the previous proposals target only traditional model checkers. Some of them need substantial human effort [22, 31], and hence are not scalable. Others [6, 35, 12], although automatic, require eagerly constructing an abstract component acting as the environment of a component being checked, making it impractical for complex large-scale systems. In contrast, DEMETER applies DIR to software model checkers by lazily and dynamically discovering all interface interactions among components, thereby significantly reducing the amount of human effort and removing any need for static program analysis to transform a system implementation or its environment into an abstract model.

Alur and Yannakakis [1] applied model checking on hierarchical state machines where the state nodes of a state machine can be ordinary states or state machines themselves. Their method leverages this hierarchical structure of the state machines to avoid exploring the same sub-state-machine multiple times. Their method applies to formal sequential hierarchical state-machine specifications only, whereas DIR targets implementation-level model checking of concurrent and distributed systems without formal specifications.

The most related method was proposed recently by Guerraoui and Yabandeh [23] to separate the exploration of system states (i.e., the combination of node-local states) and network states. The proposed method takes an optimistic approach and does not model dependencies between network transitions. This imprecision leads to loss of soundness, which has to be addressed using a compensatory validity check. In contrast, our approach tracks dependencies explicitly and ensures soundness during exploration.

State-Space Reduction. Other state-space reduction techniques, such as partial order reduction [17, 16], symmetry reduction [28], and abstraction [25, 10, 3, 13, 21], have been proposed and investigated. Those techniques are orthogonal to DIR and can often be applied together. For example, the analysis presented in Section 2.3 on the example in Figure 1 helps to show why DIR is orthogonal to partial order reduction (POR). POR states that it is sufficient to explore only one permutation order of a set of independent operations. For instance, only one order of the `Sum` in the primary and the `Ckpt` in the secondary need be explored because they are independent. Fundamentally, POR still views a system as a whole. Thus, when the `Sum` and `Ckpt` operations within one server interleave differently, POR has to re-explore the entire system. Nonetheless, combining the two reduction techniques is easy as both the global explorer and the local explorers in DIR can use POR to reduce the number of executions they explore. In fact, when integrating with MODIST, we have effectively enabled both POR and DIR. It is an interesting future direction to see whether other state-space reduction techniques are compatible with the architecture of DEMETER.

Error-Detection Techniques. Recently, symbolic execution [8, 7, 20, 9] has been used to detect errors in real systems. This technique takes program inputs as symbolic values and explores all possible execution paths by solving the corresponding path conditions. Similar to model checking, symbolic execution also confronts the problem of state-space explosion. SMART [19] applied *composition* in symbolic execution at function granularity. It checks functions in isolation, encoding the results as function summaries expressed using input preconditions and output postconditions, and then re-using those summaries when checking higher-level functions. However, their idea cannot be applied in checking concurrent/distributed systems. Zamfir and Candea [42] further enhanced symbolic execution to support concurrent systems by making thread-scheduling decisions symbolic. It is again an interesting future research direction to understand whether an idea similar to DIR would help in this scenario.

Software Verification. Many attempts have been made to verify software implementations [30, 37, 5, 24, 3, 15]. BLAST [24] and SLAM [3] combine predicate abstraction and model checking techniques to analyze and verify specific safety properties of device drivers. Model checking is complementary in that it can be used to check a bounded small state space thoroughly and to provide some assurance by attempting to find defects when complete verification is infeasible.

8. CONCLUSIONS

DEMETER provides early validation on dynamic interface reduction and closes the gap between a theoretically interesting algorithm and a practical model checking framework that demonstrates its effectiveness on representative distributed systems with real model checkers. Experiences with DEMETER further shed lights on several interesting future directions. First, removing any scalability hurdle to applying DEMETER to a large number of components could further unleash the power of this reduction. Second, further pushing the boundary of state spaces that can be completely explored could make model checking a useful tool for software reliability assurance. Third, finding bugs effectively with DEMETER requires a different thinking from covering a state sub-space completely and might need guidance with domain knowledge.

9. ACKNOWLEDGEMENTS

We thank Tisheng Chen and Yi Yang for their help at the early stage of this project, and our colleagues at System Research Group in Microsoft Research Asia for their comments and support. Sean McDirmid helped greatly in improving the writing of this paper. Charles E. Killian, Jr. provided valuable information on MACEMC. We would also thank Lorenzo Alvisi, Robbert van Renesse, and Geoffrey M. Voelker for their comments on the paper. We are grateful to the anonymous reviewers for their valuable feedback. We are particularly in debt to our shepherd Petros Maniatis for his detailed guidance and constructive suggestions. Junfeng was supported in part by NSF grants CNS-1117805, CNS-1054906 (CAREER), CNS-1012633, and CNS-0905246; and AFRL FA8650-10-C-7024 and FA8750-10-2-0253.

10. REFERENCES

[1] R. Alur and M. Yannakakis. Model checking of hierarchical state machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):273–303, 2001.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 103–122, May 2001.

[3] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[4] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *COMPOS'97: Revised Lectures from*

the International Symposium on Compositionality: The Significant Difference, pages 81–102, London, UK, 1998. Springer-Verlag.

[5] W. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, pages 1382–1396, 1989.

[6] J. Burch, E. M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In *VLSI*, pages 49–58. North-Holland, 1991.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[9] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[10] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[11] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[12] E. M. Clarke, D. Long, and K. L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 415–426, New York, NY, USA, 2006. ACM.

[14] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448, June 2000.

[15] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 352–367, 2009.

[16] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, Jan. 2005.

[17] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of LNCS. 1996.

[18] P. Godefroid. Model checking for programming languages using verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186, New York, NY, USA, 1997. ACM.

[19] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–54, New York, NY, USA, 2007. ACM.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[21] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 72–83, London, UK, 1997. Springer-Verlag.

[22] O. Grumberg and D. Long. Model checking and modular verification, May 1994.

[23] R. Guerraoui and M. Yabandeh. Model checking a networked system without the network. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, NSDI'11, Berkeley, CA, USA, 2011. USENIX Association.

[24] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the 10th international conference on Model checking software*, pages 235–239. Springer-Verlag, 2003.

[25] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70. ACM Press, 2002.

[26] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[27] G. J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Applications of Concurrency to System Design (ACSD '01)*, June 2001.

[28] C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.

[29] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 243–256, April 2007.

[30] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, 2009.

[31] K. Laster and O. Grumberg. Modular model checking of software. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 20–35, 1998.

[32] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 75–88, Dec. 2002.

[33] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, June 2007.

[34] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982.

[35] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *IEEE Int. Conf. Computer-Aided Design*, pages 130–133, 1990.

[36] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[37] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *Communications of the ACM*, 23(2):131, 1980.

[38] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[39] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[40] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.

[41] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, Dec. 2004.

[42] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334. ACM, 2010.

APPENDIX
A. PROOF SKETCH

In this section, we prove that the algorithms for the global explorer and the local explorer in Section 3.4 preserve both soundness and completeness. The proofs use the substitution rule introduced in Section 3.4 as an "axiom" that follows directly from the definitions of components, interfaces, and interface equivalence.

A.1 Soundness

Lemma A.1. *With respect to $\langle \kappa_c, \tau \rangle \in L_c$, where τ is a valid global trace, the partial-replay local system produces a valid global trace in every exploration step. A global trace is valid if its execution can occur in a real run of the checked system.*

Proof. Consider a system consisting of component c and the rest of the system R. A partial-replay local system for component c with respect to $\langle \kappa_c, \tau \rangle \in L_c$ starts from the initial state and in each step either picks an enabled transition from component c or replays τ's transitions in R. To enable replaying, the partial-replay local system tracks which of τ's transitions in R can be replayed: a transition t in τ can be replayed if and only if t is a transition from R and any transition $t' \neq t$ in $proj_R(\tau)$ satisfying $t' \preceq t$ has been replayed in previous steps. The transitions replayed in R and the interface transitions from component c always form a prefix of $proj_R(\tau)$. Therefore, at any step, there exists a prefix τ_p of τ such that $proj_R(\tau_p)$ captures all replayed transitions projected to R (including both R's internal transitions and interface transitions) and their partial order.

The partial-replay local system preserves the partial order between transitions in $proj_c$ as in the original system and between transitions in $proj_R(\tau)$ as in τ. By definition, the transitions taken in c and the interface transitions related to c form a projection of some valid trace τ_1 (i.e., $proj_c(\tau_1)$ captures all transitions in c and all the interface transitions for c). The trace that the partial-replay local system produces is therefore $subst_c(\tau_p, \tau_1)$. Due to the substitution rule, it is a valid trace.

□

Lemma A.2. *(i) For each $\langle \kappa, \tau \rangle \in G$, τ is a valid global trace, (ii) for each $\langle t_b, \tau_b \rangle \in B$, τ_b is a valid global trace, and (iii) for each $\langle \kappa_c, \tau \rangle \in L_c$ for any component c, τ is a valid global trace.*

Proof. Prove by induction on the order of the entries added into sets G, B, and L_c's.

Initially, the algorithm uses a real global execution to find a global trace to add to G. That global trace is valid by construction. For the induction step, assume that all entries in G, B, and L_c satisfy the conditions. We consider the following cases:

Case 1: A new entry $\langle \kappa_c, \tau \rangle$ is added into L_c. There must exist some $\langle \kappa, \tau \rangle \in G$ satisfying $proj_c(\kappa) = \kappa_c$. By the induction hypothesis, τ is a valid global trace.

Case 2: A new entry $\langle t_b, \tau_b \rangle$ is added to B. This is because t_b is a branching transition at trace τ_b when executing a partial-replay local system for c with respect to some $\langle \kappa_c, \tau \rangle \in L_c$ for some component c. By the induction hypothesis, τ is a valid trace. τ_b is a valid trace by the construction of the partial-replay local system due to Lemma A.1.

Case 3: A new entry $\langle \kappa_n, \tau_n \rangle$ is added into G. This is because there exists $\langle t_b, \tau_b \rangle \in B$ and $\langle \kappa_g, \tau_g \rangle \in G$ satisfying $proj_c(\kappa_g) = skel(proj_c(\tau_b))$, $\tau_n = subst_c(\tau_g, \tau_b) \circ t_b$, and $\kappa_n = skel(\tau_n)$. By the induction hypothesis, τ_b is a valid global trace and τ_g is a valid global trace. Following the substitution rule, $subst_c(\tau_g, \tau_b)$ is a valid trace. By the construction of $\langle t_b, \tau_b \rangle$, t_b is a valid transition in $subst_c(\tau_g, \tau_b)$ because it is an enabled transition from c in τ_b. Therefore, $\tau_n = subst_c(\tau_g, \tau_b) \circ t_b$ is a valid trace. □

Theorem A.3. *For any local trace τ_c that the local explorer for component c discovers, there exists a valid global trace τ, such that $\tau_c = proj_c(\tau)$.*

Proof. Follows directly from Lemma A.1 and Lemma A.2. □

A.2 Completeness

Theorem A.4. *Assume a local explorer with the eMC and the partial-replay local system explores completely the enabled transitions in a component, for any valid global trace τ_g, the global explorer eventually adds $\langle skel(\tau_g), \tau \rangle$ into G for some global trace τ. For every component c, the local explorer discovers $proj_c(\tau_g)$.*

Proof. Assume there exists a valid global trace τ_g that invalidates the theorem, i.e., either some of its projected local traces for components cannot be explored by the local explorers, or its corresponding global skeleton cannot be discovered by the global explorer. There must be a longest prefix τ_p of this global trace τ_g that satisfies the following properties: (i) the local trace $\tau_x = proj_x(\tau_p)$ for any component x has been explored by the local explorer of x and (ii) there exists a global trace τ_p^g, such that $\langle skel(\tau_p), \tau_p^g \rangle$ has been discovered by the global explorer and is therefore in G.

Let t be the subsequent transition of τ_p in τ_g: by definition of τ_p and τ_g, such a transition must exist. Without loss of generality, let t be a transition belonging to a component c. Transition t will be enabled during the local exploration of c against $\tau_c = proj_c(\tau_p)$ according to the substitution rule. We consider two cases.

Case 1: t is an internal transition. We show that $\tau_p \circ t$ satisfies (i) and (ii) as τ_p does. Because t is enabled at τ_c, the local explorer for c will take this transition, reaching the projection of $\tau_p \circ t$ to c. For any other component, the projection of $\tau_p \circ t$ is the same as that of τ_p. Because t is an internal transition, $skel(\tau_p \circ t)$ is also the same as $skel(\tau_p)$. Because $\tau_p \circ t$ is a prefix of τ_g longer than τ_p, we have a contradiction with the definition of τ_p.

Case 2: t is an interface transition. Because t is enabled at τ_c, the local explorer will take this transition. Again, we show that $\tau_p \circ t$ satisfies (i) and (ii) as τ_p does. The part of the proof about (i) is the same as in Case 1. For (ii), we need to show that the global explorer discovers $skel(\tau_p \circ t)$ through composition by substitution. Let τ_b be the global trace that the partial-replay local system constructs when reaching τ_c. We have $\tau_c = proj_c(\tau_b)$. Pair $\langle t, \tau_b \rangle$ will be reported to the global explorer. Because $\langle skel(\tau_p), \tau_p^g \rangle \in G$ and $proj_c(skel(\tau_p)) = skel(proj_c(\tau_b))$ hold, the global explorer will construct a new global trace $\tau_n = subst_c(\tau_p^g, \tau_b) \circ t$ and discovers $skel(\tau_n)$. By construction, we have $skel(subst_c(\tau_p^g, \tau_b)) = skel(\tau_p)$. Therefore, we have $skel(\tau_n) = skel(\tau_p \circ t)$, which means that (ii) holds for $\tau_p \circ t$. Again, because $\tau_p \circ t$ is a prefix of τ_g longer than τ_p, we have a contradiction with the definition of τ_p.

□