Asynchronous Programming, Analysis and Testing with State Machines

Pantazis Deligiannis[1] Alastair F. Donaldson[1] Jeroen Ketema[1] Akash Lal[2] Paul Thomson[1]

[1]Imperial College London, UK
{p.deligiannis, afd, jketema, pt1110}@imperial.ac.uk

[2]Microsoft Research, India
akashl@microsoft.com

Abstract

Programming efficient asynchronous systems is challenging because it can often be hard to express the design declaratively, or to defend against data races and interleaving-dependent assertion violations. Previous work has only addressed these challenges in isolation, by either designing a new declarative language, a new data race detection tool or a new testing technique. We present P#, a language for high-reliability asynchronous programming co-designed with a static data race analysis and systematic concurrency testing infrastructure. We describe our experience using P# to write several distributed protocols and port an industrial-scale system internal to Microsoft, showing that the combined techniques, by leveraging the design of P#, are effective in finding bugs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Asynchronous programming; state machines; concurrency; static data race analysis; systematic concurrency testing

1. Introduction

Modern computing infrastructure offers multiple computational resources, through multi-core and distributed clusters, and leaves the software to exploit concurrency for responsiveness (latency) or performance (throughput). Consequently, developing software requires expertise in concurrent programming. However, fine-grained concurrency is at odds with correctness: it makes testing and analysis of the software much harder than in the sequential case.

This paper is about programming asynchronous systems where the main concern is *responsiveness*. To achieve responsiveness, long-running sequential activities are often split into multiple shorter tasks that are scheduled asynchronously. If the tasks are not coordinated properly, the interleaving between tasks from different activities can be a source of bugs.

Multiple languages and libraries have been proposed to help address coordination issues. Examples include actor-based languages (e.g. Scala [21] and Erlang [28]), the .NET Task Parallel Library [18] and Grand Central Dispatch [2]. These approaches have mostly focused on simplifying the programming task by moving towards a declarative paradigm. No direct connection is established between language design and the analysis and testing that follows once code is written.

We present P#, a new language for high-reliability asynchronous programming co-designed with a strong static analysis to ensure race-freedom, and systematic concurrency testing techniques that can exploit the statically enforced race-freedom guarantee. Co-designing P# with accompanying static and dynamic analysis techniques makes a strong move towards ensuring high-reliability of software written in the new language.

The P# Language A P# program is composed of multiple state machines that communicate with each other by sending and receiving *events*, which are similar to messages in other asynchronous programming languages. A state can register *actions* to handle incoming events. On arrival of an event, the action registered for the event is executed. Actions are arbitrary methods written in C# with the restriction that they must be sequential, i.e. they may not spawn threads or use synchronization operations. Actions are equipped with a `send` primitive to enable sending events to other machines. The actions on states correspond to asynchronous tasks and the event-passing as well as the structure of the state machine is what imposes a coordination control over the actions.

P# is fully integrated with C#. It presents the programmer with the familiar programming and debugging environment of C#, facilitating object-oriented design. P# forces the programmer to declaratively specify asynchrony via state machines, because actions are sequential and the only way to exploit concurrency is by using multiple machines. Moreover, P# allows full reuse of legacy sequential C# code; a programmer does not have to start from scratch.

Static Data Race Analysis and Testing A P# program executes in a shared memory environment and event payloads are allowed to reference heap objects. This enables efficient event-passing without the need for deep-copying or marshaling, but leads to aliasing which can result in data races between concurrent events. This problem has been acknowledged in many message-passing systems. Solutions include runtime support for on-the-fly race detection [14], which can limit responsiveness by incurring a runtime overhead, and the use of type systems that disallow races by design [15], which are often complicated and restrictive, hampering reuse of legacy code in a fully-featured language such as C#.

P# comes with a sound static data race analysis that leverages the state machine structure of the language and the absence of concurrency inside actions to track *ownership* of objects. An action assumes ownership of any payload it receives and any object it creates; it gives up ownership of any payload it sends as part of an event. As long as each object has a unique owner, data races cannot occur. We show experimentally that our static analysis is effective

in detecting and proving absence of data races in practical examples, and has higher precision than a similar static analysis proposed in earlier work [20].

A race-free P# program can still suffer from bugs such as assertion violations, and non-determinism due to asynchrony can make such bugs hard to detect. We have embedded a *systematic concurrency testing (SCT)* framework in our P# runtime which allows the event schedulings of a P# program to be explored in a controlled manner, facilitating deterministic replay of bugs. We optimize SCT by leveraging P#'s semantics and the results of our data race analysis: for race-free programs it suffices to consider machine interleavings at event-level granularity, not at the granularity of individual memory accesses. This leads to improved performance compared to the CHESS tool [19], which also supports analysis of C# programs.

As evidence of our claim that P# enables high-reliability asynchronous programming, we present a case study porting a large asynchronous system, internal to Microsoft, to P#. The porting and subsequent analysis and testing revealed five bugs in the original system that were previously uncaught across multiple releases. We also evaluate P# and our analysis and testing infrastructure using several distributed protocols.

To summarize, our contributions are as follows:

- We present P#, a new asynchronous language co-designed with static data race analysis and testing techniques.
- We formulate a sound static analysis for proving race-freedom of P# programs, showing that it improves on previous work in terms of scalability and precision.
- We develop systematic and randomized testing strategies that leverage the P# language and static analysis to provide higher efficiency than a state-of-the-art SCT tool.
- We report on the use of P# inside Microsoft as the porting and testing target of an industrial-scale asynchronous system.

2. Related Work

The P Language Our work is inspired by the P state machine-based language [6], which unifies programming and modeling, by enabling model checking of programs using the Zing model checker [1]. P has a surface syntax for writing state machines, operating over scalar values, and exposes a *foreign-function interface* to C for encoding functionality that cannot be implemented inside the language. Only the P components of a program, not the foreign-function calls, can be analyzed using Zing. This suffices for detecting protocol-related errors, but provides no guarantees related to the foreign functions. P does not provide a systematic way of testing a whole program. In contrast, our approach enables whole-program race analysis and systematic testing, and offers a similar programming experience as in other .NET languages, including IDE debugging support, due to the embedding of P# in C#.

Actor Languages Several languages, e.g. Scala [21], Erlang [28] and SALSA [27], define an actor model for implementing asynchronous systems. The actor model is more flexible and general than the state machine-based paradigm of P#. We argue that the limitations of state machines do not hinder the design of practical asynchronous systems, and that the simplicity of state machines enables scalable and precise static analysis (Section 7).

Message-Passing and Data Races The disadvantages associated with runtime- and type-based race analysis [14, 15], discussed in the Introduction, motivate static race analysis based on ownership. SOTER [20], an ownership-based static analyzer for actor programs, is most closely related to our work (see also Section 5.5). Our novel analysis generally has higher precision than SOTER, as we show by analyzing the four worst-performing SOTER benchmarks in Section 7.

Concurrency Testing Dynamic approaches to testing of asynchronous programs include concolic execution [23] and dynamic partial-order reduction [17]. Our testing methodology is inspired by *systematic concurrency testing* [9, 12, 19] and our intent was to reuse the CHESS tool [19], as it can already analyze .NET programs. In practice, we found that we could improve on the performance of CHESS by leveraging the domain-specific nature of P# programs through a custom scheduler embedded in the P# runtime.

3. A P# Program

Figure 1 graphically represents a simple P# program implementing a master-worker asynchronous system. A *master* service and multiple *worker* services (which are `BaseService` machines) communicate with each other to update and exchange user data. A `Dispatcher` machine coordinates the services. This example is reminiscent of a proprietary Microsoft system that we ported from C# to P# (see Section 7.1).

Each outer box in Figure 1 corresponds to a different P# state machine, which is a class inheriting from the *abstract* `Machine` class (defined by P#). The example program contains three such machines: `Dispatcher`, `BaseService` and `UserService`. Member fields (represented as inner boxes labeled "Machine Local Data") can be used to store data local to a machine; static fields are disallowed. The states of a P# machine are defined by inheriting from the abstract `State` class (defined by P#). P# enforces states to be nested classes of the machine they belong to; this ensures they cannot be accessed externally. State transitions are denoted by arrows in the figure; all the available transitions are summarised in the corresponding inner box of each machine labeled "State Transitions".

The `Dispatcher` machine has a list of service machines. It can change each such machine into a master by sending an *eChangeToMaster* event, or into a worker by sending an *eChangeToWorker* event. In the `Querying` state, the `Dispatcher` loops, sending requests to the services (e.g. to update their local data).

`BaseService` is an abstract machine from which all services inherit. It contains four abstract C# methods, three of which are bound as actions (represented in the inner box labeled "Action Bindings"); a service machine *must* implement all four of these abstract methods. `BaseService` starts in the `Init` state, where it receives a `Payload` containing a unique ID. A payload in P# can be a scalar or a reference sent by a sender machine. After assigning the value to the `ID` field, `BaseService` calls `InitializeState()`. `BaseService` transitions to a new state when it receives either an *eChangeToWorker* or an *eChangeToMaster* event. In the new state, the machine may handle a different set of events. For example, in the `Worker` state the machine can handle *eUpdateState* with the `UpdateState()` action, whereas *eUpdateState* cannot be handled in the `Init` state.

Finally, `UserService` inherits from `BaseService` and implements all four abstract methods. By encapsulating the state transitions and action bindings in `BaseService`, the programmer can focus on developing custom application logic in `UserService`. Without P#, the programmer would have to manually add synchronization operations (e.g. locks) in the `UpdateState()` and `CopyState()` methods, as they can potentially race with one another. Adding synchronization operations can be a tedious and error-prone process. Instead, P# allows the C# code to be simple and sequential.

4. Language and Semantics

We present the core syntax and semantics of P#, on which we build when formulating our data race analysis in Section 5. The semantics is based on the *event-driven automata* formalism from [7],

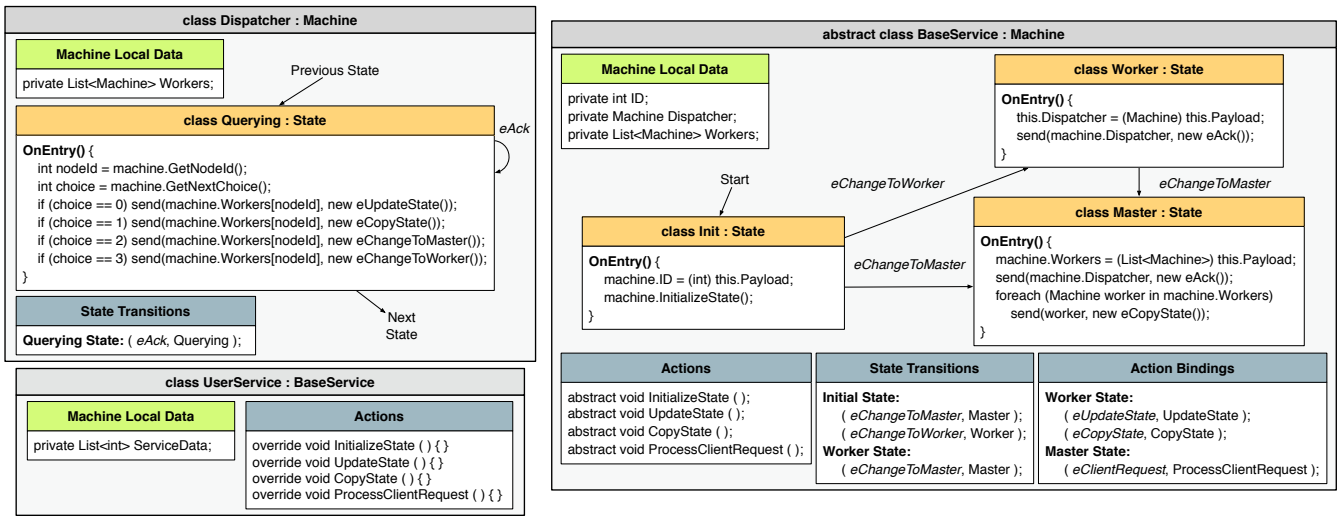

Figure 1. Graphical representation of a master-worker asynchronous system in P#.

which itself can be seen as a core calculus of P [6] minus the ability to dynamically instantiate new machines (i.e. automata). Our implementation has a feature set on par with P and *does* allow for dynamic machine instantiation.

Object-Oriented Language Machines in P# are defined as classes in an object-oriented language. The syntax of the language is presented in Figure 2. For brevity, we omit features such as exceptions and inheritance, although we support both. We use vector notation to denote sequences of declarations, e.g. $\overline{vd}$ denotes a sequence of variable declarations.

$$
\begin{array}{lll}
\text{stmt} & s & ::= \mathsf{send}_{dst}\ evt(v) \mid \mathsf{return}\ v \mid v := v \mid v := c \\
& & \mid v := v\ op\ v \mid \mathsf{this}.v := v \mid v := \mathsf{this}.v \\
& & \mid v := \mathsf{new}\ class \mid v := v.m(\overline{v}) \\
& & \mid \mathsf{if}\ (v)\ ss\ \mathsf{else}\ ss \mid \mathsf{while}\ (v)\ ss \\
\text{stmts} & ss & ::= s; ss \mid \varepsilon \\
\text{vdecl} & vd & ::= \text{type}\ v; \\
\text{mdecl} & md & ::= \text{type}\ m(\overline{vd})\ \{\overline{vd}\ \ ss\} \\
\text{cdecl} & cd & ::= \mathsf{class}\ class\ \{\overline{vd}\ \ \overline{md}\}
\end{array}
$$

Figure 2. Syntax of the object-oriented language

Each class (cdecl) consists of a sequence of member variable declarations $\overline{vd}$ followed by a sequence of method declarations $\overline{md}$. The type of each variable is either scalar (ensuring the variable stores primitive values such as integers or floats), or a reference type (such that the variable stores references to heap-allocated objects). Member variables are not directly accessible: a member v of the current class can only be accessed through an expression of the form $\mathsf{this}.v$, while a member of another class is only accessible via appropriate method calls.

A method (mdecl) has a number of formal parameters $\overline{vd}$, and its body consists of a number of local variable declarations $\overline{vd}$ followed by a sequence of statements ss, with ε the empty sequence. In statements (stmt), v refers to a local variable, a formal parameter of a method or this, with this being a constant reference to the class in which the statement occurs. We denote by c any literal scalar value and by op any operation over scalars.

We build a running example of a machine managing a linked list; we start by describing the elements of the linked list.

Example 4.1. Linked list elements are defined as follows:

```
class elem {
    int val; elem next;

    int get_val() {
      int ret;
      ret := this.val;
      return ret;
    }

    void set_val(int v) {
      this.val := v;
    }

    elem get_next() {
      elem ret;
      ret := this.next;
      return ret;
    }

    void set_next(elem n) {
      this.next := n;
    }
}
```

Thus, each element stores an integer value and a reference to a next element, and an accessor and a mutator are defined for both.

Figure 3 presents the operational semantics of the individual statements in our object-oriented language. We omit the rule for $\mathsf{send}_{dst}\ evt(v)$, which we discuss in the context of machine transitions. The rules are defined over tuples (ℓ, h, S, ss). The *local store* ℓ is a map from local variables to values (both of scalar and of reference type), the *heap* h is a map from (reference, member variable)-pairs to values, S is a *call stack*, and ss is a *sequence of statements* to be executed.

A call stack is a sequence of tuples (ℓ_s, v_s, ss_s), with ℓ_s the *local variable map* of the caller, v_s the *variable* in which the return value of the callee should be stored, and ss_s the *sequence of statements* to be executed after return from the callee. In accordance, rule RETURN pops the top frame from the stack, updates v_s and ensures ss_s is executed next.

Rules VAR-ASSIGN and CONST-ASSIGN update v with the value of v' and c, respectively. Similarly, rule OP-ASSIGN applies op to the values of v_1 and v_2 and updates v with the result (recall that op only applies to scalars).

Rule MBR-ASSIGN-TO updates the value of the member v of the object pointed to by this with the value of v'. Conversely, rule MBR-ASSIGN-FROM updates the value of the variable v with the value of the member v'.

Rule NEW-ASSIGN creates a new instance of a class $class$ by identifying an unassigned reference ref on the heap. Given ref, all member variables $\text{mv}(class)$ of $class$ are allocated by setting them to an undefined value $\perp$ (i.e. $h[(ref, mv) \mapsto \perp]_{mv \in \text{mv}(class)}$), and v is updated with ref. No constructor arguments are passed

$$\frac{\ell' = \ell_s[v_s \mapsto \ell(v)]}{(\ell, h, (\ell_s, v_s, ss_s) : S, \mathsf{return}\ v; ss) \to_s (\ell', h, S, ss_s)} \text{ (RETURN)}$$

$$\frac{\ell' = \ell[v \mapsto \ell(v')]}{(\ell, h, S, v := v'; ss) \to_s (\ell', h, S, ss)} \text{ (VAR-ASSIGN)}$$

$$\frac{\ell' = \ell[v \mapsto c]}{(\ell, h, S, v := c; ss) \to_s (\ell', h, S, ss)} \text{ (CONST-ASSIGN)}$$

$$\frac{c = \ell(v_1)\ op\ \ell(v_2) \qquad \ell' = \ell[v \mapsto c]}{(\ell, h, S, v := v_1\ op\ v_2; ss) \to_s (\ell', h, S, ss)} \text{ (OP-ASSIGN)}$$

$$\frac{h' = h[(\mathsf{this}, v) \mapsto \ell(v')]}{(\ell, h, S, \mathsf{this}.v := v'; ss) \to_s (\ell, h', S, ss)} \text{ (MBR-ASSIGN-TO)}$$

$$\frac{\ell' = \ell[v \mapsto h(\mathsf{this}, v')]}{(\ell, h, S, v := \mathsf{this}.v'; ss) \to_s (\ell', h, S, ss)} \text{ (MBR-ASSIGN-FROM)}$$

$$\frac{\neg\exists v : (ref, v) \in \mathrm{dom}(h) \qquad \ell' = \ell[v \mapsto ref] \qquad h' = h[(ref, mv) \mapsto \bot]_{mv \in \mathrm{mv}(class)}}{(\ell, h, S, v := \mathsf{new}\ class; ss) \to_s (\ell', h', S, ss)} \text{ (NEW-ASSIGN)}$$

$$\frac{\ell_{fp} = \bot[\mathsf{this} \mapsto \ell(v')][fp_i \mapsto \ell(v_i)]_{1 \le i \le n} \qquad \ell' = \ell_{fp}[v_d \mapsto \bot]_{v_d \in \overline{v_d}}}{(\ell, h, S, v := v'.m(v_1, \ldots, v_n); ss) \to_s (\ell', h, (\ell, v, ss) : S, ss_m)} \text{ (METHOD-CALL)}$$

$$\frac{\ell(v)}{(\ell, h, S, \mathsf{if}\ (v)\ ss_t\ \mathsf{else}\ ss_f; ss) \to_s (\ell, h, S, ss_t; ss)} \text{ (IF-TRUE)}$$

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{if}\ (v)\ ss_t\ \mathsf{else}\ ss_f; ss) \to_s (\ell, h, S, ss_f; ss)} \text{ (IF-FALSE)}$$

$$\frac{\ell(v)}{(\ell, h, S, \mathsf{while}\ (v)\ ss_b; ss) \to_s (\ell, h, S, ss_b; \mathsf{while}\ (v)\ ss_b; ss)} \text{ (WHILE-TRUE)}$$

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{while}\ (v)\ ss_b; ss) \to_s (\ell, h, S, ss)} \text{ (WHILE-FALSE)}$$

Figure 3. Operational semantics

upon instance creation. Instead, we assume that instance creation is always followed by one or more method calls that set the members to properly defined values (making the exact value of $\bot$ irrelevant).

Assuming a method m is defined as

$$\mathrm{type}\ m(\mathrm{type}_1\ fp_1, \ldots, \mathrm{type}_n\ fp_n)\ \{\overline{v_d}\ \ ss_m\}\,,$$

rule METHOD-CALL creates a new local store starting from the empty map $\bot$ (not to be confused with the undefined value $\bot$). The rule sets this to the value of v', initializes each formal parameter fp_i with the value of v_i, and sets each local variable v_d to some undefined value $\bot$. Finally, a new stack frame is created for the caller and execution continues with the statements ss_m of the callee. As in the case of class instantiation, we assume that each local variable is assigned a well-defined value by the callee before making use of the stored value.

Rules IF-TRUE and IF-FALSE are standard; the value of v is retrieved and the correct branch is executed. Rules WHILE-TRUE and WHILE-FALSE are similar.

Machines Given a set of event names Evt, a set of classes $\mathcal{C}$, and the set of all possible values $Vals$ that can be assigned to variables, a *machine* m is defined as a tuple $(class_m, q_m, Q_m, T_m)$, where:

- $class_m \in \mathcal{C}$ is the *class* whose methods define the statements that need to be executed on entry to each state;
- q_m is the *initial state* of m and is represented by a method of $class_m$ that does *not* have any arguments;
- Q_m is the set of *non-initial* states of m, each of which is represented by method of $class_m$ that has a *single* argument;
- T_m is the *transition function*. Given a state $q \in Q_m$ and an *event queue* $E \in \mathsf{queue}(Evt \times Vals)$, the function finds the first event $evt(val)$ in E that m is willing to handle in state q. The function then yields (i) the next state q' as a function in q and evt, (ii) val, and (iii) an event queue E' identical to E but with $evt(val)$ removed.

Example 4.2. Continuing Example 4.1, we can define a machine that manages a list. For brevity, we omit $v := \ldots$ when calling void methods, and we similarly omit return statements from the bodies of these methods. null is a special constant, denoting a null reference.

```
class list_manager {
 elem list;

 void init() {
   this.list := null;
 }

 void get(ID payload) {
   elem tmp;
   tmp := this.list;
   send_payload eReply(tmp);
 }
}
```

```
void add(elem payload) {
  elem tmp;
  tmp := this.list;
  payload.set_next(tmp);
  this.list := payload;
}
```

The $init$ method defines the initial state of the machine. The other states of the machine are defined by the method get, which sends the whole list to the machine specified by $payload$ by means of the event $eReply$, and the method add, which adds $payload$ to the list.

Assuming the machine handles events $eAdd$ and $eGet$, we define the transition function as the total function that (a) transitions to add when $eAdd$ is at the head of the event queue and (b) transitions to get when $eGet$ is at the head of the queue.

Observe that the machine potentially suffers from a data race: a reference to the list is still held by the machine after being used as a payload in the send statement of the get method.

Systems The semantics of *systems* (or programs) composed of multiple machines is now defined in terms of transitions between *system configurations* (h, M), where h is a heap shared between the machines in the system, and where M is a map from (machine) identifiers ID to machine configurations.

A *machine configuration* is a tuple (m, q, E, ℓ, S, ss), with m a machine, q the current state of the machine, E an event queue, ℓ a variable store, S a call stack (as in the object-oriented language), and ss a sequence of statements that needs to be executed before the next event in the event queue can be handled.

Given a set of (machine) identifiers ID, an *initial system configuration* is any configuration (h, M) such that for each $i \in ID$ there exists a machine m with $M(i) = (m, q_m, \varepsilon, \ell_m, \varepsilon, ss_m)$, where ε represents both an empty event queue and empty stack, where $\ell_m = \bot[v_m \mapsto \bot]$ for some variable v_m and where ss_m is defined as $v_m := \mathsf{new}\ class_m; v_m.q();$. The machine definition m occurs in the tuple, as rule RECEIVE below needs access to the transition function. The sequence of statements ss_m initializes the machine by creating an instance of the appropriate class, retaining a reference in v_m, and by executing the method corresponding to the initial state. Once the statements in ss_m have been executed, the machine is ready to handle incoming events by invoking methods of v_m.

The transitions between system configurations are defined in Figure 4. Rule INTERNAL employs the rules from Figure 3 to

$$\frac{M(i) = (m, q, E, \ell, S, ss) \quad (\ell, h, S, ss) \rightarrow_s (\ell', h', S', ss') \quad M' = M[i \mapsto (m, q, E, \ell', S', ss')]}{(h, M) \rightarrow_t (h', M')} \text{ (INTERNAL)}$$

$$\frac{M(i) = (m, q, E, \ell, S, \mathsf{send}_{dst}\ evt(v); ss) \quad M_s = M[i \mapsto (m, q, E, \ell, S, ss)] \quad M_s(dst) = (m', q', E', \ell', S', ss') \quad M' = M_s[dst \mapsto (m', q', E' : evt(\ell(v)), \ell', S', ss')]}{(h, M) \rightarrow_t (h, M')} \text{ (SEND)}$$

$$\frac{M(i) = (m, q, E, \ell, S, \varepsilon) \qquad T_m(q, E) = (q', val, E') \quad M' = M[i \mapsto (m, q', E', \ell, S, v_m.q'(val))]}{(h, M) \rightarrow_t (h, M')} \text{ (RECEIVE)}$$

Figure 4. Transition rules

execute a statement for some i. The rule updates the shared heap and local variables, stack, and sequence of statements of i appropriately. Note that this rule requires ss to be non-empty.

Rule SEND appends the event being sent to the event queue of the appropriate machine configuration (depending on dst). Observe that care needs to be taken to update the machine configuration of i before appending the event: a machine i can send an event to itself, i.e. we can have $dst = i$.

A machine transitions to a new state once no statement remains to be executed. Rule RECEIVE takes care of this by employing the transition function of m and invoking the appropriate method obtained through this function. Observe that we use a variant of call statement in this case where we use a value instead of a variable as the argument. The rule is identical to rule METHOD-CALL when restricted to a single method argument, except that instead of assigning the result of a variable evaluation to *fp*, we assign val.

5. Checking for Data Races

Two transitions in a P# program are said to *race*, if they (i) originate from different machines instances, (ii) are not separated by any other transition, and (iii) both access the same field of an object with at least one of the accesses being a write. Formally, (iii) requires the common field to be accessed using rules MBR-ASSIGN-TO and MBR-ASSIGN-FROM, with at least one being MBR-ASSIGN-TO.

If a P# program is free from data races then we can reason about the possible behaviors of the program by assuming that machines interleave with event-level granularity; there is no need to consider interleavings between individual heap access operations. Furthermore, the ability to detect or prove absence of races is valuable since races between machines are typically unintentional and erroneous, because they break the event-based communication paradigm of the language.

To detect possible data races, or to show their absence, we use an *ownership*-based static analysis. Objects are owned by machines, and ownership of an object o is transferred from a machine m to a machine m' upon m sending an event to m' with a (direct or indirect) reference to o. Once ownership has been transferred, m is no longer allowed to access o, as this would violate the fact that m' has ownership of o. If ownership is respected then data races cannot occur between machines. Violations of ownership *may* indicate that data races do occur.

To make the ownership analysis machine- and method-modular (to ensure scalability), we define a *give-up set* for each method m. For each formal parameter *fp* of m, if there exists an object o such that (i) o is *reachable* from *fp* (either directly or by transitively following references) and (ii) ownership of o is transferred by m, then *fp* is in the give-up set for m.

Once the give-up set for each method has been computed, data race-freedom can be established by considering scenarios where:

1. a formal parameter *fp* is in the give-up set for a method m';
2. a method m calls m', passing a variable v for *fp*;
3. after calling m', m goes on to access a variable v'.

It suffices to show that in each such scenario, no object o exists that can be accessed (directly or indirectly) through both v and v'.

The central theorem of this section is:

Theorem 5.1. *If each state of a machine m respects the ownership of each method invocation and send-statement (as defined in Section 5.3), then no data race can occur.*

Assumptions Below, we represent each method as a single-entry, single-exit *control flow graph* (CFG), where each CFG node consists of a *single* statement. The entry and exit nodes are denoted Entry and Exit. Employing CFGs allows us to treat conditionals, loops and sequences of statements in a uniform manner

We assume that formal parameters of methods cannot be assigned to, and that the variables passed as actual parameters during method calls are pairwise distinct (avoiding implicit creation of aliases through parameter passing). These assumptions simplify the presentation and can be satisfied through preprocessing.

Because scalar variables are passed by value, our race analysis only needs to be concerned with reference variables. Henceforth, unless stated otherwise, we thus use *variable* to only mean a variable of reference type.

5.1 Heap Overlap

Our analysis depends on knowing whether the same object is reachable from multiple variables, i.e. our analysis depends on an analysis that soundly resolves *heap overlap* between variables at different CFG nodes. Let us assume we have a heap overlap analysis at our disposal, in the form of a predicate $\mathrm{may\_overlap}$ such that for a method m:

- $\mathrm{may\_overlap}^{(N,v)}_{m,\mathrm{in}}(N', v')$ holds if there may exist a heap object o that is reachable from v' on entry to N' that is also reachable from v on entry to N.
- $\mathrm{may\_overlap}^{(N,v)}_{m,\mathrm{out}}(N', v')$ holds if there may exist a heap object o that is reachable from v' on exit from N' that is also reachable from v on entry to N.

Observe in both cases that we consider the heap objects reachable from v on *entry* to N. The reason for this is that we are interested in objects that are reachable immediately before they are given up.

Heap Overlap in Practice In practice the above predicates are implemented through an inter-procedural *taint tracking analysis*. The analysis is flow- and context-sensitive.

Given a tainted variable v, the taint tracking analysis *taints* a variable v' if there *may* exist some object that is reachable from v' that is also reachable from v. The output of the analysis consists of a function that summarizes how tainting propagates throughout methods. Given a method m, a local variable or formal parameter v of m and a node N of m, the analysis yields a function $\mathrm{tainted}^{(v,N)}_m$ that maps nodes N' of m to variables v' such that if v is assumed to be tainted on entry to N, then v' is tainted on exit from N'. In other words, $v' \in \mathrm{tainted}^{(v,N)}_m(N')$ implies that there may exist an object o such that v' can reach o on exit from N' if v can reach o on entry to N.

Our summary function is member variable *insensitive*, i.e. when we note in our analysis that a member of an object should become

tainted, we taint the whole object instead. We could improve the precision of our analysis by keeping track of individual member variables. However, in the overwhelming majority of cases tainting full objects suffices (see also Section 7.2.1).

The may_overlap predicate can now be defined as follows:

- $\text{may\_overlap}^{(N,v)}_{m,\text{in}}(N', v') \triangleq v' \in \bigcup_{P \in \text{pred}(N')} \text{tainted}^{(v,N)}_m(P)$
- $\text{may\_overlap}^{(N,v)}_{m,\text{out}}(N', v') \triangleq v' \in \text{tainted}^{(v,N)}_m(N')$

where $\text{pred}(N')$ denotes the set of predecessors nodes of N'.

Example 5.2. In Example 4.1, only this can become tainted in get_val and set_val. Moreover, it can only become tainted if it was tainted in the first place, as all other variables in these methods are scalar. For get_next, we have $\text{tainted}^{(ret,\mathsf{Exit})}_{get\_next}(\mathsf{Entry}) = \{\mathsf{this}\}$ (ret is not included in the set, as its value is overwritten in the second line of the method). The summary for get_next is similar.

Heap overlap is closely related to the well-studied problem of heap reachability, i.e. the problem of establishing whether a certain heap object o is reachable from a variable v. Hence, instead of a taint analysis, alternatively we could have tried to adapt [3], which is path-, flow-, and context-sensitive heap reachability analysis that currently seems to be among the most accurate and scalable. The analysis of [3] is complex, which makes it hard to implement. For this reason we opted instead to develop a much simpler taint analysis, which in our experience is accurate enough for our purposes (see Section 7.2.1).

5.2 Gives up Analysis

We now define for which formal parameters of a method m ownership is *given up* by that method; we denote the set of these parameters by $\text{gives\_up}(m)$. Initially setting $\text{gives\_up}(m) = \emptyset$, the give-up sets are computed as follows:

1: **repeat**
2: **for all** m **do**
3: $\text{gives\_up}(m) := \bigcup_{N \in m} \text{gives\_up}^{fp}_m(N)$
4: **end for**
5: **until** gives_up no longer changes

The function $\text{gives\_up}^{fp}_m(N)$ is defined in Figure 5, where $\text{fp}(m)$ denotes the set of formal parameters of m. As can be seen, ownership of a formal parameter *fp* is given up either when

1. some object reachable from *fp* on entry to m is also reachable from the variable passed as value argument to a send statement of m, or when
2. some object reachable from *fp* on entry to m is also reachable from the ith argument of a method m' invoked by m, where m' gives up its ith argument.

The function is formulated as a fixed-point computation, because methods may be mutually recursive. Termination occurs, as the number of methods and formal parameters is finite. The function identifies all formal parameters of a method m from which heap objects may be reachable that are also reachable from a variable occurring in a send statement.

Example 5.3. For the methods in Examples 4.1 and 4.2, no formal parameters are given up. However, if we would let the add method of $list\_manager$ forward $payload$ instead of adding it to the list, i.e. when we would replace the body by

$$\mathsf{send}_{dst}\ eAdd(payload);$$

then add would give up $payload$.

5.3 Respects Ownership Analysis

We now define the conditions under which the nodes of a method respect ownership of objects. The interesting cases are when a node represents a send operation or a method call, because these statements have the potential to erroneously transfer ownership between machines. All other types of nodes trivially respect ownership.

Suppose that N is a node of a method m, and that N represents either a method call or a send operation. If N is a method call, of the form $v := v'.m'(v_1, \ldots, v_n)$, then N *respects ownership* if ownership is respected at N for each actual parameter v_i such that the corresponding formal parameter fp_i of m' is in the give-up set fo m', i.e. $fp_i \in \text{gives\_up}(m')$. If N is a send operation, of the form $\mathsf{send}_{dst}\ evt(v)$, then N *respects ownership* if ownership is respected at N for v.

We now describe the conditions under which ownership is respected by a variable in a node. Let N be a node in method m, and let $\text{vars}(N)$ denote the set of variables occurring in N. For a variable w we say that ownership is respected for w at N if, for every node N' of m, the following conditions hold:

1. If there is a path from Entry to N through N', then
$$\neg\text{may\_overlap}^{(N,w)}_{m,\text{out}}(N', \mathsf{this})\,.$$
2. If $N' = N$, then $w \neq \mathsf{this}$ and
$$\{v \in \text{vars}(N') \mid \text{may\_overlap}^{(N,w)}_{m,\text{in}}(N', v)\} = \{w\}\,.$$
3. If there is a path from N to Exit through N', then
$$\{v \in \text{vars}(N') \mid \text{may\_overlap}^{(N,w)}_{m,\text{in}}(N', v)\} = \emptyset\,.$$

We discuss the three cases in turn. In the first case, we check whether some object reachable from this is also reachable from w. If this is the case, we may be able to access a given up object in a later machine state (through a field of the machine).

In the second case, we check whether w is equal this. If this is the case, then again we may be able to access a given up object in a later machine state. We also check that no variable other than w has access to an object reachable from w. This takes care of potential aliasing: if an object may also be accessed through other variables, then the method we invoke may be able to access the objects given up after it has given them up.

In the third case, we check whether any variable that may be used subsequent to giving up w can reach an object that was also reachable through w at the point at which w was given up. We forbid the use of any such variable.

Observe in the first case, that if some variable v is given up in a node N' on a path from Entry to N and if an object exists that is accessible through both v and w, then by symmetry an error will be flagged through the third case, as N is on a path from N' to Exit.

Example 5.4. For the method get in Example 4.2, an ownership violation will be flagged, as any object accessible through tmp in the send statement is accessible through this. This violates our first condition above.

The correctness of Theorem 5.1 now follows by the above observations regarding our respects ownership definition and gives up analysis. Observe that implementations of may_overlap do not need to cater for aliasing of formal parameters, because (1) our top-level methods representing states only have a single argument, which trivially implies that no aliasing between arguments occurs, and (2) the requirement on variables in the second case above, where we recall that all variables passed to a method are assumed to be pairwise distinct.

$$\text{gives\_up}_m^{fp}(N) = \begin{cases} \{w \in \text{fp}(m) \mid \text{may\_overlap}_{m,\text{out}}^{(N,v)}(\mathsf{Entry}, w)\} & \text{if } N = \mathsf{send}_{dst}\ evt(v) \\ \cup_{1 \le i \le n}\{w \in \text{fp}(m) \mid fp_i \in \text{gives\_up}(m') \wedge \text{may\_overlap}_{m,\text{out}}^{(N,v_i)}(\mathsf{Entry}, w)\} & \text{if } N = v := v'.m'(v_1, \ldots, v_n) \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 5. Computing the formal parameters given up by m

5.4 Extensions and Implementation

Cross-State Analysis Most false-positives in our experiments (see Section 7.2.1) originate from the payload of an event being constructed in one machine state and only being sent from a later state. This is achieved by temporarily storing the payload in a machine field. Sending the payload will lead to an ownership violation, as the sent objects may still be accessible through this. To suppress these false-positives, we run a *cross-state analysis (xSA)* upon detection of an ownership violation. The analysis is based on the observation that each machine can be seen as a CFG, where at the end of each method representing a state we non-deterministically call one of the methods representing an immediate successor state. Our analysis can now be performed on this overarching CFG once we lift all machine fields to be parameters of the methods. As payloads are now passed as parameters, the false-positives no longer occur. That xSA is sound is an immediate consequence of the soundness of the ownership analysis.

Example 5.5. We can "repair" Example 4.2 by adding the statement this.$list$:= null; after the send statement, i.e. we *reset* the $list$ member. Once we repaired our method, we need xSA to determine race-freedom, as $list$ is a member variable. Race-freedom easily follows once we lift the member variable to be a parameter of all methods in $list\_manager$.

Implementation We implemented our analysis on top of Microsoft's Roslyn[1] compiler framework. As Roslyn does not provide direct access to the underlying CFG of each method, we construct our own by querying Roslyn's abstract syntax tree (AST) interface. Once the CFG of each method has been constructed, we perform our analysis. However, instead of computing may_overlap up front, we compute the function lazily whilst performing the gives-up and respect ownership analyses. This is more efficient, as it avoids computing summaries for variables and nodes irrelevant to the analysis. Calls to libraries of which the source code is not available are handled in a conservative manner by assuming that each heap object reachable before the call is reachable from all variables involved in the call once the call returns.

Exception handling (omitted from our formal treatment) is dealt with by adding appropriate edges to the CFG. Inheritance is handled by unioning the summaries of all possible methods that can be invoked at a call site. We disallow multi-threading constructs and reflection, and assume that any imported external libraries do not use these either.

5.5 Comparison with SOTER

Although both SOTER [20] and our data race analysis perform a static ownership-based analysis, there are some fundamental differences. SOTER builds upon a field-sensitive points-to analysis. This analysis is non-modular and does not leverage an understanding of the underlying (actor) framework. As a consequence, SOTER needs to sacrifice precision to achieve scalability. Our analysis achieves scalability *without* sacrificing precision exactly by leveraging the semantics of the P# framework.

[1] `https://github.com/dotnet/roslyn`

6. Execution and Testing

The P# runtime can be used for both execution and testing. First, we show how the runtime executes a P# program; then, we discuss how it can find bugs (e.g. assertion violations and uncaught exceptions) using two custom schedulers.

6.1 The P# Runtime

The P# runtime implements the logic for executing P# programs, such as creating new machine instances, sending events, enqueueing received events, and handling dequeued events with user-defined state transitions and actions.

The P# runtime library defines the abstract classes `Machine`, `State` and `Event`. These classes need to be inherited to define machines, machine states, and events, respectively. All defined machine types must be *registered* with the runtime upon startup of the runtime; no new machine types may be defined during execution. This ensures we can easily detect the machines we need to statically analyze, which might be very hard or impossible otherwise.

Program Startup and Machine Initialization The runtime preprocesses each registered machine to build a machine-specific map from states to state transitions and action bindings. Upon startup, the runtime creates a single instance of the machine class annotated with the `Main` attribute. The runtime enforces a single main machine class, but other instances may be created during execution.

The constructor of each machine initializes the data private to the machine (e.g. the event queue), after which control is passed to the `OnEntry()` method of the initial state of the machine. This method is the entry point of the initial state and can perform user-defined operations, such as creating other machine instances and sending events. When a new event is sent, the runtime is responsible for enqueueing it in the event queue of the target machine.

Event Handling After executing the `OnEntry()` method of the initial state of a machine, the runtime invokes the event handler of the machine, which runs concurrently with the runtime and other handlers. The handler tries to dequeue an event from the event queue (implemented using a thread-safe blocking queue) and handle it appropriately. After the event has been handled, the handler attempts to dequeue the next event. If the queue is empty, the handler blocks until an event arrives or until the runtime terminates (e.g. because of an error being detected).

The runtime exits and reports an error if (i) an event can be handled in more than one way in the same state, (ii) an event cannot be handled in a state and (iii) an uncaught exception is thrown while an event handler executes.

6.2 Bug-Finding Mode

P# programs may contain concurrency bugs due to their asynchronous nature. Such bugs may only occur when machines are scheduled in a particular order. Inspired by the success of systematic concurrency testing (SCT) tools [9, 12, 19], we designed a *bug-finding mode* for the runtime, in which execution is serialized and the schedule is *controlled*. In this mode, the runtime repeatedly executes a program from start to completion, each time exploring a (potentially) different schedule. Our testing approach is fully automatic, has no false-positives and can reproduce found bugs by replaying buggy schedules. However, it does require a closed environment,

i.e. when testing we need to define and instantiate additional machines that model the environment. Like any real environment, the additional machines may be non-deterministic in nature.

When performing dynamic analysis of concurrent software, it is necessary to explore the interleavings of all *visible* operations (e.g. shared memory accesses and locks) to find all safety property violations (e.g. uncaught exceptions and deadlocks) [12]. However, if a program is race-free, only synchronizing operations need to be treated as visible, which greatly reduces the exploration space. Existing SCT tools (e.g. CHESS [19]) exploit this by only exploring interleavings of synchronizing operations while running with a race detector. If no races are found and all interleavings are explored, then no bugs were missed due to races. Otherwise, it is necessary to fallback to interleaving all visible operations.[2]

P# programs have several benefits that make them well-suited to systematic testing. First, we can verify race-freedom with our sound static analysis (see Section 5). This avoids the overhead of testing with a race detector and ensures that we only have to explore interleavings of synchronizing operations. Second, since the only synchronizing operations are the *send*, *receive* and *create-machine* methods (implemented in the runtime), it is straightforward to build a systematic testing runtime. This is in contrast to most SCT tools, which use dynamic instrumentation to intercept memory accesses and synchronizing operations [19]. In bug-finding mode, the send and create-machine methods call the runtime method `Schedule`, which blocks the current thread and releases another thread. As observed in previous work on P [6], it is not necessary to insert a scheduling point before receive operations, thus achieving a simple form of partial-order reduction [11].

We have implemented a *depth-first-search (DFS)* and a *random* scheduler (both embedded in the P# runtime). The DFS scheduler explores the schedule-tree in a depth-first manner. Each node is a schedule prefix and the branches are the enabled machines in the program state reached by the schedule prefix. The DFS scheduler is *systematic*: it explores a different schedule each time, ensuring exhaustive execution of all schedules (given enough time and resources, and assuming an acyclic state-space). A limitation of this type of systematic scheduling is that random choices made by machines must be fixed or explored systematically as well, which is at odds with the non-determinism of machines modeling the environment. In contrast, the random scheduler chooses a random machine to execute after each send and does not keep track of already explored schedules. Thus, random machines choices do not need to be controlled.

Finally, we designed the bug-finding mode to enable easy reproduction of bugs: after a bug is found, the runtime can generate a trace that represents the buggy schedule.

7. Evaluation

We report our experience of applying P# and its family of tools to a real asynchronous system used inside Microsoft. We then describe an evaluation of our analysis and testing approach on 12 P# implementations of well-known distributed algorithms.

7.1 Case Study

We used P# to model, port and test a large asynchronous system from Microsoft, used for rapid development of distributed services. We refer to this proprietary system as AsyncSystem. The system has an architecture similar to the example system from Section 3: a dispatcher and a library that exposes a set of abstract APIs. The programmer can inherit these abstract APIs to build a service. The dispatcher will take care of distributing the service. The programmer only needs to focus on developing the service logic.

Testing services in the original system was extremely challenging, as asynchronous code was mixed with the service logic. Stress testing was unable to catch bugs due to the asynchronous behavior of the system. We addressed this challenge in three steps. First, because our goal was to find bugs in the services and not the dispatcher, we modeled the latter using P#. This model captures the asynchrony internal to the dispatcher. Second, we ported the AsyncSystem library as a set of abstract P# machines, which expose the same abstract API actions as the original library. Third, we developed a P# program for each available service. The service machines inherit all transitions and action bindings from the library machines. Thus, the programmer only needs to override the exposed P# actions and implement *sequential* service logic. Table 1 shows combined statistics for the P# port of the AsyncSystem dispatcher, library and a single service (denoted AsyncSystem).

The process of porting to P#, and using our static analysis and testing framework, revealed five bugs in the original AsyncSystem. We found two of these early on during the porting process: applying the random P# scheduler to the partially ported system exposed two issues with the use of asynchrony in the original code still being used. These errors were eliminated by porting the affected parts of the system to P#. The remaining three bugs were found during analysis and testing once the porting of the system was complete.

We emphasize two key points. First, the identified bugs were indeed too hard to find using stress testing. In fact, they survived across *several* releases of the original AsyncSystem. Second, the P# user-services can execute on the actual dispatcher without performance degradation (although we leave such evaluation for future work). At the same time, linking these programs against the P# model of the dispatcher (thus creating a *pure* P# program) can efficiently exploit our analysis and testing techniques.

7.2 Experimental Results

Benchmarks We evaluate our methodology against 12 P# implementations of well-known distributed algorithms, which we separated in two benchmark suites: *SOTER–P#*, which includes P# ports of the four worst-performing Java actor programs from [20], and *PSharpBench*, which includes:

- BoundedAsync, a generic scheduler communicating with a number of processes under a predefined bound;
- German's cache coherence protocol [10];
- Lamport's Paxos consensus protocol [16];
- the two-phase distributed commit protocol [13];
- Chord [24], a distributed hash-table used for creating peer-to-peer lookup services;
- MultiPaxos [5], an advanced version of the Paxos protocol;
- Raft [22], a consensus protocol for managing a replicating log;
- the Chain Replication fault-tolerance protocol [26].

All our benchmarks are available online.[3]

Table 1 presents program statistics for the benchmark suites. The BoundedAsync, German, BasicPaxos, 2PhaseCommit, MultiPaxos and ChReplication benchmarks were ported from open source P implementations [8]. This process required three days of work (including understanding the protocols and debugging). The Chord and Raft protocols were implemented from scratch in two days using only the original papers as a reference. Finally, porting the SOTER benchmarks to P# required a full day of work.

[2] In the presence of data races and relaxed memory, even considering all interleavings of shared memory accesses may be insufficient to find all bugs.

[3] `http://multicore.doc.ic.ac.uk/tools/PSharp/PLDI15/`

						Non-racy versions				Racy versions	
							# False positives				
	Benchmarks	LoC	#M	#ST	#AB	Time (s)	(No xSA)	(xSA)	Verified?	Time (s)	Found all data races?
	AsyncSystem	5,786	14	82	51	14.883	6	2	✗	-	-
PSharpBench	BoundedAsync	416	2	8	4	4.988	1	✗	✓	4.927	✓
PSharpBench	German	718	3	35	6	4.912	✗	✗	✓	4.973	✓
PSharpBench	BasicPaxos	865	5	16	4	5.140	2	✗	✓	5.156	✓
PSharpBench	2PhaseCommit	983	6	17	12	5.008	1	✗	✓	5.466	✓
PSharpBench	Chord	1,005	3	11	17	5.212	✗	✗	✓	5.702	✓
PSharpBench	MultiPaxos	1,166	7	20	11	5.232	10	5	✗	5.205	✓
PSharpBench	Raft	1,249	4	12	33	5.118	✗	✗	✓	5.132	✓
PSharpBench	ChReplication	2,004	10	38	27	5.421	4	✗	✓	5.464	✓
SOTER–P#	Leader	275	2	3	2	4.774	✗	✗	✓	-	-
SOTER–P#	Pi	330	3	3	4	4.851	✗	✗	✓	-	-
SOTER–P#	Chameneos	536	2	4	13	4.988	✗	✗	✓	-	-
SOTER–P#	Swordfish	2,286	6	66	25	5.362	✗	✗	✓	-	-

Table 1. Program statistics and results of applying the P# static analyzer. The PSharpBench suite has both non-racy and racy versions. The reported program statistics are for the non-racy versions of the benchmarks: lines of P# code (LoC); number of machines (#M); number of state transitions (#ST); and number of action bindings (#AB). All reported execution times are in seconds and averages of 10 runs.

		CHESS DFS				P# DFS scheduler			P# random scheduler			
			RD-on	RD-off								
Benchmarks	#T	#SP	#Sch/sec	#Sch/sec	Bug found?	#SP	#Sch/sec	Bug found?	#SP	%Buggy	#Sch/sec	Bug found?
BoundedAsync	5	1127	6.73	33.32	✗	198	216.73	✗	232	6%	201.01	✓
German	6	156	18.19	88.94	✗	28	526.59	✗	28	22%	52.04	✓
BasicPaxos	10	1009	5.42	33.91	✗	186	179.19	✗	92	83%	201.62	✓
2PhaseCommit	8	281	15.92	66.16	✗	52	425.41	✗	37	3%	485.48	✓
Chord	11	1312	-	-	✓	222	-	✓	312	35%	149.57	✓
MultiPaxos	14	184	-	-	✓	27	-	✓	28	89%	3.12	✓
Raft	14	336,616	0.02	0.15	✗	61,197	2.11	✗	61,322	2%	1.47	✓
ChReplication	11	1416	-	-	✓	238	-	✓	257	100%	150.77	✓

Table 2. Results of applying CHESS and the embedded P# schedulers to the buggy PSharpBench programs for at most 10,000 executions within a 5 minute time limit. We ran CHESS in two modes: with its data race detection on (RD-on) and with its data race detection off (RD-off). We report: number of threads per execution (#T); number of scheduling points (#SP); number of explored schedules per second (#Sch/sec); and percentage of buggy schedules (%Buggy)

Our benchmarks are shared-state implementations of the original distributed algorithms. To model the environment (e.g. failures), we developed additional non-deterministic P# machines. Thus, these asynchronous P# programs are single-box simulations of distributed algorithms; this matches the approach taken in evaluation of previous work (P and SOTER).

For the buggy P# benchmarks (see Table 2) we generally aimed to gather hard to find bugs, i.e. bugs requiring a large number of schedules to occur. Most of the bugs were real mistakes that we made during the implementation of the protocols (e.g. forgetting to properly handle an event in some state). If that was not a viable option, as in the case of BasicPaxos and Multipaxos, we injected an artificial bug. Any benchmarks that required input were given a fixed input that was constant across executions. For the P and SOTER benchmarks, we used the inputs from the original implementations. For Raft and Chord, our new benchmarks, the input is irrelevant for finding bugs.

Experimental Setup We performed all experiments on a 1.9GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 8.1 Pro 64-bit.

7.2.1 Static Data Race Analysis

We evaluate the soundness, precision and scalability of our static analysis with respect to our benchmark sets.

Soundness and Precision Table 1 shows that the P# static analyzer manages to find all data races in racy versions of the PSharpBench programs. This is unsurprising, because the analyzer is sound. To confirm the results, we manually inspected the error output of the tool, and we indeed found that the tool reported all existing data races. We have also developed a regression suite that contains many complex cases of racy and non-racy P# programs (e.g. with loops, inheritance and aliasing). We have extensively exercised our analyzer on this suite to increase confidence in its soundness and fine-tune its precision.

To evaluate precision, we ran the tool on non-racy versions of the PSharpBench programs. We first ran the analyzer without *cross-state analysis (xSA)* (see Section 5.4), and then with xSA. Our tool failed to verify six benchmarks when xSA was disabled. All reported false positives were related to either (a) giving up ownership of a machine field or (b) storing a reference to be given up in such a field. With xSA enabled, the tool managed to verify all but MultiPaxos and AsyncSystem, as we discuss next. This shows that xSA is useful.

Although xSA discarded 17 out of 24 false positives, it did not manage to discard the remaining seven. In each of these seven cases a machine M_1 in a state S_1 stores a reference in a machine field a and then sends the reference to a machine M_2. In a later state S_2, M_1 sends the contents of a to a machine M_3 without first updating a to point to a new memory location. This could potentially lead to

a data race as this gives each of M_1, M_2 and M_3 access to the same memory location. However, when manually inspecting the source code, we found that the sent reference is only ever read. Hence, we believe that we can suppress these false positives by introducing a *read only* analysis.

Previous work on static data race analysis for message-passing programs includes the SOTER analyzer [20] for actor programs in Java. We ported four of the worst-performing benchmarks from [20] to P# (denoted by SOTER–P# in Table 1). While our analyzer verifies all four benchmarks, SOTER reports a number of false positives (e.g. 70 false positives in Swordfish). SOTER differs from our work in that it uses a fundamentally different static analysis that tracks ownership not just through the user defined code, but also through the actor framework (see also Section 5.5).

Performance and Scalability Table 1 also shows the execution time for the application of the P# static analyzer to our benchmarks. The reported times include xSA. The tool managed to analyze each benchmark in less than 6 seconds, except for AsyncSystem which was analyzed within 15 seconds. These results show that our analysis scales well across P# programs of varying size and complexity.

7.2.2 Finding Bugs in P# Programs

In this section, we explain that the CHESS systematic concurrency testing tool [19] works with minimal effort on our buggy P# benchmarks, but that its data race detector adds unnecessary overhead. We then show that the P# schedulers execute faster than CHESS. Finally, we show that our random scheduler is more effective at finding bugs than the CHESS and P# DFS schedulers.

CHESS and Data Races Because P# uses C#, we were able to apply CHESS to our benchmarks with minimal effort. We used our own implementation of a FIFO queue, as CHESS does not recognize the .NET 4.0 lock-free concurrency operations used by the P# runtime, i.e. a thread-safe blocking queue.

We ran CHESS on each benchmark with and without its data race detector to measure overhead (expressed in the number of schedules explored per second). We configured CHESS to perform a straightforward depth-first search of the schedule-space for at most 10,000 schedules or 5 minutes (whichever bound was reached first). Note that CHESS terminates when a bug is found: Chord, MultiPaxos and ChReplication contain bugs that were detected on the first schedule, which means that we cannot sensibly report the number of schedules per second for these benchmarks.

Table 2 shows that CHESS runs between 4 and 7.5 times faster (in terms of schedules per second) when its data race detection facilities are disabled. Since our static analysis has determined the absence of data races, we can indeed run CHESS *without* data race detection while being confident that no bugs are missed due to data races; thus, we can benefit from the increase in speed. With data race detection enabled, CHESS did not find any data races; this provides additional confidence that our static data race analysis is sound.

CHESS vs. P# DFS We ran each benchmark using the P# DFS scheduler to compare execution speed with CHESS. Exploration stopped once a bug was found or when the time/schedule limit was reached, as in the case of CHESS. Table 2 shows that the P# DFS scheduler was 7.6× faster on average (in terms of schedules per second) than CHESS with data race detection turned off.

We believe that CHESS is slower for two reasons. First, CHESS uses dynamic instrumentation (adding overhead), whereas the P# scheduler is embedded in the runtime. Second, CHESS inserts scheduling points before several synchronization operations (e.g. runtime locks), whereas the P# scheduler only needs to schedule before send and create-machine operations, which greatly reduces the schedule space.

P# Random Scheduler We also tested our buggy benchmarks using the P# random scheduler. Unlike previous tests, we continue to explore schedules after a bug is found to compute the probability that each bug occurs. Table 2 shows that the random scheduler was able to find all bugs with various probabilities. The ChReplication bug occurred 100% of the time; we believe that this bug requires only one of several random binary choices made by the non-deterministic environment in order to occur. We found two bugs in the German benchmark: an assertion violation and a livelock. Detecting livelocks in a dynamic setting is usually non-trivial. However, in this case, the livelock occurred when all but one machine had terminated; the final machine was stuck in an infinite loop continuously sending an event to itself. We easily detected the livelock after forcing the scheduler to terminate and inspecting the error output. We then imposed a depth-bound to automatically detect the livelock and ensure termination. The livelock accounts for 1415 of the 2299 buggy schedules. These deep schedules are expensive, hence the lower number of schedules per second compared to the DFS tests (which did not find the livelocks). Finally, our evaluation confirms previous claims that random concurrency testing is effective at finding bugs [4, 25].

8. Conclusion

We have presented the P# language for high-reliability asynchronous programming, co-designed with static ownership-based data race analysis and systematic concurrency testing support. Our case study involving the use of P# inside Microsoft as target for porting and testing an industrial-scale asynchronous system, and our efforts in implementing a number of distributed protocols, shows the benefits of language and analysis co-design.

In future work we plan to improve the precision of our data race analysis. We found that a reoccurring pattern of false positives involved sending the same data to multiple machines were the receivers would only read the data. We could address such false positives by introducing a read only analysis.

Acknowledgments

We would like to thank Shaz Qadeer and Ankush Desai for answering our queries regarding the P language and the P benchmarks. We also acknowledge Sophia Drossopoulou, Sriram Rajamani, Kaushik Rajan, Ganesan Ramalingam and the members of the Multicore Programming Group at Imperial for their feedback during various stages of this work. Finally, we would like to thank the anonymous reviewers and artifact evaluation committee for their comments.

This work was initiated while Pantazis Deligiannis was a research intern with Microsoft Research India. The work was further supported by a gift from Intel Corporation, EU FP7 STREP project CARP, and an EPSRC-funded PhD studentship.

References

[1] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.

[2] Apple Inc. Grand Central Dispatch (GCD) reference, accessed November 2014. URL `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`.

[3] S. Blackshear, B. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–286. ACM, 2013.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support*

for Programming Languages and Operating Systems, pages 167–178. ACM, 2010.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM, 2007.

[6] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–332. ACM, 2013.

[7] A. Desai, P. Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 709–725. ACM, 2014.

[8] A. Desai, S. Qadeer, and S. Seshia. Systematic testing of asynchronous reactive systems. Technical Report MSR-TR-2015-25, Microsoft Research, 2015.

[9] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM, 2011.

[10] S. German. Tutorial on verification of distributed cache memory protocols. In *Formal Methods in Computer Aided Design*, 2004.

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

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM, 1997.

[13] J. N. Gray. *Notes on Data Base Operating Systems*. Springer, 1978.

[14] O. Gruber and F. Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 281–301. Springer, 2013.

[15] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer, 2010.

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

[17] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.

[18] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 227–242. ACM, 2009.

[19] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 267–280. USENIX Association, 2008.

[20] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 81–90. ACM, 2011.

[21] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Inc, 2008.

[22] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

[23] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001.

[25] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 15–28. ACM, 2014.

[26] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–104. USENIX Association, 2004.

[27] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[28] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall International, 2nd edition, 1996.