Checking Compatibility of a Producer and a Consumer

Evan Driscoll,[†] Amanda Burton,[†] and Thomas Reps[†,‡]

[†]University of Wisconsin
[‡]GrammaTech, Inc.
{driscoll, burtona, reps}@cs.wisc.edu

Abstract

This paper addresses the problem of identifying incompatibilities between two programs that operate in a producer/consumer relationship. It describes the techniques that are incorporated in a tool called PCCA (**P**roducer-**C**onsumer **C**onformance **A**nalyzer), which attempts to (i) determine whether the consumer is prepared to accept all messages that the producer can emit, or (ii) find a counter-example: a message that the producer can emit and the consumer will reject.

1. Introduction

Complex systems today are made up of many communicating components. For instance, a modern fuel-injected engine has a number of sensors that send their current measurements to the engine-control unit, which decides, for instance, what the optimum fuel-air mixture should be. It emits messages to other components, such as the fuel pumps and fuel injectors, to enact its decisions.

In such systems, it is vitally important to ensure that the messages that one component sends to another are understood by the receiving component, otherwise runtime errors will ensue. Send/receive incompatibilities can often drive up the cost of developing a system because different components of a system are often developed by different development teams or different subcontractors, and thus compatibility problems may not be detected until integration time. (The cost of fixing errors found late in the development process is usually much higher than that of errors found earlier.)

As a first step toward addressing the general component-compatibility problem, this paper describes a technique for automatically determining if two components that operate in a producer-consumer relationship are compatible. In particular, the paper addresses the following problem:

> Given two programs that operate in a producer/consumer relationship, (i) determine whether the consumer is prepared to accept (and process in some way) all messages that the producer can emit, or (ii) find a counter-example: a message that the producer can emit and the consumer will reject.

Because it is necessary to use approximations to the producer's output language and the consumer's input language, the method we have developed has some limitations. In both cases, the approximations are over-approximations: i.e., the approximation of the producer's language may say that the producer can emit a certain word that can never actually be emitted by the producer; likewise, the approximation of the consumer's language may say that the consumer can accept a certain word that can never actually be accepted by the consumer. Consequently, the compatibility-checking method can have both false positives and false negatives. For this reason, our technique is really a heuristic for bug-finding, not a technique for verifying producer-consumer compatibility.

We have implemented the method in a tool called PCCA (for **P**roducer-**C**onsumer **C**onformance **A**nalyzer). Given the two source programs, along with information about which functions perform I/O (see §4.2), PCCA infers a description of the language that the producer generates and a description of the language that the consumer expects, and determines whether the former is a subset of the latter.

PCCA starts out by producing a nested word automaton (NWA) [1] P for the producer, which accepts an over-approximation of language that the producer emits. Similarly, PCCA produces an NWA C for the consumer, which accepts an over-approximation of language that the consumer accepts. In each of the nested-word languages $L(P)$ and $L(C)$, internal calls and returns in the corresponding program are made manifest in the words of the language. The goal (modulo some technicalities having to do with differences in the call and return structures of the two languages) is to determine whether $L(P) \subseteq L(C)$.

Next, PCCA "enriches" the consumer's NWA C—and hence its nested-word language $L(C)$—so that differences in the call/return structures of the producer and the consumer do not preclude answering the language-containment question. (This step is explained in more detail in §3.2.1.)

PCCA then performs a set-difference operation to compute $L(P) \setminus L(\text{Enrich}(C))$ by complementing the NWA Enrich(C), intersecting the result with NWA P. Finally, it tests whether the language of the NWA produced by the intersection is empty. If so, then $L(P) \subseteq L(\text{Enrich}(C))$; if not, then there is a counter-example to $L(P) \subseteq L(\text{Enrich}(C))$.

The techniques used in PCCA are "transport-agnostic" as long as the producer and the consumer use a stream-like interface to communicate. That is, PCCA can analyze a pair of programs—or even the *same* program as both producer and consumer—that communicate by sharing files, by sending information over sockets, by using standard I/O, or by using combinations of the above. (In the latter case, PCCA treats all the messages as if they were transmitted over the same stream.)

Organization The remainder of the paper is organized as follows: §2 provides an overview of our goals and the methods used for achieving them. §3 discusses the individual steps that make up our technique. §4 describes the prototype implementation. §5 presents experimental results. §6 discusses related work.

2. Overview

Consider the example producer and consumer shown in listings 1 and 2, respectively.

EXAMPLE 2.1. The producer is a program that monitors a sensor, and sends an update to the consumer periodically. The system uses an abbreviated protocol: if the sensor data has not changed since the last update, then it is not resent. Line [2] makes this decision.

From the standpoint of checking that the producer and consumer are compatible, even this simplified example has a number of challenging features. In particular,

1. The producer's `loop` procedure uses recursion instead of iteration. In contrast, the consumer is more straightforward: it reads values in a loop.
2. The calls to the write functions in the producer and the read functions in the consumer are organized differently. The producer calls all the `write_*` functions from the same procedure, while the consumer reads the second two fields (a `double` and

Listing 1: Example producer

```
1  sendReading(Sensor* device, int prev)
2      if device→setting == prev then
3          writeBool(false); // no change
4      else
5          writeBool(true); // change: send new data
6          writeDouble(device→setting);
7          writeBool(device→valid);
8  loop(Sensor* device, int prev)
9      ...
10     sendReading(device, prev);
11     if ... then
12         loop(device, device→setting);
13 main()
14     Sensor device;
15     loop(&device, -1);
```

Listing 2: Example consumer

```
1  updateReading(int* setting, bool* valid)
2      *setting = readDouble();
3      *valid = readBool();
4  main()
5      int setting;
6      bool valid;
7      while ... do
8          if readBool() then
9              updateReading (&setting, &valid);
10         ...
```

a `bool`) in a different procedure from the one in which it reads the first field (a `bool`).

PCCA is provided the information that the `read_*` and `write_*` functions are "special" (in that they perform I/O operations); in §4.2, we discuss how this information can be supplied to PCCA. □

Later in the paper, we will refer to a "packet" of data going from the producer to the consumer; a "packet" refers to what the producer sends or the consumer reads during one iteration of their respective loops. (That is, in this example, a packet is either just a Boolean, or a Boolean followed by a double and another Boolean.) "Packet" is an informal term used only as a shorthand in discussions in the paper, and has nothing to do with how the data is actually transported, nor does the notion of packets play an explicit role in our compatibility-checking technique. (However, in Ex. 2.6 in §2.5, we do discuss a way to make "packets" more evident in the languages of the producer's and consumer's models and why that can be advantageous.)

2.1 Nested Words and Nested Word Automata

Nested word automata (NWAs) [1] are a generalization of finite-state automata that can capture the matched call and return structure of execution traces through multi-procedure programs. For purposes of this section, the exact definition is unimportant;[1] it suffices to know that, even though they represent the matched call/return structure of a program, they have the following properties [1]:

[1] NWAs are defined formally in §3.1.

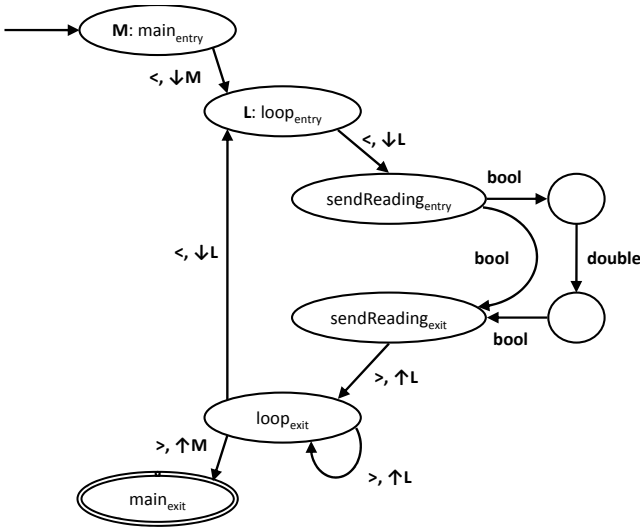

Figure 1. The producer's NWA (with ϵ-transitions collapsed). To reduce clutter, all transitions to the implicit "stuck state" are omitted. For instance, there is a transition $\delta_i(\text{sendReading}_{\text{entry}}, \text{double}, \text{stuck})$.

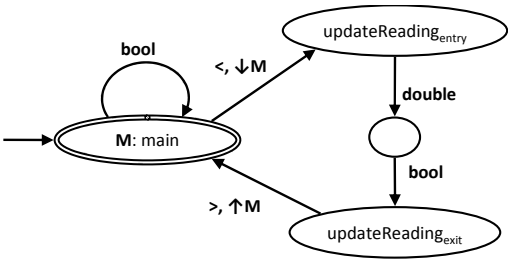

Figure 2. The consumer's NWA (with ϵ-transitions collapsed). Again, all transitions to the implicit "stuck state" are omitted to reduce clutter.

- They are closed under complementation. That is, given NWA A, one can construct an NWA A' such that $L(A') = \overline{L(A)}$.
- They are closed under intersection. That is, given NWAs A_1 and A_2, one can construct an NWA A_3 such that $L(A_3) = L(A_1) \cap L(A_2)$.
- Emptiness testing is decidable. That is, given NWA A, it is decidable whether $L(A) = \emptyset$.

Consequently, language containment is decidable. (Given NWAs A_1 and A_2, to determine whether $L(A_1) \subseteq L(A_2)$, create NWA A_3 such that $L(A_3) = L(A_1) \cap \overline{L(A_2)}$ and check whether $L(A_3) = \emptyset$.)

In our application, the alphabet consists of the types that are emitted by the producer and read by the consumer, together with distinguished call and return symbols ("$\langle$" and "$\rangle$", respectively).

For expository purposes, we talk about the producer NWA "emitting" nested words. An NWA does not actually emit anything (except a yes/no answer); what we mean is that a given word is accepted by the producer's NWA. However, if the word is accepted by the producer's NWA, that means it could be emitted by the producer, and it is often convenient to think of the NWAs as being the producer and consumer themselves, rather than models of the producer and consumer.

2.2 Inferring The I/O Format

The first step in the process is to infer an automaton that approximates the language of the producer and consumer. In the case of the producer, we wish to infer the language of all possible outputs; in the case of the consumer, we wish to infer the language of all inputs that will not drive the producer to an error state.

To do this we use a technique developed by Lim et al. [14]. Lim described the technique in terms of producing a hierarchical finite-state machine (HFSM), but we adapt it slightly to produce an NWA instead. This NWA functions somewhat like an interprocedural control-flow graph from which all "uninteresting" nodes and edges have been removed. A node in the interprocedural control-flow graph is interesting if it is either a branch or a function call/return. The entry node of `main` becomes the start node of the NWA, and the exit node of `main` becomes the sole accepting state.

Figs. 1 and 2 show the NWAs[2] that are produced from the producer and consumer shown above. (To reduce clutter, Figs. 1 and 2 do not show the transitions to an implicit "stuck state".) Call-transitions have labels of the form "$\langle, \downarrow X$", where X is the state at the source of the transition, and $\downarrow X$ means that X is pushed onto the call stack. Return-transitions have labels of the form "$\rangle, \uparrow X$", which means that the machine can make the transition only if state X is on the top of the stack; in so doing, it pops X.

Epsilon-transitions have been collapsed in both NWAs, which resulted in the removal of 7 states (and a comparable number of transitions) from each.

Knowledge about I/O Functions. The system needs information about what function calls can perform I/O. There are a number of ways such information can be provided to the system (see §4.2).

One important point is that there needs to be agreement between the producer and consumer regarding what types are used. The first, and easiest, issue related to this point is that the names of the types must agree.

The second issue is that the granularity of the I/O function specifications must agree. Consider Ex. 2.1. As written, both the producer and consumer have I/O operations expressed in terms of their constituent C types. It would also be possible to have the producer and consumer store values in a two-element structure `SensorData`, and do a "bulk read/write" with `fread()`/`fwrite()` to operate on the struct as a whole. In such a case, it would be reasonable to say that the type of that I/O operation was `SensorData`. However, the two approaches cannot be mixed: the consumer and producer need to agree on the granularity of the approach used.

Remark. The need for agreement between the producer and consumer on the granularity of types is not a fundamental limitation: it would be possible to have the user specify that `SensorData` is a {`double`, `bool`} struct at either the format-inference stage or after the NWAs are constructed, and it might even be possible to extract this information automatically from structure definitions in the code. We have not investigated these avenues at this point; however, with the current implementation the user has the ability to specify, for example, that a particular call to `fread`/`fwrite` operates on a `double` and then a `bool`. □

Finally, the discussion above and in the rest of the paper is framed from the point of view that the NWA alphabet consists of the actual programming language types used by the programs. However, our approach is more flexible. It is possible to have even finer-granularity types—in terms of specific values or constraints on values. One possibility would be to use "types" that do not correspond to C types. For instance, it would be possible to have an `int_ascii` symbol for an integer expressed in ASCII digits (e.g., the three-byte sequence "255") and `int_bin` for an integer in binary (e.g., the four bytes 0x000000FF). Example 2.6 examines a related possibility in the context of our running example. It may also be possible to extend our work to include information about the *values* that are read or written—for instance, to specify that `write_int` outputs a "4" or that `write_int` outputs a value in the range "[4,7]" (and similarly for the input operations of the consumer). Such alternatives are left for future research.

[2] Technically, what are illustrated in Figs. 1 and 2 are more properly called visibly pushdown automata (VPAs) [1], which are standard PDAs with restrictions on when the machine can access the stack. However, VPAs and their languages are equivalent in power to NWAs and their languages, and there are direct mappings from each formalism to the other.

2.3 Enriching the Consumer's NWA

It would be too restrictive to demand that the producer and consumer perform calls and returns at the same points during their executions. The NWAs that we infer from the producer and consumer allow for the same call/return behavior as the original programs, thus the nested words in the languages of the producer and consumer models contain internal call and return symbols that are not actually evident in the transmission across the wire. Performing a set-difference of the nested-word languages would require that they agree in this respect.

Ex. 2.1 illustrates the issue. Each "packet" consists of a Boolean, optionally followed by a double and a Boolean. The producer sends the entire packet within one function (`sendReading`), but the consumer reads the first Boolean, and then calls another function (`updateReading`) to read the remaining values of the packet.

The consequence of the producer and consumer having different calling structure is that the fragments of a nested word that correspond to the same packet are different in the producer's language and the consumer's language.

EXAMPLE 2.2. Consider the word emitted by a run of Ex. 2.1 in which the producer performs just one iteration—hence the word contains just a single packet—and emits "bool double bool." For the producer's NWA (Fig. 1), the corresponding nested word would be "$\langle$ $\langle$ bool double bool $\rangle$ $\rangle$", while for the consumer's NWA (Fig. 2), the nested word would be "bool $\langle$ double bool $\rangle$". These words have different nestings of "$\langle$" and "$\rangle$". □

For this reason, we modify the consumer's NWA so that it can accept words in which new occurrences of "$\langle$" and "$\rangle$" are added and existing occurrences of "$\langle$" and "$\rangle$" are deleted.

EXAMPLE 2.3. For the example discussed in Ex. 2.2, based on Ex. 2.1 and Fig. 2, the language of the consumer's enriched NWA contains not just "bool $\langle$ double bool $\rangle$" but also "$\langle$ $\langle$ bool double bool $\rangle$ $\rangle$". The latter word is in the languages of both the producer's NWA and the consumer's enriched NWA. □

2.4 Language Containment

Once we have in-hand the producer NWA and the enriched consumer NWA, determining the set difference, and thus containment, of their languages is straightforward: $L(P) \setminus L(\text{Enrich}(C)) = \emptyset$ iff $L(P) \cap \overline{L(\text{Enrich}(C))} = \emptyset$. NWAs are closed under all of these operations, so all that is necessary is to take $\text{Enrich}(C)$, complement it, intersect it with P, and test the resulting NWA for emptiness.

2.5 Further Examination of Ex. 2.1

In this section, we return to Ex. 2.1 and illustrate the types of bugs that can be discovered by our algorithm. We discuss bugs in the consumer, but similar errors in the producer would lead to similar results.

EXAMPLE 2.4. Suppose that the author of the consumer did not realize that the protocol had an abbreviation mechanism, and assumed it always sent a full `bool`–`double`–`bool` message. As a con-

sequence, he omitted the check in line [8] of listing 2 and always called `updateReading`.

Our algorithm would discover this, because the nested word "⟨⟨`bool`⟩⟨⟨`bool`⟩⟩⟩" can be emitted by the producer but would not be accepted by the consumer's NWA, no matter how the word is parenthesized. □

EXAMPLE 2.5. Suppose that the specification of the protocol changed during development to include the final `bool` field, but the implementation of the consumer was not updated and line [3] of listing 2 (which reads that field) was omitted.

This would almost certainly signify a bug, but it would *not* be detected by our tool. The reason is that there is no association between the function call on lines [3] and [5] in the consumer, which writes the first `bool` in each packet, and line [8] in the producer, which reads it. Instead, the consumer could "use" the call on `readBool` on line [8] to consume the final field of the *previous* packet, then not call `updateReading` during that iteration. □

EXAMPLE 2.6. We can modify the source code of the producer and consumer—without changing the actual protocol they use to communicate—to make it possible for our technique to detect the bug in the previous example. The problem that our technique has with detecting the previous bug is that what the producer and consumer thought were packets got out of sync. By inserting a phony I/O call at the start or end of each loop (e.g., in the ellipsis on line [9] of the producer and between lines [8] and [9] in the consumer), we can allow the inference algorithm to ensure that the the producer's and consumer's packets cannot get out of sync.

This phony call would have a type that does not appear in the packet itself; in our experiments we have called it `SEP`. The key point to realize is that this "type" does not have to have any material presence in any of the communications, and in fact the function that performs the phony I/O can be completely empty.

This idea can be generalized to "hijack" the communication-inference algorithm to ensure that *events* that should occur during the execution of the producer and consumer occur in the proper order. From this point of view, a write operation is essentially an event, during which the program only happens to perform communication. □

3. Format Inference and Containment Checking

3.1 Nested Words and Nested Word Automata

DEFINITION 3.1 ([1]). *A **nested word** $(w, \rightsquigarrow)$ over alphabet Σ is an ordinary word $w \in \Sigma^*$, together with a **nesting relation** $\rightsquigarrow$ of length $|w|$. $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. A nesting relation of length $l \geq 0$ is a subset of $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ such that*

- *Nesting edges only go forward: if $i \rightsquigarrow j$ then $i < j$.*
- *No two edges share a position: for $1 \leq i \leq l$, $|\{j \mid i \rightsquigarrow j\}| \leq 1$ and $|\{j \mid j \rightsquigarrow i\}| \leq 1$.*
- *Edges do not cross: if $i \rightsquigarrow j$ and $i' \rightsquigarrow j'$, then one cannot have $i < i' \leq j < j'$.*

*When $i \rightsquigarrow j$ holds, for $1 \leq i \leq l$, i is called a **call** position; if $i \rightsquigarrow +\infty$, then i is a **pending call**; otherwise i is a **matched call**, and the unique position j such that $i \rightsquigarrow j$ is called its **return successor**. Similarly, when $i \rightsquigarrow j$ holds, for $1 \leq j \leq l$, j is a **return** position; if $-\infty \rightsquigarrow j$, then j is a **pending return**, otherwise j is a **matched return**, and the unique position i such that $i \rightsquigarrow j$ is called its **call predecessor**. A position $1 \leq i \leq l$ that is neither a call nor a return is an **internal** position.*

***MatchedNW** denotes the set of nested words that have no pending calls or returns. **NWPrefix** denotes the set of nested words that have no pending returns.*

*A **nested word automaton** (NWA) A is a 5-tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation. The transition relation δ consists of three components, $(\delta_c, \delta_i, \delta_r)$, where*

- *$\delta_i \subseteq Q \times \Sigma \times Q$ is the transition relation for internal positions.*
- *$\delta_c \subseteq Q \times \Sigma \times Q$ is the transition relation for call positions.*
- *$\delta_r \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation for return positions.*

*Starting from q_0, an NWA A reads a nested word $nw = (w, \rightsquigarrow)$ from left to right, and performs transitions (possibly nondeterministically) according to the input symbol and $\rightsquigarrow$. If A is in state q when reading input symbol σ at position i in w, and i is an internal or call position, A makes a transition to q' using $(q, \sigma, q') \in \delta_i$ or $(q, \sigma, q') \in \delta_c$, respectively. If i is a return position, let k be the call predecessor of i, and q_c be the state A was in just before the transition it made on the k^{th} symbol; A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A **accepts** nw.* □

In our formulation, we modify the definition above to allow ϵ-transitions in δ_i only; computing the ϵ-closure of a state is straightforward. We also include an implicit "stuck state": any transitions not explicitly represented transfer to this state. By default, the stuck state is not an accepting state, but during complementation the stuck state becomes an accepting state.

In their full generality, NWAs allow there to be call/return transitions on any symbol, but we do not use this. Instead, we have two distinguished symbols: "⟨", which marks each call, and "⟩", which marks each return. In the NWAs generated to model the producer and consumer languages, only call-transitions are labeled with "⟨", and each state implicitly has internal-transitions and return-transitions labeled with "⟨" that transfer control to the stuck state. Similarly, only return-transitions are labeled with "⟩", and each state implicitly has internal-transitions and call-transitions labeled with "⟩" that transfer control to the stuck state.

For nested words of this form, we do not have to give the nesting relation explicitly: the nesting relation is implicit from the positions of the "⟨" and "⟩" symbols. For instance, a word such as "⟨ a b ⟨ c ⟩ ⟩" can only have the nesting relation $\{(1, 7), (4, 6)\}$.

3.2 Checking Containment

3.2.1 Enrichment

It is unreasonable to demand that the producer and consumer have the same call/return structure, so we introduce an "enriching" operation, denoted by Enrich, that when applied to the consumer's NWA will relax the requirement. In particular, Enrich creates new transitions in the consumer's NWA that allow it to make arbitrary calls and returns. In essence, this allows the consumer's NWA to emulate the call/return structure of the producer's NWA.

The transformation is defined as follows:

DEFINITION 3.2. *Given NWA $A = (Q, \Sigma, q_0, \delta, F)$, augment δ with the following transitions:*

1. *For every state p, introduce a call transition $\delta_c(p, \langle, p)$.*
2. *For every pair of states (p, q), introduce a return transition $\delta_r(p, q, \rangle, p)$.*
3. *For every call transition $\delta_c(p, \langle, q)$ in the original NWA, introduce a new ϵ-transition $\delta_i(p, \varepsilon, q)$.*
4. *For every return transition $\delta_r(p, p', \rangle, q)$ in the original NWA, introduce a new ϵ-transition $\delta_i(p, \varepsilon, q)$.*

□

Items 1 and 2 allow the consumer's enriched NWA to perform extra call or return moves to emulate the producer NWA, while items 3 and 4 allow the consumer's enriched NWA to omit call or return moves, in case the producer has fewer.

EXAMPLE 3.3. The example discussed in Exs. 2.2 and 2.3 requires all four steps: to match the producer, the consumer needs to add two call-transitions to the beginning of the nested word, add two matching return-transitions to the end of the nested word, and remove the "extra" call between the first "bool" and "double" and its corresponding return. □

While in theory it is possible to either enrich the consumer to match the producer or enrich the producer to match the consumer, in practice only the former produces reasonable results. The goal of the containment is to determine the emptiness $L(P) \setminus L(C)$. Enriching an NWA increases the size of its language, so this operation adds some error E to one of the operands, resulting in either $(L(P) \cup E) \setminus L(C)$ or $L(P) \setminus (L(C) \cup E)$. Unfortunately, the error introduced by enriching the producer's NWA invariably leads to false positives: for the consumer to accept everything that the enriched producer emits, the consumer would have to accept every possible call structure of every word the producer emits.

3.2.2 Complement and Intersection

As mentioned in §2.4, determining the set difference of the producer NWA and the enriched consumer NWA, is straightforward: $L(P) \setminus L(\text{Enrich}(C)) = \emptyset$ iff $L(P) \cap \overline{L(\text{Enrich}(C))} = \emptyset$, and NWAs support all of the required operations.

3.2.3 Checking Emptiness

Although other algorithms are known for checking whether the language of an NWA is empty [1], for completeness we describe a new algorithm that we devised, which harnesses previously known operations for answering reachability queries on pushdown systems (PDSs). One of the advantage of our approach is that the PDS reachability operation used below (*post**) supports *witness tracing* [24], which can be useful for tracing non-emptiness answers back to potential bugs in the producer and consumer when the answer to the query is that the NWA's language is non-empty. That is, the PDS *post** operation can be run in a mode that returns a counter-example, which serves as an explanation of why the language of an NWA is non-empty.

To describe the algorithm, it is necessary to review some known results about PDSs.

DEFINITION 3.4. *A **pushdown system** (PDS) is a three-tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of **control locations**, Γ is a finite set of **stack symbols**, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of **rules**. A **configuration** of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $u \in \Gamma^*$. The rules define a collection of **transition relations** $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$, then $\langle p, \gamma u \rangle \Rightarrow \langle p', u' u \rangle$ for all $u \in \Gamma^*$.* □

Because the number of configurations of a PDS is unbounded, it is useful to use finite automata to describe certain infinite sets of configurations.

DEFINITION 3.5. *A **configuration automaton** that defines a language of configurations of PDS $\mathcal{P} = (P, \Gamma, \Delta)$ is a finite-state automaton $\mathcal{C} = (S, \Gamma, \rightarrow, P, F)$, where S is a finite set of states, $\mathcal{C}$ uses $\mathcal{P}$'s set of stack symbols Γ as its alphabet, $\rightarrow \subseteq S \times \Gamma \times S$ is the transition relation, the set of initial states consists of $\mathcal{P}$'s set of control locations P (which must be a subset of S), and $F \subseteq S$ is the set of final states. We say that a configuration $\langle p, u \rangle$ is **accepted** by configuration automaton $\mathcal{C}$ if $\mathcal{C}$ can accept u (in the ordinary sense from the theory of finite-state automata) when it is started in the state p; that is, $p \xrightarrow{u}{}^* s$, where $s \in F$. A set of configurations is said to be **regular** if some configuration automaton accepts it.* □

Let $\Rightarrow^*$ denote the reflexive transitive closure of $\Rightarrow$. For a set of configurations C, $pre^*_{\mathcal{P}}(C) \stackrel{\text{def}}{=} \{c' \mid \exists c \in C : c' \Rightarrow^* c\}$ and $post^*_{\mathcal{P}}(C) \stackrel{\text{def}}{=} \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$—i.e., backward and forward reachability, respectively, with respect to transition relation $\Rightarrow$. When C is a regular language of configurations, automata for the configuration languages $pre^*_{\mathcal{P}}(C)$ and $post^*_{\mathcal{P}}(C)$ can be constructed by algorithms that run in time polynomial in the size of $\mathcal{P}$ [3, 10].

Given an NWA A, the first step of checking whether $L(A) = \emptyset$ is to convert A to a PDS $\mathcal{P}_A$.

DEFINITION 3.6. *Given NWA $A = (Q, \Sigma, q_0, \delta, F)$, we define PDS $\mathcal{P}_A = (\{s\}, Q, \Delta)$, where each transition of A is converted to one or two rules in Δ, as follows:*

- *For each transition $(q, \sigma, q') \in \delta_i$, Δ has a rule $\langle s, q \rangle \hookrightarrow \langle s, q' \rangle$.*
- *For each transition $(q_c, \sigma, q_e) \in \delta_c$, Δ has a rule $\langle s, q_c \rangle \hookrightarrow \langle s, q_e\, q_c \rangle$.*
- *For each transition $(q_x, q_c, \sigma', q_r) \in \delta_r$, Δ has two rules, $\langle s, q_x \rangle \hookrightarrow \langle s_x, \epsilon \rangle$ and $\langle s_x, q_c \rangle \hookrightarrow \langle s, q_r \rangle$.*

□

In our application, the initial state of the producer's NWA is $\texttt{main}_{\texttt{entry}}$, and the only final state is $\texttt{main}_{\texttt{exit}}$. Assuming that `main` is never invoked recursively, we really only care about perfectly-matched words (MatchedNW words) and whether the set of perfectly-matched words is empty. To test this condition, we create configuration automata for the languages of initial-state and final-state configurations (where all words have an empty stack)

$$\begin{aligned} L(\text{InitialConfigurations}) &= \{\langle s, q_0 \rangle\} \\ &= \{\langle s, \texttt{main}_{\texttt{entry}} \rangle\} \\ L(\text{FinalConfigurations}) &= \{\langle s, f \rangle \mid f \in F\} \\ &= \{\langle s, \texttt{main}_{\texttt{exit}} \rangle\} \end{aligned}$$

We can check whether the set of perfectly-matched words is empty by answering the question of whether there is a path in the transition relation $\Rightarrow$ from a configuration in $L(\text{InitialConfigurations})$ to a configuration in $L(\text{FinalConfigurations})$. One way to answer this question is to check whether the language of the finite-state automaton constructed as follows is empty:

$$\text{FinalConfigurations} \cap post^*_{\mathcal{P}_A}(\text{InitialConfigurations}). \quad (1)$$

(Note that this reduces the question of NWA emptiness to a question about the emptiness of the language of an ordinary finite-state automaton.)

Remark. The more general question of NWA emptiness when non-perfectly-matched words are of interest can also be addressed using Eqn. (1): one merely has to use more elaborate languages of initial and final configurations. □

3.3 Other Possible Approaches using Known Language-Theoretic Techniques

A different approach to the producer-consumer compatibility-checking problem would be to cast the problem in terms of context-free languages instead of nested-word languages. For instance, it is possible to infer a context-free grammar for each of the producer and consumer programs by the following method:

- Each procedure P in the program corresponds to a nonterminal N_P, for which there is a single production with N_P on the left-hand side.

- The right-hand side of the production for N_P consists of a regular expression over the terminals and nonterminals to capture the I/O operations and call sites, respectively, of P.

For Ex. 2.1, context-free grammars that capture the structure of the producer and the consumer are

PRODUCER:

$$\begin{aligned} \text{main} &\rightarrow \text{loop} \\ \text{loop} &\rightarrow \text{sendReading}\ (\epsilon \mid \text{loop}) \\ \text{sendReading} &\rightarrow (\text{bool} \mid (\text{bool}\ \text{double}\ \text{bool})) \end{aligned}$$

CONSUMER:

$$\begin{aligned} \text{main} &\rightarrow (\text{bool}\ (\epsilon \mid \text{updateReading}))^* \\ \text{updateReading} &\rightarrow \text{double}\ \text{bool} \end{aligned}$$

For compatibility checking, such an approach immediately runs into the problem that the context-free languages are not closed under either complementation or intersection.

For other program-analysis problems that involve multiple context-free languages (e.g., either multiple processes [22, 4, 6] or a single process with two or more independent context-free features [23]) one has the option of modeling each feature/process as a context-free language, then abstracting all but one of the languages as regular languages, intersecting the remaining context-free language with the intersection of the regular languages, and testing whether the resulting context-free language is empty. For instance, Kodumal and Aiken have devised a modeling formalism—set constraints annotated with regular expressions [12]—expressly for this purpose.

Thus, an alternative algorithm for finding possible producer-consumer compatibility bugs would involve

1. inferring a context-free grammar for each of the two programs,
2. using an algorithm that approximates a context-free language by a regular one, such as the one due to Mohri and Nederhof [20] to approximate $L(C)$,
3. complementing the regular approximation of $L(C)$,
4. testing whether the intersection of $L(P)$ with the complement of the regular approximation of $L(C)$ is empty.

The hypothetical method outlined above is incomparable with the method presented in §3. To make this statement more precise,

- Let CFLtoReg denote some operation for approximating a context-free language by a regular language.
- Let Enrich denote the nested-word-automaton enrichment operation defined in Defn. 3.2.
- Let Forget be the operation that (i) first drops the nesting relation, and then (ii) removes all occurrences of "$\langle$" and "$\rangle$" from each word of a nested-word language.

For instance, the context-free languages L(PRODUCER) and L(CONSUMER) equal Forget(L(listing 1)) and Forget(L(listing 2)), respectively.

Now suppose that A is some nested-word automaton. We have

$$\text{CFLtoReg}(\text{Forget}(L(A))) \supseteq \text{Forget}(L(A)) \subseteq \text{Forget}(L(\text{Enrich}(A))).$$

It remains for future work to investigate whether one method is better than the other in practice.

Another alternative would be to use the idea of a *parenthesis grammar* [19], which is a context-free grammar in which each production's right-hand side is enclosed in a pair of parentheses. In the case of grammars like those discussed above, in which each nonterminal corresponds to a procedure, we could add call and return markers around each production. For instance, parenthesis grammars for the producer and consumer are as follows, where the symbols "$\langle$" and "$\rangle$" denote "call" and "return", respectively:

PRODUCER:

$$\begin{aligned} \text{main} &\rightarrow \langle\ \text{loop}\ \rangle \\ \text{loop} &\rightarrow \langle\ \text{sendReading}\ (\epsilon \mid \text{loop})\ \rangle \\ \text{sendReading} &\rightarrow \langle\ (\text{bool} \mid (\text{bool}\ \text{double}\ \text{bool}))\ \rangle \end{aligned}$$

CONSUMER:

$$\begin{aligned} \text{main} &\rightarrow \langle\ (\text{bool}\ (\epsilon \mid \text{updateReading}))^*\ \rangle \\ \text{updateReading} &\rightarrow \langle\ \text{double}\ \text{bool}\ \rangle \end{aligned}$$

However, although parenthesis languages are closed under the language-difference operation [11], that would not help us because the producer's language and the consumer's language are allowed to have a different organization of their calling structures. Parenthesis languages are closed under union and intersection—but, because parenthesis languages can consist of only well-parenthesized words, they are not closed under concatenation and complementation.

In essence, the approach adopted in §3 is close to the parenthesis-grammar approach, but encodes the parenthesis grammars in the more flexible formalism of nested-word automata.

4. Implementation

This section describes a prototype implementation of the ideas presented in §3 in a tool called PCCA (**P**roducer-**C**onsumer **C**onformance **A**nalyzer).

4.1 Removing Irrelevant Procedures

To reduce the size of the inferred NWA, PCCA prunes procedures that cannot possibly participate in I/O operations. If there is no matched path (i.e., a path that produces a MatchedNW word) from the entry of procedure P to the exit of procedure P along which an I/O procedure is invoked, P can be discounted entirely. (For instance, in the consumer's NWA, every nested-word fragment that can be generated for matched paths from P's entry to P's exit would consist entirely of matched call and return markers; however, that fragment can also be produced in the reduced version of the consumer's NWA by "spinning" on a state, using the transitions that were introduced during the enriching operation—see Defn. 3.2.)

The first phase of PCCA uses CodeSurfer/C [8] to build an interprocedural control-flow graph for the program being analyzed. The control-flow graph is traversed to create the NWA. However, an additional analysis artifact CodeSurfer can supply is a call graph and its strongly connected components; we use these structures to perform the pruning operation.

PCCA performs a traversal over the strongly connected components in a reverse topological order, looking at each procedure. When it encounters a procedure that either calls an I/O procedure directly or calls a procedure that was previously marked, it marks that procedure and everything else in the same SCC. When PCCA does another walk over the program to generate the NWA, it only traverses procedures that it marked during the filtering step, ignoring any others.

This is analogous to performing a control-dependence slice on the program from the I/O nodes and only analyzing procedures that appear in the slice, except that it operates at the granularity of procedures.

As illustrated in columns 3 and 4 of Fig. 3 (see §5), the effect of pruning is substantial, reducing the number of procedures by as much as 99.6%. (Larger programs see much more benefit than smaller ones.)

4.2 Seeding the System with I/O Functions

PCCA requires information about (i) what function calls of the producer can perform output, and (ii) what function calls of the consumer can perform input. There are a number of ways such information can be supplied to PCCA:

1. The user can provide a list of I/O functions (e.g. `readBoolean`, `writeInt`, as in the example) and their associated types. For calls to standard functions such as `puts`, PCCA is already equipped with such mappings.
2. For calls to `printf`- or `scanf`-style procedures, if the format string is a constant in the code, PCCA will parse the string to determine the types being operated on.

   The implementation is flexible enough so that the producer or consumer can contain user-defined procedures with `printf`/`scanf`-like format-strings, provided that the format-string syntax is either the same as what is used by `printf` or what is used by `scanf`. PCCA just needs to know the name of the procedure and which formal parameter holds the format string.
3. For calls to `fread`/`read`-like procedures, we can infer the type being operated on by looking at the type of the data parameter before its cast to `void*`.
4. If all else fails, the user can supply comments that annotate procedure-call sites to specify that a particular call site performs either input or output. The annotation includes the type that is operated on. This method also allows the user to selectively choose only some call sites on a particular procedure.
5. Finally, the list of procedure-call sites that the tool should consider to be I/O functions is explicitly materialized in a text file, so the user can add, remove, or change procedure-call sites in that list, or even write another script to create it. This interface facilitates making extensions to PCCA to supply it with additional information about I/O functions. (In fact, in the current version of PCCA, the techniques described in items [1]–[3] are implemented by one program, and the technique described in item 4 is implemented by a second program.

5. Experiments

To test the capabilities of PCCA, we ran it on a small corpus of examples (whose characteristics are listed in columns 2 and 3 of Fig. 3). The inference of the NWAs for the producers and consumers was performed on a machine with 2.83 GHz Core 2 Quad and 3 GB of memory running 32-bit Red Hat Linux Enterprise 5. The language-containment check was performed on a single core of a two-processor quad-core 2.26 GHz Xeon processor running Windows XP 64-bit, with 12 GB of memory.

The experiments were designed to test whether PCCA would detect bugs in producer-consumer pairs that were buggy, correctly identify (presumably) correct code as having the language containment property, and scale to realistic programs.

Each example consisted of a pair of programs—a producer and a consumer. In several cases, we used the program as both the producer and the consumer, which makes sense for programs such as `gzip`, `bzip2`, and `dia`, where they produce output that is meant to be processed by the same program.

The examples are as follows[3]:

- `ex-prod/ex-cons` make up our running example (stubs for the I/O functions are included in the count),
- `ex-prod-bug/ex-cons-bug` are buggy versions of the running example, modified as described in Ex. 2.6 with the separator to mark the packets,
- `ez-ntp-server/ez-ntp-client` are a server and client for the network time protocol (the communication we are testing here is from the server to the client),
- `gzip` and `bzip2` are the common Unix compression/decompression utilities, and
- `dia` is software for creating diagrams.

Two of the tests required minor modifications. `gzip` uses input and output operations much like the one in our running example, except that they are implemented as macros. Because our inference tool works over the control-flow graph generated by CodeSurfer/C, these macros are not visible, so we replaced the macro definitions with functions. (Making them into stub functions is sufficient for our analysis.) The `ez-ntpd` server forks off threads to handle each request using `pthread_create`, which takes a function pointer to the function that handles that request. Because the program does not include the source for `pthread_create`, CodeSurfer/C does not know about the transfer of control, and so the server looks like it does not actually respond to the request (and thus does not emit anything); we changed this to a call to the function directly. Having source code for `pthread_create` or an appropriate model would obviate the need for this step.

As shown in Fig. 3, PCCA reports that some commonly-used programs operate in a correct manner with regard to their I/O behavior. PCCA also detects synthetic programming errors in small examples, as shown by the second pair of examples.

6. Related Work

Inferring Input or Output Formats of Programs. PCCA is built, in part, on top of the file-format-extractor tool FFE/x86 [14], which extracts output formats from x86 executables. PCCA's I/O extraction components were created from FFE/x86 by (i) retargeting it to work on C and C++ source code,[4] and adding the capability to infer input formats as well as output formats.

Inferring input formats of executables has received much attention lately, particularly in the context of protocol reverse engineering for network security [5, 9, 25, 15, 16]. However, most of this work involves the use of dynamic-analysis techniques.

Komondoor and Ramalingam developed methods to recover an object-oriented data model from a program written in a weakly-typed languages, such as Cobol [13]. It is capable of recovering information about the record structure of entities that occur in a file, as well as information about subtyping relationships between such entities.

Checking Compatibility/Conformance. Rajamani and Rehof [21] developed a way to check that an implementation model I extracted from a message-passing program conforms to a specification S. Their goal was to support modular reasoning on models; they established that if I conforms to S and P is any environment in which P and S do not get stuck waiting in vain to send or receive messages, then P and I also do not get stuck.

A related question is checking conformance of software components as software evolves and components are replaced or upgraded. Clarke et al. [7] survey several approaches that have been devised to answer the question, including interface automata, behavioral subtyping [17], input/output-based compatibility of upgrades [18], and model checking.

References

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[3] Our experiments can be found at `http://www.cs.wisc.edu/wpis/examples/pcca/`

[4] Our job was made easier because FFE/x86 was based on the x86 version of CodeSurfer [2], which makes use of some of the CodeSurfer/C infrastructure [8].

Test	#Lines of code	#Funcs Original	#Funcs Reduced	#NWA vertices	#Static I/O ops	Times (seconds) Infer NWA	Times (seconds) ¬Consumer	Times (seconds) ∩	Times (seconds) Total	OK?
ex-prod	43	11	3	16	5	2.87	0.49	0.0	6.12	Y
ex-cons	26	7	2	11	2	2.87				
ex-prod-bug	43	11	3	17	5	2.65	0.47	0.0	9.57	N
ex-cons-bug	25	7	2	11	3	6.67				
ez-ntp-server	622	7	2	32	1	4.38	0.56	0.0	11.58	Y
ez-ntp-client	678	6	1	51	1	6.57				
gzip-prod	4213	87	15	271	18	120	665	0.13	843	Y
gzip-cons	4213	87	15	367	47	57				
bzip2-prod	5413	108	15	357	8	132	877	0.62	1124	Y
bzip2-cons	5413	108	13	322	10	115				
dia-prod	128972	4554	18	137	7	435	4.11	0.25	885	Y
dia-cons	128972	4554	14	196	5	446				

Figure 3. Experimental results.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables, (tool demonstration paper). In *Comp. Construct.*, 2005.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, 1997.

[4] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.*, pages 62–73, 2003.

[5] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Conf. on Comp. and Commun. Sec.*, 2007.

[6] S. Chaki, E. Clarke, N. Kidd, T. Reps, and T. Touili. Verifying concurrent message-passing C programs with recursive calls. In *Tools and Algs. for the Construct. and Anal. of Syst.*, 2006.

[7] E. Clarke, N. Sharygina, and N. Sinha. Program compatibility approaches. In *Formal Methods for Components and Objects*, 2005.

[8] CodeSurfer system. www.grammatech.com/products/codesurfer.

[9] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Conf. on Comp. and Commun. Sec.*, 2008.

[10] A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.

[11] D. Knuth. A characterization of parenthesis languages. *Inf. and Control*, 11(3), 1967.

[12] J. Kodumal and A. Aiken. Regularly annotated set constraints. In *Prog. Lang. Design and Impl.*, 2007.

[13] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *Working Conf. on Rev. Eng.*, 2007.

[14] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *Working Conf. on Rev. Eng.*, 2006.

[15] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Dist. Syst. Security*, 2008.

[16] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Found. of Softw. Eng.*, 2008.

[17] B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints. In H. Bowman and J. Derrick, editors, *Formal Methods for Distributed Processing: An Object Oriented Approach*. Cambridge Univ. Press, 2001.

[18] S. McCamant and M. Ernst. Early identification of incompatibilities in multicomponent upgrades. In *European Conf. on Obj.-Oriented Prog.*, 2004.

[19] R. McNaughton. Parenthesis grammars. *J. ACM*, 14(3), 1967.

[20] M. Mohri and M.-J. Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, chapter 9. Kluwer Acad., 2001.

[21] S. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Computer Aided Verif.*, 2002.

[22] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *Trans. on Prog. Lang. and Syst.*, 22(2):416–430, 2000.

[23] T. Reps. Undecidability of context-sensitive data-dependence analysis. *Trans. on Prog. Lang. and Syst.*, 22(1):162–186, Jan. 2000.

[24] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.

[25] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Network and Dist. Syst. Security*, 2008.