Checking Conformance of a Producer and a Consumer*

Evan Driscoll, Amanda Burton, and Thomas Reps
Computer Sciences Department, University of Wisconsin – Madison
{driscoll,burtona,reps}@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 considers ill-formed.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability

General Terms

Algorithms, Reliability

Keywords

producer-consumer compatibility, language containment, visibly pushdown automata

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 what the optimum fuel-air mixture should be. It emits messages to other components, such as the fuel pumps and fuel injectors, to carry out its decisions.

*Supported by NSF under grants CCF-0540955, CCF-0810053, CCF-0904371, and CCF-0701957; by ONR under grants N00014-{09-1-0510, 10-M-0251}, by ARL under grant W911NF-09-1-0413; and by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

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 occur. Send/receive incompatibilities can 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.)

Consider an example system made up of the producer and consumer shown in Figs. 1 and 2, respectively. The producer is a program that monitors a sensor, and periodically sends a "packet" of data to the consumer.[1] The system uses an abbreviated protocol: if the sensor data has not changed since the last message, then only the Boolean literal `false` is sent. Line 2 in Fig. 1 makes this decision.

As presented, these components are correct: both "speak" the same protocol. However, consider a buggy version of the consumer that does not account for the possibility that the producer sends an abbreviated message, and always expects the full packet. This code is shown in Fig. 3.

To find this bug, can we reason about the languages over which each component operates? In the consumer, we know that the `updateReading` function always reads a `double` and then a `bool`. Furthermore, each time through the loop in the buggy version of `main`, the consumer reads a `bool` then the `double`–`bool` sequence from `updateReading`. Thus we can determine that the input language of the buggy consumer, expressed as a regular expression over types that the consumer reads, is `(bool double bool)+`. Similarly, we can determine that the output language of the producer, expressed as a regular expression, is `(bool | bool double bool)+`.

From these two language descriptions we can see that one of the components is buggy: the string `bool bool`, is in the producer's language but not in the consumer's. This disparity suggests that some execution of the producer could output two Boolean values, but *no* execution of the consumer would expect to read that message.

A similar analysis suggests that Fig. 2's consumer is correct. The language it expects is `(bool (double bool)?)+`, which is equivalent to what we inferred for the producer.

We describe a technique for determining whether two components are compatible, which proceeds along lines similar to this example: we infer a model of the output language of the producer, infer a model of the input language of the

[1]We use *packet* to refer to the data that the components communicate each time through their "loop".

```
 1 sendReading(Sensor* device, int prev)
 2     if device→setting == prev then
 3         writeBool(false);
 4     else
 5         writeBool(true);
 6         writeDouble(device→setting);
 7         writeBool(device→valid);
 8 loop(Sensor* device, int prev)
 9     ... // update device with new readings
10     sendReading(device, prev);
11     if ... then
12         loop(device, device→setting);
13 main()
14     Sensor device;
15     loop(&device, -1);
```

Figure 1: Example producer

```
 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         ... // do something with current readings
```

Figure 2: Example consumer

consumer, and determine whether the two descriptions are compatible. However, we also investigate using a richer family of languages than just regular languages for the format descriptions, and thus the check of whether the models are compatible is more than straight language-containment.

This paper addresses the following problem: *Given two programs that operate in a producer/consumer relationship, (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 but the consumer considers ill-formed.*

We have implemented our technique 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 §4.1), PCCA infers a description of the language that the producer generates and a description of the language that the consumer expects, and (roughly speaking) determines whether the former is a subset of the latter.

PCCA starts out by creating an automaton P that models the producer, which accepts an over-approximation of the language that the producer emits. We have two versions of PCCA: one creates a pushdown automaton (PDA) and the other creates a standard finite automaton (FA). Similarly, PCCA produces an automaton C for the consumer, which accepts an over-approximation of the language that the consumer expects. Our goal becomes determining whether $L(P) \subseteq L(C)$. For the FA version, we do this directly. For the PDA version, this question is undecidable with a direct

```
1 main()
2     while ... do
3         readBool();
4         updateReading (&setting, &valid);
5         ... // do something with current readings
```

Figure 3: Example buggy consumer. (`updateReading` is the same as in Fig. 2.)

approach; thus we actually use a restricted form of PDAs called *visibly pushdown automata* [3, 2] (VPAs) to model the components' behaviors. While less powerful than full PDAs, VPAs provide enough power to recognize important classes of non-regular languages, such as balanced parentheses.

Unlike PDAs, the restrictions on VPAs allow them to retain closure under complementation and intersection, and thus language containment is decidable. However, when using VPAs we must make internal calls and returns explicit in the words of the languages that PCCA infers from the components. If PCCA were to check containment of these languages directly, we would be requiring that the two components have the same internal call/return structure, which is undesirable. To compensate, PCCA "enriches" the consumer's VPA C—and hence its 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.)

To test language containment, PCCA performs a set-difference operation to compute $L(P) \setminus L(C)$ (for the FA version) or $L(P) \setminus L(\text{Enrich}(C))$ (for the VPA version) by complementing the righthand automaton intersecting the result with P. Finally, it tests whether the language of the VPA produced by the intersection is empty. If so, then the subset holds, and PCCA reports the components are compatible; if not, then there is a counter-example in the difference and PCCA reports the components are incompatible.

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 playing the roles of both producer and consumer—that share files, send information over sockets, or use standard I/O.

Because it is necessary to approximate each component's language, the technique we have developed has some limitations. Both approximations are over-approximations: i.e., the producer's model may say that the producer can emit a string that can never actually be emitted by the producer, and similarly for the consumer. Consequently, the compatibility-checking technique can have both false positives and false negatives, which puts PCCA among other bug-hunting approaches, such as work developed by Engler et al. [16, 22] as well as many other authors.

Our contributions can be summarized as follows:

- We describe our approach to format inference, inspired by Lim et al. [23], and its application to component compatibility.
- The model of the program can be described using a context-free language, but testing language containment for CFLs is not decidable. We describe how to side-step this problem by using VPAs, making the internal calls and returns of the components explicit, and performing our Enrich operation.

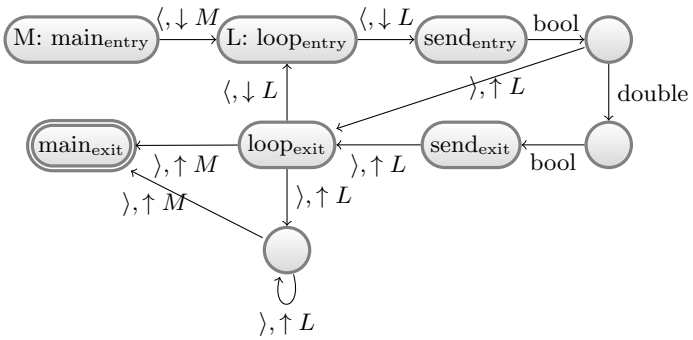

Figure 4: The producer's VPA. To reduce clutter, all transitions to the implicit "stuck state" are omitted and sendReading is abbreviated as send.

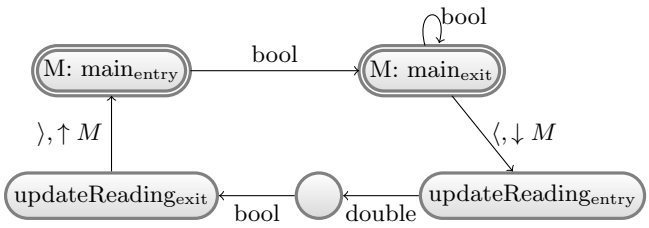

Figure 5: The consumer's VPA.

- We compare the results of a VPA-based model to those obtained by using ordinary finite automata.
- We describe a new algorithm for determining the emptyness of a VPA, which builds on existing work related to pushdown systems.
- We implemented our techniques in a tool called PCCA, and demonstrate its utility on several examples.

Organization. The remainder of the paper is organized as follows: §2 discusses our goals and the methods we use to achieve them. §3 discusses the indiv steps that that make up our technique when PCCA is operating using VPAs. §4 describes the prototype implementation. §5 presents experimental results. §6 discusses related work. §7 has a brief discussion of future work.

2. OVERVIEW

We now give a description of our technique, framed around how it operates on the example considered in the Introduction. We first consider the producer and the correct version of the consumer, as presented in Figs. 1 and 2, respectively. (Ex. 2.3 covers the buggy version of the consumer.)

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

PCCA is provided the information that the `read_*` and `write_*` functions are "special" (in that they perform I/O); in §4.1, we discuss how this information can be supplied.

As mentioned earlier, PCCA is able to use two different formalisms to model the components: visibly pushdown automata (VPA) and standard finite automata (FA).

2.1 Visibly Pushdown Automata

From each component we infer a visibly pushdown automaton when operating in that mode. VPAs [3] are a restriction of ordinary pushdown automata, and can be used in program analysis to capture the matched call and return structure of execution traces through multi-procedure programs. In essense, each alphabet symbol describes whether a VPA is allowed to push or pop a symbol from its stack. We formally define VPAs in §3.1, but for purposes of this section it suffices to know that, even though they share some power with general PDAs, determining language containment of two VPAs remains decidable, in contrast to PDAs.

In our application, the alphabet consists of the types that are emitted by the producer and read by the consumer, as well as distinguished call and return symbols $\langle$ and $\rangle$.

For expository purposes, we talk about the producer automaton "emitting" strings. Neither FAs nor VPAs actually emit anything (except a yes/no answer); what we mean is that a given string is accepted by the producer's automaton. However, if a string is accepted by the producer's automaton, that means it could be emitted by the producer program, and it is often convenient to think of the automata as being the components themselves, rather than models.

2.2 Inferring The I/O Format

The first step in the process is to infer an automaton that approximates the language of each component. 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 expected inputs.

The idea behind our technique is to create automata that mimic the control-flow behavior of the source programs. Each automaton that PCCA generates has the same language as one created by transliterating the program's interprocedural control-flow graph (ICFG) in the following manner:

1. There is one state $\tilde{c}$ for each ICFG node c.
2. If a call site c can call an I/O function that outputs or expects a value of type τ, we add a transition on τ from $\tilde{c}$ to its corresponding return node. In the VPA model, this transition does not modify the stack.
3. If a call site c can call a non-I/O function f with entry node f_e and exit node f_x, we add one transition from $\tilde{c}$ to $\tilde{f_e}$ and a second transition from $\tilde{f_x}$ to the corresponding return site. In the FA model, both transitions are ε transitions. In the VPA model, the first transition is on the symbol $\langle$ and pushes $\tilde{c}$ onto the VPA's stack, and the second transition is on the symbol $\rangle$ under the condition that $\tilde{c}$ is at the top of the stack (which is then popped).
4. All other transitions in the ICFG become ε-transitions. In the VPA version, these do not modify the stack.
5. The entry node of main becomes the start state, and the exit node becomes the sole accepting state.

However, if we used this naive translation, the result-

ing automata would be extremely large, which would cause problems during the determinization phase of PCCA.[2]

Instead of treating the ICFG as a whole, PCCA proceeds procedure-by-procedure through the program. For each procedure, it looks at the *intra*procedural CFG and carries out the above translation, except that step 3 is replaced by the following:

- If a call site c can call a non-I/O function f, we add an *internal* transition from $\tilde{c}$ to the corresponding return, labeled with a generated symbol `call_f`.

(Also, the VPA's starting and accepting states are the entry and exit nodes of that procedure.)

Even in the VPA version, because each procedure's automation has no transitions that modify the stack (stack operations only happen on a call or return), it can be interpreted as a standard finite-state machine. We do this, and use the standard algorithms to determinize and minimize each procedure's machine. (PCCA's implementation uses the OpenFST library for this purpose [1].) The efficiency upshot is that this technique turns what would be a multiplicative factor into an additive one, thus dramatically reducing the time spent in determinization.

Once we have the collection of minimized automata, we combine all the automata into one and "restore" the call and return transitions. We replace each transition that moves from state c to r when reading a symbol `call_f` with a pair of transitions that match those in the original step 3:

- We add a transition from c to f's entry point. In the FA version, this is an ε transition; in the VPA version, it is on the symbol $\langle$ and pushes c onto the stack.
- We add a transition from the exit point of f to r. In the FA version, this is an ε transition; in the VPA version, it is on the symbol $\rangle$ and requires that c appears at the top of the stack (which is then popped).

Finally, we have to perform one more determinization step in case connecting the procedures adds nondeterminism.

This translation essentially abstracts the program to its control flow only: data is not considered. One could envision a higher-fidelity translation that weaves selected data elements (or abstractions of data elements) into the automata we infer, but of course there is a trade-off between precision and automaton size.

Figs. 4 and 5 show the VPAs that are inferred from the code in Figs. 1 and 2, respectively. (To reduce clutter, Figs. 4 and 5 have ε-transitions collapsed, which removes 7 states and a comparable number of transitions in each automaton.) 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. The FA version is similar, except that all call and return transitions are replaced with ε transitions.

Knowledge about I/O functions. PCCA needs information about what function calls can perform I/O. There are a number of ways the user can provide such information (see §4.1).

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 our example. 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.

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 VPAs are constructed, and it should even be possible to extract this information from struct 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`. □

2.3 Enriching the Consumer's VPA

This section applies to the VPA version of PCCA only; §3.2 describes the operation formally.

It would be too restrictive to demand that the producer and consumer perform calls and returns at corresponding moments during their executions. The VPAs that we infer from the producer and consumer follow the same call/return behavior as the original programs; thus the strings in the languages of the producer and consumer models contain internal call and return symbols that are not actually present in the messages between components. Checking containment of the languages of the inferred models would require that the components agree in this respect.

Our running example 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 substrings that correspond to the same packet are different in the producer's language and the consumer's language.

EXAMPLE 2.1 Consider the string `bool double bool`, emitted by Fig. 1's code when the producer performs just one iteration—hence the string contains just a single packet. For the producer's VPA (Fig. 4), the string would be $\langle$ $\langle$ `bool double bool` $\rangle$ $\rangle$, while for the consumer's VPA (Fig. 5), the corresponding string would be `bool` $\langle$ `double bool` $\rangle$. These strings have $\langle$ and $\rangle$ in different locations. □

To accommodate the different nesting structures, we "enrich" the consumer's VPA so that it can use nondeterminism to guess when the producer makes an internal call or return and insert the corresponding symbol into its own strings.

EXAMPLE 2.2 For the example discussed in Ex. 2.1, the language of the consumer's enriched VPA contains not just `bool`

[2]As shown in §5, determinization dominates execution time.

⟨ `double bool` ⟩ but also ⟨ ⟨ `bool double bool` ⟩ ⟩. The latter string is in the languages of both the producer's VPA and the consumer's enriched VPA. □

After enrichment, a counterexample to the language containment (and an indication of incompatibility) is a string that the producer's NWA can emit where it is impossible to add and/or remove balanced parentheses and arrive at a string that the consumer's NWA accepts.

EXAMPLE 2.3 For the buggy consumer in Fig. 3, the original language contains strings such as `bool` ⟨ `double bool` ⟩, but not, for instance `bool bool` (which is in the producer's VPA's language). Denote by C_e the VPA inferred for the buggy consumer. No matter how you add parentheses to `bool bool`, you will not arrive at a string in the language of Enrich(C_e); this will be a counterexample to language containment. □

If an analyst knows that both components use the same call/return structure, he can omit the enrichment step to obtain a more precise comparison of the two languages. Without the approximation caused by Enrich, a "compatible" result is more credible; however, if there is uncertainty in the call/return assumption, an "incompatible" result is less credible.

2.4 Language Containment

Once we have the producer automaton P and the consumer automaton C' (for the VPA version, C' is the enriched consumer automaton), determining the set difference, and thus containment, of their languages is straightforward: $L(P) \setminus L(C') = \emptyset$ iff $L(P) \cap \overline{L(C')} = \emptyset$. Both FAs and VPAs are closed under all of these operations, so all that is necessary is to take C', complement it, intersect it with P, and test the resulting VPA for emptiness. §3 discusses this step in greater detail for VPAs.

2.5 Helping PCCA Improve its Results

We now return to the running example to illustrate how the programmer could improve the results of the analysis. We start by describing a bug that PCCA would not be able to find, and then explain how to modify the code—but without changing the actual protocol—so that the bug *is* found.

Suppose that the specification of the protocol changed during development: the final `bool` field was not originally needed, but was added later. Suppose that the implementation of the producer *was* changed to emit this field, but the consumer was not updated. (In other words, line 7 in Fig. 1 was added at the time the specification changed. The consumer *should* have been changed to add line 3 in Fig. 2, but that line was erroneously omitted.)

This situation 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 producer, which writes the first `bool` in each packet, and line 8 in the consumer, 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.

We can modify the source code of the producer and consumer to make it possible for our technique to detect the previous bug. The problem that our technique has with detecting this 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 make the packet divisions visible to PCCA, allowing it to check that the producer's and consumer's packets cannot get out of sync.

The phony calls 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 compatibility 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 fact that the program communicates is only incidental.

3. FORMALIZATION OF VPA-BASED CONTAINMENT CHECKING

This section discusses the details of how PCCA determines whether the producer's language is a subset of the (enriched) consumer's when using visibly pushdown automata (VPAs). Using VPAs provides a potential benefit over FAs (see §3.3) but introduces a number of complications to the process that are not present with standard finite automata.

3.1 Visibly Pushdown Automata

DEFINITION 3.1 ([3]) A VPA is a pushdown automaton (PDA) that operates on a tagged alphabet and whose stack accesses are restricted by the current symbol. A **tagged alphabet** $\tilde{\Sigma}$ is a partition of a normal alphabet into disjoint subsets Σ, Σ_c, and Σ_r. In our application, $\Sigma_c = \{\langle\}$, $\Sigma_r = \{\rangle\}$, and Σ is the set of types on which the program being analyzed operates. (In program analysis, pairs of symbols in Σ_c and Σ_r are often used to model program calls and returns; they can also be used to represent other matched entities such as opening and closing XML tags.)

A **visibly pushdown automaton** V is a 6-tuple $(Q, \tilde{\Sigma}, \Gamma_\perp, \delta, q_0, F)$ where Q is the set of states, $\tilde{\Sigma}$ is a tagged alphabet, $\Gamma_\perp$ is the stack alphabet (with $\perp$, a special bottom-of-stack symbol, and $\Gamma = \Gamma_\perp \setminus \{\perp\}$), $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The transition relation, δ, is the union of three components:

- $\delta_i \subseteq (Q \times \Sigma) \times Q$
- $\delta_c \subseteq (Q \times \Sigma_c) \times (Q \times \Gamma)$
- $\delta_r \subseteq (Q \times \Sigma_r \times \Gamma_\perp) \times Q$

A VPA M reads its input and makes transitions on each symbol as follows. If the current symbol is σ, the current state is q, and γ is at the top of the stack, then:

- If $\sigma \in \Sigma$, then M selects a transition $((q, \sigma), q')$ from δ_i and changes its control state to q'.
- If $\sigma \in \Sigma_c$, M selects a transition $((q, \sigma), (q', \gamma'))$ from δ_c, pushes γ' onto its stack, and changes to state q'.
- If $\sigma \in \Sigma_r$, M selects a transition $((q, \sigma, \gamma), q')$ from δ_r, pops γ from its stack, and changes to state q'.

The VPA accepts its input if there is a run that ends in a final state $f \in F$.

The behavior above can be expressed as limiting the operation of a standard PDA in the following way: when reading $\sigma \in \Sigma$, the VPA cannot access the stack; when reading

$\sigma_c \in \Sigma_c$, the VPA must push exactly one symbol; and when reading $\sigma_r \in \Sigma_r$, the VPA must pop exactly one symbol. In this way, the VPA's stack accesses are visible in each input.

We refer to the $\sigma \in \Sigma$ as **internal** symbols, the $\sigma_c \in \Sigma_c$ (and the positions in a string at which they appear) as **calls**, and $\sigma_r \in \Sigma_r$ (and their positions) as **returns**. □

We take $\Gamma = Q$ and construct VPAs that, when in state q with a call as the current symbol, push q onto the stack.[3] We also allow internal ε-transitions in the natural way.

3.2 Enrichment

As discussed at a high level in §2.3, 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 VPA will relax the requirement. Enrich creates new transitions in the consumer's VPA that allow it to make arbitrary calls and returns. In essence, this allows the consumer's VPA to emulate the call/return structure of the producer's VPA. Enrich is defined as follows:

DEFINITION 3.2 Given VPA $A = (Q, \Sigma, q_0, \delta, F)$, 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 VPA, introduce a ε-transition $\delta_i(p, \varepsilon, q)$.
4. For every return transition $\delta_r(p, p', \rangle, q)$ in the original VPA, introduce a ε-transition $\delta_i(p, \varepsilon, q)$. □

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

EXAMPLE 3.3 The example discussed in Exs. 2.1 and 2.2 requires all four steps: to match the producer, the consumer needs to add two calls to the beginning of the input string, add two matching returns to the end of the input string, and remove the "extra" call between the first "bool" and "double" and its corresponding return. □

While in theory it is possible either to enrich the consumer to match the producer or enrich the producer to match the consumer, in practice only the former is reasonable. The goal of the containment check is to determine the emptiness $L(P) \setminus L(C)$. Enriching a VPA enlarges 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 VPA 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 string the producer emits.

3.3 Benefits of Using VPAs

There are several kinds of automata that we could have chosen. For instance, the FA version of PCCA models each program as a single finite automaton. This approach removes the need for the enrich operation because calls and returns are not represented explicitly in the languages.

The trade-offs between VPAS and FAs mirror trade-offs that one can make in traditional interprocedural dataflow analysis. The simplest way of performing such analysis is to build the ICFG and run the analysis as if call and return edges were just normal intraprocedural control-flow edges. However, that approach loses precision because of suprious data flows from one call site c_1, into the called function f, and then out the return edge to a different call site c_2. A similar kind of imprecision can affect the FA version of PCCA.

For instance, Fig. 6 shows a producer and consumer for which FAs and VPAs produce different results. Due to space constraints we omit diagrams of the inferred automata. The FA version of PCCA infers `int | char int char` for the language of the consumer, but `int | char int | int char | char int char` for the producer. The producer's language contains two words that are not in the language of the consumer, thus the FA version of PCCA reports that the components are incompatible.

One way of getting around this problem is to perform function inlining: each call site c gets its own copy of the procedure f, which is only called from c and only returns to c. This eliminates the suprious control flows, but at the cost of a potentially exponentially larger model. It would be possible to do exactly the same thing in our domain: create a single FSM, but inline procedures.

A more sophisticated mechanism for eliminating these spurious flows uses context-free-language reachability techniques [31]. This marks each call/return edge pair with a distinct set of matched parentheses; possible executions of the program correspond only to strings with matched parentheses. The dataflow problem can be formulated so that only flows along such well-matched paths are considered. Our use of VPAs closely mirrors this approach for the producer.[4]

```
1 outputInt()
2     writeInt();
3 producerMain()
4     if ... then
5         outputInt();
6     else
7         writeChar(); outputInt(); writeChar();
8 inputInt1()
9     readInt();
10 inputInt2()
11    readInt();
12 consumerMain()
13    if ... then
14        inputInt1();
15    else
16        readChar(); inputInt2(); readChar();
```

Figure 6: Components that illustrate the benefits of VPAs

[3]Following [3], this restriction is called a "weakly-hierarchical VPA", and does not reduce the expressiveness.

[4]CFL-reachability distinguishes acceptable return edges from unacceptable ones by whether the brackets match; our VPAs distinguish them by whether the corresponding call site is on the VPA's stack.

The code in Fig. 6 benefits from this increase in precision. The VPA's constraints on the return transitions from the exit node of `outputInt` to each of the two return sites restricts the data flow, thus the producer's language is inferred to be ⟨ `int` ⟩ | `char` ⟨ `int` ⟩ `char`. The VPA version of PCCA reports that the two components are compatible.

Unfortunately, this benefit only applies to the producer's model: the enrich operation we do to the consumer essentially makes a regular approximation out of the original. We have not investigated applying the ideas of inlining to obtain increased precision, although we think it would be possible.

In other words, using VPAs to model the components provides a way to obtain a context-sensitive analysis in one of the components without the exponential blowup of inlining.

3.4 Complement and Intersection

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

3.5 Checking Emptiness

Although other algorithms are known even for general PDAs, for completeness we describe a new algorithm that we devised, which harnesses previously known operations for answering reachability queries on pushdown systems (PDSs). Our approach is purely automata-theoretic, and does not translate the VPA language to a context-free grammar. We can use the *witness tracing* [32] feature supported by the PDS reachability operation (*post**) to trace non-emptiness answers back to a string that is in the producer's VPA's language but not in the consumer's; such a string suggests a potential bug in one of the components.

To describe the algorithm, it is necessary to review some known results about PDSs [5, 14].

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 **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 $\mathcal{P}$ [5, 14].

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

DEFINITION 3.6 Given VPA $A = (Q, \tilde{\Sigma}, Q, \delta, q_0, F)$, 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 $((q, \sigma), q') \in \delta$, Δ has a rule $\langle s, q \rangle \hookrightarrow \langle s, q' \rangle$.
- For each transition $((q, \sigma_c), (q', \gamma)) \in \delta$, Δ has a rule $\langle s, q \rangle \hookrightarrow \langle s, q'\, q \rangle$. (This pushes q' onto the stack, the top of which is currently q.)
- For each transition $((q, \gamma, \sigma_r), q') \in \delta$, Δ has two rules, $\langle s, q \rangle \hookrightarrow \langle s_x, \varepsilon \rangle$ and $\langle s_x, \gamma \rangle \hookrightarrow \langle s, q' \rangle$. (Conceptually this can be thought of as a single transition $\langle s, q\, \gamma \rangle \hookrightarrow \langle s, q' \rangle$ of a prefix rewriting system [8].)

One can interpret this conversion as simply moving the information in the VPA's finite control into the top symbol of the stack. □

In our application, the initial state of the producer's VPA is $\texttt{main}_{\texttt{entry}}$, and the only final state is $\texttt{main}_{\texttt{exit}}$. Assuming that `main` is never invoked recursively, we only consider perfectly-matched strings (those with balanced calls and returns) and whether the set of perfectly-matched strings is empty. To test this condition, we create trivial configuration automata for the languages of initial-state and final-state configurations (where the machine has 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 strings is empty by answering the question of whether there is a path in the transition relation $\Rightarrow$ from a configuration in L(InitialConfigurations) to a configuration in L(FinalConfigurations). One way to answer this question is to check whether the language of the finite-state automaton constructed as follows is empty:

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

(This reduces the question of VPA emptiness to emptiness of the language of an ordinary FA.)

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

4. IMPLEMENTATION

This section describes a prototype implementation of the ideas presented in §2 and §3 in a tool called PCCA (**P**roducer-**C**onsumer **C**onformance **A**nalyzer).

PCCA has two phases: inference and compatibility. During the inference phase, PCCA uses CodeSurfer/C [10] to per-

form pointer analysis and build an interprocedural control-flow graph (ICFG) and call graph for each component. It traverses the ICFG to create a list of all call sites that (directly) call an I/O function (see §4.1), then traverses the call graph to determine which procedures to prune (see §4.2). It then traverses the ICFG again to create the automaton for each procedure as described in §2.2, minimizes each of them, and combines them into our model of the program.

During the compatibility phase, PCCA reads the automaton produced for each component and proceeds with the compatibility check according to PCCA's mode. For the VPA-mode, PCCA actually uses a formalism called Nested-Word Automata (NWAs) instead of VPAs, but each is essentially an alternative expression of the other [3]. We use an extension to the WALi library that implements NWAs [6].

4.1 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. 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 to a particular procedure.
4. 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 call sites in that list, or even generate it by different means. (In fact, in the current version of PCCA, the techniques described in items 1 and 2 are implemented by one program, and the technique described in item 3 is implemented by a second program.)

4.2 Removing Irrelevant Procedures

To reduce the size of the inferred VPA, PCCA prunes procedures that cannot possibly participate in I/O operations. If there is no path 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. One of the first steps of PCCA is to traverse the call graph generated by CodeSurfer, determine which procedures can transitively call an I/O function, and ignore all others. As illustrated in columns 3 and 4 of Fig. 7 (see §5), the effect of pruning is substantial, reducing the number of procedures by as much as 90%.

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. 7). The experiments were run on a system with dual quad-core, 2.27GHz Xeon E5520s processors; however, PCCA is entirely single-threaded. The system has 12 GB of memory, and runs Red Hat Enterprise Linux 5.

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. We also compared the results between the FA and VPA-based modes of operation to determine whether the potential benefits discussed in §3.3 arose.

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 that read and write the same format.

The examples are as follows:[5]

- *ex-prod*/*ex-cons* make up our running example (stubs for the I/O functions are included in the count),
- *ex-prod*/*ex-cons-fig3* uses the buggy version of the consumer presented in Fig. 3,
- *ex-prod-§2.5*/*ex-cons-§2.5* are buggy versions of the running example, modified as described at the end of §2.5 with the separator to mark the packets,
- *gzip* and *bzip2* are the common Unix compression/decompression utilities,
- *gzip-fixed* uses a modified version of *gzip* (discussed below) to eliminate an erroneous report,
- *png2ico* is an image-conversion program, which we compare to a hand-written specification.

Reported times are the median of 5 runs. The numbers for the FA version use NWAs with no call or return transitions. This gives an apples-to-apples comparison with NWAs, but is slower than an alternative implementation that converts each NWA to an OpenFST acceptor, determinizes with OpenFST, and converts back. All times are less than 1 sec. with the latter approach. There *is* an intrinsic cost to using an NWA representation, but we feel that most of the difference between our FA numbers and OpenFST's indicates room for improving the WALi implementation. (That would improve the NWA version as well.)

We also performed an informal experiment using the VPA version without Enrich (as mentioned at the end of §2.3). We tested programs that read and write trees in infix and prefix notation. Both the standard VPA version of PCCA and the no-Enrich version reported that the infix components are compatible with each other, that the prefix components are compatible with each other, and that each is incompatible with the other. As discussed in §2.3, the compatibility results are more credible for the no-Enrich version; the incompatibility results are more credible for the standard VPA version.

Omitting the Enrich step also dramatically decreased determinization time; even the *gzip-fix-cons* could be determinized in less than one second. Thus, it might be beneficial to try to combine enrichment and determinization.

Two of the tests, *gzip* and *png2ico*, required relatively minor modifications. *gzip* uses input and output operations

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

Test	LOC	#Funcs Orig.	#Funcs Pruned	$\|Q\|$	#I/O	Infer aut.	VPA version (sec.) ¬C	VPA version (sec.) Total	VPA version (sec.) OK?	FA version (sec.) ¬C	FA version (sec.) Total	FA version (sec.) OK?
ex-prod	43	11	3	9	4	2.12	0.35	4.90	Y	0.10	4.76	Y
ex-cons	26	7	2	5	3	2.24						
ex-prod	43	11	3	9	4	2.12	0.16	4.49	N	0.10	4.61	N
ex-cons-fig3	25	7	2	5	3	2.09						
ex-prod-§2.5	43	11	3	10	5	2.29	0.70	4.87	N	0.10	5.09	N
ex-cons-§2.5	25	7	2	5	3	2.40						
gzip-prod	4396	100	17	51	25	26.3	123	177	N*	101	157	N*
gzip-cons	4396	100	24	71	50	27.8						
gzip-prod	4396	100	17	51	25	26.3	583	646	Y	102	156	Y
gzip-fix-cons	4389	100	24	73	51	27.8						
bzip2-prod	5772	121	15	32	8	26.3	47.7	102	Y	47.1	101	Y
bzip2-cons	5772	121	13	29	10	27.4						
png2ico-prod	806	39	1	22	29	9.48	14.4	33.1	Y	0.16	10.1	Y
ico-spec-cons	n/a	n/a	n/a	26	28	n/a						

Figure 7: The experiments. "LOC" is lines of code, "orig." is the number of functions in the program, "pruned" is that number after pruning. $|Q|$ is the number of states in the inferred automaton (equal between the two variants). "# I/O" is the (static) number of calls to I/O functions. "Infer aut." is the time (sec.) to produce the automata for every procedure in the program. (The output of this step is used for both the VPA and FA versions.) For both the VPA and FA version, ¬C is the time (sec.) to determinize and complement the automaton. (Determinizing each procedure's FA is not included in this time, but takes a negligible amount of time in all experiments.) "Total" is the end-to-end time for analysis, including the inference step. "OK?" reports the output of PCCA.

much like those in our running example, except implemented as macros. Because PCCA uses the control-flow graph generated by CodeSurfer/C, these macros are not visible, so we replaced the macro definitions with functions. In addition, *gzip* calls the function that actually performs the compression or decompression through a function pointer. CodeSurfer/C performs points-to analysis, but PCCA does not yet take such indirect calls into account; thus we modified the source to call the function directly. (This is not a fundamental limitation of our technique, though imprecise pointer analysis could lead to further imprecision.) A final modification that applies in a similar manner to both *gzip* and *png2ico* will be described in their respective sections.

As shown in Fig. 7, PCCA reports that some commonly-used programs operate in a correct manner with regard to their I/O behavior, regardless of the automaton model used. PCCA also detects synthetic programming errors in small examples, as shown by the second pair of examples.

As can be seen in the results, the potential VPA benefits did not appear to affect the results of the analysis. (PCCA does report different results for the example in §3.3, but we do not include that experiment in Fig. 7.) This result surprised us, and in the future we plan to look at additional examples to see whether any of them benefit from VPAs.

gzip. The analysis of *gzip* reported a erroneous bug in the distributed version; we examine the issues more closely here. For *gzip*, the actual compressed data appears as just a sequence of bytes, so the compatibility check essentially is testing the compatibility of the code that reads and writes the header and footer. Fig. 8 describes the header format of a *gzip* file. The code that writes this header (in `zip.c`) corresponds very closely to the header format:

```
put_byte(GZIP_MAGIC[0]); /* magic header */
put_byte(GZIP_MAGIC[1]);
put_byte(DEFLATED);      /* compression method */
...
put_byte(flags);         /* general flags */
put_long(time_stamp);
...
put_byte((uch)deflate_flags); /* extra flags */
put_byte(OS_CODE);
```

For this code, PCCA infers the format specified in Fig. 8.

However, the code that *reads* the header is reported to be incompatible; this is a false positive. Unlike the output functions, input is always done one byte at a time:

```
stamp  = (ulg)get_byte();
stamp |= ((ulg)get_byte()) << 8;
stamp |= ((ulg)get_byte()) << 16;
stamp |= ((ulg)get_byte()) << 24;
```

Because the consumer reads the `time_stamp` field as four `byte`s instead of one `long`, it appears incompatible. This is similar to the issue of granularity of types discussed in §2.2.

To address this, we replaced this code (and similar code that reads `long` fields in the footer) with a new `get_long` function. This function can be implemented in terms of four bytewise reads; as long as PCCA is told that `get_long` performs I/O, PCCA will recognize the call as reading a `long`. (In addition to helping PCCA, we feel that the modified code is cleaner: by having the code for reading and writing a `long` in one place, it is easier for the programmer to see that those functions agree, for instance by reading and writing the bytes in the same order. It should even be possible to use our techniques to perform this check as well, by giving different types to each byte in the `long`.)

After making this change, PCCA reports that the programs are compatible. It is unclear why there is such a dramatic difference between the time it takes to determinize each version of the consumer in the VPA version. The input VPAs are of almost identical size and makeup, but it appears that the extra `long` alphabet symbol in the revised version causes

ID1, ID2	Fixed constants; *gzip*'s "magic number"
CM	Compression algorithm
FLG	Flags, as a bitmap
MTIME	The modification time of the original file
XFL	Compression-method-specific flags
OS	ID of the OS where the file was compressed

Figure 8: The specification of *gzip*'s header format. Each field is 1 byte except for MTIME, which is 4.

the determinized VPA to be much bigger (176 states vs. 27). (Note that neither of these automata are minimal; it could be that the extra size in the revised version could be reduced to be more in line with the original version.) The sizes of the two automata in the FA version are much closer.

png2ico. For *png2ico*, we demonstrate a slightly different application of our techniques. Instead of comparing a producer to a consumer, we compare a producer to a manually-crafted specification acting as the consumer. This checks that the producer emits only messages that are allowed by the specification. In the case of *png2ico*, we see that the program indeed appears to conform to the specification.

We manually crafted an automaton that describes the format of an icon file [18] and used that as the consumer. For the ICO format, this was reasonably straightforward and took less than two hours. The automaton allows PCCA to check header information, similar to *gzip* but with a much richer format. An icon file can hold several different images. In addition to a global header (that mainly says how many images there are), there is a directory that gives the offset and other information about each image and a header for the image data itself. We can check all of this, leaving only the raw image data itself appearing as a "meaningless" byte stream. (We cannot check that the image headers actually appear at the correct offsets, however.)

While most of the output from *png2ico* is done through the functions `WriteByte`, `WriteWord`, and `WriteDWord`, there are three places where a raw write is done using `fwrite`. Two of these locations write a sequence of raw bytes of an image to the file. We could reasonably infer just `byte*` for those calls (similar to how the actual compressed data comes across in *gzip*), however we decided to put in a bit of extra effort to obtain higher confidence in the result. The two `fwrite` calls correspond to the "xor mask" and "and mask" of the bitmap. We manually specified that the first `fwrite` call outputs "xor mask" bytes and the second call outputs "and mask" bytes, and required that each bitmap in the icon file contains a sequence of "xor mask" bytes followed by a sequence of "and mask" bytes. However, there is one call to `WriteByte` amongst those writing the "xor mask", so we had to manually change the type of that call to match that of the preceding `fwrite`. (We repeated the experiment but just used `byte*`, and PCCA still reported compatibility.)

The third call to `fwrite` is used instead of a sequence of four `WriteByte` calls; the reason the author choose this is not clear. We replaced this `fwrite` with the four individual `WriteByte` calls. In the future we hope to implement the ability to automatically break apart a write such as this, to make such manual intervention unnecessary.

We only report the results for the version with specific types. The other variants we tried did not have much effect.

6. RELATED WORK

Inferring Input or Output Formats of Programs. PCCA's format-inference techniques, as well as the problem in general, was inspired by the File Format Extractor tool (FFE) by Lim et al. [23]. FFE infers output models of x86 executables using a weighted pushdown system. We skip the WPDS step, performing the procedure discussed in §2.2 instead. The minimization we perform on each procedure's model gets us the benefits intended by FFE's use of WPDSs, and produces far smaller automata.

Inferring input formats of executables has received much attention lately, particularly in the context of protocol reverse engineering for network security [7, 12, 34, 24, 25]. 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 weakly-typed languages, such as Cobol [21]. 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 [30] 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; they established that if I conforms to S and P is any environment in which P and S cannot starve waiting to send or receive messages, then P and I also cannot starve.

A related question is checking conformance of software components as software evolves and components are replaced or upgraded. Clarke et al. [9] survey several approaches that have been devised to answer the question, including interface automata, behavioral subtyping [26], input/output-based compatibility of upgrades [27], and model checking.

There have also been many papers on session types, starting with [17, 15, 29]. In some sense, this body of work has the same goal that we have—helping to ensure that different components communicate properly—but their approach is far different. Session types, at a high level, convey much the same information as our inferred languages. (For instance, in the syntax of [17], "`↑int; (↑char & ↑double)`" is the type of a component that emits an `int` followed by either a `char` or `double`.) Some recent work, e.g. [19], is integrating session types into common programming languages.

In most of this literature, session types need to be incorporated into the language being used to write the components, which means they cannot be applied to legacy software without rewriting it. In contrast to these papers, our work analyzes existing C/C++ code for compatibility by inferring the format. In return for this re-engineering, session types support richer interactions than we currently do; most notably, it can specify bidirectional communications.

Recently there have been advances in inferring session types, which is much closer to our goal. Mezzina [28] and

Collingbourne and Kelly [11] each developed such an algorithm. Collingbourne's is particularly related, as they implemented their technique in a source-to-source translator for C++. However, neither paper really gives enough information on how it performs in practice to compare to PCCA.

In a similar vein – and actually of equivalent power – are channel contracts from the Singularity OS [13]. Channel contracts specify a protocol between two endpoints as a state machine, where each state specifies messages that each endpoint can send or receive. Fähndrich et al. describe an analysis that verifies certain memory-safety properties in programs that use channel contracts. More recent work has analyzed channel contracts with respect to deadlocks [33] and developed formal type theories for channels [4].

7. FUTURE WORK

We have a number of extensions to the basic idea described in this paper in mind; we describe two of them here. The discussion in this paper is framed from the point of view that the VPA 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. To do this, we would use types that do not correspond to those in C. 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). (§2.5 described how a particular way of using a phony I/O call to help PCCA detect bugs; this idea expands that technique.) It should 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). This would require a change to the compatibility portion as well.

Second, there are some engineering tasks that should make it easier to get more helpful results. For example, we can implement the technique mentioned in the remark at the end of §2.2, where we make it possible to break apart automatically a structure or array type into its component fields. This would allow the producer and consumer to use data structures that are organized differently from one another but have the same semantic meaning, and would avoid the most invasive changes we had to make in our experiments.

8. REFERENCES

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *CIAA*, 2007.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.

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

[4] V. Bono, C. Messa, and L. Padovani. Typing copyless message passing. In *ESOP*, 2011.

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

[6] A. Burton, A. Thakur, E. Driscoll, , and T. Reps. WALi: Nested-word automata. TR-1675, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, July 2010.

[7] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *CCS*, 2007.

[8] D. Caucal. On the regular structure of prefix rewriting. In *CAAP*, 1990.

[9] E. Clarke, N. Sharygina, and N. Sinha. Program compatibility approaches. In *FMCO*, 2005.

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

[11] P. Collingbourne and P. Kelly. Inference of session types from control flow. *ENTCS*, 238(6), 2010.

[12] W. Cui, M. Peinado, K. Chen, H. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *CCS*, 2008.

[13] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys*. 2006.

[14] A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *ENTCS*, 9, 1997.

[15] S. Gay, V. Vasconcelos, and A. Ravara. Session types for inter-process communication. TR-2003-133, Dept. of Computing Sci., Univ. of Glasgow, March 2003.

[16] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI*, 2002.

[17] K. Honda. Types for dyadic interaction. In *CONCUR*. 1993.

[18] J. Hornick. Icons. `http://msdn.microsoft.com/en-us/library/ms997538.aspx`.

[19] R. Hu, D. Kouzapas, O. Pernet, N. Yoshida, and K. Honda. Type-safe eventful sessions in java. In *ECOOP*. 2010.

[20] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.

[21] R. Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *WCRE*, 2007.

[22] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *OSDI*, 2006.

[23] J. Lim, T. Reps, and B. Liblit. Extracting output formats from executables. In *WCRE*, 2006.

[24] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, 2008.

[25] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *FSE*, 2008.

[26] 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.

[27] S. McCamant and M. Ernst. Early identification of incompatibilities in multicomponent upgrades. In *ECOOP*, 2004.

[28] L. Mezzina. How to infer finite session types in a calculus of services and sessions. In D. Lea and G. Zavattaro, editors, *Coordination Models and Languages*. 2008.

[29] O. Nierstrasz and M. Papathomas. Viewing object as patterns of communicating agents. In *OOPSLA*, 1990.

[30] S. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *CAV*, 2002.

[31] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[32] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, TUM, Munich, Germany, July 2002.

[33] Z. Stengel and T. Bultan. Analyzing Singluarity channel contracts. In *ISSTA*. 2009.

[34] G. Wondracek, P. Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *NDSS*, 2008.