# Software Component Protocol Inference

(General Examination Report)

Tao Xie
Department of Computer Science and Engineering
University of Washington
taoxie@cs.washington.edu

30 May, 2003

## Abstract

Component-based software development has increasingly gained popularity in industry. While correct component usage is critical to successful reuse of components, the expected component usage is rarely specified explicitly. To address this issue, one recent area of research has been to infer specifications of protocols or sequencing constraints using both static and dynamic techniques. This paper explores the research area of software component protocol inference with a focus on dynamic inference techniques. A framework is proposed to compare the existing dynamic inference techniques. Along with the framework, some static inference techniques are covered in brief discussions. In the end, directions for future work are suggested to push the state of the art forward.

## 1. Introduction

Component-based software development has become an emerging discipline that manages the growing complexity of software systems [Szy98]. In component-based software development, software components are the building blocks of a software system. According to Szyperski [Szy98], "a software component is defined as a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". It is necessary for a component to have its interfaces explicitly specified. These interfaces describe services that the component provides to and requests from other components. These services can be in the form of messages exchanged through the component interfaces. Component interface protocols, also called component protocols, are sequencing constraints that a component should obey in message exchanging.

Component protocols can be specified as parts of documentation written for the developers of component client code. Given component protocols, static verification tools can analyze component client code to check whether the component usage conforms to the protocols [DF01][TB01][DLS02][BRBY00]. In addition, the conformance of component client code to the protocols can be checked at runtime [BRBY00]. Despite the fact that component protocols are quite useful, components

are often not equipped with them. Recently research on dynamic and static component protocol inference has attempted to address this issue. Dynamic protocol inference techniques discover protocols from execution traces collected while the component is being used [WML02][ABL02][RR01]. Static protocol inference techniques deduce sequencing constraints by statically analyzing component code [WML02] or component client code [LCED01][ECHCC01][WD01]. In this paper, both inference techniques are discussed. The remainder of this paper is organized as follows: Section 2 presents and compares the existing protocol inference techniques, Section 3 proposes some future work, and Section 4 concludes.

# 2. Component Protocol Inference

This section describes the previous work on component protocol inference. Section 2.1 uses an example to illustrate how a component protocol is specified. Section 2.2 discusses the component protocol inference problems and identifies several technical challenges. Section 2.3 proposes a dynamic inference framework in which the existing dynamic techniques are compared. Section 2.4 briefly discusses two types of static inference techniques.

## 2.1 Component Protocol Specifications

A component protocol is usually bidirectional. It specifies sequencing constraints on the messages that can be both sent and received by a component through an interface [YS97]. However, in some research contexts, a component protocol defines sequencing constraints on only those messages that can be received by a component, but not the ones sent by the component. We term this type of component protocol *unidirectional component protocol*. Unidirectional component protocols are commonly used to specify the correct usage of an Application Programming Interface (API) provided by a component [BRBY00] [WML02]. Conversely, bidirectional component protocols are usually used to specify collaborations between components [CMK98]. Here we primarily discuss components written in Object-Oriented (OO) programming languages, e.g. classes in Java or C++, and unidirectional component protocols for these components.

The notion of protocol originates from path expressions, which are used to synchronize concurrent operations on an object [CH74]. A common way to express protocols is to model them as Finite State Automaton (FSA) [HU79]. There are usually three ways to specify an FSA: a state transition diagram (STD) [YS97], a regular grammar or parser accepting the valid sequences [Flo95], and a regular expression generating the valid sequences [VL91]. Here we use an STD to represent an FSA protocol. In an STD, each edge is labeled by a method call and each node represents the state that the component is in

[ABL02][RR01][CW98]. Whaley et al. [WML02] use an STD to express a component protocol in a slightly different way. In their STD, nodes represent method calls, and transition edges represent "followed-by" sequencing constraints. However, their STD can be easily translated into the one we use. To illustrate the different protocol inference techniques, we will use the class `java.util.zip.zipOutputStream` as an example [Jav03]:

```
public class ZipOutputStream extends DeflaterOutputStream implements ZipConstants {
   public ZipOutputStream(OutputStream out);
   public static final int DEFLATED;
   public static final int STORED;
   public void close() throw IOException;
   public void closeEntry() throw IOException;
   public void finish () throws IOException;
   public void putNextEntry(ZipEntry e) throws IOException;
   public void setComment(String comment);
   public void setLevel(int level);
   public void setMethod(int method);
   public synchronized void write(byte[] b, int off, int len) throws IOException;
}
```

We excerpt the usage description of this class's APIs from the book *Java in a Nutshell* by Flanagan [Fla97] as follows:

> This class is a subclass of `DeflaterOutputStream` that writes data in API file format to an output stream. Before writing any data to the `ZipOutputStream`, you must begin an entry within the ZIP file with `putNextEntry()`. The `ZipEntry` object passed to this method should specify at least a name for the entry. Once you have begun an entry with `putNextEntry()`, you can write the contents of that entry with the `write()` methods. When you reach the end of an entry, you can begin a new one by calling `putNextEntry()` again, or you can close the current entry with `closeEntry()`, or you can close the stream itself with `close()`. Before beginning an entry with `putNextEntry()`, you can set the compression method and level with `setMethod()` and `setLevel()`. The constants `DEFLATED` and `STORED` are the two legal values for `setMethod()`. If you use `STORED`, the entry is stored in the ZIP file without any compression. If you use `DEFLATED`, you can also specify the compression speed/strength tradeoff bypassing a number from 1 to 9 to `setLevel()`.

This description specifies two main types of constraints: sequencing constraints and argument value constraints. The former is expressed by protocol specifications, and the latter by operational specifications in the form of pre/post-conditions [Mey97].

Based on the description, a protocol for this class has been formally specified by Butkevich et al. [BRBY00]. We translate the protocol into an STD, which is shown in Figure 1. In the STD, the starting and ending states are marked with `S` and `E`. The states in the left-top and right-top portions are marked with `DEFLATED` and `STORED` respectively for the purpose of illustration. In order to distinguish the method calls of `setMethod()`that transit the component to different states, we also specify the argument value of `setMethod()`in the end of the method name. Note that neither the protocol specified by Butkevich et al. [BRBY00] nor a standard FSA protocol distinguishes these `setMethod()`'s.

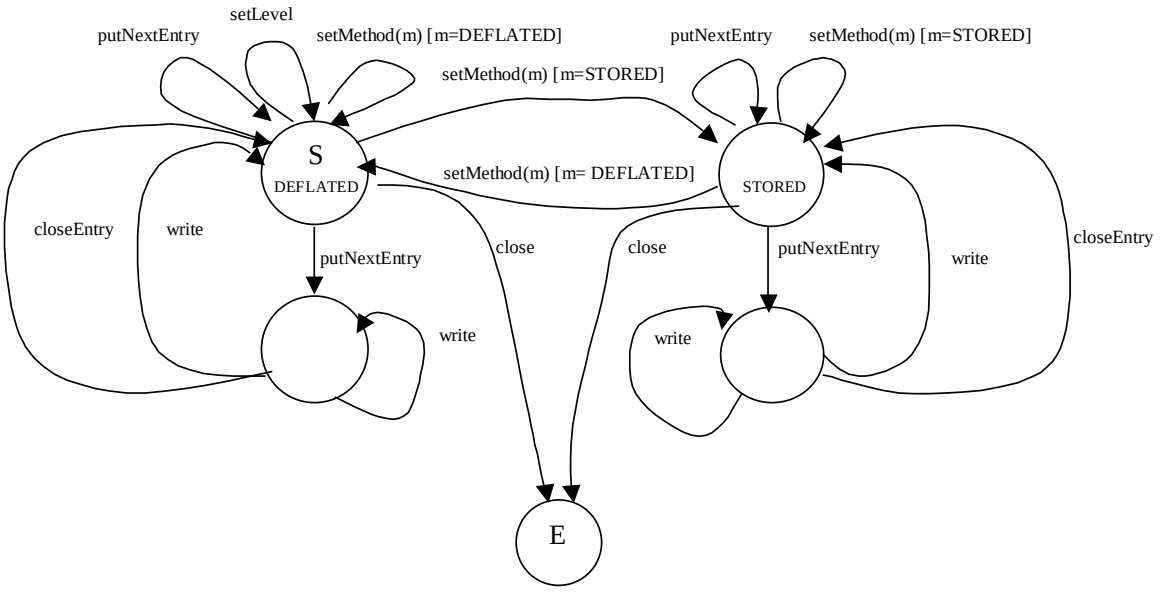



**Figure 1. The protocol for `java.util.zip.zipOutputStream` in STD**

The protocol could be extended to specify the methods `setComment()` and `finish()`, which have not been described in the Flanagan book. There are two potential reasons for the protocol of an interface not to specify a method in the interface. One is that there are no sequencing constraints on the method and thereby leaving it out of the protocol makes the specification succinct. The other is that there are some sequencing constraints on the method, which renders the protocol incomplete. We looked into the source code of `zipOutputStream` and found that no sequencing constraints were imposed on `setComment()` but there was a sequencing constraint between `putNextEntry()` and `finish()`.

When an FSA protocol captures all the legal method sequences in an interface, any method call that does not follow the transitions in the FSA is considered as protocol violation. However, as we have discussed earlier, an FSA protocol can leave out the method calls that no sequencing constraints are imposed on, e.g. `setComment()`. To accommodate this, if a method in a component interface does not occur on any transition edge of the FSA protocol, it is legal to call this method at whatever state the component is in. The underlying assumption is that calling this method does not change the state of the component [BRBY00].

## 2.2 Protocol Inference Problem

Although component protocols are rather useful in assuring the correct component usage, few components have accompanying protocol specifications. The usage of a component interface is usually documented informally in a natural

language, such as the one for `ZipOutputStream` in the Flanagan book. Moreover, even when developers are willing to write down the formal specifications, they may make mistakes by making wrong assumptions or for being careless. We found that the formal protocol specifications written by Butkevich et al. [BRBY00] mistakenly put the `setLevel()` transition in the `STORED` state. To tackle these problems, there have been a variety of attempts to infer protocol specifications dynamically or statically. The dynamic protocol inference problem is described as follows:

*Given a method set* M, *which contains all the methods in a component interface* I, *and the traces of* M *collected while they are used, infer the sequencing constraints on* M *in the form of FSA.*

Similar to the process discovery problem described by Cook et al. [CW98], the protocol inference problem can be cast in terms of a previously investigated *grammar inference* problem. The grammar inference problem is to discover a grammar for a regular language provided with sample sentences or sequences in that language [AS83]. There have been various algorithms on inferring a grammar or an FSA given some positive and negative sequences. Positive sequences are those sequences that are valid whereas negative sequences are those that are invalid. Previous research has shown that it is very difficult to learn the best grammar for a regular language from both positive and negative sequences, and that it is much more difficult to learn from only positive sequences [Gol78]. In the dynamic protocol inference problem, the traces of *M* can be collected while *M* is used correctly or incorrectly. In previous work, the traces are usually collected from running systems that supposedly exercise the correct usage of *M*. Indeed, these traces may contain a few invalid sequences caused by some faults in the systems, but these faults are unidentified before the inference is performed.

The static protocol inference problem is described as follows:

*Given a method set* M, *which contains all the methods in a component interface* I, *and the component code or the component client code that uses* M, *infer the sequencing constraints on* M *in the form of FSA.*

The static protocol inference problem receives the component code or the component client code as inputs. The component code can be used to infer sequencing constraints if the component is defensively programmed to detect illegal sequence of method calls. The component client code can provide some static information on how *M* is used, as opposed to the dynamic information provided by the traces of *M* collected from client code executions.

We have identified the following four technical challenges for the protocol inference problems:

**Separation/composition of constraints:** Not all the methods in *M* are closely related with respect to sequencing constraints. Sometimes there are no sequencing constraints being imposed on some methods in *M*, such as `setComment()` in `zipOutputStream`. Sometimes sequencing constraints on *M* are conceptually separated into several groups, each of which operates on a particular type of scenario. For example, the `zipOutputStream` protocol is divided into two groups: `DEFLATED` and `STORED`, respectively corresponding to the left and right portions of Figure 1. Sometimes the separated groups are combined by some additional sequencing constraints. For example, the transitions of `setMethod()` connect these two groups. Sometimes the same method may occur in multiple groups, such as most methods in `zipOutputStream`. Sometimes the method calls from multiple groups can be intermingled in any order as long as they obey the sequencing constraints in their own group. Inferring a protocol with one single FSA for them makes the protocol inaccurate or too complicated, which is analogous to using a single FSA to model multiple concurrent FSAs [Har87]. During protocol inference, it is challenging to find an appropriate way to separate and compose the sequencing constraints.

**Data-dependent transitions:** The standard FSA deals with events only. No data is associated with events or the FSA. Sometimes this is insufficient in specifying the component protocol. For example, considering the argument value of `setMethod()` is important for specifying accurate state transitions. Extended Finite State Machine (EFSM) allows data variables associated with an FSA [Hol91]. A guard condition on data variables or method parameters can also be on a transition. If the method whose name is associated with the transition is called and the condition becomes true, then the transition may be taken. In the presence of data-dependent transitions, the protocol inference problem becomes more difficult.

**Overgeneralization/over-restrictiveness**: An inferred FSA protocol is accurate if it accepts all legal sequences and rejects all illegal sequences. If an inferred FSA protocol accepts some illegal sequences, it is called overgeneralization. For example, usually the previous dynamic inference techniques cannot separate the `DEFLATED` and `STORED` states. Then an illegal sequence, a method `setLevel()`being called immediately after `setMethod(STORED)`, is accepted by the inferred FSA protocol. When only positive sample sequences are given to an inference algorithm, the algorithm has no way to determine when it is overgeneralizing [CW98]. On the other hand, if an inferred FSA protocol rejects some legal sequences, it is called over-restrictiveness. This is an inherent problem for dynamic inference techniques.

**Robustness to noise:** It is possible that traces or component client code with noise is given to inference algorithms. Noise may include some illegal method sequences due to some faults in the component client. It is equally conceivable that noise arises when some collected method calls do not involve any sequencing constraints. For example, `setComment()` method calls are noise in traces of the `zipOutputStream` class. It is challenging for algorithms to infer a protocol satisfactorily in the presence of noise.

## 2.3 Overview of Previous Work

**Table 1. Summary of previous work**

| Previous work | Target language/system | Analysis type | Results |
|---|---|---|---|
| Whaley et al. [WML02] | Java | Static/Dynamic | FSA |
| Reiss et al. [RR01] | Java/C++ /C | Dynamic | FSA |
| Ammons et al. [ABL02] | C | Dynamic | FSA |
| Cook et al. [CW98] | Software process | Dynamic | FSA |
| El-Ramly et al. [ESS02] | Interactive system | Dynamic | Frequently recurring usage patterns |
| Lie et al. [LCED01] | C protocol code | Static | Models to model checkers |

Previous related work is summarized in Table 1. Whaley et al. [WML02] use both static and dynamic techniques to extract multiple FSA submodels for the interface of a Java class. Reiss et al. [RR01] employ dynamic techniques to encode program execution traces in a variety of forms including FSA. By observing how C programs interact with an API or abstraction datatype, Ammons et al. [ABL02] use dynamic techniques to infer frequent interaction patterns as an FSA. The above three lines of work are directly tackling the component protocol inference problems.

Although some lines of work do not directly aim at the component protocol inference problems, their inference techniques can be borrowed to tackle the problems. For instance, Cook et al. [CW98] apply dynamic techniques to discover an FSA model of the software process behavior from process events. El-Ramly et al. [ESS02] use dynamic techniques to discover frequently recurring patterns from traces of system-user interactions. Lie et al. [LCED01] resort to static techniques to extract protocol models from C implementation, which are checked with a model checker.

## 2.4 Dynamic Protocol Inference Framework

We propose a dynamic protocol inference framework to compare the previous dynamic protocol inference techniques. The dynamic protocol inference is divided to four phases: trace collection, scenario extraction, protocol inference, and protocol usage. In the trace collection phase, specific types of trace data for the methods in a component interface are collected while

systems are running. In the scenario extraction phase, the collected trace data are grouped into a set of scenarios, each of which is composed of interdependent method calls. In the protocol inference phase, common patterns among the scenarios (i.e. protocols) are inferred. In the protocol usage phase, the inferred protocols are used to aid a variety of activities. The summary of the previous work within this framework is presented in Table 2.

**Table 2. Summary of dynamic inference work within the framework**

| Previous work | Trace collection | Scenario extraction | Protocol inference | Protocol usage |
|---|---|---|---|---|
| Whaley et al. [WML02] | Method calls, Class/Object IDs | Object-based, Slicing by member fields | Separation of state-modifying and state-preserving methods | Test suite characterization, Software auditing |
| Reiss et al. [RR01] | Method calls, Class/Object IDs | Object-based | *k*-tails algorithm | n/a |
| Ammons et al. [ABL02] | Method calls, Argument/return values | Flow dependence, Simplification/standardization | sk-strings algorithm | Trace verification, Specification debugging |
| Cook et al. [CW98] | Process events | n/a | RNet algorithm<br>*k*-tails algorithm<br>Markov algorithm | Process validation |
| El-Ramly et al. [ESS02] | Screen IDs | Interaction-based | IPM2 algorithm | Legacy system reengineering |

In this table, the trace collection column shows the types of collected data traces. The scenario extraction column shows the criteria used to group the traces into scenarios. The protocol inference column shows the names of the inference algorithms or the used key techniques. The protocol usage column shows the activities that use the inferred protocols. The detailed descriptions of these four phases for the previous work are presented in Sections 2.4.1-2.4.4.

Like other dynamic analysis techniques, dynamic protocol inference techniques can be performed either *online* or *offline*. An online technique performs the first three phases while the system under analysis is still running, and inferred protocols are available when the system terminates. An offline technique usually does the postmortem analysis on the collected traces after the system has terminated. Whaley et al. use an online technique but the other four lines of research adopt offline techniques. This online technique is more scalable to large systems since it does not need to store any intermediate traces. In addition, its protocol inference algorithm is so simple that the overhead of runtime analysis is very low.

### *2.4.1 Trace Collection*

In trace collection, different types of trace data can be collected, and different collection mechanisms can be used. Since the key elements in a component protocol are the method calls that can be invoked on the component, these method calls are collected as a source to infer the protocol. In fact, a method call can be seen as the interval between two events: the method

entry and exit events. The traces of a method call can be recorded at the method entry point, the method exit point, or both. There can be several attributes for a method call. One of them is the signature of the method being called. Another one is the sequencing information delivered by the order in which the method call is recorded. If we record the time when the method call occurs, we can also derive the sequencing information from it. All previous dynamic inference techniques record both the signature and the sequencing information of a method call. In addition, there are a couple of other attributes for a method call that can be recorded:

- Class ID and Object ID (for OO languages): The collected Class IDs and Object IDs for method calls can help to group them by object. This grouping operation is to be performed in the scenario extraction phase.
- Method arguments and return values: We cannot group by object the calls of methods written in non-OO languages, since there are no explicit objects associated with them. Instead, the collected arguments and return values for method calls can be used to group them based on their data dependency. In OO languages, the fields in a class can be seen as implicit arguments and return values for each method in the class. The class field values can also be collected for a method call. The collected class field values and arguments for method calls can be used to infer data-dependent transitions in a protocol.

For a method call, Reiss et al. collect class ID and object ID but no arguments or return values. Whaley et al. do not explicitly collect raw trace data but track each object of a specific class individually – in other words – they implicitly keep track of the class ID and object ID of a method call. Whaley et al. do not track arguments or return values for a method call, either. Ammons et al. collect arguments and return values to group C method calls. Cook et al. and El-Ramly et al. respectively collect process events and screen IDs, whose counterparts are method calls in program traces.

There are three common ways to collect dynamic information from Java program executions: instrumenting source code, instrumenting bytecode, or using the standard Java Virtual Machine Profiling Interface (JVMPI). The last way collects dynamic information from the environment where a program is running. Whaley et al. use the Byte Code Engineering Library [Dahm00] to instrument Java bytecode and Reiss et al. use the JVMPI to collect traces. Similarly, there are two common ways to collect traces from C/C++ program executions: instrumenting source code or instrumenting executable. Reiss et al. patch executables to insert hooks, which are used to generate trace files. By using EEL Executable Editing Library [LS95], Ammons et al. edit executables to replace the calls to an original API method with the calls to a wrapper for the API method.

Either the component code or client code can be instrumented to collect traces on how a component interface is used. The component code is instrumented by Whaley et al., and the client code by Ammons et al. Instrumenting the component code "does it once for all (clients)" without touching the client code. On the other hand, instrumenting the client code has a better control of the instrumentation scope without requiring the availability of the component code. For example, if some places in a system do not correctly use a component interface, we can leave these places out of the instrumentation scope. However, in the client code, it is not easy to statically identify all the call sites to the component interface in the presence of function pointers, dynamic binding, or polymorphism. To address this problem, more recent work by Ammon uses a dynamic linker to redirect API calls to wrappers without editing the executable [Amm03].

When traces of a component interface are collected by instrumenting the component code or using the JVMPI, the internal usage of the component shall be filtered out. For example, in the method body of `putNextEntry()`, if the current entry is not closed, then `closeEntry()` is called. This `closeEntry()` method call is internal usage, which is not invoked by component clients through the interface. To identify and ignore internal usage of the object, Whaley et al. maintain knowledge of the local call stack whereas Reiss et al. post-process the collected traces.

### *2.4.2 Scenario Extraction*

A component usage scenario consists of interdependent method calls to a component interface. Given the collected traces, scenario extraction needs to determine which method calls to group together, which to put separately, and which to ignore completely. A common way to split the traces into scenarios is to group the method calls by object, since only method calls on the same object are considered to be interdependent. For example, a method call of `write()` on an object of `zipOutputStream` may be dependent on a method call of `putNextEntry()` on the same object, but is independent of the one on a different object. Reiss et al. and Whaley et al. group the Java or C++ method calls on an object together. However, method calls in C program traces cannot be grouped in the same way, since there are no explicit objects or instances of library APIs. Instead, we need to consider some specific arguments and return values of method calls to identify scenarios. For example, in the file interface of C library, a pointer to a FILE data structure is returned by a `fopen()` function and then can be passed to the function `fprintf()`, `fscanf()` or `fclose()` as the first argument and the function `fread()` or `fwrite()`as the fourth argument. In order to group method calls into scenarios, we can specify that the return value of `fopen()` is linked to the first argument of `fprint()`, `fscanf()`or `fclose()` and the fourth argument of `fread()` or

`fwrite()`. Then all those method calls related to the same `fopen()` are grouped as a scenario. Reiss et al. briefly describe the above approach but do not present its details or implementation.

Ammons et al. also develop a set of techniques to extract scenarios from C program traces. At first, users specify a list of attributes (method arguments or returns) that define and use an abstract object, where defining an object means changing its state. When there are multiple types of abstract objects, users can specify one list of attributes for each type. For example, the return of `fopen()` and the first argument of `fclose()` define a file object. The first argument of `fprintf()`, `fscanf()` and `fclose()`, and the fourth argument of `fread()` and `fwrite()` use a file object. Then given the list of attributes, a flow dependence analysis, is performed on the traces. This analysis is a dynamic version of the reaching definitions problem. For example, the flow dependence analysis can infer the flow dependence graph in Figure 2 for the sequence of

`fopen():return=0x4000120,fprintf(fp=0x4000120),fscanf(fp=0x4000120),fclose(fp=0x4000120).`

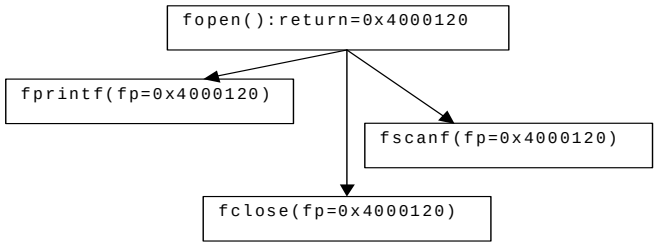



**Figure 2. Flow dependence for a method call sequence**

Users supply a set of scenario seeds, each of which is a function signature. Then a scenario is extracted around each function call in the traces that matches a scenario seed. Given a user-tunable parameter *N*, each scenario contains at most *N* ancestors and at most *N* descendants of the seed function call, where ancestors and descendants are defined along the chain of flow dependent interactions. The scenario extractor further simplifies and standardizes the scenarios before passing them to the protocol inference algorithm. Simplification eliminates the arguments or return values that do not carry flow dependence in any traces. Standardization converts a scenario into a scenario string by replacing the remaining arguments or return values with symbolic variables, as well as by reordering the function calls. The function calls in a scenario are partially ordered by flow, anti, and output dependences, which means a scenario corresponds to a directed acyclic graph (DAG). Reordering puts two scenarios with the same DAG into the same total order.

In traces of OO programs, each method call is explicitly associated with an object, which corresponds to an abstract object defined by Ammons et al. for C programs. However, only using objects to group method calls into scenarios is too

coarse grained. A subset of the method calls on an object might be relatively independent of another subset of the method calls on the same object. Whaley et al. provide a fine-grained mechanism, called model slicing by member fields, to extract scenarios from Java program traces. Instead of inferring one single FSA model, Whaley et al. infer *n* FSA submodels for a class that has *n* fields. Each submodel contains the methods that access the same field. If a method accesses multiple fields, it can be contained in multiple submodels. In a submodel, the method calls of the same object are grouped into a scenario. The rationale behind this mechanism is that if methods do not refer to the same variable, they are unrelated and their relative ordering is independent of each other. For example, the FSA submodel that access the `entry` field of `ZipOutputStream` only contains the methods of `putNextEntry()`, `closeEntry()`, `finish()` and `write()`. So these method calls on the same object are grouped into a scenario while keeping their relative ordering.

An event stream represents a sequence of events. It is one execution of a process corresponding to a sequence of method calls on an object in program traces. Cook et al. use one event stream, instead of multiple event streams, to infer an FSA model of the process behavior. Yet their inference techniques can also be applied to multiple event streams. El-Ramly et al. group screen IDs into multiple scenarios, each of which records a trace of interaction between the system user interface and one of its users.

### *2.4.3 Protocol Inference*

Given a set of scenarios, protocol inference algorithms infer frequent method call patterns as a protocol, usually in the form of FSA. The protocol inference can be treated as a learning activity. Provided with some observations (scenarios), the learner (protocol inference tool) finds a general rule (protocol), which explains the given observations (scenarios) and predicts future observations (scenarios). There have been a variety of FSA learning algorithms, one of which is the *k*-tails algorithm [BF72]. Reiss et al, Ammons et al. and Cook et al. use variants of this algorithm to infer FSA protocols. The key notion of *k*-tails algorithm is that a state is defined by what future behavior can ensue. If two different histories have the same future behavior, then they are put in the same state in the FSA. The future, called the *k*-tail, is defined as the next *k* method calls, where *k* is a parameter to the algorithm. Reiss et al. merge two states if they have a *k*-tail in common whereas Cook et al. merge two states if one includes all the *k*-tails of the other one. Since fewer states are merged by Cook et al., their inferred FSA is larger, in the company of stricter constraints, than the one inferred by Reiss et al..

For example, by running a program using `zipOutputStream` APIs with different inputs [Dar98], we can generate two sample traces of API calls (Figure 3). The methods in the same trace are invoked on the same object, and the methods from

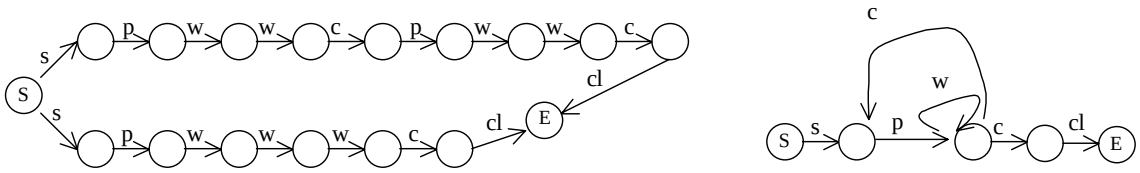
different traces are invoked on different objects. By grouping the scenarios based on object, we can get two scenarios, which correspond exactly to the two traces.

```
setMethod(),putNextEntry(),write(),write(),closeEntry(),putNextEntry(),write(),write(),closeEntry(),close()

setMethod(),putNextEntry(),write(),write(),write(),closeEntry(),close()
```

**Figure 3. Two sample traces of `zipOutputStream` usage**

To simplify the representation, we use the first letter of each method name to represent the method call except that we use `cl` to represent `close()` in order to distinguish it from `closeEntry()`. We can first construct an initial FSA to accept only these two scenarios, as is shown in Figure 4.a. The initial FSA uses a starting state and an ending state to connect these two sequences. And then we can merge the states that have a 2-tail in common recursively as Reiss et al. do. The final inferred FSA is shown in Figure 4.b.




a. Initial FSA

b. FSA generated by 2-tails algorithm

**Figure 4. An example of *k*-tail algorithm (k=2)**

Both Reiss et al. and Cook et al. enhance the *k*-tails algorithm to better handle loops. Moreover, Cook et al. enhance the algorithm to handle noise in the scenarios. If noise is assumed to be method calls with low frequency, then a state with very few members in the final FSA may indicate a place where the *k*-tail method call sequences are erroneous. Therefore, the states whose members are fewer than a user-specified threshold could be thrown away. Ammons et al. use an off-the-shelf learner of Probabilistic Finite State Automaton (PFSA) [RP97]. PFSA is a nondeterministic finite automaton in which each edge is weighted by how often the edge is traversed while accepting sequences. The learner is a variation of the *k*-tails algorithm. To handle noise, Ammons et al. remove the edges with low frequency, whereas Cook et al. remove the states with low frequency.

Cook et al. also augment the RNet algorithm [DM94], a statistical approach based on neural network, to infer FSAs. But they found that the RNet algorithm is not sufficiently mature to be used in the process discovery problem. In addition, they develop the Markov algorithm, which looks at the neighboring past and future behavior to define a state. It assumes that the probability of a process being in some state only depends on the previous state that the process was in. To handle noise, it removes the sequences whose probability and number of occurrences are below user-specified thresholds. Then an FSA is

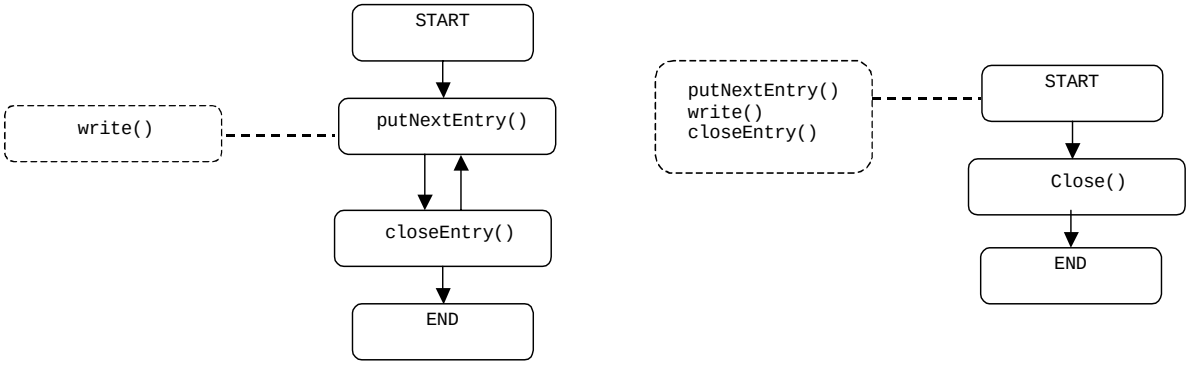
generated from the remaining event sequences based on their probabilities. Their evaluations show that the Markov algorithm is more robust than the RNet or *k*-tails algorithm in the presence of noise.

Different from other work using complicated inference algorithms, Whaley et al. use a simple protocol inference algorithm. It first determines the state-modifying methods for a submodel. State-modifying methods are those methods that write the field associated with the submodel. Calls of the state-modifying methods advance the state of an object, and they are represented by state-modifying transitions in the FSA submodel. Similarly, the algorithm determines the state-preserving methods for the submodel, which are those methods that only read the field. Calls of a state-preserving method do not advance the state of an object, but they may still have sequencing constraints. These calls are represented by state-preserving transitions in the FSA submodel. For a state-preserving method, its transitions define the set of states for which calling the method is legal.

At runtime, for a submodel, each object keeps track of the last state-modifying method that was called on that object. At object construction time, the last state-modifying method is initialized to be a special `START` method. At object destruction time, a special `END` method, treated as a state-modifying method, is hypothetically called. Assume at one point, the last state-modifying method for a submodel on an object is *m*. If a state-preserving method *n* for the submodel is called on the object, a state-preserving transition from the *m* to *n* is added to the submodel. If a state-modifying method *m'* for the submodel is called on the object, a state-modifying transition from *m* to *m'* is added to the submodel. At the same time, the last-modifying method is updated to be *m'*.

For example, after the program executions that generate the traces in Figure 3, the dynamically extracted submodel for the `entry` field is shown in Figure 5.a. In this figure, state-preserving and state-modifying transitions are represented by dotted lines and solid arrow lines respectively. The submodel can only contain the methods of `putNextEntry()`, `closeEntry()`, `finish()` and `write()`, in which only `write()` is a state-preserving method. The history of the last state-modifying method for the first trace is `putNextEntry()`, `closeEntry()`, `putNextEntry()` and `closeEntry()`, and the history for the second trace is `putNextEntry()` and `closeEntry()`. The inferred submodel shows that `write()` can only be called after a `putNextEntry()`is called. The dynamically extracted submodel for the `closed` field is shown in Figure 5.b. The submodel can only contain the methods of `putNextEntry()`, `closeEntry()`, `finish()`, `write()` and `close()`, in which only `close()` is a state-modifying method. The inferred submodel shows that after calling `close()`, none of `putNextEntry()`, `closeEntry()`, `finish()`or `write()` is allowed to be called. Note that in a FSA submodel inferred by

Whaley et al., a state-modifying method can occur at only one place, whereas any method can occur at multiple places in a FSA inferred by other dynamic inference techniques.




(a). Extracted submodel for the `entry` field

(b). Extracted submodel for the `closed` field

**Figure 5. Dynamically extracted submodels for the `entry` and `closed` field in `zipOutputStream`**

El-Ramly et al. develop the IPM2 algorithm to discover frequently occurring interaction patterns. Given a set of scenarios and some parameters, it outputs all maximal qualified patterns in the scenarios, each of which is a sequence. The algorithm is robust to noise. For example, given certain parameters, the algorithm can find two patterns for two scenarios: 1,3,2,3,4,3 and 2,3,2,4,1,3, where the numbers represent screen IDs. The first pattern is 2,3,4 and the second one is 3,2,4,3.

In sum, Whaley et al. effectively separate the constraints. It is achieved by slicing model by member fields in the scenario extraction phase, and by separating the state-modifying and state-preserving methods in the protocol inference phase. However, they do not combine the separated submodels by making connections between them. On the other hand, Reiss et al., Ammons et al., and Cook et al. can infer a single global view of sequencing constraints, but all of them lack good mechanisms to separate the constraints. In addition, only Ammons et al., Cook et al., and El-Ramly et al. provide some mechanisms to handle noise in protocol inference.

### *2.4.4 Protocol Usage*

The inferred protocols can be used in a variety of activities. Sometimes the whole inferred protocols are provided for human inspection. Whaley et al. use the inferred protocol to characterize test suites and understand the system under test. By inspecting the inferred protocols, a programmer who is not familiar with the system can gain deeper understanding of the code. Developers can use the inferred protocols as the starting point to construct a complete protocol specification. In addition, several protocol inference algorithms in the previous work require user-specified parameters. By inspecting the inferred

protocols, users can tune the parameters and rerun the algorithms to improve the results. Reiss et al. tuned different values for the parameter *k* in their *k*-tails algorithm by inspecting the results. They found that the value of 3 is a good candidate for *k*.

Sometimes some tool supports are provided to narrow down the scope of the inferred protocols or traces for human inspection. Whaley et al. develop a dynamic protocol model checker to report the protocol discrepancies between the intended and the implemented APIs. This may be useful for finding errors during program evolution. Ammons et al. develop program verification tools to distinguish program traces that satisfy a protocol specification from program traces that do not satisfy the specification. The tools provide the violating traces and the violated specifications for human inspection. Inspecting these traces is a way to debug a specification, but sometimes there may be hundreds of traces. More recent work by Ammons et al. [AMBL03] uses concept analysis [Wil82] to automatically cluster similar traces together. This can help a human to debug a specification more effectively.

Additionally, Cook et al. use the process validation technique to compare the inferred process model with a prescribed model to measure the deviation of the actual behavior from the intended one. El-Ramly et al. sort the discovered interaction patterns so that users can more effectively find out useful interaction patterns. These patterns provide the basis for reengineering the software system into web-accessible components, each of which supports one of the patterns.

## 2.5 Static Protocol Inference Techniques

### *2.5.1 Static Analysis of Component Code*

More and more components are being defensively coded to detect illegal usage of their interfaces. Checks are put into the component code and if a check fails, then an appropriate exception is thrown or an error-indicating return code is returned. In particular, to detect illegal method call sequences, a state value is often implicitly or explicitly encoded in a Java class field and then the validity of the state is checked before performing an operation. For example, in the beginning of the `write()`method body, the `entry` field is checked to see if it is null. If so, an exception is thrown. By looking into other methods' implementations, we found that only `putNextEntry()` assigns a non-null value to the `entry` field and `closeEntry()` assigns a null value to it. It is natural to infer that the call of a `putNextEntry()` must precede the call of a `write()` and no call of `closeEntry()` between them. We can infer, only based on the fact that `closeEntry()` assigns a null value to it, that the call of a `closeEntry()` immediately followed by the call of a `write()` is not allowed. On the basis of this notion, Whaley et al. develop a static protocol inference algorithm. For each method *m*, the algorithm first identifies those fields and

predicates that guard the throws of exceptions. To make the analysis feasible, it restricts the analysis to only consider certain simple predicates. These predicates test a field with a constant null or a constant integer, and the field is not written before being tested. Then the algorithm finds those methods that set these fields to cause the exceptions. It uses a constant propagation analysis on each method to determine whether any of these fields is set to some constant at its exit. An illegal transition from *m* to *m'* is found if *m* sets a constant value to a field, making its value satisfy the exception-throwing condition in *m'*. A protocol can be attained by complementing the illegal transitions with regards to the relevant methods.

### *2.5.2 Static Analysis of Component Client Code*

By analyzing component client code, we can statically attain some component usage information, which can help to infer component protocols. Lie et al. use pattern matching and program slicing techniques to automatically extract model descriptions from code and then feed the extracted models to a model checker. The code analyzed by them implements the cache coherence protocol, which is different from component protocols discussed in this paper. However, their techniques can also be applied in analyzing component client code and extracting component usage from the code. Along this line of research, Wagner et al. [WD01] statically infer a FSA for system calls in application code and use it to detect intrusions. Engler et al. [ECHCC01] statically infer temporal rules from the Linux code to find bugs by assuming the norm is the behavior that is observed most of the time. They analyze the code to find method sequences that instantiate a fixed set of temporal rule templates, as exemplified by "no method call *a* after *b*" and "method call *b* must follow *a*".

# 3. Future Work

In this section, two lines of future work are proposed in the area of component protocol inference to tackle some challenges presented in Section 2.2. The first is to investigate the component protocol inference in the specific context of component testing. The second is to improve the existing component protocol inference techniques in a general context.

## 3.1 Component protocol inference and component testing

The previous dynamic inference work uses traces collected while the component is being used by client systems. Usually the scenarios extracted from them are treated as positive samples for the inference algorithms. In component testing, some test cases are developed to test whether the component correctly throws exceptions when illegal sequences are invoked. We can usually identify a negative sample by observing whether the execution of a test case throws an exception. Therefore, in

component testing, both positive and negative samples can be given to the inference algorithms. This can alleviate the overgeneralization problem of the inferred protocols.

One concern of inferring protocol in component testing is that component test cases may be insufficient to infer actual usage of a component. This is called the test insufficiency problem, which is also faced by other dynamic inference techniques. There are several measures to alleviate this problem. More and more developers adopt the test-first principle, as advocated by the Extreme Programming (XP) development process [Bec00]. This principle requires unit tests or component tests to be constructed and maintained before, during, and after the source code is written. At the same time, we have seen some efforts in writing extensive test suites for the Java class libraries [Mau03]. These trends lead to the expectation that more component tests are to be available, which can alleviate the problem of test insufficiency. At the same time, we propose a way to automatically generate more tests, rather than relying only on the tests written by human. We can borrow the idea of static protocol inference by Whaley et al. to dynamic analysis in component test generation. We can automatically generate test cases to exercise each pair of method calls among the methods accessing the same field. Based on the symptom of exception throwing, we can infer the likely sequencing constraints in a way similar to Whaley et al.'s static analysis. This dynamic inference approach can complement their static approach by handling more cases of exception throwing, but in an unsound way. In addition, we propose to use an inferred protocol to guide automatic test data generation for exercising the protocol in a systematic way. By observing whether test executions throw exceptions, we can expose some illegal transitions that are accommodated by the inferred protocol because of its overgeneralization. The traces of these exception-causing tests are added to the inputs to the inference algorithms as negative samples. Then a better protocol can be further inferred, which can be used to guide the test data generation in the next iteration. Iterations go on until no generated tests can throw exceptions. This forms a feedback loop between testing and inferred protocol specifications.

Inferred protocols can be used to characterize test suites. Toward this end, Whaley et al. provide the whole set of inferred protocols for human inspection. We propose to use protocol violations to select automatically generated component tests to augment the existing tests written by humans. We can select those automatically generated tests that violate the inferred protocol. Protocol violations may indicate the insufficiency of the existing tests or errors in the component code. We have done some work on using inferred operational specifications (in the form of pre/postconditions) to aid unit test data selection [XN03]. We propose to use the inferred protocol specifications in the similar way. This work is similar to the work on the

feedback loop between testing and inferred specifications, but does not require that the automatic test generation be based on protocol specifications.

## 3.2 Improvements on component protocol inference

Apart from the improvements discussed in Section 3.1, this section will propose more improvements in a general setting. Whaley et al. propose some techniques to separate constraints by slicing a protocol model by member fields and distinguishing state-preserving and state-modifying methods. In contrast, others present a global view to compose all constraints together. The former lacks good composition of constraints whereas the latter lacks good separation of constraints. We propose an eclectic approach by inferring a hierarchical protocol, which can be specified by using hierarchical concurrent FSAs [Har87]. To attain a hierarchy, concept analysis [Wil82] is to be performed on the methods and the fields they access.

No previous work tackles the data-dependent transition problem. We propose to use some heuristics to tackle this problem. The key issue in tackling this problem is to determine what data are related to a component mode, which is a named group of states. We borrow the notion from the mode definition in SCR requirement specifications [Hen80]. For example, the `method` field of `ZipOutputStream` contains the data value of `DEFLATED` or `STORED`, which determines a component mode. We observed that in the class implementation, there are five conditionals in different methods to compare the `method` field with the value of `DEFLATED` or `STORED`. By statically analyzing the code and using statistics, we can identify this type of fields and their data range. During trace collection, we can collect the data values of method arguments and class fields at method entry and exit by using the Daikon front-end [ECGN01]. The values of relevant fields will help to determine which component mode the component is in before or after calling a method. Later we can infer one submodel for the method calls staying in the same component mode. Transitions between modes can be inferred from those method calls whose entry and exit states are in different modes. Organizing sequencing constraints in component modes can better separate and compose the constraints.

Since different clients may use the same set of methods in a component interface quite differently, we propose to perform cluster analysis on the scenarios before feeding them to the inference algorithms. Each cluster is to be used to infer a protocol. The set of resulting protocols can be used collectively. We also propose to modify the IPM2 algorithm by El-Ramly et al. to infer frequent method call patterns from the scenarios and cluster the scenarios based on these patterns. Michail [Mic99] analyze the source code to discover association rules that identify certain library components. These components are often

reused in combination by clients. We propose to use similar techniques to infer association rules among component APIs, which can guide the clustering of scenarios.

We also identify a new problem for our future work that is related to both operational and protocol specifications. For example, in the `putNextEntry(ZipEntry e)` method, it is important to specify what method calls need to be invoked on object `e` before invoking `putNextEntry` with `e` as argument, if there are any. In the precondition of `putNextEntry`, operational specifications inferred by Daikon [ECGN01] may specify them as object field value constraints for `e`, but here we intend to express them as sequencing constraints on the method calls of object `e`. The protocol inference techniques in previous work can be borrowed to tackle this problem. This problem is also related to bi-directional component protocols for collaborations [CMK98].

# 4. Conclusions

In this paper, we discussed the component protocol inference problems and identified several technical challenges. We proposed a framework for dynamic protocol inference and compared the previous work that uses dynamic analysis techniques. We also briefly discussed the previous work using static analysis techniques. By identifying the limitations of the previous work to tackle those technical challenges, we suggested some lines of future work in this area.

# 5. References


[ABL02] G. Ammons, R. Bodik and J. R. Larus. Mining Specification. In Proceedings of Principles of Programming Languages (POPL02), Portland, Oregon, pp. 4-16, January 2002.

[AMBL03] G. Ammons, D. Mandein, R. Bodik, J. R. Larus. Debugging Temporal Specifications with Concept Analysis. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, California, June 2003.

[Amm03] G. Ammons. Strauss. A Specification Miner. Ph.D. dissertation, University of Wisconsin, Madison, Department of Computer Science, May. 2003.

[AS83] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. ACM Computing Surveys, 15(3), pp. 237-269, 1983.

[BF72] A. Ÿ W. Biermann and J. Ÿ A. Feldman. On the synthesis of finite-state machines from samples of their behaviour. IEEE Transactions on Computers, 21, pp. 591-97, 1972.

[BRBY00] S. Butkevich, M. Renedo, G. Baumgartner, and M. Young. Compiler and tool support for debugging object protocols. In Proceedings of the 8th ACM International Symposium on the Foundations of Software Engineering, pp. 50-59, November 2000.

[CH74] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions, in Operating Systems, International Symposium, Rocquencourt, vol. 16, Lecture Notes in Computer Science, pp. 89-102, 1974.

[CMK98] I. Cho, J.D. McGregor, and L. Krause. A protocol based approach to specifying interoperability between objects. In Proceedings of Technology of Object-Oriented Languages (TOOLS 26), pp. 84 –96, Aug 1998.

[CW98] J. E. Cook and A. L. Wolf. Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology, vol. 7, no. 3, pp. 215-249, July 1998.

[Dahm00] M. Dahm. byte code engineering library. http://bcel.sourceforge.net, 2000.

[Dar98] C. Darby. ZIP Data Compression with Java , e-Business Advisor, August 1998. http://www.j-nine.com/pubs/javazip/jkzip.java

[DF01] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 59-69, June 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, June 2002.

[DM94] S. Das and M. Mozer. A unified gradient-descent/clustering architecture for finite state machine induction. In Proceedings of the 1993 Conference on Morgan Kaufmann Advances in Neural Information Processing Systems, vol. 6. Morgan Kaufmann Publishers Inc., San Francisco, CA, pp. 19–26, 1994.

[ECGN01] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, vol. 27, no. 2, pp. 1-25, Feb. 2001.

[ECHCC01] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In Proceedings of Eighteenth ACM Symposium on Operating Systems Principles, pp. 57-72, October 2001.

[ESS02] M. El-Ramly, E. Stroulia, and P. Sorenson. Interaction-Pattern Mining: Extracting Usage Scenarios from Run-time Behavior Traces, In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 2002.

[Fla97] D. Flanagan. Java in a Nutshell: A Desktop Quick Reference. O'Reilly & Associates, Sebastopol, California, 2nd edition, 1997.

[Flo95] G. Florijn. Object Protocols as Functional Parsers, Proceedings of the ECOOP `95, Springer LNCS 952, pp. 351-373, Aug. 1995.

[Gol78] E. Gold. Complexity of automatic identification from given data. Inf. Control 37, pp. 302–320, 1978.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems, Science of Computer Programming., vol 8, pp. 231-274, 1987.

[Hen80] K. L. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Applications, IEEE Transactions on Software Engineering, 6, 1, pp. 2-13, 1980.

[Hol91] G. J. Holzman. Design and Validation of Computer Protocols. Prentice-Hall International, 1991.

[HU79] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison Wesley, 1979.

[Jav03] Java 2 SDK. Standard Edition. http://java.sun.com/j2se/1.4.1/, accessed in May 2003.

[LCED01] D. Lie, A. Chou, D. Engler, and D. Dill. A simple method for extracting models from protocol code. In Proceedings of the 28th Annual International Symposium on Computer Architecture, pp. 192-203, July 2001.

[LS95] J. R. Larus and E. Schnarr. EEL. Machine-independent executable editing. In Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI'95), La Jolla, CA, pp. 291-300, June 1995.

[Mau03] The Mauve Project. http://sources.redhat.com/mauve/, accessed in May 2003.

[Mey97] B. Meyer. Object-Oriented Software Construction, New York, London: Prentice Hall, Second Edition, 1997.

[Mic99] A. Michail. Data mining library reuse patterns in user-selected applications. In 14th IEEE International Conference on Automated Software Engineering, pp. 24-33, 1999.

[Nie93] O. Nierstrasz. Regular types for active objects. In Proceedings of the OOPSLA '93 Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 1-15, October 1993.

[RR01] S. P. Reiss and M. Renieris. Encoding program executions. In Proceedings of the International Conference on Software Engineering, pp. 221-230, May 2001.

[RP97] A. V. Raman and J. D. Patrick. The sk-strings method for inferring PFSA. In Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97), 1997.

[Szy98] C. Szyperski. Component Software - Beyond Object-Oriented Programming, First Edition, Addison-Wesley, 1998.

[TB01] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces, SPIN 2001, Workshop on Model Checking of Software, LNCS 2057, pp. 103-122, May 2001.

[VL91] J. van den Bos and C. Laffra. PROCOL: A Concurrent Object-Oriented Language with Protocols Delegation and Constraints, Acta Informatica, Springer-Verlag, pp. 511-538, 1991.

[Wil82] R. Wille. Restructuring lattice theory: an approach based on lattices of concepts. Ordered Sets, pp. 445-470, 1982.

[WD01] D. Wagner and D. Dean. Intrusion detection via static analysis. In Proceedings of the IEEE Symposium on Security and Privacy, pp. 156-169, May 2001.

[XN03] T. Xie and D. Notkin. Exploiting Synergy Between Testing and Inferred Partial Specifications. In Proceedings of ICSE 2003 Workshop on Dynamic Analysis, Portland, Oregon, pp. 17-20, May 9, 2003.

[YS97] D. M. Yellin and R. E. Storm. Protocol Specifications and Component Adaptors. TOPLAS 19(2), pp. 292-333, 1997.