Fault Origin Adjudication

Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic*
Department of Computer and Information Science
University of Pennsylvania

15 February 2000

Abstract

When a program P fails to satisfy a requirement R supposedly ensured by a detailed specification S that was used to implement P, there is a question about whether the problem arises in S or in P. We call this determination *fault origin adjudication* and illustrate its significance in various software engineering contexts. The primary contribution of this paper is a framework for formal fault origin adjudication for network protocols using the NS/2 simulator and the SPIN model checker. We describe our architecture and illustrate its use in a case study involving a standard specification for packet radio routing.

1 Introduction

It is generally accepted that non-trivial computer programs will have faults. It is also well known that faults can derive from each of several stages of the software engineering lifecycle. For instance, a program P may deviate from its detailed specification S. But it is also possible that the specification S was incorrect because it failed to ensure high-level user requirements R. In this paper we introduce a technique for automated analysis to determine which of these two possibilities obtains, assuming that high-level requirements have been properly expressed and a deviation from them has been found. We call this process *Fault Origin Adjudication (FOA)*.

To see a characteristic example, suppose a development project is implementing a standard specification S of a communication protocol. This protocol is expected to have a property R, but testing of the program reveals that the program fails to satisfy R for some test input W. Clearly the problem needs to be repaired, but the *way* it needs to be repaired depends on the origin of the fault. If the program does not conform to the standard, then this may be the cause of the failure: the program should be revised to conform. This conformance should ensure interoperability with other implementations of S. Moreover, the design of S was probably intended to guide the implementor to a program that satisfies R. However, if the standard does not ensure the requirement, then the standard specification may be the origin of the difficulty. In the worst case, no conformant implementation of the standard will satisfy the property R. In a less extreme case, there is a risk that some implementations will conform with the standard but not satisfy R, leading to potential failures. Thus, if the fault lies with S, then the matter needs to be referred to the standards body and a revision of S should be made.

Of course, this reflects an ideal world in which standards are sufficiently unambiguous that one knows how to create a conformant implementation. Moreover, one must know what properties a conformant implementation is meant to satisfy. Even if this ideal is not achieved, it is important to have techniques appropriate to address the problem as rigorously or formally as possible. In fact, the problem of fault origin adjudication arises naturally in commercial practice on a daily basis. To see a recent concrete example, consider the standard for the Java Virtual Machine (JVM). It is considered crucial that implementations of the JVM conform to its standard since this ensures interoperability between implementations (indeed, at least one lawsuit was fought over incompatibilities). It is equally important that implementations of the JVM satisfy certain requirements, especially concerning security. Consider, for instance, the requirement R that the JVM is type safe and the specification S of the JVM in early versions of [20]. This S provided a specification of class loaders that left open the possibility of conformant implementations P that did not satisfy R. Indeed, there were several such implementations, including the Sun JDK implementations in versions 1.1.x, the Netscape implementations up to 4.05, and versions of the Microsoft implementation of Java through August of 1999.[1] Subsequently the specification has been repaired to address this problem [19], and it has also been addressed in at least the Sun JDK version 1.2.

Networking software often displays a similar pattern. For instance, distance vector routing, which was used in the early Internet and is still widely used in the Internet today, was given a standard specification [14, 22]. Other documents provided analysis of the protocol [23] and characterized its applicability [21]. High level requirements included, for instance, the expectation that router advertisements would eventually allow the routers to determine shortest paths [2]. Implementations of RIP that conform to the standard will satisfy this property. Indeed, this property has been checked using formal verification [6]. RIP is a seasoned standard based on a well-understood mathematical foundation. Many (probably most) other protocols are less well-understood, including new protocols being developed as Internet Drafts (`http://www.ietf.org/ID.html`). In general, it is challenging to determine whether a standard satisfies a requirement before there is a conformant implementation to test it. Once

*Email: `bkarthik@saul.cis.upenn.edu`, `gunter@cis.upenn.edu`, and `davor@saul.cis.upenn.edu`

[1] This example is due to Drew Dean. Dirk Balfanz, Ed Felten, Dan Wallach, and Drew Dean created (different) programs casting integers to object references in each of these implementations.

an implementation exists, tests can be run with sample input and deviations from the requirements lead to insights about whether the standard enforces the requirements. If the implementation conforms to the standard but violates a requirement in testing, then the standard does not enforce the requirement for conformant implementations.

In this paper we describe a framework for fault origin adjudication and a technique for using the NS/2 simulator and the SPIN model checker to support this framework in the context of network simulations. Our approach is applicable to safety properties that can be characterized using traces of network events like the transmission of a packet, assuming that the standard can be adequately modeled using SPIN. This approach is applied to a case study for illustration, namely the Ad Hoc On Demand Distance Vector (AODV) routing protocol for packet radio ([9] is S) and NS/2 simulations based on an implementation P of AODV by the Monarch group (`http://www.monarch.cs.cmu.edu`). These are analyzed relative to various requirements R for the AODV protocol, such as freedom from loops [27]. We have treated the task of finding deviations from R in another work [4].

The paper is divided into five sections. The second section discusses software artifacts and the relationship between them in terms of our reference model [13, 3]. In the third section we use this foundation to derive a framework for analyzing fault origin adjudication and describe an approach to formal analysis for fault origin adjudication for network protocols using SPIN and NS/2. This framework is applied to AODV in the fourth section. The fifth section provides analysis and conclusions.

2 Relating Software Artifacts

A software project generates a variety of artifacts in the form of code, documentation, and orally communicated (or uncommunicated) assumptions. It is helpful to use a 'reference model' for these artifacts as a foundation for classification and analysis. We have proposed a general model called *WRSPM* in [13] as an extension of work of Jackson and Zave [17, 18, 30] and applied it in a case study [3]. We use it again in this paper as a strategy for characterizing the issues in fault origin adjudication. Software projects do not always generate all of the artifacts in the reference model, and it is uncommon for any of them to be described with mathematical rigor, let alone formally. However, this paper will illustrate how the classification is useful in generating formal models from real-world artifacts. In effect, the reference model helps bridge the gap between formalism and reality. Before looking at the FOA framework and our networking case study it is helpful to have some background on the reference model and an extended toy example of the concepts we aim to explore.

2.1 The WRSPM Reference Model

The WRSPM reference model consists of five artifacts classified into two overlapping groups as depicted in Figure 1. Here W, the 'world', describes assumptions about the operational environment, and R represents a set of requirements to be met by a program. The goal of a programming project is to produce a program P that satisfies the requirements R when it is run on a programming platform M in an environment that satisfies the restrictions W. The role of the

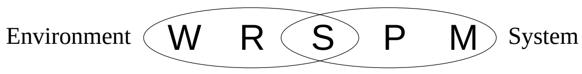

Figure 1: Five Software Artifacts

specification S is to provide enough information for a programmer to build such a program. Compliance with the specification is supposed to guarantee satisfaction of the requirements.

Our earlier work describes the expected relationships in a somewhat general form [13, 3]. In this paper we make simpler assumptions and focus on systems whose observable behavior is a finite trace of events generated during the execution. In that sense, we can regard R, S and P as sets of traces of events:

- R represents the set of all event traces that satisfy the requirements.
- S represents the set of event traces allowed by the standard specification.
- P represents the set of event traces that a given implementation can produce.

We say that the triple (R, S, P) is *safe* if $P \subseteq S \subseteq R$ (Figure 2). In other words, all program behaviors are allowed

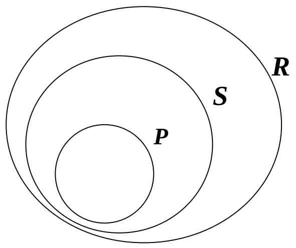

Figure 2: Safe Artifacts

by the standard and the standard guarantees fulfillment of the requirements. Notice that a program that does nothing (*i.e.* whose set of traces is empty) refines every standard, and, similarly, a contradictory standard refines every requirement.

In practice, the trace sets admitted by the artifacts can intersect arbitrarily so traces can generally fall into any of seven disjoint regions depicted in Figure 3. We will view

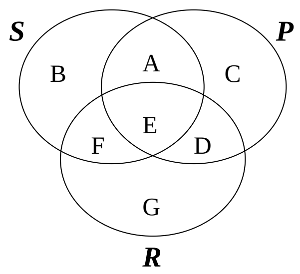

Figure 3: Artifacts in the general case

traces falling outside of all P,S and R as irrelevant for our

Table 1: Seven kinds of traces

Region	Meaning
A	Standard-allowed traces that break the requirements and are realizable by the program.
B	Standard-allowed traces that break the requirements and are not realizable by the program.
C	Program-realizable traces that break the requirements and are disallowed by the standard.
D	Program-realizable traces that satisfy the requirements and are disallowed by the standard.
E	Program-realizable traces that satisfy both the standard and the requirements.
F	Standard-allowed traces that satisfy the requirements and are not realizable by the program.
G	Traces that satisfy the requirements, but are not allowed by the standard and are not realizable by the progem.

Table 2: GBS Standard Specification S_1

function: Find

given: Nondecreasingly sorted array $a[0..n]$

arguments: number x, indices l and u

Step 1: If u is smaller than l, immediately return false, otherwise proceed with Step 2.

Step 2: If u equals l then proceed with Step 3, otherwise proceed with Step 4

Step 3: If x equals $a[l]$, immediately return true, otherwise immediately return false.

Step 4: Pick an index m such that $l \leq m \leq u$.

Step 5: If x equals $a[m]$, immediately return true, otherwise proceed with Step 6.

Step 6: If x is smaller than $a[m]$, recursively invoke Find with arguments x, l, m and return that result, otherwise proceed with Step 7.

Step 7: Recursively invoke Find with arguments x, m, u and return that result.

study. Interpretations of the seven regions are given in Table 1.

When the artifacts are safe, all traces are containted in regions E,F and G. Notice that in that case we have

$$\mathrm{E} = P,\ \mathrm{F} = S - P,\ \mathrm{G} = R - S,$$

so E, F and G correspond to the three disjoint regions of the diagram in Figure 2. Any trace falling into A,B,C or D indicates a violation of the safety refinement between the artifacts. Traces in regions A or B break the inclusion $S \subseteq R$. The difference is that traces in A are realizable by the program, while traces in B are not. This makes 'bug-hunting' in A more practical than in B. Analogously, regions C and D break the inclusion $P \subseteq S$. Traces in either C or D fail to satisfy the standard. However, traces in C also break the requirements, which makes them eaiser to detect.

2.2 A Toy Example

We now provide an example to illustrate each of the four kinds of safety violations (A,B,C and D) on an example.

Consider a problem of searching for a number in a sorted array. Suppose that $a[0..n]$ is a nondecreasingly sorted array of numbers. Given a number x and a pair of indices (l, u), where $0 \leq l \leq u \leq n$, the program should return true if x appears among the elements $a[l], a[l+1], \ldots, a[u]$, otherwise it should return false. Indices l and u represent boundaries of the searching region.

Let us consider a solution which in each step breaks the array into two parts, eliminates one of them, and recursively searches for x in the other part. This idea generalizes binary search, so we will refer to the algorithm as Generalized Binary Search (GBS). An attempted specification of GBS is given in Table 2. This could be viewed as a standard where different implementors may choose different ways of splitting the array based on their own criteria. The concept is similar to one that appears in programming language standards when a compiler writer is given the lattitude to evaluate the arguments of a function in any order he or she chooses since this enables various optimizations. For our purposes it is unimportant whether there is really any advantage in allowing this lattitude for binary search.

A computation proceeds in a sequence of recursive calls of the function Find. Each call involves three arguments—a number x and indices l and u. Therefore, we can define observable behavior to be the sequence of such (x, l, u)-triples in the order in which they appear during the computation, together with the final answer returned. For instance, suppose that the array is $a[0..4] = [2, 30, 80, 100, 114]$. Consider the following traces:

1. $(30, 0, 4) \to (30, 1, 4) \to (30, 1, 2) \to \mathsf{true}$
2. $(30, 0, 4) \to (30, 2, 4) \to (30, 2, 3) \to (30, 2, 2) \to \mathsf{false}$

The first trace is allowed by the GBS standard S_1, but the second trace is not (in the very first step, number 30 was dropped out of the searh region—the region changed from [0..4] to [2..4], but 30 appears only on position 1).

We will also allow *incomplete* traces to be regarded as observable behaviors. This makes the set of observable behaviors prefix-closed, which matches the usual intuition. For example, the trace $(30, 0, 4) \to (30, 1, 4)$ is also considered valid with respect to S_1, since it is a prefix of a complete trace allowed by S_1.

Our requirement needs to be a *safety* property on the set of all traces. This constraint is a natural consequence of our semantic model, in which behaviors are *finite* traces. Safety properies are traditionally [1] characterized as properties which are violated by finite trace prefixes. However, *all* of our traces (including violating counterexamples) are finite, so our model makes sense only in the context of safety properties. Liveness properties require a different notion of refinement from the ordinary subset inclusion on the sets of traces.

One way to ensure safety is to set an upper bound on the number of computation steps. This is much like using timers in the networking context, where we require that an action be carried out within a certain time window. In this example, we require that each computation reaches the correct answer within n steps, where n is the size of the initial search region. Table 3 describes the requirement precisely.

Table 3: GBS Requirement R

R is the prefix closure of the set of all traces of the form $(x, l, u) \to T \to a$, where the following holds:

- T is any trace such that $|T| < u - l + 1$. ($|T| :=$ the number of "arrows" in T.)
- $a = \begin{cases} \mathsf{true} & \text{if } x \in \{a[l], a[l+1], \ldots a[u]\} \\ \mathsf{false} & \text{otherwise} \end{cases}$

Table 4 gives an implementation of the standard in

Table 4: GBS implementation P

```
int Find (int x, int l, int u)
{
  int m;

  if (u<l) return 0;
  if (u==l) return (x==a[l]);
  m = (l + u) / 2;
  if (x == a[m]) return 1;
  if (x < a[m])
    return (Find (x,l,m));
  else
    return (Find (x,m,u));
}
```

C. Notice the difference between the implementation and the standard in the choice of the pivoting position (m). The standard allows m to range arbitrarily over the set $\{l, l+1, \ldots, u\}$, while the implementation always chooses middle element as the pivot. For instance, another implementation might have chosen pivot points randomly in the allowed range.

Errors in the standard (A and B cases)

Suppose that our sorted array is $a[0..2] = [5, 6, 10]$. If we invoke `Find (10,0,2)`, we get a divergent computation:

$$(10, 0, 2) \to (10, 1, 2) \to (10, 1, 2) \to (10, 1, 2) \to \cdots$$

Any finite prefix of this computation will be a behavior realizable by P. In particular, consider the behavior

$$T_1 = (10, 0, 2) \to (10, 1, 2) \to (10, 1, 2) \to (10, 1, 2).$$

The initial search region (positions 0 through 2) has the size 3, but T_1 did not produce a correct answer in the first 3 steps. Therefore, $T_1 \notin R$. On the other hand, T_1 is realizable by P and S_1. According to the Figure 3, this is an example of an A-region error.

In this case, P is a correct implementation of the standard but still produces an error. This indicates an error in the standard. In fact, S_1 allows many more incorrect behaviors besides those that can be demonstrated by P. For instance, it is easy to see that, with the same array a, S_1 allows the trace

$$T_2 = (5, 0, 2) \to (5, 0, 2) \to (5, 0, 2) \to (5, 0, 2)$$

by repeatedly picking the pivot position $m = 2$. Just like T_1, trace T_2 also breaks the requirements, but it is not obtainable through P. In our diagram T_2 corresponds to a B-region error.

Errors in the implementation (C and D cases)

In the previous scenario, P was a valid implementation of an incorrect standard S_1. We can try to fix this by revising a standard. Table 5 shows a new standard S_2. Framed parts

Table 5: GBS Standard Specification S2

function: Find

given: Nondecreasingly sorted array $a[0..n]$

arguments: number x, indices l and u

Step 1: If u is smaller than l, immediately return false, otherwise proceed with Step 2.

Step 2: If u equals l then proceed with Step 3, otherwise proceed with Step 4

Step 3: If x equals $a[l]$, immediately return true, otherwise immediately return false.

Step 4: Pick an index m such that $\boxed{l \le m < u}$.

Step 5: If x equals $a[m]$, immediately return true, otherwise proceed with Step 6.

Step 6: If x is smaller than $a[m]$, recursively invoke Find with arguments $\boxed{x, l, (m-1)}$ and return that result, otherwise proceed with Step 7.

Step 7: Recursively invoke Find with arguments $\boxed{x, (m+1), u}$ and return that result.

contain differences between the two versions.

Consider the trace T_1 again, now with the new standard. The trace is produced by P, it breaks the requirement R, but this time it is not allowed by the standard S_2. Hence, in the context (R, S_2, P), trace T_1 is a C-region error.

Finally, we look at the case when our implementation P produces a correct behavior which does not comply to the standard S_2. Let the array be the same ($a[0..2] = [5, 6, 10]$). Consider the behavior of P on input $(4, 0, 2)$:

$$T_3 = (4, 0, 2) \to (4, 0, 1) \to (4, 0, 0) \to \mathsf{false}.$$

This computation took three steps. It can be shown that any S_2-compliant computation would solve this instance in at most two steps. Therefore, $T_3 \notin S_2$. Yet T_3 clearly satisfies R. This classifies T_3 as a D-region error with respect to (R, S_2, P).

One can show that standard S_2 indeed ensures R. Program P is an incorrect implementation of S_2 that can produce both correct and incorrect behaviors. Furthermore, there are correct behaviors of P that violate the standard (e.g. T_3). An categorized overview of the errors is given in Table 6.

3 Fault Origin Adjudication (FOA) Framework

Recall the counterexamples T_1 and T_3 from the previous section. T_1 was a violation of the standard S_2 which, at the

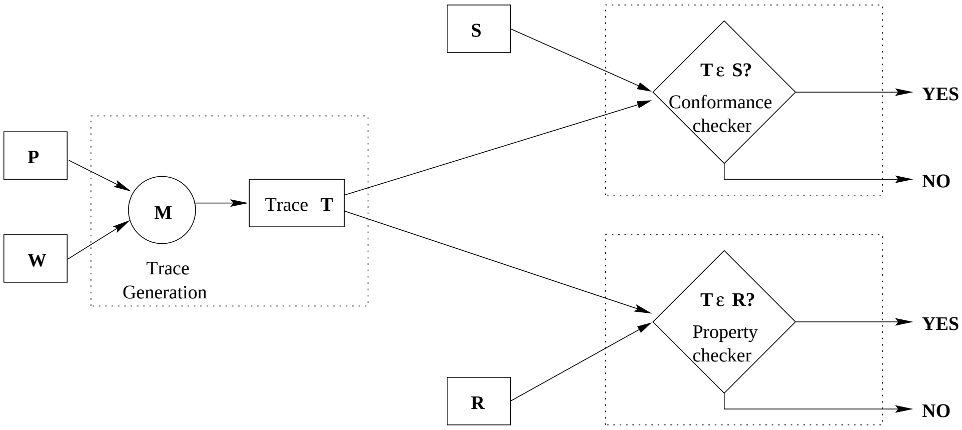

Figure 4: Fault Origin Adjudication Framework

Table 6: Overview of the errors

	A	B	C	D
(R, S_1, P)	T_1	T_2	empty	empty
(R, S_2, P)	empty	empty	T_1	T_3

same time, violated the requirement. On the other hand T_1 only violated the standard, but not the requirement. Imagine a simple validation framework which tests implementation behaviors against requirements but not against the standard. Such a framework would correctly recognize T_1 as a failure, but it would not find anything wrong with T_3. Discovering the problem with T_3 would be useful because it would inform the standardization process and might be important for interoperability with other implementations of the standard. This shows the value of testing with respect to the standard as well as testing with respect to the requirements. This idea is the basis of our FOA framework.

Figure 4 shows the abstract view of the FOA framework. The framework consists of three parts: *trace generator* (to the left), *conformance checker* and *property checker*:

Trace generator takes as its input program P and *scenario* W. A scenario typically includes inputs to P and certain 'world assumptions' about the environment in which P runs. P and W are fed into the machine (M) which runs them and produces a trace T.

Property checker takes as its input a trace T (produced by the trace generator or some other way) and requirements R. It tests whether T satisfies the requirements and outputs a yes/no answer.

Conformance checker takes as its inputs a trace T and a standard specification S. It tests whether T is allowed by the standard and outputs a yes/no answer.

A combination of answers from the conformance checker and the property checker enables us to reason about the fault origin. Table 7 describes the four possible outcomes and their interpretations. Notice that this framework catches only errors which can be demonstrated through a particular *implementation*. This includes only three kinds of errors (A,C and D). Detection of B-region errors would involve

Table 7: FOA outcomes and their interpretations

		Property check	
		$T \in R$	$T \notin R$
Conformance check	$T \in S$	E	A
	$T \notin S$	D	C

Region	Interpretation and remedy
E	Everything OK.
D	Incorrect implementation of the standard. Correct the program.
C	Incorrect implementation of the standard. Correct the program.
A	Incorrect standard. Revise the standard and the program accordingly.

(random) simulations of the standard specification, or model checking. The latter will usually be computationally infeasible for the general case.

3.1 Network Protocols

The general FOA framework is as described in Figure 4. For the particular case of network protocols, we describe a methodology for each of the three phases required for the adjudication. Many network protocols have the following characteristics which guide the way the three phases must be implemented:

1. The protocol is specified in the form of a reasonably precise standard. While low-level details are left to the implementations, aspects like packet formats are clearly specified.

2. The software runs concurrently as processes on a number of nodes geographically dispersed in a network.

3. The processes communicate by asynchronously passing packets to one another.

4. The network is dynamically changing and communication is unreliable.

5. A number of actions are carried out under real-time constraints based on timers.

6. The requirements are typically safety properties of the packets injected into the system. Liveness properties are typically converted to safety properties by stipulating a time limit for an action to occur.

We now show how network software is tested to produce traces (*trace generator*). We then describe how the traces can be checked for requirements using SPIN (*property checker*). Finally, we show how SPIN can be used to check if the traces are conformant with the specification (*conformance checker*). In this manner, we develop a complete system for the fault origin adjudication of network protocols with respect to a wide range of properties.

The tools that we use are the Network Simulator NS/2 and the model-checker SPIN. NS/2 [12, 28] is popular among protocol designers for performance analysis of new or modified network protocol designs. Given simulator code for the protocol in C++ and a network scenario, NS/2 carries out a discrete event simulation of the network for a specified length of time. The simulation produces a trace which is normally used for measuring performance. SPIN [16, 15], on the other hand, is a model-checker. It takes as input an abstract model of a protocol written in the specification language—Promela. SPIN can simulate the protocol along with an environment model. Given a property written in Linear Temporal Logic [24] or in Promela, SPIN can also *verify* by exhaustive state-space search that the protocol model satisfies the property.

3.2 Protocol Testing - Trace Generation

The concurrent nature of network routing software makes it rather difficult to test in real situations. Therefore, established network protocol design practice involves using network simulators like NS/2 to test the protocol code for a variety of artificial scenarios. We use NS/2 to carry out the simulations and produce the resulting trace. The inputs to the system are as shown in the Figure 5, which essentially

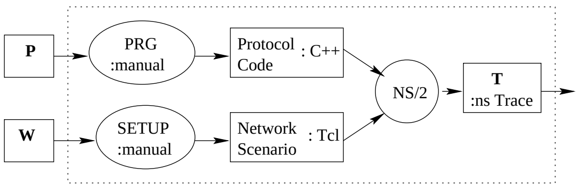

Figure 5: Trace Generation Using NS/2

expands the corresponding part of Figure 4.

The protocol is typically encoded as a C++ program that reacts to incoming packets and possibly produces outgoing packets. A network scenario is a description of the number of nodes in the network, their topology, and a description of data sources and sinks. Scenarios are written in Tcl and provided as input to NS/2. Given the above inputs, NS/2 carries out a random (seeded) simulation of the network up to a specified time and produces a trace of all the packets generated and transported through the network. The output trace is in a specific NS/2 trace format with one line for each packet transmission or reception event.

3.3 Property Checking

Given a trace produced by NS/2, we need to check that it satisfies the requirements R. This check can be carried out by a number of methods [4]. In this paper, we show how to use SPIN. The Property Checker is as shown in Figure 6.

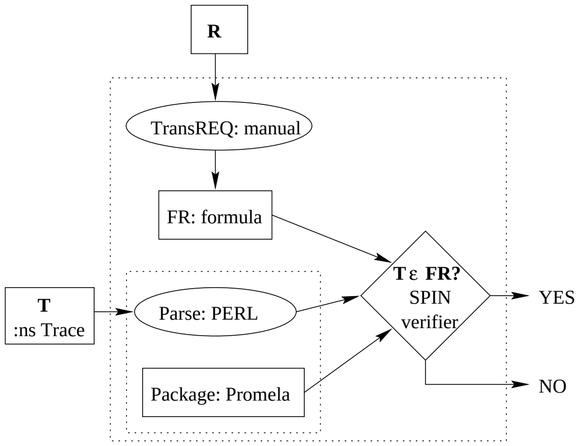

Figure 6: Property Checking Using SPIN

First, the NS trace needs to be translated into something that SPIN can understand. We do this in two steps. The parser, written in PERL, parses the NS packet trace format and decomposes each packet event into its constituent fields which SPIN can read. Then a Promela process re-packages the fields into SPIN packet events. This translation mechanism has to be written just once for each protocol packet type.

Next the requirements R are encoded into Promela. This is straightforward for safety properties—for each property we encode a monitoring process which checks if the trace so far is conformant to the property. Promela is particularly attractive for encoding these properties because

1. Communication is a primitive in Promela, so properties about packets do not need to be encoded in a different formalism.

2. Promela has dynamic process creation, so monitoring processes can be spawned on-demand.

3. There is a well-known method for converting Linear Temporal Logic formulae to Promela processes.

Once all the above modules are in place, the trace is fed into a SPIN execution of the property checker. If the trace fails to satisfy the property, the execution halts and declares an error. If no errors are found, the trace conforms to the requirements.

3.4 Conformance Checking

The conformance checker re-uses the parsing programs described in the previous section for translating NS packet traces to SPIN events. For conformance, we need to check if the input packet trace could have been produced by the specification S. The specification in this case is in the form of a protocol standard. Network protocol standards are typically designed based on a prototype implementation. However, there are usually some implementation issues that are left to the programmers discretion. In addition, there are ambiguities that stem from the incompleteness of the specification. Therefore, although the standard is designed around an executable prototype, it is typically abstract and non-deterministic.

The main elements of conformance checking are as shown in Figure 7. The standard first needs to be encoded in a

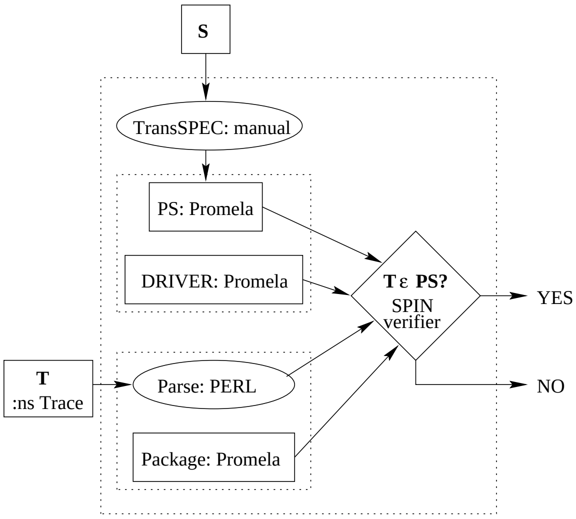

Figure 7: Conformance Checking Using SPIN

formal framework for analysis. This formalism should support non-determinism and concurrent processes. Moreover, it is important for the encoding to be as abstract as the original specification. Therefore, the formalism must support high-level communication and abstract data structures. We choose Promela as the formal language for our encoding. Promela has all the above features and has been widely used as an abstract language for real communication protocols. Importantly, Promela models can be automatically analyzed using SPIN.

We encode the standard as a Promela process that runs at one node. We encode the environmental assumptions in the form of auxiliary processes. This model of the standard can be model-checked for various properties in SPIN, including deadlock-detection and satisfaction by LTL formulae. However, the huge state space of the encoding makes it impractical to check it for network of more than 2 or 3 nodes.

Once the standard is encoded in Promela, we encode a *driver* for the protocol. The driver reads the input packet trace, and checks whether the protocol standard allows the sequence of events described in the trace. To carry out this check, the driver essentially determinizes the standard with respect to the input trace and attempts to force it to generate the trace. If the standard fails to generate the trace, the trace is not conformant to the standard. This determinization is simple for most network protocols, although it need not be in general.

Finally, the trace is fed to a SPIN execution of the standard and its driver. If the driver manages to consume the entire trace without error, the trace is conformant with the standard. If not, then the trace must have been generated by a faulty implementation of the standard.

4 Case Study

We apply FOA as described above to analyze Ad-Hoc On-Demand Distance Vector routing [27, 26, 9] (AODV), an emerging standard for routing in packet radio networks. We find traces that fall in all the categories described.

4.1 Ad Hoc On Demand Distance Vector Routing

In mobile, wireless networks, nodes communicate with each other on links that have limited range. When the nodes communicate without the use of a central base station there is a need for protocols and algorithms that allow a group of nodes to cooperate in transporting data from one node to another. Such a protocol implicitly creates an ad hoc internetwork where each participating node acts as a router. A typical example of the network's operation is as shown in Figure 8—when one node needs to communicate with an-

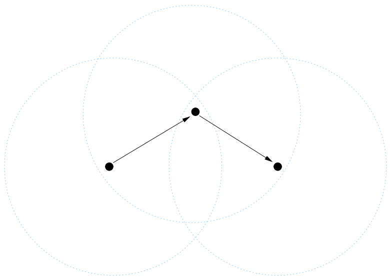

Figure 8: Packet Radio Routing

other which is not in its range, it asks an intermediate node to forward its packets. The intermediate node is a router that provides packets a hop on the way to their destinations.

Ad Hoc On Demand Distance Vector routing (AODV) is one of the protocols that has been proposed as a standard for such wireless networks. In AODV, intermediate forwarding nodes store dynamic information about paths to the destination for which they need to forward data. The term 'distance vector' refers to the fact that the only information they store for each destination node is the length of the 'best' path, and the next node on this path. We refer to this path information at each node as the route to the destination. Since nodes keep moving around, routes need to be frequently re-computed with respect to the new topology. AODV provides an algorithm for the distributed computation of routes, along with a protocol for realizing the algorithm.

However, as a result of the dynamic nature of a mobile network, routes are frequently out-of-date—the paths that they refer to do not exist any more. Moreover, since simple distance vector protocols just store 'short-sighted' information about the path, they cannot easily detect that the path has been broken unless the break is in their immediate vicinity. In particular, there could be routing loops as shown in Figure 9, that are created for periods of time. Loops are the worst kind of route inconsistency—they are difficult to detect, and they consume lots of bandwidth in the meanwhile.

AODV has been designed to avoid routing loops. This is done by attaching recency information to paths and ensuring that the path information at nodes is kept as up-to-date as possible. The recency information is in the form of sequence numbers issued by the destination. AODV thus maintains

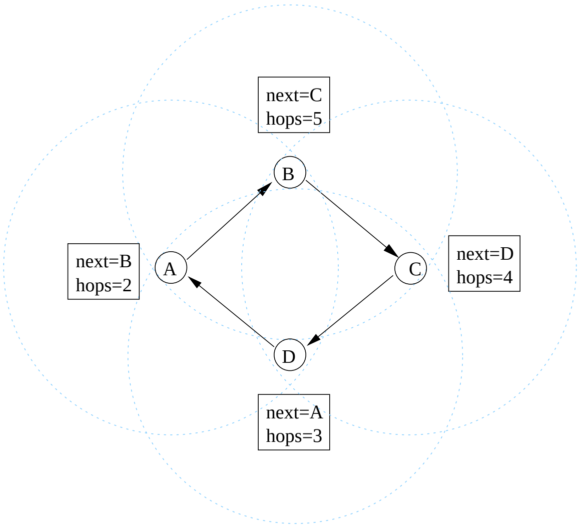

Figure 9: Loop in a Distance Vector protocol

on-demand routes for destination nodes, consisting of a hop-count to the destination (length of the path), the next node on the path, and the sequence number issued to the path.

4.2 Framework

We analyze AODV with respect to the framework described in earlier sections. The mapping of the software artifacts is as shown in the Table 8. The requirement R asserts that,

Table 8: Framework Mapping for AODV

P	Monarch AODV Code (C++, Simulator specific)
M	NS/2 Simulator
W	Random Scenarios, S1, S2, S3
S	AODV Version 2 [26]
R	Loop Freedom

in any run of the protocol over any network, the routes to a destination should never form a loop. We call this *loop freedom*. The specification S is the second version of the AODV standard, available as an IETF Internet Draft [26]. The implementation P we consider is the prototype implementation of AODV for the NS/2 simulator, written in C++ by the CMU Monarch group (`http://www.monarch.cs.cmu.edu`). For the tests, we start out with a set of large randomly generated network scenarios. In this paper, we present three simple scenarios which illustrate all the errors that can be detected by the FOA framework.

4.3 SPIN Modeling

In order to analyze the AODV traces produced by NS/2, we carry out the following steps:

1. Decide on packet formats in Promela.
2. Write a Perl program to parse the NS/2 packet format and extract the fields relevant to AODV. Write a dual Promela program that takes these fields and packages them up as a Promela AODV packet.
3. Encode the requirements as formulae in Promela.
4. Encode the specification in Promela.
5. Finally we write a driver for the specification in Promela.

After all the above modules are in place, we need to carry out the two tests

Property Checking We feed traces generated by NS/2 into the SPIN execution of the formulae and the parsing programs. If any of the properties is violated at any point in the trace, the execution terminates with an error; the test outputs NO. Otherwise, it reaches the end of the trace without incident and the trace outputs YES.

Conformance Checking We feed the same packet traces into the SPIN execution of the standard along with the driver and parsing programs. If the driver fails to force the standard to generate the trace, the test outputs NO. If the standard succeeds in generating the entire trace, the test answers YES.

4.4 Tests & Results

Network software is typically simulated for large randomly selected topologies. We simulate the AODV protocol in NS/2 for a number of such scenarios, and find errors in the AODV code, as well as the standard. We present three simple scenarios which illustrate the different errors encountered. We use the traces produced by NS/2 for these scenarios to help us adjudicate the origin of faults in AODV.

Scenario S1

S1 as shown in Figure 10. A solid line between two nodes indicates that the two nodes have a wireless link between them—they are within range of each other.

S1 is the simplest scenario involving three nodes—A, B and D. A wants to continuously send packets to D. Therefore, the aim of the system is to ensure A knows to send all these packets to B, and B knows to forward them to D. We simulate this scenario in NS/2 and produce a trace T1. Using FOA analysis, we find that T1 passes the property check, but it fails the conformance check.

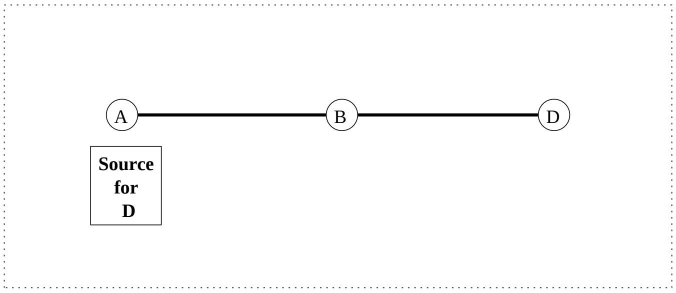

Figure 10: Scenario S1

Scenario S2

In Scenario S2, shown in Figure 11, we make the nodes mobile. In particular, after 10 seconds, the destination node D moves out of range of the other nodes. At the same time, B has started sending packets to D. Now the system must adapt to this change of topology and A, B must discover that they can no longer reach D. We simulate this scenario to generate trace T2. Using FOA analysis, we find that T2 violates the requirements as well as fails to conform to the standard.

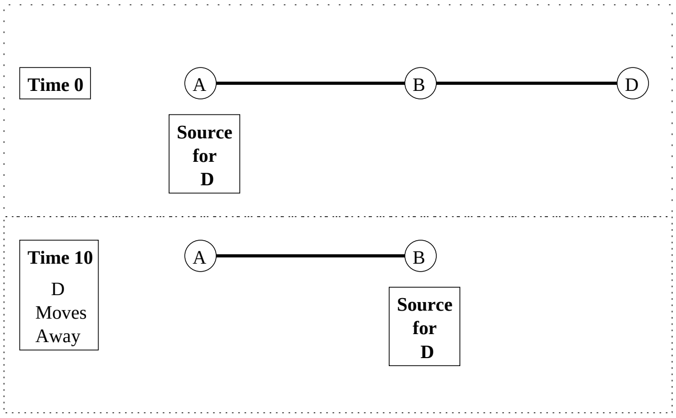

Figure 11: Scenario S2

Scenario S3

Finally, in Scenario S3, shown in Figure 12, A needs to send packets to D, but D is never reachable by A,B or C. After 10s, A moves away from B and C. At the same time, B starts sending packets to A. In this case, A,B and C must first discover that D is unreachable and then B, C must discover that A is unreachable. We simulate this scenario to produce a trace T3. T3 conforms to the standard, but it fails to satisfy the requirement.

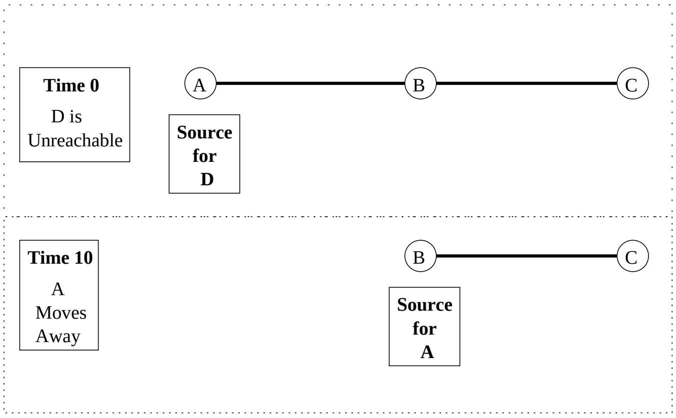

Figure 12: Scenario S3

The three traces obtained above fall in different categories with respect to the FOA framework as shown in Table 9.

Table 9: AODV Traces

Trace	Error Region
T1	D
T2	C
T3	A

T1 and T2 indicate that the implementation is faulty in that it does not correctly implement the standard. Indeed, we have since found at least three bugs in the simulator code which cause it to be non-conformant with respect to the standard. More importantly, the simulator implementation of AODV allows routing loops. However, T3 indicates that the standard itself is faulty in that it allows routing loops to be formed. We have since confirmed at least four conditions under which the standard allows loops to be formed. As a result, the standard itself needs to be revised if it is to guarantee loop freedom.

The AODV standard has reached a fouth version [9] by the time of this writing. Whether any of the issues we discuss here have been resolved in subsequent versions of the artifacts in Table 8 does not affect the point of our case study, which is to illustrate the FOA framework. However, efforts are underway to address all of the points raised in this paper. We prepared a discussion [5] of the issues known to us concerning the standard and this was circulated on the Manet mailing list. The fault in Scenario S1 was located by ourselves working with Mike Berry. The faults in Scenarios S2 and S3 were located by ourselves based on a similar fault found by Joshua Broch and Dave Maltz reported in their 1998 Mobicomm tutorial.

5 Conclusion and Analysis

We have described the idea of fault origin adjudication and presented a framework for applying NS/2 and SPIN to carry it out for a broad class of network protocols and properties. We illustrated this technique for a non-trivial case study based on distance vector routing for packet radio using the AODV protocol. Let us now consider briefly some further aspects of the assumptions and scope of the approach.

A salient feature of our approach has been that we pay attention to even seemingly minute differences in software artifacts, believing that these aberrations often give rise to important faults. However, we do not deal directly with some of the actual artifacts. In particular, our automated analysis does not treat S or R directly; instead we analyze Promela encodings of S and R. Nevertheless, we believe that this level of encoding is justifiable. Any attempt at automated analysis must require that all the artifacts be formalized. The only question is whether Promela induces a level of unnatural encoding—can the Promela model be considered *equivalent* to the original artifact? For many (perhaps most) network protocols, R and S can indeed be naturally expressed in Promela, since it is: (a) abstract enough to elide detail, (b) expressive enough to represent non-deterministic behavior and temporal formulae, and (c) especially tailored for describing communication protocols. Having said that, one can conceive of requirements that will be difficult to express in Promela (any property to do with arithmetic or sets for instance), and standards that will difficult to encode (for example ones that involve complex data structures).

FOA can be viewed as a testing activity. Our framework provides formal support for it in cases with certain attributes.

- The framework is not bound to any specific language or a tool. The FOA procedure can be carried out as long as the following elements are provided:
  1. An implementation, standard specification, and requirements.
  2. A way for the user to collect observable behaviors from the implementation (trace generator).
  3. A mechanism (or a tool) for testing whether a given behavior satisfies the requirements (property checker).
  4. Requirements formally stated in a language that can be understood by the property checker.
  5. A mechanism (or a tool) for testing whether a given behavior satisfies the standard specification (conformance checker).
  6. A faithful formal model of the standard specification given in a language that can be understood by the conformance checker.
- The framework can be applied only for the analysis of safety properties.
- The framework can detect and adjudicate errors, but it does not generally provide a proof of system correctness.
- The framework can detect only three out of four kinds of errors discussed (Table 7). The case which is not handled is the B-region.

The last two shortcomings could be addressed by attempting a direct verification of the refinement between S and R through an exhaustive check. This approach does potentially offer a stronger result, but unfortunately it does not scale well. The state space of S, which needs to be explored during the exhaustive check, grows roughly exponentially with the size of S. This can be a serious problem even with examples of moderate sizes. On the other hand, FOA is more resistant to such problems, since checking if given trace satisfies S and R is typically fast.

An interesting extension of the work presented would be to investigate how to 'steer' trace generation to produce traces of a particular interest (e.g. those that would illustrate some interesting scenarios). One possibility would be to use ideas similar to those presented in [7, 8]. They describe a methodology for automatic generation of test oracles (based on the set of requirements), that resolve non-determinism in a way that forces the system into a stress scenario. Several other ideas that focus on generation and analysis of test-oracles for various logics are described in [10, 29, 25, 11]These ideas could be used to increase the coverage of the test suits.

Also, our approach considers only traces generated by a particular implementation. This is what prevents us from discovering errors in the B-region. However, certain tools (like SPIN) can generate traces from nondeterministic specifications, by using randomized or some other kinds of simulations. This could be used as an alternative (direct) method for finding errors in the standard. An interesting question is which method works better in practice? Comparisons could be carried out through a number of case studies.

Acknowledgments

We would like to express gratitude for assistance we received from the following people: Martín Abadi, Mike Berry, Drew Dean, David Dill, Dave Johnson, Dave Maltz, Madanlal Musuvathi, Charles Perkins and researchers in the Monitoring and Checking group at Penn. This research was supported by DARPA Contract F30602-98-2-0198 and ARO Contract DAAG-98-1-0466.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:73–132, 1993.

[2] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1991.

[3] Karthikeyan Bhargavan, Carl A. Gunter, Elsa L. Gunter, Michael Jackson, Davor Obradovic, and Pamela Zave. The Village Telephone System: A case study in formal software engineering. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics 11th International Conference TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 49–66, Canberra, Australia, September 1998. Springer.

[4] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolosky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, August 2000.

[5] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Analyzing the occurrence of loops in aodv. Manet Mailing List, December 1999.

[6] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, February 2000. `http://www.cis.upenn.edu/~hol/papers/rip.ps`.

[7] J. Callahan and F. Schneide. Specification-based testing using model checking. In *1996 SPIN Workshop, Rutgers University*, August 1996.

[8] John R. Callahan, Stephen M. Easterbrook, and Todd L. Montgomery. Generating test oracles via model checking, 1998. Unpublished, working paper. Available at `http://research.ivv.nasa.gov/projects/blurbs/FormalTesting.html`.

[9] Santanu Das, Charles E. Perkins, and Elizabeth M. Royer. Ad hoc on demand distance vector (AODV) routing. Internet-Draft Version 4, IETF, October 1999.

[10] Laura K. Dillon and Y.S. Ramakrishna. Generating Oracles from Your Favorite Temporal Logic Specifications. In *Proc. 4th ACM SIGSOFT Symp. Foundations of Software Engineering, San Francisco*, pages 106–117, October 1996.

[11] Laura K. Dillon and Q. Yu. Oracles for Checking Temporal Properties of Concurrent Systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT'94)*, volume 19, pages 140–153, December 1994. Proceedings published as Software Engineering Notes.

[12] Kevin Fall and Kannan Varadhan. *ns Notes and Documentation*. The VINT Project, 2000.

[13] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. `http://www.cis.upenn.edu/~hol/papers/vts.ps`, 2000. To appear in IEEE Software.

[14] C. Hendrick. Routing information protocol. RFC 1058, IETF, June 1988.

[15] Gerard J. Holzmann. SPIN-formal verification. Web Page. Available at `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[16] Gerard J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[17] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–64. IEEE Computer Society Press, 1992.

[18] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15–24. IEEE Computer Society Press, 1995.

[19] Sheng Liang and Gilad Bracha. Dynamic class loading in the java virtual machine. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98)*, volume 33 of *SIGPLAN Notices*, pages 36–44, Vancouver, British Columbia, Canada, October 1998.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[21] G. Malkin. Rip version 2 applicability statement. RFC 1722, IETF, November 1994.

[22] G. Malkin. Rip version 2 carrying additional information. RFC 1723, IETF, November 1994.

[23] G. Malkin. Rip version 2 protocol analysis. RFC 1721, IETF, November 1994.

[24] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1991.

[25] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient Specification-Based Test Oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[26] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on demand distance vector (AODV) routing. Internet-Draft Version 2, IETF, March 1998.

[27] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 90–100, February 1999.

[28] The VINT Project. The ns Network Simulator. Web Page. Available at `http://www-mash.CS.Berkeley.EDU/ns`.

[29] D.J. Richardson, S. Leif Aha, and T.O. O'Malley. Specification-Based Oracles for Reactive Systems. In *14th International Conference on Software Engineering*, May 1992.

[30] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, 6(1):1–30, January 1997.