A Robust Transport Protocol for Run-Time Fault Detection

G. Noubir K. Vijayananda, H. J. Nussbaumer
Swiss Federal Institute of Technology, Lausanne,
Computer Engineering Department, EPFL-DI-LIT,
CH-1015, Lausanne, Switzerland
noubir, vijay@di.epfl.ch

Abstract

Run-time fault detection in communication protocols is essential to detect faults that cannot be detected during the testing phase. We propose a signature-based method to detect run-time faults. A *polynomial* using the state and event information as coefficients is used to transform a sequence of states and events into a number (signature). The static signature corresponding to the correct execution of the protocol is compared with the run-time signature. This technique is more reliable, faster, and efficient compared to existing techniques. The states and events are assigned values such that multiple paths leading to the same state result in a unique signature. This reduces the number of comparisons required to verify the correct execution of the protocol. In this paper, we present *eXTP4*, an extended transport layer protocol that facilitates run-time fault detection.

1 Introduction

In a distributed system environment, communication protocols play a major role in providing connectivity for meaningful exchange of information. They are designed to provide reliable services to the users. Protocol verification and validation techniques are used to ensure the correctness of the specification of communication protocols. However, there is no strategy to ensure that the execution of the protocols is error-free. Dormant errors due to coding, memory problems, and external disturbances are not considered in above mentioned techniques. Unless some kind of run-time checking is done, this type of faults may remain undetected until a partial or complete breakdown of the communication system. In order to ensure the continuous availability and quality of services offered by these systems, it is necessary to detect and diagnose these run-time faults.

Several fault-detection techniques for communication protocols can be found in the literature. Finite State Machine (FSM) methods have been used to detect run-time faults in communication protocols. At run-time, an external observer is used to monitor the exchanged messages. These messages are compared with the FSM model for their correctness. In [2], Bouloutas et al. use the FSM approach to detect faults in machines whose behavior is described using FSMs. Observers that are modeled using FSMs are used to detect out-of-sequence messages and incorrect state transitions. In [11], Wang et al. decompose the FSM representing the communication protocol into several FSMs. Each FSM is represented by an observer. All the observers operate in parallel to detect faults in the communication protocol. In [3], Diaz et al. use an observer to monitor and detect faults in an implemented system. The observer is modeled using the specifications of the implemented system. In [9], model-based reasoning is used to detect and diagnose deviations from the specified behavior of the communication protocols. In most of these methods, the observer is complex as the system under observation.

We use a new signature-based method to detect run-time faults in communication protocols [7]. A *polynomial* with the state and event information as coefficients, is used to transform a sequence of states and events into a number (signature). An event corresponds to a message exchanged between the entities. The *static signature* which corresponds to the correct sequence of messages, is computed and stored in a table. During the execution of the communication protocol, an observer computes the *run-time signature* and compares it with the static signature. A fault is detected when the run-time signature differs from the static signature. This method is simpler compared to the FSM method and can detect a broader class of faults. After a fault has been detected, the localization of the fault maybe done in an off-line manner using other techniques like model-based diagnosis **??**.

In the past, communication protocols have been modified for the purpose of improving their performance. In [10, 6, 4], a high speed transport protocol has been proposed where the current state of the sender machine is transmitted at regular intervals in dedicated messages. The extension proposed by the authors is restricted to the data transmission phase and resulted in a significant improvement in the throughput over conventional protocols. In this paper, we propose extensions to communication protocols which facilitate the detection of run-time faults using our signature-based method. Extensions are proposed to include the state information in the messages exchanged between communication entities.

This paper is organized as follows. In section 2, we present the definitions and terminologies used in this paper. In section 3, we discuss the fault detection technique. Section 4, presents our signature generation technique. In section 5, we describe the use of

a new signature-based method for run-time fault detection in communication protocols. In section 6, the fault-detection mechanism is illustrated using an example from TP4 (TP4 is the Transport Class4 protocol). The extensions to the communication protocols that facilitate this new fault-detection scheme are introduced in section 7. Transport Class 4 protocol is used to describe some implementation details of the proposed extensions.

2 Definitions and Terminologies

We recall the definition of finite state machines and give our definition of paths, correct paths, legal state paths, and legal event paths.

2.1 Definitions

Definition 1: The communication protocol is modeled as an FSM. An FSM is a 3-tuple $A = (Q, \Sigma, \delta)$, where Q denotes the set of all possible states $\{S_1, \ldots, S_n\}$. Σ is the set of all possible events $\{E_1, \ldots, E_n\}$. In this paper, the term event and message are used to denote the messages exchanged between entities. δ denotes the state transition function ($\delta : Q \times \Sigma \rightarrow Q$). Each state S_i (event E_i) is assigned a value s_i (e_i) from an algebraic finite field F (*Galois* field).

Definition 2 A path C is defined as an alternating sequence of states and events (e.g., $S_1E_1S_5E_4 \cdots E_{n-1}S_n$).

The *state path* is the sub-sequence of states derived from the path by deleting all the events and retaining the states in their original order (e.g., $S_1S_5S_3 \ldots S_n$).

The *event path* is the sub-sequence derived from the path by deleting all the states and retaining all the events in their original order (e.g., $E_1E_4 \cdots E_{n-1}$).

Definition 3: `Last(`C`)` of a path C is the last state in C, (e.g., $last(S_2S_1S_7S_3) = S_3$). `First(`C`)` of a path C is the first state in C (e.g., $first(S_2S_1S_7S_3) = S_2$). `Penultimate(`$C$`)` is the last but one state in a path (e.g., $Penultimate(S_2S_1S_7S_3) = S_7$).

Definition 4: A path C is *correct* if and only if for every sub-sequence $S_iE_jS_k$ of C; $\delta(S_i, E_j) = S_k$.

A state path $C : S_1S_5S_3 \cdots S_n$ is a *legal state path* if and only if for every sub-sequence S_iS_j of C, there exists an event $E \in \Sigma$ such that $\delta(S_i, E) = S_j$. Otherwise, the state path is an *illegal state path.*

An event path $C : E_1E_3E_2 \cdots E_n$ is a *legal event path* if and only if for every sub-sequence E_iE_j of C, there exists three states $(S_k, S_l, S_m) \in Q^3$ such that $\delta(S_k, E_i) = S_l$ and $\delta(S_l, E_j) = S_m$. Otherwise, the event path is an *illegal event path.*

3 Fault Detection Technique

At run time, messages are exchanged between communication entities. Let $E_1, E_2, ..E_k$ be a sequence of messages exchanged, where $E_1, E_2, ..E_k \in \Sigma$. Let S_0 be the initial state of the communication protocol, and $S_1, S_2, ..S_k$ be the subsequent states of the communication protocol such that $\delta(S_{i-1}, E_i) = S_i, i = 1..k$. The path represented by $S_0E_1S_1E_2S_2..E_kS_k$ is transformed into a number using a signature function Φ. This function is a polynomial with the state and event values as coefficients.

$\Phi : \Pi \rightarrow F$ represents the signature function that transforms a full path into a signature. F represents signature space. Π represents the set of all possible paths. The correct behavior of the communication protocol is represented by the set of all correct paths: Π_C. The signature corresponding to a correct path is known as the *static signature.* This signature is associated with the last state in the correct path. Thus every state has a static signature.

The correct behavior of the communication protocol is represented by the set of all correct paths: Π_C. The signature corresponding to a correct path is known as the *static signature.* This signature is associated with the last state in the correct path. Thus every state has a static signature. The run-time path which represents the sequence of messages exchanged between the entities is transformed into a signature using Φ. This represents the run-time signature. Let S_l be the last state in the run-time path. The run-time signature is compared with the static signature of S_l. A fault is detected when the run-time signature does not match the static signature of S_l.

4 Signature Generation

4.1 Requirements

Given a communication protocol modeled by a FSM, we would like to detect illegal paths. The signature function Φ which is used in the transformation must have the following property:

$$\exists p < 1; Probability[\Phi(C_1) = \Phi(C_2) | C_1 \neq C_2] < p \quad (1)$$

p is called the *masking probability* and a good signature function has a low masking probability.

Formally, the problem is stated as follows:

Given a FSM $A = (Q, \Sigma, \delta)$ and a state S_0 with predefined signature equal to zero (Without loss of generality, we can assume the signature of S_0 to be equal to zero) such that all the other states are reachable from S_0, we require a function that maps every path beginning at state S_0 onto a unique value from an algebraic field F.

In the following section, we introduce a signature-generation technique that satisfies property (1). Furthermore, to reduce the number of comparisons, the signature of all correct paths leading to a state S_n must be the same. This requires only one comparison when state S_n is reached.

4.2 Path Signature

We present three kinds of signatures: the *full-path*, the *state*, and the *event* signature. The main idea is to associate a path with a univariate polynomial. The signature of a path C is the evaluation of its associated polynomial at a given point x_0.

4.3 Choice of the Signature Function

A polynomial using the state and event information is used as signature function. We choose a polynomial signature function for the following reasons:

Polynomials are a well studied mathematical function and their properties are well understood. It is easy to compute the masking probability p defined in equation 1. The state and event-assignment problem

can be solved algebraically. The run-time signature can be computed in a step-wise manner using Horner rules.

4.3.1 Full-path signature

The computation of the *full-path* signature requires the state and event information. Each state and event is assigned a value that is used to compute the signature. The signature is evaluated using the path polynomial at a given point x_0. Using this signature one can detect illegal and incorrect paths. The states and the events must be visible to the signature generator. It is well suited for developing and implementing self-checking programs. The polynomial associated with a path C is defined as follows:

$P_C(x) = \sum_{i=0}^{n-1}(s_i x^{2(n-i)} + e_i x^{2(n-i)-1}) + s_n$

where:

s_i: state value, e_i: event value, n: length of the path.

The signature function Φ is defined as $\Phi(C) = P_C(x_0)$. The event (state) signature is computed using only the event (state) information. The event signature cannot detect all illegal state paths, and the state signature cannot detect all illegal event paths.

$P_C(x) = \sum_{i=0}^{n-1} s_i x^{n-i} + s_n$

$P_C(x) = \sum_{i=0}^{n-1} e_i x^{n-i-1}$

4.4 Probability of Detecting Illegal Paths

Theorem 1 *Prob($\Phi(C_1)$=$\Phi(C_2)$ | $C_1 \neq C_2$) = $\frac{1}{|F|}$ where $|F|$ is the size of the working algebraic field.*

Sketch of Proof: The probability that two different paths have the same signature is equal to the probability that two different polynomials evaluate to the same value at a given point x_0. This is equal to the probability that a polynomial evaluates to a value *sgn* at x_0. For any given working field F, the size of the set of polynomials of degree n is given by $|F|^{n+1}$. By symmetry, the size of the set of polynomials that evaluates to *sgn* at point x_0 is $|F|^n$. Let P_{C2} be the polynomial corresponding to path C_2. Then the probability that the polynomial P_{C2} evaluates to *sgn* is $|F|^n/|F|^{n+1} = 1/|F|$. □

Corollary 1 *The probability that an illegal path is undetected is equal to $\frac{1}{|F|}$, where $|F|$ is the size of the working algebraic field.*

Sketch of Proof: The probability that an illegal path is undetected is equal to the probability that an illegal path has the same predefined signature as the legal path. From theorem 1, this probability is equal to $\frac{1}{|F|}$. □

The probability of detecting an illegal path can be increased by choosing a large Galois field. However, a large Galois field increases the complexity of computing the signature and the size of the signature.

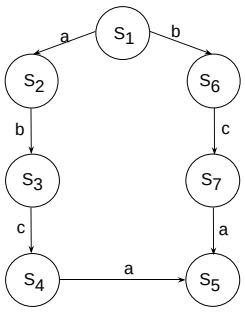

Figure 1: Parallel path.

4.5 Signatures of Parallel Paths

When many paths lead to the same state, they are called *parallel paths*. In order to reduce the number of comparisons, all parallel paths should result in the same signature. For example, in Figure 1 in the ideal case, the signature generated at the end of paths $S_1aS_2bS_3cS_4aS_5$ and $S_1bS_6cS_7aS_5$ are equal.

To solve this problem, we have to assign appropriate values to the events and states such that the signature of all parallel paths are equal. We denote this problem as the *event-state assignment* problem. When assigning values to the events and states, two different events or states have to be associated with two distinct values; otherwise they are indistinguishable. The same signature is computed for both correct and wrong event. This makes it impossible to detect an incorrect transition as a result of a wrong event.

As the signature of a path is equal to the evaluation of its associated polynomial at a given point, solving the event-state assignment problem is equivalent to solving a system of linear equations.

In the case of full-path signature, having the same signature for two paths is equivalent to solving a linear equation. In Figure 1, the linear equation is as follows:

$s_1x^8+ax^7+s_2x^6+bx^5+s_3x^4+cx^3+s_4x^2+ax+s_5 = s_1x^6 + bx^5 + s_6x^4 + cx^3 + s_7x^2 + ax + s_5$

This equation can be simplified as follows:

$s_1(x^8-x^6)+s_2x^6+s_3x^4+s_4x^2-s_6x^4-s_7x^2+ax^7 = 0$

The advantage lies in the fact that only one signature is required for comparison with the run-time signature. This significantly reduces the complexity of the observer compared with the state adjustment technique (It treats graphs with loops, in the same manner as graphs without loops and finally the state or event values can be computed by solving a linear set of equations). Its disadvantage is the restriction on the topology of the control flow graph.

Theorem 2 gives an idea about the structure of a graph for which the edge-assignment problem can be solved. The state-assignment problem and the event-state-assignment problems can be solved in a similar manner. We denote by $E^-(S)$ the set of edges that leads to state S and S^- the set of states S_i for which there exists an event e_i such that $\delta(S_i, e_i) = S$. S_0 is a state for which the signature is given an initial value (e.g., 0) and from which all other states are reachable.

Theorem 2 *The event-assignment problem is solvable if* $|E| > |E^-(S_0)| + \sum_{S \neq S_0}(|E^-(S)| - 1)$

Sketch of Proof: The event-assignment problem is equivalent to solving a system of linear equations. We have $|E|$ unknowns and $|E^-(S_0)| + \sum_{S \neq S_0}(|E^-(S)| - 1)$ equations. This is because for each state S different from S_0 we have $|E^-(S)|$ different paths. These paths have the same signature, which is equivalent to $|E^-(S)| - 1$ equations. For S_0, since the signature of all the paths that leads to it is predefined, we have $|E^-(S_0)|$ equations. Since this system of linear equations has a second member equal to 0, and the number of equations is greater than the number of unknowns, we can find a large number of solutions.

The only problem is that two different edges may be assigned the same value, and thus they are indistinguishable. □

We now describe the algorithm for generating the system of equations that has to be satisfied to solve the *state-assignment* problem. The values of the states can be computed by solving the system of equations generated by this algorithm. The system of equations for computing the values of the events can be generated in a similar manner. The functions used in this algorithm have been defined in section 2.1. S^- denotes the set of states s_i for which there exists an event e_i such that $\delta(S_i, e_i) = S$.

Algorithm 1 *state-assignment problem(input: A, output: System)*

$System \leftarrow \{\}$
For all $S \in Q - \{S_0\}$ such that $|S^-| > 1$
  Let $S_1 \in S^-; C_0$ a state path such that: $first(C_0) = S_0; last(C_0) = S; penultimate(C_0) = S_1$
  For all $S_i \in S^- - \{S_1\}$;
    Let C_i a state path such that $first(C_i) = S_0; last(C_i) = S; penultimate(C_i) = S_i$
    $eqn \leftarrow \Phi(C_0) = \Phi(C_i)$
    $System \leftarrow System \cup \{eqn\}$
If $|S_0^-| > 0$ **then**
  For all $S_i \in S_0^-$;
    Let C_i a state path such that $first(C_i) = S_0; last(C_i) = S_0; penultimate(C_i) = S_i$
    $eqn \leftarrow \Phi(C_i) = 0$
    $System \leftarrow System \cup \{eqn\}$
end

4.6 Computation of Run-time Signature

The signature can be generated at run-time in the following manner.

Let x_0 be a given value chosen from a *Galois* field, and $S_1E_1S_2E_2S_3E_3 \ldots E_{n-1}S_n$ be the full path. $s_1, s_2, s_3, \ldots s_n$ are the values of the states and $e_1, e_2, e_3, \ldots e_{n-1}$ the values of events. At run-time, the signature is computed after every event. Horner rules are used in computing the signature [8].

$Signature(i) := (Signature(i-1) * x_0 + s_i) * x_0 + e_i$

where $Signature(0) = 0$ and $Signature(i)$ is the signature after the i^{th} event has occurred.

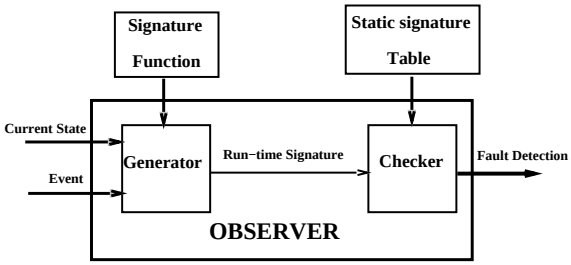

Figure 2: Observer Module

4.7 Advantages

1. Simple: It is step-wise computable and requires at most two addition and multiplication operations at each step.

2. State, event, and full-path signatures can be computed using similar polynomials and the technique described in section 4.6.

3. All correct paths leading to the same state have the same static signature. This reduces the number of comparisons to verify the correctness of the run-time path.

4. The state and event-assignment problem can be solved in an algebraic manner. This is elegant and simple compared to heuristic methods used for MISR (Multiple Input Shift Register) signatures used in [5].

5. The masking probability defined in section 4.1 can be easily computed.

5 Detection of Run-time Faults

An *observer* is used to detect run-time faults. Figure 2 shows the components of the observer. The observer consists of a *generator* and a *checker*. During the execution of the program, the *run-time* signature is computed by the generator (In this paper, the word *program* refers to the implementation of the communication protocol). The technique described in section 4.6 is used to generate the run-time signature. The checker compares the run-time signature with the static signature stored in a table called *static-signature table*. The comparison is performed when the program reaches the checking state.

When the observer is internal (i.e, it is a part of the program), we have a self-checking program. However, the observer and the program is subjected to the same perturbations. In the case of an external observer, the environment of the observer is different from the program (The environment of a program includes the hardware and the software required to execute the program). Hence, it is not affected by the perturbations of the program's environment. In this paper, we address the issues related to the external observer.

5.1 Static-signature Table

This table contains the signature corresponding to the correct execution of the communication protocol. The content of this table depends on the availability of the *current state* information to the observer.

If the current state information is available to the generator then this table has two columns: *checking*

state and *signature*. This table contains the static signature for every checking state. Otherwise, this table has two columns represented by *previous signature* and *current signature*. The signature from the previous comparison is used to find the static signature for the current comparison. After the comparison, the current signature becomes the previous signature for the next comparison. In this case, the checker needs to know when the program reaches a checking state.

It is possible to reduce the number of comparisons by decreasing the size of the set of checking states Q_c. When the size of Q_c is less than the size of Q, the FSM corresponding to the checker is reduced in size and is represented by (Q_c, Σ, δ_c). This also reduces the number of rows in the static signature table. The advantage of this technique is the reduction in the size of the table. However, this increases the *error latency*. Error latency is defined as the time between the occurrence of an error and its detection.

In the following sections, we present methods for detection faults. We discuss the structure of the static-signature table, requirements, advantages and disadvantages of each scheme.

5.2 Without State Information

In this case only the event information is available to compute the run-time signature. The checker is notified when the program reaches the checking state. When the notification is received, the checker compares the run-time signature with the static signature.

Since the external checker does not have access to the state information, the table contains the previous and current signature. The previous signature corresponds to the run-time signature when the last comparison was made. The comparison is made when the checker receives the notification from the program. After the comparison is made, the current signature becomes the previous signature for the next comparison.

The program must be modified to notify the checker when the program reaches the checking state. A single bit is sufficient for this notification. The generator and checker are independent of the program. The signature function and the static-signature table are the inputs to the observer.

5.3 With State Information

In this case, the *full-path* signature is computed based on the state and event information. This method requires the program under observation to transmit the state information. It is necessary to extend the communication protocol to support this requirement.

The static-signature table has two columns: *current state* and *signature*. The static signature gives the signature for the current state and is compared with the run-time signature. The size of the static-signature table can be reduced by reducing the size of the set of checking states.

The communication protocol must be extended to transmit the current state information. Every message that is exchanged between the entities must contain the current state of the program. The signature function, the static-signature table and the checking states are the inputs to the observer.

5.4 Discussion

In general, the size and complexity of the observer are lower compared to those of the communication protocol. This argument is supported by the dimensions of the static signature table. The FSM table corresponding to the program has two dimensions (state and event). The *current state* and *event* are used to compute the *next state*. The static-signature table has only one dimension (state). For the given state (previous signature), it gives the correct signature. The observer uses only the *current state* or the *previous signature* to determine the static signature. This reduces the complexity of the observer by one dimension. Moreover, the number of entries in the static-signature table can be minimized by reducing the number of checking states.

The fault-detection mechanism is independent of the program under observation. The *static-signature table* and the *signature function* depend on the communication protocol. They are the inputs to the observer. Therefore, the observer can be used to detect run-time faults in any communication protocol. Moreover, the observer is not affected by the perturbations of the program's environment. The overhead of transmitting the state information to the fault-detection module is higher than the overhead of transmitting the signal to perform the comparison. However, the probability of detecting a fault is higher when the state information is used to compute the run-time signature (refer to section 4.2).

5.5 Type of Detected Faults

1. Illegal state/event path: When the FSM incorrectly changes the state and traverses a different path, it results in an incorrect signature. The probability of detecting the illegal path is $1 - \frac{1}{|F|}$.

2. Out-of-order events: When the events occur out-of-order, the run-time signature does not match with the static signature. These type of faults are detected when the *full-path* or the *event* signature is used for comparison.

6 Example Using TP4

The FSM model of TP4 is shown in Figure 3(a). The visible events are CR, CC, AK, DR, DC, DT, CR', CC', and AK'. The input and output events are distinguished with a $'$. For the sake of clarity, the tuple *(CR, CC, AK, DR, DC, DT, CR', CC', AK')* is denoted by *(a,b,c,d,e,f,a',b',c')*. Using algorithm 1, the event-assignment problem is transformed into the following system of equations:

$$\begin{cases} \Phi(ab'c) & = & \Phi(a'bc') \\ \Phi(ab'cc) & = & \Phi(ab'c) \\ \Phi(ab'cc) & = & \Phi(ab'cf) \\ \Phi(ab'cde) & = & 0 \end{cases} \quad (2)$$

The third equation in system (2) gives the condition that events c and f must have the same value. The other equations of system (2) can be transformed

into a system of linear equations: $A[abcdea'b'c']^t = [00000000]^t$, where matrix A is given below:

$$A = \begin{bmatrix} x^2 & -x & 1 & 0 & 0 & -x^2 & x & -1 \\ x^4 & 0 & x^2 & x & 1 & 0 & x^3 & 0 \\ x^2 - x & 0 & 1 & 0 & 0 & 0 & x-1 & 1 \end{bmatrix}$$

Solving this system gives a solution: $[c'ec]^t = B[abda'b']^t$, where B is given below:

$$B = \begin{bmatrix} x & -x & 0 & -x^2 & 1 \\ -x^3 & 0 & -x & 0 & -x^2 \\ x - x^2 & 0 & 0 & 0 & 1-x \end{bmatrix}$$

If we assign the value 2 to x_0 and *(a,b,d,a',b')=(1,2,3,4,5)*, then the tuple *(c',e,c)* has to be equal to *(-13,-34,-7)*. The solution is as follows: *(a,b, c,d,e,a',b',c') = (1,2,-7,3,-34,4,5,-13)*. Figure 3(b) shows the static signature for every state. Table 1 gives the static-signature table corresponding to this FSM. This signature is computed using the event information.

Prev. Sg.	Current Sg.
0	4, 1
1	7
4	10
7	7, 17
10	7
17	0

Table 1: Static-Signature Table for 3(b).

Figure 4 shows the reduced FSM with the checking states, $Q_c = (Closed, Open)$. Table 2 gives the static signature corresponding to this FSM.

Prev. Sg.	Current Sg.
0	7
7	0, 7

Table 2: Reduced Static-Signature Table

The event-state-assignment can be solved in a similar manner. A possible solution is shown in Figure 3(c). This solution is *(Closed, WFCC, WFCC', Open, WFTRESP, AKWAIT, Closing) = (0, 0, 0, 0, 0, 2, 0)* and *(CR, CC, AK, DR, DC, DT, CR', CC', AK') = (1, 2, -30, 3, -172, -30, 4, 5, -62)*.

More details on solving the event-state-assignment problem are discussed in [7].

6.1 Fault Detection

We now describe the technique for generating the run-time signature for the TP4 protocol whose FSM is shown in Figure 3(a). This FSM corresponds to the correct execution of TP4. Table 3 shows the table of static signatures for this example.

Consider the following correct path: (Closed CR WFTRESP CC' AKWAIT AK Open). This corresponds to the connection-establishment phase of TP4. Table 4 shows the signature computed after each event. Both the event and state values are used in computing the signature. The value of x_0 is 2. The technique described in section 4.6 is used to compute the signature. The signature computed after each event corresponds to the static signature in Table 3. This means that the chosen path is correct.

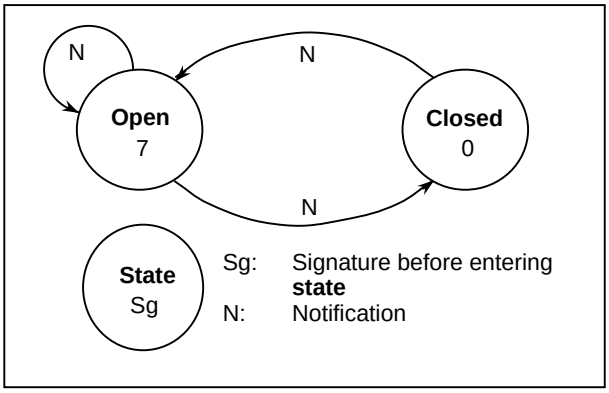

Figure 4: TP4 Protocol: Reduced FSM

State	Signature
Closed	0
WFCC	4
WFCC'	18
Open	10
WFTRESP	1
AKWAIT	9
Closing	43

Table 3: Static-Signature Table

Now, let us consider an incorrect path: (Closed CR WFTRESP DR Closing DC Closed). Table 5 shows the computation of the signature. After event *DR*, the run-time signature computes to 7, which does not match with the static signature of *Closing* (i.e., 43). Thus, a fault is detected after event *DR*.

Path	Computation	Sg.
Closed CR	$0 * 2 + 1$	*1*
Closed CR WFTRESP DR	$(1 * 2 + 0) * 2 + 3$	$7 \neq 43$

Table 5: Example 2: Incorrect Path

7 An Extended Transport Communication Protocol

In this section, we describe the extensions to the communication protocol required to support the computation and comparison of run-time signatures. The extensions do not modify the communication protocol but require the addition of some parameters to transmit the current state information. The original communication protocol can still be used to communicate with entities that use the extended communication protocol. Two extensions are proposed in this section.

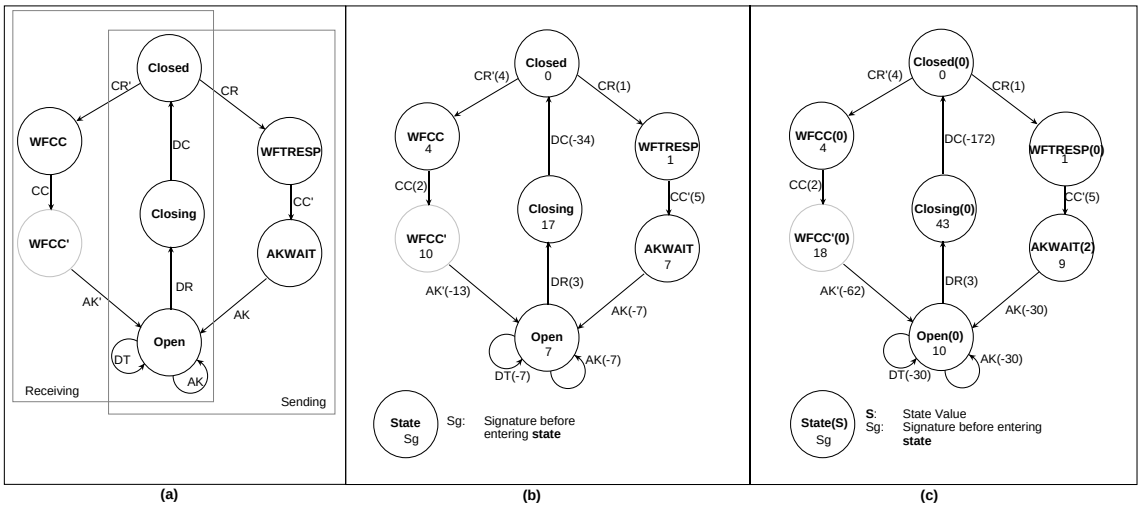

Figure 3: TP4 Protocol: state and event-assignment

Path	Computation	Sg.
Closed CR	$0 * 2 + 1$	*1*
Closed CR WFTRESP CC'	$(1 * 2 + 0) * 2 + 5$	*9*
Closed CR WFTRESP CC' AKWAIT AK	$(9 * 2 + 2) * 2 + (-30)$	*10*

Table 4: Example 1: Correct Path

Extension 1: The current state of the program is included in every message exchanged between the communication entities. Every state of the program is assigned an integer value called the *state value*. The state value is used to compute the run-time signature.

Extension 2: When the program reaches the checking state, a notification is included in the message. A single bit is sufficient to include this information. When this bit is set, the checker compares the run-time signature with the static signature for correctness.

Transport Class 4 (TP4) [1, 8] is used as an example to illustrate the two extensions. The proposed extensions to TP4 do not affect its functionality. The eXtended TP4 (eXTP4) can communicate with an existing TP4 implementation without any modifications to the TP4 implementation.

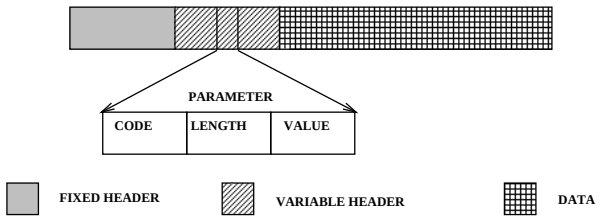

Figure 5: Transport Protocol Data Unit.

A Transport Protocol Data Unit (TPDU) has a fixed header, a variable header, and data (see Figure 5). The fixed header contains the mandatory parameters. The variable header contains the optional parameters. Each optional parameter in the header is encoded in the following manner:

parameter = (code, length, value)

The *code* represents the type of the parameter and the *length* indicates the number of bytes used to represent the *value*. The optional parameters in the TP4 header facilitates the inclusion of the *current state* value in every message. The state information is treated as an optional parameter. It is ignored by TP4 and can be used by eXTP4 to compute the run-time signature.

For TP4, the value of $|Q|$ is nine. This requires one byte to represent the state information. Four bits are sufficient to represent 9 states. However, the minimum length of the value is one byte. So we use only the first four bits of value to include the current state information. Three additional bytes are required to include the current state information in every message. Extension 2 requires only a single bit to be included in the message when the program reaches the checking state. Since the minimum length of the value of a parameter is one byte, three additional bytes are required to include this information.

The code for the new parameters (*current state* and *checking state*) must be chosen such that there is no conflict with the code of existing TPDU parameters.

7.1 Overhead Due to Extensions

Table 6 shows the length of headers of different Transport Protocol Data Units. The overhead for transmitting the current state or the checking state information is three bytes and the overhead percentage is calculated as a percentage of the header length.

It does not take into account the user data which is of variable length and is also a part of the TPDU.

PDU TYPE	*hdr length*	*Overhead(%)*
CR	*35*	*8.5*
CC	*35*	*8.5*
DT (Normal)	*8*	*37.5*
DT (Extended)	*11*	*27.3*
ED (Normal)	*8*	*37.5*
ED (Extended)	*11*	*27.3*
AK (Normal)	*12*	*25*
AK (Extended)	*17*	*17.6*
AKf(Normal)	*23*	*13*
AKf(Extended)	*27*	*11.1*
EA (Normal)	*8*	*37.5*
EA (Extended)	*11*	*27.3*
DR	*10*	*30*
DC	*9*	*33.3*

Table 6: Overhead for transmitting state information

7.2 Discussion

The number of bits required to include the state information is $log_2|Q|$, where $|Q|$ represents the number of states of the communication protocol. Only one bit has to be included in the message when the program reaches the checking state. In general, the communication overhead for Extension 1 is much more than that of Extension 2. The overhead of Extension 2 is independent of $|Q|$. On the contrary, the overhead of Extension 1 depends on $|Q|$. However, this overhead permits the detection of a broader class of protocol faults. In the case of TP4, both the extensions require the same number of bytes. Extension 1 includes the current state information in every message. Extension 2 does not include the checking state information in every message. Thus, Extension 2 still has less overhead compared to Extension 1.

It is possible to reduce the overhead of each extension *modifying* the protocol. The information about the current state can be encoded in $log_2|Q|$ bytes. The information for the checking state can be included in a message using a single bit. This results in a modification of the protocol and requires new encoders and decoders to exchange the messages. The modified protocol cannot communicate with the original protocol.

8 Summary and Conclusions

In this paper, we have introduced a new method for generating path signatures based on polynomials. This method is simple, step-wise computable and requires at most two addition and multiplication operations for each computation of the run-time signature. This method generates unique signatures that can detect *illegal transitions* and *out-of-order* events. Hence, this method is more reliable than other existing methods which cannot detect out-of-order events. Our *observer* is much simpler compared to the observers used in FSM-based methods. We have proposed extensions to communication protocols to facilitate the application of this method to detect run-time faults in communication protocols. We have also shown how the extensions can be applied to TP4 without modifying the original protocol, though some additional overhead is produced. *eXTP4* is an extended Transport Protocol that facilitates run-time fault detection using our signature based technique.

Acknowledgments

The authors are grateful to Dr. Berthe Choueiry for reading the paper and providing valuable comments. This paper was partially supported by the Swiss PTT Project F&E N°309.

References

[1] Transport Protocol Specification for Open System Interconnection for CCITT Applications, CCITT X.224, ISO 8073. International Telecommunication Union, 1988.

[2] A. Bouloutas, G.W. Hart, and M. Schwartz. Simple Finite-State Fault Detectors for Communication Networks. *IEEE Trans. on Communications*, 40(3):477–479, March 1992.

[3] M. Diaz, G. Juanole, and J. Courtait. Observer-A concept for Formal On-Line Validation of Distributed Systems. *IEEE Trans. on Software Engg.*, 20(12):900–912, 1994.

[4] B.T. Doshi, P.K. Johri, A.N. Netravali, and K. Sabnani. Flow Control Performance of a High Speed Protocol. *IEEE Trans. on Communications*, 41(5):707–719, 1993.

[5] Régis Leveugle. *Analyse de Signature et Test en Ligne Integré sur Silicium.* PhD thesis, Institut National Polytechnique de Grenoble, 1990.

[6] A.N. Netravali, W.D. Roome, and K. Sabnani. Implementation of a High Speed Transport Protocol. *IEEE Trans. on Communications*, 38(11):2010–2023, 1990.

[7] G. Noubir and K. Vijayananda. Signature-based Fault Detection for Communication Protocols. Technical Report 94/93, DI-LIT, Swiss Federal Institute of Technology at Lausanne, 1995.

[8] H.J. Nussbaumer. *Computer Communication Systems*, volume 1 & 2. Wiley Chichester, 1989.

[9] M. Riese. *Model-Based Diagnosis of Communication Protocols.* PhD thesis, Swiss federal Institute of Technology at Lausanne, Switzerland, 1993.

[10] K. Sabnani and A. Netravali. A High Speed Transport Protocol for Datagram/Virtual Circuit Networks. In *SIGCOMM'89,Symposium on Communications Architecture and Protocols*, 1989.

[11] C. Wang and M. Schwartz. Fault Detection with Multiple Observers. *IEEE Trans. on Networking*, 1(1):48–55, 1993.