Chapter 14

FORMAL MODELING AND ANALYSIS OF THE MODBUS PROTOCOL

Bruno Dutertre

Abstract Modbus is a communication protocol that is widely used in SCADA systems and distributed control applications. This paper presents formal specifications of Modbus developed using PVS, a generic theorem prover; and SAL, a toolset for the automatic analysis of state-transition systems. Both formalizations are based on the Modbus Application Protocol, which specifies the format of Modbus request and response messages. This formal modeling effort is the first step in the development of automated methods for systematic and extensive testing of Modbus devices.

Keywords: Modbus, formal methods, modeling, test-case generation

1. Introduction

A distributed control system—sometimes called a SCADA system—is a network of devices and computers for monitoring and controlling industrial processes such as oil production and refining, electric power distribution, and manufacturing plants. The network requirements for these applications include real-time constraints, resilience to electromagnetic noise, and reliability that are different from those of traditional communication networks. Historically, the manufacturers of control systems have developed specialized, often proprietary networks and protocols, and have kept them isolated from enterprise networks and the Internet.

This historical trend is now being reversed. Distributed control applications are migrating to networking standards such as TCP/IP and Ethernet. This migration is enabled by the increased sophistication of control devices. Also, the migration provides increased bandwidth and functionality, and economic benefits. Control systems are now using the same technologies and protocols as communication networks, and the separation that existed between control networks and other networks is disappearing. SCADA systems are now often connected to conventional enterprise networks, which are often linked to the Internet.

This interconnection increases the risk of remote attacks on industrial control systems, which could have devastating consequences. Intrusion detection systems and firewalls may provide some protection, but it is important that the control devices are reliable and resilient to attacks. Detecting and eliminating vulnerabilities in these devices is essential to ensuring that industrial control networks are secure.

Extensive testing is a useful approach to detecting vulnerabilities in software. It has the advantage of being applicable without access to the source code. As is well known, security vulnerabilities often reside in parts of the software that are rarely exercised under normal conditions. Traditional testing methods, which attempt to check proper functionality under reasonable inputs, can fail to detect such vulnerabilities. To be effective, security testing requires wide coverage. The set of test-cases must cover not only normal conditions, but also inputs that are not likely to be observed in normal device use. Indeed, flaws in handling malformed and unexpected inputs have been exploited by many attacks on computer systems, including the ubiquitous buffer-overflow attacks.

A major challenge to exhaustive testing is the generation of relevant test-cases. It is difficult and expensive to manually generate large numbers of test-cases to achieve adequate coverage. An attractive alternative is to generate test-cases automatically using formal methods. This is accomplished by constructing test-cases mechanically from a specification of the system under test and a set of testing goals called "test purposes." The concept has been successfully applied to hardware, networking and software systems [1, 5–7, 12, 13]. This paper explores the application of similar ideas to SCADA devices. More precisely, we focus on devices that support the Modbus Application Protocol [8], a protocol widely used in distributed control systems.

Automated test-case generation for Modbus devices requires formal models of the protocol that serve as a reference, and algorithms for automatically generating test-cases from such models. We present two formal models that satisfy these goals. The first model is developed using the PVS specification and verification system [11]. This model captures the Modbus specifications as defined in the Modbus standard [8]: it includes a precise definition of valid Modbus requests and, for each request class, the specification of acceptable responses. The PVS model is executable and can be used as a reference for validating responses from a device under test. Given a test request r and an observed response m, it is possible to determine whether the device passes or fails the test by executing the PVS model with input r and m.

The second model is designed for automated test generation, i.e., for constructing Modbus requests that satisfy a test purpose. This model is developed using the SAL environment for modeling and verifying state transition systems [2, 3]. In this approach, test-case generation is translated to a state-reachability problem that can be solved using SAL's model checking tools. This approach is more efficient and more powerful than attempting to generate test-cases directly from PVS specifications.

The Modbus standard is very flexible, and devices are highly configurable. Modbus-compliant devices may support only a subset of the defined functions, and, for each function, devices may support a subset of the possible parameters. Several functions are left open as "user definable." To be effective, a testing strategy must be tailored to the device of interest and specialized to the functions and parameters supported by the device. Special care has been taken to address this need for flexibility. The formal models presented in this paper are easily customized to accommodate the different features and configurations of Modbus devices.

2. Modbus Protocol Overview

Modbus is a communication protocol widely used in distributed control applications. Modbus was introduced in 1979 as a serial-line protocol for communication between "intelligent" control devices. It has become a *de facto* standard implemented by many manufacturers and used in a variety of industries.

The Modbus serial-line specifications describe the physical and link-layer protocols for exchanging data [10]. Two main variants of the link-layer protocol are defined and two types of serial lines are supported. In addition, the specifications define an application-layer protocol, called the Modbus Application Protocol, for controlling and querying SCADA devices [8].

Modbus was subsequently extended to support other types of buses and networks. The Modbus Application Protocol assumes an abstract communication layer that allows devices to exchange small packets. Serial-line Modbus remains an option for implementing this communication layer, but other networks and protocols may be used. Many modern SCADA systems implement the communication layer using TCP, as described in the Modbus over TCP/IP specifications [9].

2.1 Modbus Serial Protocol

Figure 1 presents a typical architecture employing the Modbus serial protocol. Several devices are connected to a single bus (serial line) and communicate with a central controller. Modbus uses a master-slave approach to control access to the shared communication line and prevent message collisions. Communication is initiated by the controller (master device), which issues commands (requests) on the bus, usually destined for a single device (slave). This slave device may then access the bus and transmit the response to the master. Slave devices do not communicate directly with each other, nor do they transmit data without a request from the master device. Modbus also permits multicast messages from a master to several slave devices, but these transactions have no response messages.

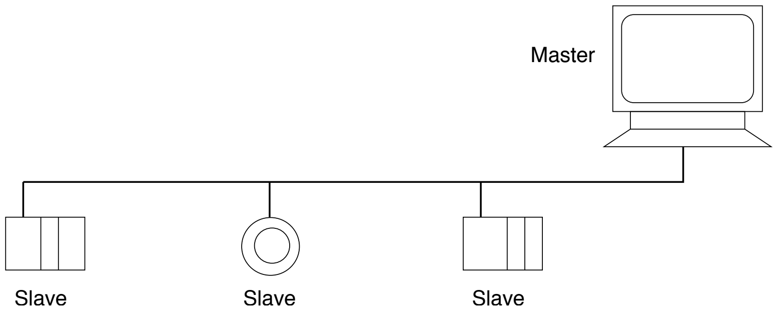

Figure 1. Modbus serial-line architecture.

The Modbus specification defines a physical layer and describes packet formatting, device addressing, error checking and timing constraints. Several design decisions have an impact on the Modbus Application Protocol and on Modbus over TCP/IP.

- The application protocol follows the same master-slave design as the serial line protocol. Each transaction in the application layer is a simple request-response exchange initiated by the master and addressed to a single slave device.
- Modbus requests and responses are required to fit in a single serial-line frame. The maximum length of a Modbus frame is 256 bytes. One byte is reserved for the device address and two bytes for CRC error checking. Therefore, the maximum length of a request or response is 253 bytes.

2.2 Modbus TCP Protocol

The Modbus TCP protocol uses TCP as the communication layer, but attempts to remain compatible with the Modbus serial protocol. The specification defines an embedding of Modbus packets into TCP frames and assigns a specific IP port number (502) for Modbus TCP. The frame includes the usual IP and TCP headers, followed by a Modbus-specific header and the payload. The payload is limited to 253 bytes to maintain compatibility with the Modbus serial protocol. Several fields in the Modbus TCP header are also inherited from the serial protocol [9].

Modbus TCP has economic advantages because of the wide availability of TCP- and TCP/IP-compatible networks. It is also more flexible. However, from a security perspective, migrating to TCP/IP introduces vulnerabilities and adds considerable complexity. The master-slave architecture presented in Figure 1 can be implemented using relatively simple devices since most of the protocol control and functionality are incorporated in the master device. The situation is reversed in Modbus TCP: the master is a "TCP client" and the slave devices are "TCP servers." As far as networking is concerned, devices that support Modbus TCP must implement all (or a significant subset) of the features of a TCP/IP server. The TCP client/server semantics is also more general than the simple master-slave model. For example, multiple Modbus transactions can be sent concurrently to a single device and a device may accept connections from different clients. Interested readers are referred to [9] for additional details and guidance on Modbus TCP implementations.

2.3 Modbus Application Protocol

The application protocol is common to all variants of Modbus. As mentioned earlier, this protocol is very simple as it was originally intended for the Modbus master-slave protocol on serial lines. Almost all transactions consist of a request sent by the master to a single slave, followed by a response from the slave device. The few exceptions are transactions involving requests sent by the master that require no response messages on the part of the slave devices.

The majority of Modbus requests are commands to read or write registers in slave devices. The Modbus standard defines four main classes of registers:

- **Coils:** Single-bit, readable and writable registers
- **Discrete Inputs:** Single-bit, read-only registers
- **Holding Registers:** 16-bit, readable and writable registers
- **Input Registers:** 16-bit, read-only registers

Individual registers in each category are identified by a 16-bit address. There is no requirement for a Modbus device to support the full range of addresses or the four types of registers. The four address spaces are allowed to overlap. For example, a single control bit may have its own address as a coil and be part of a 16-bit holding register.

Table 1 shows the format of a Modbus command and the associated responses. The first byte of the request identifies a specific command, in this example, function code `0x02` corresponding to *Read Discrete Inputs*. The rest of the request contains parameters. The example command has two parameters: a start address between `0x0000` and `0xFFFF` (in hexadecimal) and the number of discrete inputs to read expressed as a 16-bit integer. The device may either reject the request and send an error packet or return the requested data in a single packet. The format of a valid response is also indicated in Table 1: the first byte is a copy of the function code in the request, the second byte contains the size of the response (if n bits are requested, this byte is $\lceil n/8 \rceil$), and the rest of the packet is the data itself. An error packet contains a copy of the function code of the request with the high-order bit flipped and an exception code that indicates the reason for the failure. Figure 2 summarizes how the request should be processed and how the exception code should be set in case of failure.

Read and write commands are all similar to the example in Table 1. Other commands in the Modbus Application Protocol are related to device identification and diagnostics. Every command starts with a one-byte function code,

Table 1. Example Modbus command and associated responses.

Request		
Function Code	1 byte	0x02
Start Address	2 bytes	0x0000 to 0xFFFF
Quantity	2 bytes	1 to 2000

Response		
Function Code	1 byte	0x02
Byte Count	1 byte	N
Data	N bytes	

Error		
Error Code	1 byte	0x82
Exception Code	1 byte	01 to 04

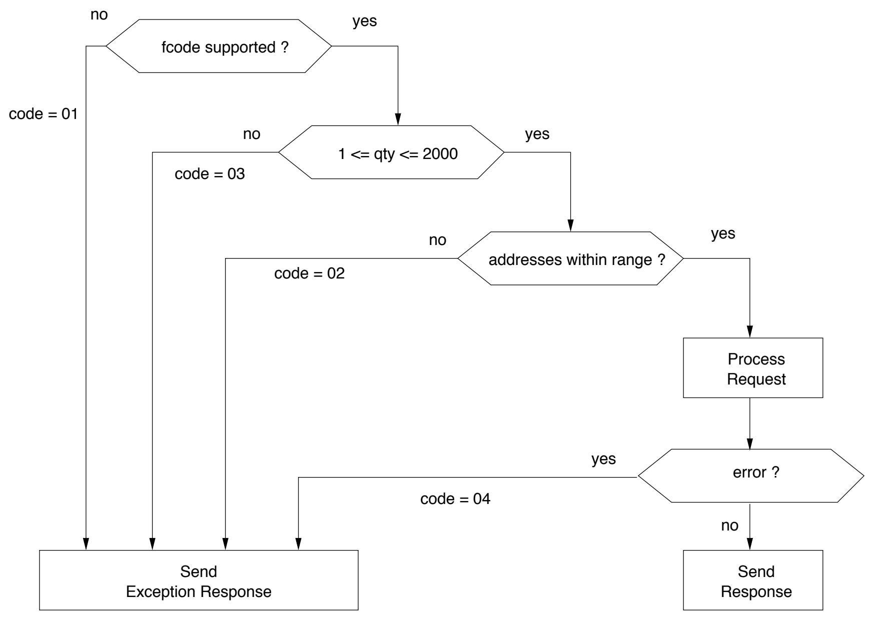

Figure 2. Example command (processing and error reporting).

a value between 1 and 127. Following the function code are parameters that are specific to the function code (e.g., register addresses and quantity) and optionally other data (e.g., values to write in registers). Correct execution is indicated by sending back a response packet whose first byte (function code) is the same as the command, and other data (e.g., response to a read command). A failure to execute the command is indicated by responding with a two-byte error packet.

The Modbus standard defines the meaning of nineteen of the 127 possible function codes. The other function codes are either unassigned and reserved for future use, or they are reserved for legacy implementations and are not available, or they are user defined. A device is allowed to support only a subset of the public functions, and within each function code to support only a subset of the parameters. The only requirement is for the device to return an appropriate error packet to signal that a function is not supported or that an address is out of range.

The protocol standard is loose and flexible. Modbus-compliant devices may vary widely in the functions, number of registers and address spaces they support. The interpretation of user-defined function codes is not specified by the standard and different devices may assign different interpretations to the same user-defined code.

3. Formal Specification

The formal models discussed in this paper deal with the Modbus Application Protocol. As summarized previously, the transactions in this protocol consist of a single request followed by a single response. Both requests and responses are small packets of at most 253 bytes in length. Because it is so simple, the Modbus Application Protocol is not a good candidate for traditional formal verification, whose goal is typically to prove some non obvious but critical property. Instead, our formal modeling and analysis efforts are geared to support extensive, automated testing of Modbus devices.

Given a formal model, we show how to automatically derive test scenarios, i.e., specific Modbus requests, and check whether the device issues correct responses. Because test-cases can be generated automatically and the model can be specialized for a given device, this approach enables extensive testing, beyond checking for compliance with the Modbus standard. For example, the method enables tests of how a device responds to a variety of malformed requests that it would not receive during normal operation (e.g., requests that are too long or too short, and requests containing bad function codes or unsupported addresses). Our goal is to help detect vulnerabilities in Modbus devices, including buffer overflows and other flaws.

3.1 PVS Model

Our first formal model of Modbus was developed using the PVS specification and verification system [11]. PVS is a general-purpose interactive theorem prover based on higher-order logic. Details on the PVS specification language, theorem prover and other features may be found on the PVS website (pvs.csl.sri.com). Our full PVS specification of the Modbus Application Protocol is available in an extended version of this paper [4]. The specification is a straightforward formalization of the Modbus Application Protocol standard [8]. The PVS model defines the exact format of the Modbus requests and, for each request type, it specifies the format of the valid responses and possible error messages.

The definition of well-formed requests is summarized as follows:

- The type `raw_msg` represents arbitrary packets (raw messages), which are modeled as byte arrays of length from 1 to 253.
- Correct formatting of requests is defined by a succession of predicates and subtypes of `raw_msg`:
  - Given a raw message `m`, `standard_fcode(m)` holds if the function code of `m` is one of the nineteen assigned codes. A `pre_request` is a raw message that satisfies the predicate `standard_fcode`.
  - Given a prerequest `sr`, `acceptable_length(sr)` holds if the length of `sr` is within the bounds specified by Modbus for the function code of `sr`. A `request` is a prerequest that satisfies `acceptable_length`.
  - Given a request `r`, `valid_data(r)` holds if `r` is well formed. This predicate captures constraints on the number and ranges of request parameters that depend on the function code.

In summary, a valid request is a raw message that satisfies the three predicates `standard_fcode`, `acceptable_length` and `valid_data`.

The definition of acceptable responses to a given request follows the same general scheme and involves a series of PVS predicates. The main predicate `acceptable_response(v, r)` is `true` whenever `r` is a possible response to a valid request `v`. The definition takes into account the function code and parameters of `v` and checks that the response `r` (raw message) has the appropriate format.

The predicates and types summarized so far are device neutral. They capture the general formatting requirements of Modbus [8]. Since devices are not required to implement all the functions and since different devices may support different address ranges, it is useful to specialize the PVS specifications to device characteristics and configurations. For this purpose, the PVS model is parameterized and includes device-specific properties such as supported function codes and valid address ranges for coils, discrete inputs, holding registers and input registers. Once these are specified, the final PVS definition is the predicate `modbus_response(m, r)` that captures formatting and device-specific constraints. The predicate holds when `r` is a response for the given device to a properly-formatted request `m`. Constraints on error reporting are included in this definition.

3.2 Applications

The PVS formalization is as an unambiguous and precise specification of the Modbus Application Protocol. The specification is executable, so it can be used as a reference implementation. For example, it is possible to check that the responses to the bad requests `[|0|]` and `[|1|]` are as specified in the standard:

```
modbus_resp([| 0 |], illegal_function(0));
==>  TRUE

modbus_resp([| 1 |], illegal_function(1));
==>  FALSE

modbus_resp([| 1 |], illegal_data_value(1));
==>  TRUE
```

Request `[|0|]` is a packet containing the single byte 0, i.e., a packet of length 1 with invalid function code 0. The corresponding response must be the packet `illegal_function(0)`. For packet `[|1|]`, the function code is valid and supported by the device, but the format is incorrect. The error code in such a case is required to be `illegal_data_value(1)`. The following examples illustrate the responses to a read command:

```
modbus((: READ_COILS, 0, 10, 0, 8 :), (: READ_COILS, 1, 165 :));
==>  TRUE

modbus((: READ_COILS, 0, 10, 0, 8 :), (: READ_COILS, 10, 165 :));
==> FALSE

modbus((: READ_COILS, 0, 10, 0, 8 :), (: READ_COILS, 2, 165, 182 :));
==> FALSE
```

The read command above is a request for the values of 8 coils starting at address 10. A valid response must consist of a copy of the function code `READ_COILS`, followed by a byte count of 1, followed by an arbitrary 8-bit value (first line above). The second line shows a badly-formatted response with an incorrect byte count of 10. In the third line, the response has the correct format, but it does not match the request because it returns the values of 16 instead of 8 coils.

4. Automated Test-Case Generation

Traditional software testing typically relies on exercising a piece of software using hand-crafted inputs. This method does not scale well for moderately complex software, as the cost of generating interesting test-cases by hand is prohibitive. In many cases, it is possible to automate the generation of test-cases from formal specifications. This section outlines such an approach, which is developed to test devices that run the Modbus Application Protocol.

Most test-case generation tools require a model of the expected behavior (such as the PVS specifications of Modbus presented in the previous section). The goal is to generate input data for the system under test and check whether the system's behavior in response to the input data satisfies the specifications. To guide the search, additional constraints may be specified on the input data so that particular aspects of the system under test are exercised. The extra constraints are often called "test purposes" or "test goals." For example, one may want to test the response of a device to a specific class of commands by giving an adequate test purpose.

To some extent, PVS and the formal specifications presented previously can be used for test-case generation. This is accomplished by using PVS's built-in mechanisms to find counterexamples to postulated properties. This method is explained in detail in [4]. However, a test-case generation procedure using PVS is limited because it relies exclusively on random search. Specifically, the PVS procedure randomly generates arrays of bytes of different lengths, and checks these byte arrays until one is found that satisfies the test purpose. For many test purposes, such a naïve random search has a low probability of success.

To address this issue, we have constructed a new model of Modbus, which is specifically intended for test-case generation. This model was developed using the SAL toolkit. SAL provides many more tools for exploring models and searching for counterexamples than PVS, and is, thus, better suited for test-case generation. In particular, a test purpose can be encoded as a Boolean satisfiability problem and test-cases can be generated using efficient satisfiability solvers.

4.1 SAL Model

SAL, the Symbolic Analysis Laboratory, is a framework for the specification and analysis of concurrent systems modeled as state transition systems. SAL is less general than PVS, but it provides more automated forms of analysis, including several symbolic model checkers, a bounded model checker based on SAT solving for finite systems, and a more general bounded model checker for infinite systems. Descriptions of these tools and the SAL specification language can be found in [2, 3]. Additional details are available at the SAL website (sal.csl.sri.com).

The SAL model is presented in [4]. The model is intended to support automated test-case generation by constructing Modbus requests that satisfy given constraints (test purposes). For this reason, and unlike the PVS formalization discussed previously, the SAL model covers only half of the Modbus Application Protocol, namely, the formatting of Modbus requests.

The SAL model relies on the observation that the set of well-formatted Modbus requests is a regular language; such a language can be recognized by a finite-state automaton. In fact, the SAL model is essentially a finite-state automaton written in SAL notation. This automaton is defined as module `modbus` whose state variables are shown in Figure 3 (in SAL, "module" is a synonym for state-transition system). The `modbus` module has a single one-byte input variable `b`. Its internal state consists of the ten local variables listed in Figure 3. All these variables have a finite type, so the module is a finite state machine.

```
INPUT
   b: byte
LOCAL
   aux: byte,
   stat: status,
   pc: state,
   len: byte,
   fcode: byte,
   byte_count: byte,
   first_word: word,
   second_word: word,
   third_word: word,
   fourth_word: word
```

Figure 3. Interface of the `modbus` module in SAL.

The main state variables are `pc` and `stat`, which record the current control state of the module and a status flag. Informally, the SAL module reads a Modbus request as a sequence of bytes on input variable `b`. Each input byte is processed and checked according to the current control state `pc`. The check depends on the position of the byte in the input sequence and on the preceding bytes. If an error is detected, a diagnostic code is stored in the `stat` variable; otherwise, the control variable `pc` is updated and the module proceeds to read the next byte. Processing of a single packet terminates when a state with `pc = done` is reached. At this point, `stat = valid_request` if the packet is well formed or `stat` has a diagnostic value that indicates why the packet is not well formed. For example, the diagnostic value may be `length_too_short`, `fcode_is_invalid`, `invalid_address`, and so on. In addition to computing the status variable, the SAL module extracts and stores important attributes of the input packet such as function code and packet length.

Figure 4 shows a fragment of the SAL specifications that extracts the first word of a packet (16-bit number that follows the function code). The specification uses guarded commands, written `condition --> assignment`. The condition refers to the current state and the assignment defines the values of the variables in the next state. The identifier `X` refers to the current value of a variable `X`, and `X'` refers to the value of `X` in the next state.

Default processing of the first word is straightforward: when control variable `pc` is equal to `read_first_word_byte1`, the first byte of the word is read from input `b` and stored in an auxiliary variable `aux`. Then, `pc` is updated to `read_first_word_byte2`. On the next state transition, the full word is computed from `aux` and `b`, and stored in variable `first_word`. Special checking is required if `fcode` is either `DIAGNOSTIC` or `READ_FIFO_QUEUE`. Otherwise, control variable `pc` is updated to read the second word of the packet. If the function code is `DIAGNOSTIC`, additional checks are performed on the first word and state variable `stat` is updated. An invalid word is indicated by setting `stat` to `diagnostic_subcode_is_reserved`. Otherwise, `stat` is set either to `valid_request` (to indicate that a full packet was read with no errors) or to

```
[] pc = read_first_word_byte1 -->
       aux' = b;
       pc' = read_first_word_byte2

[] pc = read_first_word_byte2 AND fcode = DIAGNOSTIC -->
       first_word' = 256 * aux + b;
       stat' = IF reserved_diagnostic_subcode(first_word')
                THEN diagnostic_subcode_is_reserved
                ELSIF first_word' = RETURN_QUERY_DATA
                THEN valid_request
                ELSE unknown
                ENDIF;

       aux' = len - 3;
       %% aux' = number of extra bytes for RETURN_QUERY_DATA

       pc' = IF reserved_diagnostic_subcode(first_word')
              THEN done
              ELSIF first_word' = RETURN_QUERY_DATA
              THEN read_rest
              ELSE read_second_word_byte1
              ENDIF

[] pc = read_first_word_byte2 AND fcode = READ_FIFO_QUEUE -->
       first_word' = 256 * aux + b;
       stat' = valid_request;
       pc' = done

[] pc = read_first_word_byte2 AND fcode /= DIAGNOSTIC AND
       fcode /= READ_FIFO_QUEUE -->
       first_word' = 256 * aux + b;
       pc' = read_second_word_byte1
```

Figure 4. SAL Fragment: Reading the first word of a packet.

`unknown` (to indicate that more input must be read and more checking must be performed).

Just like the PVS model discussed previously, the SAL model can be specialized to the features of a given Modbus device. For example, one may specify the exact set of function codes supported by the device and the valid address ranges for each function.

4.2 Test-Case Generation Using SAL

By using a state-machine model to specify the format of Modbus requests, the problem of test-case generation is transformed into a state-reachability problem. Given an input sequence of n bytes $b_1, \dots, b_n$, the SAL `modbus` machine performs n state transitions and reaches a state s_n. We determine whether $b_1, \dots, b_n$ is a well-formed Modbus request by examining the values of variables `pc` and `status` in state s_n:

- if `pc` = `done` and `stat` = `valid_request` then $b_1, \dots, b_n$ is a well-formatted request;

- if `pc` $=$ `done` and `stat` $\neq$ `valid_request` then $b_1, \ldots, b_n$ is an invalid request;
- if `pc` $\neq$ `done` then the status of $b_1, \ldots, b_n$ is not known yet (this means that $b_1, \ldots, b_n$ is an incomplete packet that may extend to a valid request).

To generate an invalid Modbus request of length n, we search for a sequence $b_1, \ldots, b_n$ that satisfies the second condition. This problem is solved using SAL's bounded model checker. In general, a bounded model checker searches for counterexamples to a property P that have a fixed length n. In SAL, given a state machine M, a counterexample is a finite sequence of n state transitions:

$$s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$$

such that s_0 is an initial state of M and one of the states s_i violates P. The property must be negated to use bounded model checking for test-case generation. For example, to obtain an invalid packet, we search for a counterexample to the following property:

```
test18: LEMMA modbus |- G(pc = done => stat /= invalid_data);
```

This lemma states that property `pc` $=$ `done` $\Rightarrow$ `stat` $\neq$ `invalid_data` is an invariant of module `modbus`. In other words, it postulates that in all reachable states of `modbus`, either `pc` $\neq$ `done` or `stat` $\neq$ `invalid_data`. This property is not true, and a counterexample is exactly what is needed: a sequence of bytes that reaches a state where `pc` $=$ `done` and `stat` $=$ `invalid_data`. Counterexamples to this property can be obtained by invoking SAL's bounded model checker as follows:

```
sal-bmc flat_modbus test18 -d 20
```

This searches for a counterexample to lemma `test18` defined in input file `flat_modbus.sal`. The option `-d 20` specifies a search depth of 20 steps. The resulting counterexample, if any, will be of length 20 or less. The counterexample produced is the sequence of bytes $[4, 128, 0, 254, 64]$. The state reached after this byte sequence is:

```
stat = invalid_data
pc = done
len = 5
fcode = 4
byte_count = 0
first_word = 32768
second_word = 65088
```

The values of `pc` and `stat` are as required; the other variables give more information about the packet generated: its length is 5 bytes and the function code is 4 (*Read Input Register*). For this command, the first word is interpreted as the address of a register and the second word as the number of registers to read. The second word is incorrect because the maximum number of registers allowed in the command is 125. As defined in the Modbus standard, the answer to this command must be an error packet with a code corresponding to "invalid data." Property `test18` is a test purpose designed to construct such an invalid request. By modifying the property, it is possible to search for test cases that satisfy other constraints. For example, the lemma:

```
test20: LEMMA modbus |-
          G(pc = done AND stat = valid_request => len < 200);
```

is a test purpose for a valid request of at least 200 bytes. More complex variants are possible. A variety of scenarios are discussed in [4].

The search algorithm employed by the finite-state bounded model checker is based on converting the problem into a Boolean satisfiability (SAT) problem and using a SAT solver. Any solution to the resulting SAT problem is then converted back to a sequence of transitions, which forms a counterexample or test-case. Although SAT solving is NP-complete, modern SAT solvers can routinely handle problems with millions of clauses and hundreds of thousands of variables. This approach to test-case generation is much more efficient than the random search used with PVS. In SAL, test-case generation is guided by the test purpose. Because the SAT solver used by SAL is complete, the method finds a solution whenever one exists. Given any satisfiable test purpose, `sal-bmc` will generate a test-case. For example, `sal-bmc` can easily construct valid requests, which have a very low probability of being generated by the PVS random search.

The time required for generating test-cases is short, usually a few seconds when running `sal-bmc` on a 3 GHz Intel PC. However, the cost of the search grows as the search depth increases, since the number of variables and clauses in the translation to SAT grows linearly with the depth. Longer test-cases are more expensive to construct; still, the time required is acceptable in most cases. For example, any counterexample to `test20` must be at least 200 bytes long. Finding such a counterexample requires a search depth at least as high; `sal-bmc -d 204` finds such a solution in 80 seconds by solving a SAT problem with more than 500,000 Boolean variables and 2,000,000 clauses. Thus, a large set of test-cases can be generated for many scenarios at little cost. This enables extensive testing of device compliance with Modbus specifications as well as testing for device vulnerabilities by generating a variety of malformed requests. It is also possible to target a specific model or device configuration by modifying the device-specific features of the SAL model. A large number of device-specific test-cases can be generated automatically within a few minutes.

5. Conclusions

The framework presented in this paper supports the systematic and extensive testing of control devices that implement the Modbus Application Protocol. It helps increase the robustness and security of Modbus devices by detecting vulnerabilities that conventional testing methods may easily miss. The framework relies on two main components. The first is a formal PVS specification of the Modbus protocol, which serves as a reference when checking whether or not device responses to test requests satisfy the standard. The second component is a state-machine model of Modbus requests, which functions as an automated test-case generator. Both the models accommodate the variability in Modbus functions and parameters. Furthermore, they are parameterized and can be rapidly specialized to the features of the Modbus device being tested.

Our future research will adapt the formal models to facilitate online monitoring of Modbus devices; by monitoring Modbus requests and responses, it should be possible achieve accurate, high-coverage intrusion detection. Another research objective is to automatically derive intrusion detection sensors from formal models of the expected function and behavior of Modbus devices in a test-case environment.

Acknowledgements

This work was partially supported by the Institute for Information Infrastructure Protection (I3P) under Award 2003-TK-TX-0003 from the Science and Technology Directorate of the U.S. Department of Homeland Security.

References

[1] P. Ammann, P. Black and W. Majurski, Using model checking to generate tests from specifications, *Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, pp. 46–54, 1998.

[2] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea and A. Tiwari, SAL 2, in *Proceedings of the Sixteenth International Conference on Computer Aided Verification (LNCS 3114)*, R. Alur and D. Peled (Eds.), Springer, Berlin-Heidelberg, Germany, pp. 496–500, 2004.

[3] L. de Moura, S. Owre and N. Shankar, The SAL Language Manual, Technical Report SRI-CSL-01-02, SRI International, Menlo Park, California, 2003.

[4] B. Dutertre, Formal Modeling and Analysis of the Modbus Protocol, Technical Report, SRI International, Menlo Park, California, 2006.

[5] A. Gargantini and C. Heitmeyer, Using model checking to generate tests from requirements specifications, in *Proceedings of the Seventh European Software Engineering Conference (LNCS 1687)*, O. Nierstrasz and M. Lemoine (Eds.), Springer, Berlin-Heidelberg, Germany, pp. 146–162, 1999.

[6] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur and Y. Wolfsthal, Coverage-directed test generation using symbolic techniques, in *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (LNCS 1166)*, M. Srivas and A. Camilleri (Eds.), Springer, Berlin-Heidelberg, Germany, pp. 143–158, 1996.

[7] G. Hamon, L. de Moura and J. Rushby, Generating efficient test sets with a model checker, *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, pp. 261–270, 2004.

[8] Modbus IDA, MODBUS Application Protocol Specification v1.1a, North Grafton, Massachusetts (www.modbus.org/specs.php), June 4, 2004.

[9] Modbus IDA, MODBUS Messaging on TCP/IP Implementation Guide v1.0a, North Grafton, Massachusetts (www.modbus.org/specs.php), June 4, 2004.

[10] Modbus.org, MODBUS over Serial Line Specification and Implementation Guide v1.0, North Grafton, Massachusetts (www. modbus.org/specs.php), February 12, 2002.

[11] S. Owre, J. Rushby, N. Shankar and F. von Henke, Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS, *IEEE Transactions on Software Engineering*, vol. 21(2), pp. 107–125, 1995.

[12] S. Rayadurgam and M Heimdahl, Coverage based test-case generation using model checkers, *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, 2001.

[13] V. Rusu, L. du Bousquet and T. Jéron, An approach to symbolic test generation, in *Proceedings of the Second International Conference on Integrated Formal Methods (LNCS 1945)*, W. Grieskamp, T. Santen and B. Stoddart (Eds.), Springer, Berlin-Heidelberg, Germany, pp. 338–357, 2000.