Chapter 14

FORMAL MODELING AND ANALYSIS OF THE MODBUS PROTOCOL*

Bruno Dutertre

Abstract Modbus is a communication protocol widely used in SCADA systems and distributed control applications. We present formal specifications of Modbus developed using PVS, a generic theorem prover, and SAL, a toolset for automatic analysis of state-transition systems. Both formalizations are based on the *Modbus Application Protocol Specification* [9], the application-layer part of the Modbus standard, which specifies the format of Modbus command and response messages. The goal of the formal modeling was the study of automated methods for systematic and extensive testing of Modbus devices.

Keywords: SCADA Protocol, Modbus, Test-case Generation, Formal Methods

1. Introduction

A distributed digital 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 automated 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 and often proprietary networks and protocols, and 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

*This study was supported by the Institute for Information Infrastructure Protection (I3P) under grant 2003-TK-TX-2003 from the Office of Domestic Preparedness of the U.S. Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Office of Domestic Preparedness.

migration is enabled by the increased sophistication of control devices. It provides increased bandwidth and functionality, and economical benefits. Control systems are now using the same technologies and protocols as communication networks, and the separation between control and other networks is disappearing. SCADA systems are now largely connected to conventional enterprise networks, which themselves 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 in addition one would hope that the end devices are reliable and resilient to attacks. Detecting and removing vulnerabilities in control devices is essential to security.

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 rarely exercised under normal conditions. Traditional testing methods, which attempt to check proper functionality under reasonable input, can fail to detect such vulnerabilities. To be effective, security testing requires high coverage. A large set of test cases must be used that covers not just normal conditions but also input that is not likely to be observed in ordinary device use. Flaws in handling of malformed or unexpected input have been the source of many attacks on computer systems, such as buffer-overflow attacks.

A major challenge to such exhaustive testing is the generation of relevant test cases. It is difficult and expensive to generate by hand a large number of test cases that achieve sufficient coverage. An alternative is to generate test cases automatically, using formal method techniques. This relies on constructing test cases mechanically from a specification of the system under test and a set of testing goals called *test purposes*. This idea has been 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 target control devices that support the Modbus Application Protocol [9], 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 to satisfy these two goals. A first model is developed using the PVS specification and verification system. This model captures the Modbus specifications as defined in the Modbus standard [9]: it includes a precise definition of valid Modbus requests and, for each request class, the specification of the 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, one can determine whether the device passed or failed this test by executing the PVS model with input r and m.

In addition, we present another model designed for automated test generation, that is, for the construction of Modbus requests that satisfy a test purpose. This model was developed using the SAL environment for modeling and verifying state transition systems. Using this model, test-case generation translates to a state-reachability problem that can be solved using the model checking tools available in SAL. This approach is more efficient and powerful than attempting to generate test cases directly from the PVS specifications.

The Modbus standard is very flexible, and devices are highly configurable. The standard allows for a Modbus-compliant device to support only a subset of the defined functions, and for each function to support a subset of the possible parameters. Several functions are left open as user definable. To be effective, a testing strategy must then be tailored to the device at hand and specialized to the functions and parameters this device supports. Special care has been taken to address this need for flexibility. The formal models presented can be easily modified and customized to take into account the features and different configurations of Modbus devices.

2. An Overview of Modbus

Modbus is a communication protocol widely used in distributed control applications. Modbus was initially introduced in 1979 by Modicon (a company now owned by Schneider Electric) 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 physical and link-layer protocols for exchanging data [8]. Two main variants of the link-layer protocol are defined and two different types of serial lines are supported. In addition, the specifications define an application-layer protocol, known as the Modbus Application Protocol, for controlling and querying devices [9].

Subsequently, Modbus was extended to support other types of buses or 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. Nowadays, many Modbus systems implement the communication layer using TCP, as described in The Modbus over TCP/IP specification [10].

2.1 Modbus over Serial Line

Figure 1 sketches the typical architecture of Modbus over serial lines. 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. Communica-

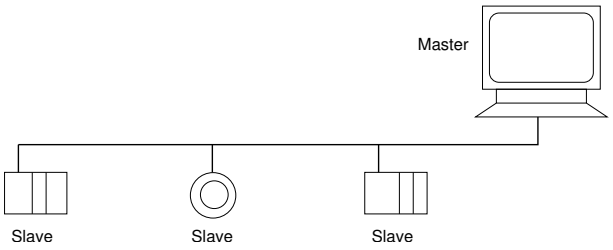

Figure 1. Modbus over Serial Line: Master/Slave Architecture

tion is initiated by the controller (master node), which issues commands on the bus, usually destined for a single device. This device (slave node) may then access the bus and begin transmission in response to the master command. Slave nodes do not directly communicate with each other and do not transmit data without a request from the master node.

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 at the application layer is a simple query-response exchange initiated by the master and addressed to a single device. Requests and responses fit in a single serial-line frame.
- The maximal length of a Modbus frame is 256 bytes. One byte is the device address and two bytes are used for CRC. The maximal length of a query or response is 253 bytes.

2.2 Modbus over TCP/IP

Modbus over TCP/IP uses TCP as the communication layer, but attempts to remain compatible as much as possible with the serial-line protocol. The specification defines an embedding of Modbus packets into TCP frames and assigns a specific IP port number (502) for the Modbus protocol. The frame includes the usual IP and TCP headers, followed by a Modbus-specific header and by the payload. To maintain compatibility with Modbus over serial lines, the payload is limited to at most 253 bytes. Several fields of the Modbus header are also inherited from Modbus over serial lines [10].

Supporting Modbus over TCP has economical 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 probably introduces vulnerabilities and adds considerable complexity. The master-slave

architecture illustrated in Figure 1 can be implemented by relatively simple devices since most of the protocol control and functionality are implemented in the master node. The situation is reversed in the TCP framework: the master node is a *TCP client* and the devices are *TCP servers*. As far as networking is concerned, devices that support Modbus over TCP/IP 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 a simple master-slave model. For example, multiple Modbus transactions can be sent concurrently to the same device and a device may accept connections from different clients. The *Modbus Messaging on TCP/IP Implementation Guide* [10] gives guidelines on these issues.

2.3 The Modbus Application Protocol

The common part of all variants of Modbus is the application protocol. As mentioned earlier, this protocol was initially intended for the Modbus master-slave protocol on serial lines. It is a very simple protocol: Almost all transactions consist of a request sent by a node to a single device, followed by a response from that device. The few exceptions are a small number of transactions made of a single command with no response back.

The majority of the requests are commands to read or write registers in a device. The Modbus standard defines four main classes of registers as follows:

Coils: single-bit, readable and writable

Discrete Input: single-bit, read-only

Holding: 16-bit, readable and writable

Input: 16-bit, read-only

Individual registers in each category are identified by a 16-bit address. There is no guarantee or requirement for a 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.

Figure 2 shows the format of an example Modbus command and the associated responses. The first byte of the request identifies a specific command—in this example, function code 0x02 for *Read Discrete Inputs*. The rest of the request are parameters. The example command has two parameters: a start address between 0x0000 and 0xFFFF (in hexadecimal) and the number of discrete inputs to read stored as a 16-bit integer. The device may either reject such a request and send an error packet or return the requested data in a single packet. The format of a valid response is indicated in Figure 2: 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 were requested, this byte is $\lceil n/8 \rceil$), and the rest of

Request

Function code	1 byte	0x02
Start address	2 bytes	from 0 to 0xFFFF
Quantity	2 bytes	from 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 Modbus Command and Associated Responses

the packet is the data itself. An error packet contains a copy of the request's function code with the high-order bit flipped and an exception code that indicates the reason for the failure. The diagram of Figure 3 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 Figure 2. Other commands in the Modbus Application Protocol are related to device identification and diagnostic. Every command starts with a function code, which is a single byte between 1 and 127. The command then contains 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 with the same first byte as the command, and other data (e.g., response to a read command). Failure is indicated by responding with a two-byte error packet.

The Modbus standard defines the meaning of 19 out of the 127 possible function codes. Other function codes are either unassigned and reserved for future use, or reserved for legacy products and not available, or 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 standard is very 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 it is possible for different devices to assign different interpretations to the same user-defined code.

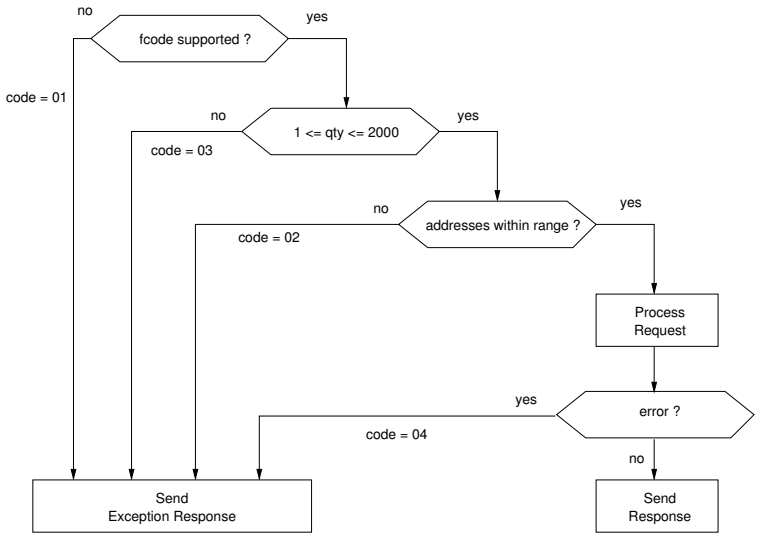

Figure 3. Example Command: Processing and Error Reporting

3. Formal Specification

The formal models we have developed focus on the Modbus Application Protocol. As summarized previously, all 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 nonobvious but critical property. Instead, our modeling and analysis work has focused on formal models that can be used for extensive testing of Modbus devices.

Given such models, we show how to automatically derive test scenarios, that is, specific Modbus requests, and check whether the device answers properly. Because test cases can be generated automatically and the models 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 one to test how a device responds to a variety of malformed requests that it may not be expected to receive in normal operation (e.g., requests too long or too short, containing bad function codes or unsupported addresses). Our goal is for this technology to help detect vulnerabilities in Modbus devices, including buffer overflows and other bugs.

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 can be found on the PVS website: `http://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 standard [9]. 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 can be 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 `m`'s function code is one of the nineteen assigned codes. A `pre_request` is a raw message that satisfies predicate `standard_fcode`.

  Given a prerequest `sr`, `acceptable_length(sr)` holds if the length of `sr` is within the bounds specified by the standard 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 consists of a succession 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` (a raw message) has the appropriate format.

The predicates and types summarized so far are device neutral. They capture the general formatting requirements of Modbus [9]. 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 both 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 main utility of the PVS formalization is as an unambiguous and precise specification of the Modbus Application Protocol. As it is written, the PVS specification is executable, so it can be used as a reference implementation. For example, one can check that the response to the bad requests `[|0|]` and `[|1|]` is 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 that contains the single byte 0, that is, a packet of length 1 with invalid function code 0. The corresponding answer must be the packet `illegal_function(0)`. For packet `[|1|]`, the function code is valid and supported by the device but the format is wrong. The error code in such a case is required to be `illegal_data_value(1)`.

The following examples illustrate 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 value 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: the byte count of 10 is incorrect. In the third line, the response is correctly formatted but it does not match the request: it returns 16 rather than 8 coils.

4. Automated Test-Case Generation

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

The general method employed by most test-case generation tools requires a model of the expected behavior (such as the PVS specifications of Modbus presented previously). The goal is to generate input data for the system under test and check whether the system's behavior in response to this input satisfies the specifications. To guide the search, one can specify additional constraints 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 can be done via PVS's built-in mechanisms for finding counterexamples to postulated properties. This method is explained in detail in [4]. However, test-case generation with PVS is limited because it relies exclusively on random search. The PVS procedure we apply works by randomly generating arrays of bytes of different lengths, and checking these byte arrays one by one, until one is found that satisfies the test purpose. For many test purposes, this naïve random search has a low probability of success.

To solve this, we have built a different model of Modbus that is specifically intended for test-case generation. This model is described in the next section and 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 analy-

```
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 4. Interface of the `modbus` Module in SAL

sis 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] and [3], and at `http://sal.csl.sri.com/`.

The full SAL model is given in [4]. The model is intended to support automated test-case generation by constructing Modbus requests that satisfy given constraints (the 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 requests.

The SAL model relies on a simple observation: the set of well-formatted Modbus requests is a regular language. It can then be defined by a finite-state automaton. The SAL model is essentially such an automaton written in the SAL notation. This automaton is defined as module `modbus` whose state variables are shown in Figure 4. In SAL, *module* is a synonym for state-transition system. The `modbus` module has a single input variable `b`, which is a byte. Its internal state consists of the ten local variables given in Figure 4. All these variables have a finite type, so the full module is a finite state machine.

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, then 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, we have `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

```
[] 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 5. SAL Fragment: Reading the first word of a packet

so forth. In addition to computing the status variable, the SAL module extracts and stores important attributes of the input packet such as the function code or the packet length.

Figure 5 shows a fragment of the SAL specifications that extracts the first word of a packet, that is, the 16-bit number that follows the function code. The specification uses guarded commands, written `cond --> assignment`. The *condition* refers to variables of current state and the *assignment* defines the value of 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 `unknown` (to indicate that more input must be read and more checking must be performed).

Just like the PVS model discussed previously, our 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 formatting of Modbus requests, we have turned test-case generation into a state-reachability problem. Given an input sequence of n bytes $b_1, \ldots, b_n$, the SAL `modbus` machine will perform n state transitions and reach a state s_n. We can determine whether $b_1, \ldots, b_n$ 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 $b_1, \ldots, b_n$ 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 can then search for a sequence $b_1, \ldots, b_n$ that satisfies the second condition.

This problem can be solved using the SAL bounded model checkers. In general, a bounded model checker searches for counterexamples of a fixed length n to a property P. In SAL, given a state machine M, such a counterexample is a finite sequence of n state transitions

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

such that s_0 is an initial state of M and one of the states s_i violates P. To use bounded model checking for test-case generation we just need to negate the property. 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`, we have either `pc` $\neq$ `done` or `stat` $\neq$ `invalid_data`. This property is not true, and a counterexample is exactly what we need: 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 then be of length 20 or less. The counterexample produced is the sequence of bytes $[4, 128, 0, 254, 64]$ and the state reached after that sequence is as follows

```
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 and the other variables give more information about the packet generated: its length is 5 bytes and the function code is 4 (command *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 as the maximum number of registers allowed in such commands 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, one can 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. Examples are given in [4] for a variety of scenarios.

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 into 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 that was 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 will find 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 for generating test cases is typically 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 then more expensive to construct than shorter ones. Still, the runtime remains 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 s by solving a SAT problem with more than 500,000 boolean variables and 2,000,000 clauses. Overall, it is then possible to generate a large set of test cases for many scenarios at little cost. This enables extensive testing of the compliance of a device with the 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 in a few minutes of runtime.

5. Conclusion

We have presented a framework to support systematic, extensive testing of control devices that implement the Modbus Application Protocol. Our aim is to increase the robustness and security of these devices by detecting potential vulnerabilities that conventional testing methods may easily miss. The framework relies on two main components. A formal specification of the protocol written in PVS serves as a reference to check whether the observed responses to a test request satisfy the standard. A state-machine model of Modbus requests serves as an automated test-case generator. Both models are designed to accommodate the high variability in supported functions and parameters that the Modbus standard allows. The models are parameterized and can be rapidly specialized to the features of a device under test.

In future work, we plan to adapt the formal models presented here for online monitoring. By monitoring online the requests and responses from a control system, we hope to achieve accurate and high-coverage intrusion detection. Our next goal is the automatic derivation of intrusion-detection sensors from

precise formal models of the expected function and behavior of Modbus devices in a test-case environment.

References

[1] P. E. Ammann, P. E. Black, and W. Majurski, Using model checking to generate tests from specifications, *Second International Conference on Formal Engineering Methods (ICFEM '98)*, pp. 46–54, 1998.

[2] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, SAL 2, *Computer-Aided Verification (CAV'04)*, LNCS 3114, 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, 2003.

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

[5] A. Gargantini and C. Heitmeyer, Using model checking to generate tests from requirements specifications, *Software Engineering—ESEC/FSE '99*, LNCS 1687, 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, *First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, LNCS 1166, pp 143–158, 1996.

[7] G. Hamon, L. de Moura, and J. Rushby, Generating efficient test sets with a model checker, *2nd International Conference on Software Engineering and Formal Methods*, pp. 261–270, 2004.

[8] Modbus-IDA, *Modbus over Serial Line: Specification and Implementation Guide V1.0*, 2002.

[9] Modbus-IDA, *Modbus Application Protocol Specification V1.1a*, 2004.

[10] Modbus-IDA, *Modbus Messaging on TCP/IP: Implementation Guide V1.0a*, 2004.

[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*, 21(2):107–125, 1995.

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

[13] V. Rusu, L. du Bousquet, and T. Jéron, An approach to symbolic test generation, *2nd International Workshop on Integrated Formal Method (IFM'00)*, LNCS 1945, pp. 338–357, 2000.