SRI International

October 11, 2006

Formal Modeling and Analysis of the Modbus Protocol

Bruno Dutertre

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.

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Abstract

Modbus is a communication protocol widely used in SCADA systems and distributed control applications. This report describes formal models of the Modbus protocol. Two formalizations of Modbus are presented: the first was developed using the PVS generic theorem prover and the second was developed using SAL, an environment for automatic analysis of state-transition systems using model checking tools. Both formalizations are based on the *Modbus Application Protocol Specification* [9], the application-layer part of the Modbus standard. This application-layer protocol specifies the format of command and response messages that a distributed control application can use to communicate with sensor or actuator devices. The goal of the formal modeling effort was to study automated methods for systematic and extensive testing of Modbus devices.

Contents

List of Figures

Chapter 1

Introduction

Distributed digital control systems —sometimes called SCADA systems— have been used for many years in manufacturing, transport, power distribution and generation, and other industries. A distributed control system consists of a network of devices and computers for monitoring, sensing, 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 common networking standards such as TCP/IP and Ethernet. The reasons for this migration include increased sophistication of the control devices, increased bandwidth and reliability of traditional networking technology, and many 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 typically linked to wider networks such as the Internet.

This interconnection increases the risks 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 potential 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 attempts 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 in supporting 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 formal 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 report 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 serves as a reference and algorithms for automatically generating test cases from such models. In this report, we develop two formal models to satisfy these two goals. First, we present a formal specification of the Modbus Application Protocol developed using the PVS specification and verification system. This model captures the Modbus specifications as defined in the Modbus standard [9]. The PVS model includes a precise definition of valid Modbus requests and, for each request class, the specification of the acceptable responses. This 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 formal 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 the SAL environment. 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. It 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. In addition, 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 in this report can be easily modified and customized to take into account the features and different configurations of Modbus devices.

The next chapter gives an overview of the main features of Modbus. The details of the PVS specifications are presented in Chapter 3. The SAL model and test-case generation method are discussed in Chapter 4. Complete specifications are given in Appendix A and B.

Chapter 2

An Overview of Modbus

Modbus is a communication protocol widely used in distributed control applications (SCADA systems), especially in the oil and gas sector. 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]. The application protocol was originally intended for devices connected via the Modbus serial-line protocol.

Subsequently, the Modbus specifications were 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. Increasingly, TCP/IP is being used as the communication layer for Modbus. The Modbus over TCP/IP specification [10] describes how to implement the Modbus communication layer using TCP.

2.1 Modbus over Serial Line

Figure 2.1 sketches the typical architecture of systems that use 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. Communication 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 —namely, the types of serial lines that are supported— 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:

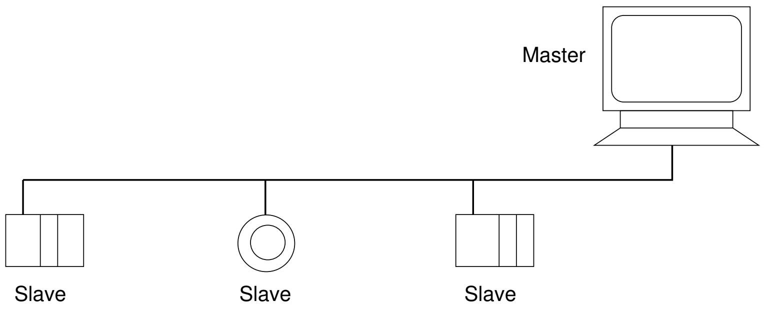

Figure 2.1: Modbus over Serial Line: Master/Slave Architecture

- 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 node and addressed to a single device. Both requests and responses fit in a single serial-line frame.
- Each device on the same serial line has an 8 bit address. Addresses 0 and 248 to 255 are reserved. There can be at most 247 devices on a single line.
- 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 a communication protocol rather than a dedicated serial-line protocol. Nonetheless, Modbus over TCP/IP attempts to remain compatible as much as possible with the serial-line protocol. The specifications defines an embedding of Modbus packets into TCP frames and assigns a specific IP port number (502) for the Modbus protocol. Figure 2.2 sketches how Modbus packets are encapsulated into TCP frames. 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: the header contains a unit identifier byte, which plays the same role as the device address in the serial-line specification. The unit identifier allows Modbus gateways to connect serial-line Modbus devices with TCP/IP networks. The other header fields are a 2-byte transaction identifier, a 2-byte protocol identifier that always must be 0, and the length of the payload (plus one) [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 2.1 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*

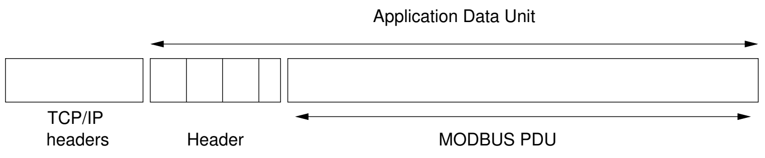

Figure 2.2: Modbus over TCP/IP

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 designed to be supported by the Modbus master-slave protocol for serial lines. The Modbus Application Protocol is very simple. Almost all transactions consist of a request sent by a node to a single device, followed by a response from that device. Both requests and responses fit in a single serial-line frame of at most 253 bytes. 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 registers that are both readable and writable

Discrete Input: single-bit, read-only register

Holding Registers: 16-bit registers, readable and writable

Input Registers: 16-bit, read-only registers.

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 2.3 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. On receiving such a request, 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 indicated in Figure 2.3: 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 then this byte is $\lceil n/8 \rceil$), and

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 2.3: Example Modbus Command and Associated Responses

the rest of 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 2.4 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 2.3. 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 from 1 to 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 user-defined codes are in the ranges 65 to 72 and 100 to 110, and can be used freely to support functionalities outside the standard. 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. It is possible for different devices to assign different interpretations to the same user-defined function code.

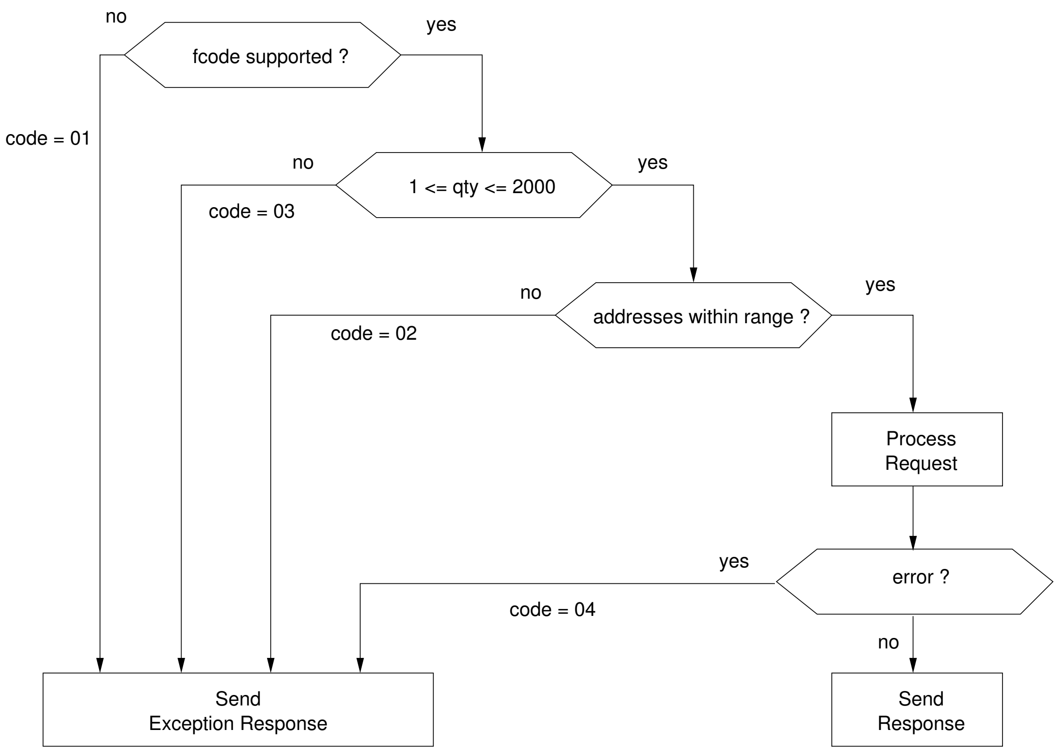

Figure 2.4: Example Command: Processing and Error Reporting

Chapter 3

Formal Specification

As mentioned previously, Modbus can be decomposed into an abstract communication layer and the Modbus Application Protocol. The communication layer is either provided by the Modbus serial-line master-slave protocol or implemented on TCP/IP. The most interesting target for this I3P project is Modbus over TCP/IP, since the serial-line version of Modbus is mostly used in older systems. Modern SCADA systems in the oil and gas industries employ Modbus over TCP/IP in majority.

Since TCP is not a SCADA-specific protocol, the formal models we have developed focus on the Modbus Application Protocol. As summarized in Section 2.3, all transactions in this protocol are very simple and consist of a single request followed by a single response. Both requests and responses are small packets of at most 253 bytes in length. All the complexities in Modbus over TCP/IP reside in TCP/IP rather than in the application protocol itself. Seen in isolation, 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 formal modeling and analysis work has focused on developing precise models of the protocol for enhancing the security of distributed control systems that employ Modbus.

For this purpose, we have developed precise formal models that can be used to support 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 devices 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/`. Documentation, example applications, and the system itself can be downloaded from this website.

Our full PVS specification of the Modbus Application Protocol is given in Appendix A. The specification is a straightforward formalization of the standard as defined in [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.

3.1.1 Main Model Elements

The details of the PVS specifications can be found in Appendix A. The definition of well-formed requests can be summarized as follows:

- The type `raw_msg` represents arbitrary packets (raw messages). Packets are modeled in PVS as arrays of bytes of length from 1 to 253.
- Function `fcode(m)` returns the function code of an a raw message `m`, that is, the first bye of `m`.
- The following PVS predicates are defined for function codes
  - `assigned_fcode(f)` is `true` if `f` is one of the nineteen public function codes. This set of codes is divided into function codes that exist for serial-line devices only and function codes that all devices may support.
  - `reserved_fcode(f)` is `true` if `f` is one of the codes used by legacy devices and that should not be used by others.
  - `user_defined_fcode(f)` is `true` if `f` is one of the user-defined function codes.
  - `exception_fcode(f)` is `true` if the high-order bit of `f` is 1.
  - `invalid_fcode(f)` is `true` if `f` is equal to 0.

  These predicates divide the set of function codes in disjoint categories as specified in the standard.[1]
- Correct formatting of requests is then 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 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.

[1]We chose to introduce a special category for function code $f = 0$, which is not discussed at all in the standard.

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

The PVS definition of acceptable responses to a given request follows the same general scheme. Given a valid request `r`, predicate `acceptable_response(v, r)` is `true` whenever `r` is a possible response to 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 given in the Modbus standard [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.

3.1.2 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,[2] 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:

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

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

<PVSio> 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 answer to this packet is 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:

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

[2]This is not true in general for PVS specifications, but it is true here.

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

<PVSio> 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.

Chapter 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. However, 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.

4.1 Test-Case Generation Using PVS

To some extent, PVS and the formal specifications presented previously can be used for test-case generation. Let us assume one wishes to generate input for a Modbus device, that is, request packets, with the purpose of examining responses of the device to illegal function codes. To generate corresponding test cases with PVS, we want to generate a request packet `m` that satisfies the property

```
EXISTS b: modbus_response(m, illegal_function(b))
```

This formalizes the property "the response to `m` is an error packet with function code `b` and exception code 1" (1 is the exception code defined by the standard to report illegal functions).

PVS is a generic theorem prover normally used to formally prove properties, but it is quite common in practice that the properties one tries to prove are actually not true. To help the user in detecting that a property is false, PVS includes a randomized procedure that attempts to find counterexamples. We can use this mechanism for automatically generating test cases. For example, to find a request `m` that satisfies the previous property, we ask PVS to find a counterexample to the following theorem:

```
random_test1: LEMMA
   FORALL m, b: NOT modbus_resp(m, illegal_function(b))
```

This lemma is actually false. It is the negation of the previous property. A counterexample to `random_test1` is a packet `m` (together with a byte `b`) for which the lemma is not true. Such a counterexample satisfies the property

```
modbus_resp(m, illegal_function(b))
```

and is what we are looking for. PVS's random search is invoked on the lemma as follows:

```
random_test1 :

  |-------
 {1}   FORALL m, b: NOT modbus_resp(m, illegal_function(b))

 Rule? (random-test :count 100 :size 20)
```

This results in the following counterexample:

```
The formula is falsified with the substitutions:
   m ==> (# len := 13,
            b := LAMBDA (x: below(13)):
                  IF x = 0 THEN 18
                  ELSIF x = 1 THEN 19
                  ELSIF x = 2 THEN 4
                  ELSIF x = 3 THEN 9
                  ELSIF x = 4 THEN 0
                  ELSIF x = 5 THEN 4
                  ELSIF x = 6 THEN 19
                  ELSIF x = 7 THEN 15
                  ELSIF x = 8 THEN 11
                  ELSIF x = 9 THEN 9
                  ELSIF x = 10 THEN 9
                  ELSIF x = 11 THEN 3
                  ELSE 4
                  ENDIF #)
   b ==> 18
```

The generated counterexample is printed in PVS's syntax. It represents a packet of length 13 whose first byte is equal to 18. One can check that this packet is a valid test case for the purpose given previously. The packet is an incorrect Modbus request with an illegal function code as required.

Using this approach, one can generate a variety of Modbus requests and use these as test cases. However, the PVS-based test-case generation method 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. For example, the probability that a random array of bytes is a well-formed Modbus request is very low. So any test purpose that attempts to check the response of a device to a valid request is very hard to satisfy using random search.

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.

```
modbus: MODULE =
  BEGIN
    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 4.1: Interface of the `modbus` Module in SAL

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.

4.2 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. Complete descriptions of these tools and of the SAL specification language can be found in [4], [2], and [3]. SAL is available at `http://sal.csl.sri.com/`.

The full SAL model we developed for Modbus is given in Appendix B. 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 given in Appendix B is essentially such an automaton written in the SAL notation. This automaton is defined as module `modbus` whose interface is specified in Figure 4.1. In SAL, module is 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 4.1. 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

```
%% read first word
%% DIAGNOSTIC and READ_FIFO_QUEUE are treated specially
%% all other requests cannot be checked yet: second word is needed
[] 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 to read 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 4.2: SAL Fragment: Reading the first word of a packet

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 was 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 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 4.2 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 of the form `condition --> 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 the `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 either set 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.

4.3 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

$$b_1, \dots, b_n,$$

the SAL `modbus` machine will perform n state transitions and reach a state s_n. We can 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 $b_1, \dots, b_n$ is an invalid request
- if `pc` $\neq$ `done` then the status of $b_1, \dots, b_n$ is not known yet. This means that $b_1, \dots, b_n$ 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 $b_1, \dots, b_n$ that satisfies the second condition.

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

$$s_0 \rightarrow s_1 \dots \rightarrow s_n$$

such that s_0 is an initial state of M and one of the states s_i violates P. To use bounded model checking as a test-case generation tool 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 $\texttt{pc} = \texttt{done} \Rightarrow \texttt{stat} \neq \texttt{invalid\_data}$ is an invariant of module modbus. In other words, it postulates that in all reachable states of modbus, we have either $\texttt{pc} \neq \texttt{done}$ or $\texttt{stat} \neq \texttt{invalid\_data}$. This property is not true, and a counterexample is exactly what we need: a sequence of bytes that reaches a state where $\texttt{pc} = \texttt{done}$ and $\texttt{stat} = \texttt{invalid\_data}$.

Counterexamples to this property can be obtained by using SAL's bounded model checkers. Since the modbus module is finite, both the finite-state bounded model checker (sal-bmc) and the bounded model checker for infinite systems (sal-inf-bmc) can be used. For example, sal-bmc can be invoked as follows:

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

This searches for a counterexample to lemma test18 defined in input file flat_modbus.sal. This file contains the full Modbus specification shown in Appendix B. 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 by sal-bmc is the following sequence of bytes $[4, 128, 0, 254, 64]$. The state reached after reading that sequence is displayed as follows

```
 --- System Variables (assignments) ---
 aux = 254
 stat = invalid_data
 pc = done
 len = 5
 fcode = 4
 byte_count = 0
 first_word = 32768
 second_word = 65088
 third_word = 0
 fourth_word = 0
```

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 constraint. 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 Appendix B that correspond to a variety of error 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 or less when running `sal-bmc` on a 3GHz 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. As a consequence, longer test cases are 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.

Chapter 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-cases 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.

Bibliography

[1] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In John Staples, Michael G. Hinchey, and Shaoying Liu, editors, *Second International Conference on Formal Engineering Methods (ICFEM '98)*, pages 46–54, Brisbane, Australia, December 1998. IEEE Computer Society.

[2] Leonardo de Moura. SALenv: Tutorial. Technical report, Computer Science Laboratory, SRI International, 2003. Available at `http://sal.csl.sri.com/documentation.html`.

[3] Leonardo de Moura, Sam Owre, Harald Ruess, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In *Computer-Aided Verification (CAV'04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer-Verlag, July 2004.

[4] Leonardo de Moura, Sam Owre, and Natarajan Shankar. The SAL Language Manual. Technical Report SRI-CSL-01-02, Computer Science Laboratory, SRI International, 2003. Available at `http://sal.csl.sri.com/documentation.html`.

[5] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Software Engineering—ESEC/FSE '99: Seventh European Software Engineering Conference and Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162, Toulouse, France, September 1999. Springer-Verlag.

[6] Daniel Geist, Monica Farkas, Avner Landver, Yossi Lichtensteitn, Shmuel Ur, and Yaron Wolfsthal. Coverage-directed test generation using symbolic techniques. In *First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 1996.

[7] Grégoire Hamon, Leonardo deMoura, and John Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, September 2004. IEEE Computer Society.

[8] Modbus-IDA. *Modbus over Serial Line: Specification and Implementation Guide V1.0*, December 2002. Available at `http://www.modbus.org`.

[9] Modbus-IDA. *Modbus Application Protocol Specification V1.1a*, June 2004. Available at `http://www.modbus.org`.

[10] Modbus-IDA. *Modbus Messaging on TCP/IP: Implementation Guide V1.0a*, June 2004. Available at `http://www.modbus.org`.

[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, February 1995.

[12] Sanjai Rayadurgam and Mats Heimdahl. Coverage based test-case generation using model checkers. In *Eighth International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 83–91, Washington DC, April 2001. IEEE Computer Society.

[13] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *2nd International Workshop on Integrated Formal Method (IFM'00)*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357, Dagstuhl, Germany, November 2000. Springer-Verlag.

Appendix A

PVS Specifications

```
%
% PVS Specification of the Modbus Application Protocol
% extracted from: Modbus Application Protocols Specification V1.1a
%
% Modbus functions not covered by this spec.
% - read_file_record
% - write_file_record
% - modbus encapsulated interface subfunctions
% All other functions are specified
%

modbus : THEORY

  BEGIN

  %---------------
  %  Basic types
  %---------------

  byte: TYPE = subrange(0,255)

  word: TYPE = subrange(0, 65535)

  posword: TYPE = { w: word | w > 0 }


  % Conversions
  bytes2word(hi, low: byte): word = 256 * hi + low

  hi_byte(w: word): byte = ndiv(w, 256)

  lo_byte(w: word): byte = rem(256)(w)



  %------------------------------------------------
  %  Raw message = an array of at most 253 bytes
  %------------------------------------------------
```

```
raw_msg_length: TYPE = subrange(1,253)

raw_msg: TYPE = [# len: raw_msg_length,
                   b: [below(len) -> byte] #]


m, m1, m2: VAR raw_msg


%------------------------------------------------------------
%  Equality predicate.
%  This is necessary for using PVSIO and the ground evaluator
%  This is semantically the same as ordinary equality but
%  can be compiled properly by the ground evaluator.
%  Right now, (m1 = m2) does not compile property.
%------------------------------------------------------------

equals(m1, m2): bool =
   m1‘len = m2‘len AND (FORALL (i: below(m1‘len)): m1‘b(i) = m2‘b(i));

%% check that equals is the same as ordinary equality
raw_message_equals: LEMMA equals(m1, m2) IFF m1 = m2


%-----------------------
% Message constructors
%-----------------------

x, y, z: VAR byte;

% One-byte message
[||](x): raw_msg =
    (# len := 1, b := lambda (i: below(1)): x #)

% Two-byte message
[||](x, y): raw_msg =
    (# len := 2, b := lambda (i: below(2)): if i=0 then x else y endif #)





%---------------------------------
%  Function and subfunction codes
%---------------------------------

READ_COILS: byte = 1

READ_DISCRETE_INPUTS: byte = 2

READ_HOLDING_REGISTERS: byte = 3

READ_INPUT_REGISTERS: byte = 4

WRITE_SINGLE_COIL: byte = 5
```

```
WRITE_SINGLE_REGISTER: byte = 6

READ_EXCEPTION_STATUS: byte = 7   % serial line only

DIAGNOSTIC: byte = 8            % serial line only

GET_COMM_EVENT_COUNTER: byte = 11  % serial line only

GET_COMM_EVENT_LOG: byte = 12      % serial line only

WRITE_MULTIPLE_COILS: byte = 15

WRITE_MULTIPLE_REGISTERS: byte = 16

REPORT_SLAVE_ID: byte = 17        % serial line only

READ_FILE_RECORD: byte = 20

WRITE_FILE_RECORD: byte = 21

MASK_WRITE_REGISTER: byte = 22

READ_WRITE_MULTIPLE_REGISTERS: byte = 23

READ_FIFO_QUEUE: byte = 24

ENCAPSULATED_INTERFACE_TRANSPORT: byte = 43


%
% Subfunction codes for diagnostic
% (serial line only)
% These are 16 bit long
%
RETURN_QUERY_DATA: word = 0

RESTART_COMMUNICATION_OPTION: word = 1

RETURN_DIAGNOSTIC_REGISTER: word = 2

CHANGE_ASCII_INPUT_DELIMITER: word = 3

FORCE_LISTEN_ONLY_MODE: word = 4

CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER: word = 10

RETURN_BUS_MESSAGE_COUNT: word = 11

RETURN_BUS_COMMUNICATION_ERROR_COUNT: word = 12

RETURN_BUS_EXCEPTION_ERROR_COUNT: word = 13

RETURN_SLAVE_MESSAGE_COUNT: word = 14

RETURN_SLAVE_NO_RESPONSE_COUNT: word = 15

RETURN_SLAVE_NAK_COUNT: word = 16
```

```
 RETURN_SLAVE_BUSY_COUNT: word = 17

 RETURN_BUS_CHARACTER_OVERRUN_COUNT: word = 18

 CLEAR_OVERRUN_COUNTER_AND_FLAG: word = 20


 %
 % Modbus Encapsulated Interface types
 % (i.e., subfunction  codes for EncapsulatedInterfaceTransport)
 %
 MEI_CANopen: byte = 13

 MEI_ReadDeviceIdentification: byte = 14


 %-------------------
 %  Exception codes
 %-------------------

 ILLEGAL_FUNCTION: byte = 1

 ILLEGAL_DATA_ADDRESS: byte = 2

 ILLEGAL_DATA_VALUE: byte = 3

 SLAVE_DEVICE_FAILURE: byte = 4

 ACKNOWLEDGE: byte = 5

 SLAVE_DEVICE_BUSY: byte = 6

 MEMORY_PARITY_ERROR: byte = 8

 GATEWAY_PATH_UNAVAILABLE: byte = 10

 GATEWAY_TARGET_DEVICE_FAILED_TO_RESPOND: byte = 11



 %--------------------------
 %  Function code checking
 %--------------------------

 % 0 is the only invalid fcode
 invalid_fcode(f: byte): bool = f = 0


% reserved codes: used by legacy devices but not to be used by others
 reserved_fcode(f: byte): bool =
   f=9 OR f=10 OR f=13 OR f=14 OR f=41 OR f=42
       OR f=90 OR f=91 OR f=125 OR f=126 OR f=127

 % user-defined codes: available to the user/unspecified behavior
 user_defined_fcode(f: byte): bool =
   (65 <= f AND f <= 72) OR (100 <= f AND f <= 110)

 % function codes only for serial-line devices
 assigned_serial_line_fcode(f: byte): bool =
    f = READ_EXCEPTION_STATUS OR
    f = DIAGNOSTIC OR
    f = GET_COMM_EVENT_COUNTER OR
    f = GET_COMM_EVENT_LOG OR
    f = REPORT_SLAVE_ID

 % function codes actually defined in the spec and
 % applicable to serial-line and non-serial-line devices
 assigned_general_fcode(f: byte): bool =
    f = READ_COILS OR
    f = READ_DISCRETE_INPUTS OR
    f = READ_HOLDING_REGISTERS OR
    f = READ_INPUT_REGISTERS OR
    f = WRITE_SINGLE_COIL OR
    f = WRITE_SINGLE_REGISTER OR
    f = WRITE_MULTIPLE_COILS OR
    f = WRITE_MULTIPLE_REGISTERS OR
    f = READ_FILE_RECORD OR
    f = WRITE_FILE_RECORD OR
    f = MASK_WRITE_REGISTER OR
    f = READ_WRITE_MULTIPLE_REGISTERS OR
    f = READ_FIFO_QUEUE OR
    f = ENCAPSULATED_INTERFACE_TRANSPORT


 % assigned codes: union of the 2 previous
 assigned_fcode(f: byte): bool =
    assigned_general_fcode(f) OR assigned_serial_line_fcode(f)


 % unassigned public codes: not to be used
 unassigned_fcode(f: byte): bool =
    1 <= f AND f <= 127 AND
       NOT (user_defined_fcode(f) OR assigned_fcode(f) OR reserved_fcode(f))


 % exception code: any code with high-order bit=1
 exception_fcode(f: byte): bool = f >= 128




 %
 % Consistency checks:
 % - predicates cover all the cases
 % - categories are disjoint
 %

 f: VAR byte

 check_complete1: LEMMA
      invalid_fcode(f)
   OR reserved_fcode(f)
   OR user_defined_fcode(f)
   OR assigned_serial_line_fcode(f)
   OR assigned_general_fcode(f)
   OR unassigned_fcode(f)
   OR exception_fcode(f)


 check_disjoint1: LEMMA
       NOT (invalid_fcode(f) AND reserved_fcode(f))
   AND NOT (invalid_fcode(f) AND user_defined_fcode(f))
   AND NOT (invalid_fcode(f) AND assigned_serial_line_fcode(f))
   AND NOT (invalid_fcode(f) AND assigned_general_fcode(f))
   AND NOT (invalid_fcode(f) AND unassigned_fcode(f))
   AND NOT (invalid_fcode(f) AND exception_fcode(f))

   AND NOT (reserved_fcode(f) AND user_defined_fcode(f))
   AND NOT (reserved_fcode(f) AND assigned_serial_line_fcode(f))
   AND NOT (reserved_fcode(f) AND assigned_general_fcode(f))
   AND NOT (reserved_fcode(f) AND unassigned_fcode(f))
   AND NOT (reserved_fcode(f) AND exception_fcode(f))

   AND NOT (user_defined_fcode(f) AND assigned_serial_line_fcode(f))
   AND NOT (user_defined_fcode(f) AND assigned_general_fcode(f))
   AND NOT (user_defined_fcode(f) AND unassigned_fcode(f))
   AND NOT (user_defined_fcode(f) AND exception_fcode(f))

   AND NOT (assigned_serial_line_fcode(f) AND assigned_general_fcode(f))
   AND NOT (assigned_serial_line_fcode(f) AND unassigned_fcode(f))
   AND NOT (assigned_serial_line_fcode(f) AND exception_fcode(f))

   AND NOT (assigned_general_fcode(f) AND unassigned_fcode(f))
   AND NOT (assigned_general_fcode(f) AND exception_fcode(f))

   AND NOT (unassigned_fcode(f) AND exception_fcode(f))



 %--------------------------------------------
 %  Subcode checking for diagnostic function
 %--------------------------------------------

 valid_diagnostic_subcode(f: word): bool =
   f = RETURN_QUERY_DATA OR
   f = RESTART_COMMUNICATION_OPTION OR
   f = RETURN_DIAGNOSTIC_REGISTER OR
   f = CHANGE_ASCII_INPUT_DELIMITER OR
   f = FORCE_LISTEN_ONLY_MODE OR
   f = CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER OR
   f = RETURN_BUS_MESSAGE_COUNT OR
   f = RETURN_BUS_COMMUNICATION_ERROR_COUNT OR
   f = RETURN_BUS_EXCEPTION_ERROR_COUNT OR
   f = RETURN_SLAVE_MESSAGE_COUNT OR
   f = RETURN_SLAVE_NO_RESPONSE_COUNT OR
   f = RETURN_SLAVE_NAK_COUNT OR
   f = RETURN_SLAVE_BUSY_COUNT OR
   f = RETURN_BUS_CHARACTER_OVERRUN_COUNT OR
   f = CLEAR_OVERRUN_COUNTER_AND_FLAG
```

```
reserved_diagnostic_subcode(f: word): bool =
   NOT valid_diagnostic_subcode(f)



%-----------------------
%  Checks on MEI types
%-----------------------

valid_mei_type(f: byte): bool =
   f = MEI_CANopen OR f = MEI_ReadDeviceIdentification

reserved_mei_type(f: byte): bool =
   NOT valid_mei_type(f)






%-----------------------------------------------
%  Function code = first byte of a raw message
%-----------------------------------------------

fcode(m): byte = m‘b(0)


%
% Requests to reject based on fcode
% - these requests should be ignored by any device
%
invalid_fcode(m): bool =
  invalid_fcode(fcode(m)) OR
  unassigned_fcode(fcode(m)) OR
  exception_fcode(fcode(m))

%
% Reserved fcode: should also be ignored, but
% may be used in legacy devices.
%
reserved_fcode(m): bool = reserved_fcode(fcode(m))

%
% User-defined fcode: undocumented but available for use.
% May have different usage in different devices.
%
user_defined_fcode(m): bool = user_defined_fcode(fcode(m))


%
% Serial-line only fcode: should be ignored by devices
% that use Modbus/TCP
%
serial_line_only_fcode(m): bool = assigned_serial_line_fcode(fcode(m))

%
```

```
% Generic fcode: behavior is specified by the standard for
% any device
%
general_fcode(m): bool = assigned_general_fcode(fcode(m))



%----------------------------------------------------------
% requests we care about = those defined in the standard
%----------------------------------------------------------

standard_fcode(m): bool = general_fcode(m) OR serial_line_only_fcode(m)

pre_request : TYPE = { m | standard_fcode(m) }

sr: VAR pre_request



%-------------------------------------------------
%  Length constraints for the standard requests
%  - this just gives minimal and maximal length
%  - further checking is required for variable-
%  length requests.
%-------------------------------------------------

acceptable_length(sr): bool =
    (fcode(sr) = READ_COILS AND sr`len = 5)
 OR (fcode(sr) = READ_DISCRETE_INPUTS AND sr`len = 5)
 OR (fcode(sr) = READ_HOLDING_REGISTERS AND sr`len = 5)
 OR (fcode(sr) = READ_INPUT_REGISTERS AND sr`len = 5)
 OR (fcode(sr) = WRITE_SINGLE_COIL AND sr`len = 5)
 OR (fcode(sr) = WRITE_SINGLE_REGISTER AND sr`len = 5)
 OR (fcode(sr) = READ_EXCEPTION_STATUS AND sr`len = 1)
 OR (fcode(sr) = DIAGNOSTIC AND sr`len >= 5)
 OR (fcode(sr) = GET_COMM_EVENT_COUNTER AND sr`len = 1)
 OR (fcode(sr) = GET_COMM_EVENT_LOG AND sr`len = 1)
 OR (fcode(sr) = WRITE_MULTIPLE_COILS AND 6 <= sr`len AND sr`len <= 253)
 OR (fcode(sr) = WRITE_MULTIPLE_REGISTERS AND 6 <= sr`len AND sr`len <= 253)
 OR (fcode(sr) = REPORT_SLAVE_ID AND sr`len = 1)
 OR (fcode(sr) = READ_FILE_RECORD AND 2 <= sr`len AND sr`len <= 245)
 OR (fcode(sr) = WRITE_FILE_RECORD AND 2 <= sr`len AND sr`len <= 245)
 OR (fcode(sr) = MASK_WRITE_REGISTER AND sr`len = 7)
 OR (fcode(sr) = READ_WRITE_MULTIPLE_REGISTERS AND
                      10 <= sr`len AND sr`len <= 253)
 OR (fcode(sr) = READ_FIFO_QUEUE AND sr`len = 3)
 OR (fcode(sr) = ENCAPSULATED_INTERFACE_TRANSPORT AND
                       2 <= sr`len AND sr`len <= 253)



% request: standard requests that satisfy the above length constraints

request: TYPE = { sr | acceptable_length(sr) }

req: VAR  request
```

```
%--------------------------------
% Common subfields of a request
%--------------------------------

first_word(m | m‘len >= 3): word = bytes2word(m‘b(1), m‘b(2))

second_word(m | m‘len >= 5): word = bytes2word(m‘b(3), m‘b(4))

third_word(m | m‘len >= 7): word = bytes2word(m‘b(5), m‘b(6))

fourth_word(m | m‘len >= 9): word = bytes2word(m‘b(7), m‘b(8))



%-------------------------------------------------
% Check subcodes for diagnostic and MEI requests
%-------------------------------------------------

standard_subcode(req): bool =
    (fcode(req) = DIAGNOSTIC AND valid_diagnostic_subcode(first_word(req)))
 OR (fcode(req) = ENCAPSULATED_INTERFACE_TRANSPORT AND valid_mei_type(req‘b(1)))




%------------------------------------------
% Check data contents in a request
% including additional length constraints
%------------------------------------------

% diagnostic requests:
% - length must be 5 unless the subcode is 0 (RETURN_QUERY_DATA)
%   the data field in RETURN_QUERY_DATA must have 2 * N bytes
%   for some N>=1 (so req‘len must be odd in that case)
% - data field must be 0 for all subcodes except
%     RESTART_COMMUNICATION_OPTION,
%     CHANGE_ASCII_INPUT_DELIMITER,
%     RETURN_QUERY_DATA
% - allowed data field for RESTART_COMMUNICATION_OPTION:
%     0x0000 or 0xFF00 (65280 decimal)
% - allowed data filed for CHANGE_ASCII_INPUT_DELIMITER:
%     one character, 00 in low order byte
%
valid_data_diagnostic(req | fcode(req) = DIAGNOSTIC): bool =
    (first_word(req) = RETURN_QUERY_DATA => odd?(req‘len))
AND (first_word(req) /= RETURN_QUERY_DATA => req‘len = 5)
AND (first_word(req) = RESTART_COMMUNICATION_OPTION =>
        (second_word(req) = 0 OR second_word(req) = 65280))
AND (first_word(req) = CHANGE_ASCII_INPUT_DELIMITER => req‘b(4) = 0)
AND (first_word(req) /= RETURN_QUERY_DATA
      AND first_word(req) /= RESTART_COMMUNICATION_OPTION
        AND first_word(req) /= CHANGE_ASCII_INPUT_DELIMITER
```

```
           =>  second_word(req) = 0)


% write multiple coils:
% - length must be 6 + N where N = byte count = 6th byte in the request
% - N must be equal to ceil(M/8) where M = second word of the request
% - M must be between 1 and 0x7b0 = 1968 (not clear why it’s not 1976)??
%
valid_data_write_multiple_coils(
      req | fcode(req) = WRITE_MULTIPLE_COILS): bool =
  LET M = second_word(req), N= req‘b(5) IN
    1 <= M AND M <= 1968 AND M <= 8 * N AND 8 * N < M + 8 AND req‘len = 6 + N


% write multiple registers:
% - length must be 6 + N where N = byte count = 6th byte
% - N must be equal to 2 * M  where M = second word of the request
% - M must be between 1 and 123
%
valid_data_write_multiple_registers(
     req | fcode(req) = WRITE_MULTIPLE_REGISTERS): bool =
  LET M = second_word(req), N = req‘b(5) IN
    1 <= M AND M <= 123 AND N = 2 * M AND req‘len = 6 + N


% read file record: to be done
valid_data_read_file_record(req | fcode(req) = READ_FILE_RECORD): bool = false


% read file record: to be done
valid_data_write_file_record(req | fcode(req) = WRITE_FILE_RECORD): bool = false



% read/write multiple registers
% - read quantity (2nd word) must be between 1 and 125 (0x7d)
% - write quantity (4th word) must be between 1 and 121 (0x79)
% - byte count must be 2 * write_quantity
% - length must be byte_count + 10
valid_data_read_write_multiple_registers(
     req | fcode(req) = READ_WRITE_MULTIPLE_REGISTERS): bool =
  LET read_qty = second_word(req),
      write_qty = fourth_word(req),
      byte_count = req‘b(9)
  IN
     1 <= read_qty AND read_qty <= 125 AND 1 <= write_qty AND write_qty <= 121
        AND byte_count = 2 * write_qty AND req‘len = 10 + byte_count


% encapsulated interface transport:
% either MEI_CANopen type or MEI_ReadDeviceIdentification
% read device identification type:
valid_data_MEI(
     req | fcode(req) =  ENCAPSULATED_INTERFACE_TRANSPORT): bool =
     (req‘b(1) = MEI_CANopen)
  OR (req‘b(1) = MEI_ReadDeviceIdentification AND
      req‘len = 4 AND 1 <= req‘b(2) AND req‘b(2) <= 4)
```

```
% data validity check: general case
valid_data(req): bool =
     (fcode(req) = READ_COILS OR fcode(req) = READ_DISCRETE_INPUTS =>
        1 <= second_word(req) AND second_word(req) <= 2000)
 AND (fcode(req) = READ_HOLDING_REGISTERS OR fcode(req) = READ_INPUT_REGISTERS =>
        1 <= second_word(req) AND second_word(req) <= 125)
 AND (fcode(req) = WRITE_SINGLE_COIL =>
         (second_word(req) = 0 OR second_word(req) = 65280)) %% (0x0000 or 0xFF00)
 AND (fcode(req) = DIAGNOSTIC =>
         valid_data_diagnostic(req))
 AND (fcode(req) = WRITE_MULTIPLE_COILS =>
         valid_data_write_multiple_coils(req))
 AND (fcode(req) = WRITE_MULTIPLE_REGISTERS =>
          valid_data_write_multiple_registers(req))
 AND (fcode(req) = READ_FILE_RECORD =>
          valid_data_read_file_record(req))
 AND (fcode(req) = WRITE_FILE_RECORD =>
          valid_data_write_file_record(req))
 AND (fcode(req) = READ_WRITE_MULTIPLE_REGISTERS =>
          valid_data_read_write_multiple_registers(req))
 AND (fcode(req) = ENCAPSULATED_INTERFACE_TRANSPORT =>
          valid_data_MEI(req))




%---------------------------------------------------------------------------
% Constraints on responses to requests that satisfy all
% previous constraints: standard_fcode, acceptable_length, and valid_data
%---------------------------------------------------------------------------

v, v1, v2: VAR (valid_data)
r, r1, r2: VAR raw_msg


% Possible responses to a read coil request with valid address
% (TBD: padding requirement. If M is not a multiple of 8,
% then the last bits of the last byte of the response should all be 0).
acceptable_response_read_coils((v | fcode(v) = READ_COILS), r): bool =
   LET M = second_word(v) IN
     fcode(r) = READ_COILS AND r‘len >= 2 AND
       r‘len = 2 + r‘b(1) AND M <= 8 * r‘b(1) AND 8 * r‘b(1) < M + 8

% Possible responses to a read discrete input request
% (same as above)
acceptable_response_read_discrete_inputs(
            (v | fcode(v) = READ_DISCRETE_INPUTS), r): bool =
   LET M = second_word(v) IN
     fcode(r) = READ_DISCRETE_INPUTS AND r‘len >= 2 AND
       r‘len = 2 + r‘b(1) AND M <= 8 * r‘b(1) AND 8 * r‘b(1) < M + 8


% Acceptable responses to read holding register:
```

```
% length must match the request
acceptable_response_read_holding_registers(
            (v | fcode(v) = READ_HOLDING_REGISTERS), r): bool =
   LET M = second_word(v) IN
     fcode(r) = READ_HOLDING_REGISTERS AND r‘len = 2 + 2 * M
        AND r‘b(1) = 2 * M


% Acceptable responses to read holding register: length must match the request
acceptable_response_read_input_registers(
            (v | fcode(v) = READ_INPUT_REGISTERS), r): bool =
   LET M = second_word(v) IN
     fcode(r) = READ_INPUT_REGISTERS AND r‘len = 2 + 2 * M
        AND r‘b(1) = 2 * M



% Response to write single coil: must be a copy of the request
acceptable_response_write_single_coil(
             (v | fcode(v) = WRITE_SINGLE_COIL), r): bool = equals(r, v)

% Response to write single register: must be a copy of the request
acceptable_response_write_single_register(
             (v | fcode(v) = WRITE_SINGLE_REGISTER), r): bool = equals(r, v)


% Response to read exception status: 2bytes with first equal fcode in request
acceptable_response_read_exception_status(
             (v | fcode(v) = READ_EXCEPTION_STATUS), r): bool =
      r‘len = 2 AND fcode(r) = READ_EXCEPTION_STATUS



% responses to diagnostic requests depend on subfunction
% (This assumes that the device is not in the "listen-only mode")
acceptable_response_diagnostic((v | fcode(v) = DIAGNOSTIC), r): bool =
  % subfunctions whose response is to echo back the request
  IF first_word(v) = RETURN_QUERY_DATA
     OR first_word(v) = RESTART_COMMUNICATION_OPTION
     OR first_word(v) = CHANGE_ASCII_INPUT_DELIMITER
     OR first_word(v) = CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER
     OR first_word(v) = CLEAR_OVERRUN_COUNTER_AND_FLAG
  THEN equals(r, v)

  % other subfunctions: response is a 5 byte message
  % with same subcode and fcode as v
  ELSIF first_word(v) /= FORCE_LISTEN_ONLY_MODE THEN
    fcode(r) = DIAGNOSTIC AND r‘len = 5 AND first_word(r) = first_word(v)

  ELSE % FORCE_LISTEN_ONLY_MODE --> no response
    FALSE
  ENDIF


% get_comm_event_counter: same subfcode + 16 bits of data
acceptable_response_get_comm_event_counter(
               (v | fcode(v) = GET_COMM_EVENT_COUNTER), r): bool =
   r‘len = 5 AND fcode(r) = GET_COMM_EVENT_COUNTER


% get_comm_event_log: three 16-bit registers are returned
% and between 0 and 64 event bytes
acceptable_response_get_comm_event_log(
              (v | fcode(v) = GET_COMM_EVENT_LOG), r): bool =
   r‘len >= 8 AND r‘len <= 72 AND r‘len = 2 + r‘b(1) AND
      fcode(r) = GET_COMM_EVENT_LOG


% response to write multiple coils: echo back the five first bytes of v
acceptable_response_write_multiple_coils(
              (v | fcode(v) = WRITE_MULTIPLE_COILS), r): bool =
   r‘len = 5 AND fcode(r) = WRITE_MULTIPLE_COILS AND
      first_word(r) = first_word(v) AND second_word(r) = second_word(v)

% response to write multiple registers: echo back the five first bytes of v
acceptable_response_write_multiple_registers(
              (v | fcode(v) = WRITE_MULTIPLE_REGISTERS), r): bool =
   r‘len = 5 AND fcode(r) = WRITE_MULTIPLE_REGISTERS AND
      first_word(r) = first_word(v) AND second_word(r) = second_word(v)


% report slave ID: not clear from the standard/device specific.
acceptable_response_report_slave_id(
             (v | fcode(v) = REPORT_SLAVE_ID), r): bool =
     r‘len >= 3 AND fcode(r) = REPORT_SLAVE_ID


% file record stuff: ignored for now


% mask write: echo back the request
acceptable_response_mask_write_register(
              (v | fcode(v) = MASK_WRITE_REGISTER), r): bool = (r = v)


% read/write multiple registers
acceptable_response_read_write_multiple_registers(
              (v | fcode(v) = READ_WRITE_MULTIPLE_REGISTERS), r): bool =
   LET read_qty = second_word(v) IN
     r‘len = 2 + 2 * read_qty AND fcode(r) = READ_WRITE_MULTIPLE_REGISTERS
       AND r‘b(1) = 2 * read_qty


% read FIFO queue:
% byte count (two bytes???)
% fifo count (two bytes) = N (between 0 and 31)
% N register values of two bytes each
acceptable_response_read_fifo_queue(
              (v | fcode(v) = READ_FIFO_QUEUE), r): bool =
   5 <= r‘len AND fcode(r) = READ_FIFO_QUEUE AND r‘len = first_word(r) + 3
      AND r‘len = 5 + 2 * second_word(r) AND second_word(r) <= 31


% Modbus encapsulated interface: ignored for now.
% Just set the constraint on fcode
acceptable_response_MEI(
              (v | fcode(v) = ENCAPSULATED_INTERFACE_TRANSPORT), r): bool =
   r‘len >= 2 AND fcode(r) = ENCAPSULATED_INTERFACE_TRANSPORT
      AND r‘b(1) = v‘b(1)




% Global predicate
acceptable_response(v, r): bool =
    (fcode(v) = READ_COILS AND
     acceptable_response_read_coils(v, r))
 OR (fcode(v) = READ_DISCRETE_INPUTS AND
     acceptable_response_read_discrete_inputs(v, r))
 OR (fcode(v) = READ_HOLDING_REGISTERS AND
     acceptable_response_read_holding_registers(v, r))
 OR (fcode(v) = READ_INPUT_REGISTERS AND
     acceptable_response_read_input_registers(v, r))
 OR (fcode(v) = WRITE_SINGLE_COIL AND
     acceptable_response_write_single_coil(v, r))
 OR (fcode(v) = WRITE_SINGLE_REGISTER AND
     acceptable_response_write_single_register(v, r))
 OR (fcode(v) = READ_EXCEPTION_STATUS AND
     acceptable_response_read_exception_status(v, r))
 OR (fcode(v) = DIAGNOSTIC AND
     acceptable_response_diagnostic(v, r))
 OR (fcode(v) = GET_COMM_EVENT_COUNTER AND
     acceptable_response_get_comm_event_counter(v, r))
 OR (fcode(v) = GET_COMM_EVENT_LOG AND
     acceptable_response_get_comm_event_log(v, r))
 OR (fcode(v) = WRITE_MULTIPLE_COILS AND
     acceptable_response_write_multiple_coils(v, r))
 OR (fcode(v) = WRITE_MULTIPLE_REGISTERS AND
     acceptable_response_write_multiple_registers(v, r))
 OR (fcode(v) = REPORT_SLAVE_ID AND
     acceptable_response_report_slave_id(v, r))
 OR (fcode(v) = MASK_WRITE_REGISTER AND
     acceptable_response_mask_write_register(v, r))
 OR (fcode(v) = READ_WRITE_MULTIPLE_REGISTERS AND
     acceptable_response_read_write_multiple_registers(v, r))
 OR (fcode(v) = READ_FIFO_QUEUE AND
     acceptable_response_read_fifo_queue(v, r))
 OR (fcode(v) = ENCAPSULATED_INTERFACE_TRANSPORT AND
     acceptable_response_MEI(v, r))



% Quick check
same_fcode: LEMMA acceptable_response(v, r) => fcode(v) = fcode(r)




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% DEVICE-SPECIFIC DATA: to be rewritten for every device
% - must include: which functions the device supports
%                 which addresses are available
% - assumption I’ve made: address ranges all start at 0
%   for the four address spaces.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Example device (that does not accept any serial-line command)
supported_function(fcode: byte): bool =
   fcode = READ_COILS OR fcode = READ_DISCRETE_INPUTS OR
   fcode = READ_HOLDING_REGISTERS OR fcode = READ_INPUT_REGISTERS OR
   fcode = WRITE_SINGLE_COIL OR fcode = WRITE_SINGLE_REGISTER OR
   fcode = WRITE_MULTIPLE_COILS OR fcode = WRITE_MULTIPLE_REGISTERS


% Address ranges: last address + 1
% can be set to zero if the address space is empty.
coil_address_range: word = 50                     %% 0 to 49 are valid
discrete_input_address_range: word = 60           %% 0 to 59 are valid
holding_register_address_range: word = 12         %% 0 to 11 are valid
input_register_address_range: word = 10           %% 0 to 9 are valid


% address checking: check whether addresses from a to a + q - 1
% are within the device’s address spaces.
valid_coil_addresses(a: word, q: posword): bool =
   0 <= a AND a + q <= coil_address_range

valid_discrete_input_addresses(a: word, q: posword): bool =
   0 <= a AND a + q <= discrete_input_address_range

valid_holding_register_addresses(a: word, q: posword): bool =
   0 <= a AND a + q <= holding_register_address_range

valid_input_register_addresses(a: word, q: posword): bool =
   0 <= a AND a + q <= input_register_address_range




%-----------------------------------
% Check address range of a request
%-----------------------------------

valid_address(v): bool =
      (fcode(v) = READ_COILS OR fcode(v) = WRITE_MULTIPLE_COILS
          => valid_coil_addresses(first_word(v), second_word(v)))
  AND (fcode(v) = READ_DISCRETE_INPUTS
          => valid_discrete_input_addresses(first_word(v), second_word(v)))
  AND (fcode(v) = READ_HOLDING_REGISTERS OR
       fcode(v) = WRITE_MULTIPLE_REGISTERS
          => valid_holding_register_addresses(first_word(v), second_word(v)))
  AND (fcode(v) = READ_INPUT_REGISTERS
          => valid_input_register_addresses(first_word(v), second_word(v)))
  AND (fcode(v) = WRITE_SINGLE_COIL => valid_coil_addresses(first_word(v), 1))
  AND (fcode(v) = WRITE_SINGLE_REGISTER OR
       fcode(v) = MASK_WRITE_REGISTER
```

```
          => valid_holding_register_addresses(first_word(v), 1))
  AND (fcode(v) = READ_WRITE_MULTIPLE_REGISTERS
          => valid_holding_register_addresses(first_word(v), second_word(v))
              AND valid_holding_register_addresses(third_word(v), fourth_word(v)))




%-----------------------------------------------------
% Requests accepted by the device:
% - satisfy the constraints acceptable_length,
%   valid_data, and valid_address
% - contain a function code that’s standard and
%   supported by the device
%-----------------------------------------------------

accepted_request: TYPE =
   { v | valid_address(v) AND supported_function(fcode(v)) }

g, g1, g2: VAR accepted_request




%%%%%%%%%%%%%%%%%
%  FULL SPEC    %
%%%%%%%%%%%%%%%%%

%----------------------
% Exception responses
%----------------------

%-----------------------------------------------------------
% Exception code: to signal an error the device generates
% an exception pdu, with first-byte = function code of the
% request that caused the error + 0x80.
%
% This does not say what to do if the function code in the
% request already has its high-order bit set.
% I’ve made up something as follows.
%-----------------------------------------------------------

mk_exception(f: byte): byte = IF f <= 127 THEN f+128 ELSE f ENDIF

% code 1 - illegal function code or function not supported by the device
illegal_function(fcode: byte): raw_msg =
           [| mk_exception(fcode), ILLEGAL_FUNCTION |]

% code 2 - bad address for the device
illegal_data_address(fcode: byte): raw_msg =
           [| mk_exception(fcode), ILLEGAL_DATA_ADDRESS |]

% code 3 - bad data in request
% (anything that makes valid_data or acceptable_length fail)
illegal_data_value(fcode: byte): raw_msg =
           [| mk_exception(fcode), ILLEGAL_DATA_VALUE |]

% code 4 - device failure. the request was accepted but
%          the device failed somehow
slave_device_failure(fcode: byte): raw_msg =
           [| mk_exception(fcode), SLAVE_DEVICE_FAILURE |]




%------------------------------------------------------------------
% Definition of the response predicate:
% - modbus_response(m, r)
% means that r is a legal response to the request m
%
% Device characteristics are encapsulated into the two predicates
% supported_function and valid_address
%------------------------------------------------------------------

modbus_response(m, r): bool =
  IF NOT standard_fcode(m) OR NOT supported_function(fcode(m)) THEN

      r = illegal_function(fcode(m))

  ELSIF NOT acceptable_length(m) OR NOT valid_data(m) THEN

      r = illegal_data_value(fcode(m))

  ELSIF NOT valid_address(m) THEN

      r = illegal_data_address(fcode(m))

  ELSE %% valid request: either a device failure or a valid response

      acceptable_response(m, r) OR r = slave_device_failure(fcode(m))

  ENDIF



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  EXECUTABLE SPECIFICATIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Replaced = by the equals predicate
modbus_resp(m, r): bool =
  IF NOT standard_fcode(m) OR NOT supported_function(fcode(m)) THEN

      equals(r, illegal_function(fcode(m)))

  ELSIF NOT acceptable_length(m) OR NOT valid_data(m) THEN

      equals(r,  illegal_data_value(fcode(m)))

  ELSIF NOT valid_address(m) THEN

      equals(r, illegal_data_address(fcode(m)))
  ELSE %% valid request: either a device failure or a valid response

      acceptable_response(m, r) OR equals(r, slave_device_failure(fcode(m)))

  ENDIF


%------------------------------------------------------------------
% To help readability in PVSIO, we write packets as lists of bytes
% and use translations between lists and packets
%------------------------------------------------------------------

map2list(n: nat, f: [below(n) -> byte]): RECURSIVE list[byte] =
   IF n = 0 THEN null
   ELSE cons(f(0), map2list(n - 1, lambda (i: below(n - 1)): f(i+1)))
   ENDIF
 MEASURE n

packet2list(m): list[byte] = map2list(m‘len, m‘b);

list2packet(l: list[byte]): raw_msg =
  LET n = length(l) IN
    IF 1 <= n AND n <= 253
    THEN (# len := n, b := lambda (i: below(n)): nth(l, i) #) :: raw_msg
    ELSE [| 0 |]
    ENDIF

l, l1, l2: VAR list[byte]
b: VAR byte

modbus(l1, l2): bool = modbus_resp(list2packet(l1), list2packet(l2))

% Example random test-case generation
random_tests: LEMMA FORALL l: NOT modbus(l, l)

random_test1: LEMMA FORALL m, b: NOT modbus_resp(m, illegal_function(b))

random_test2: LEMMA FORALL l, b: NOT modbus(l, packet2list(illegal_function(b)))


END modbus
```

Appendix B

SAL Specifications

```
%
% Flat model of the modbus protocol.
% This is an automaton that recognizes the valid modbus requests.
%
% The specifications are derived from the standard:
% "Modbus Application Protocols Specification V1.1a"
% and from the PVS specifications of the protocol in modbus.pvs
%


flat_modbus: CONTEXT =

BEGIN


  %------------------------------
  %  Input is a string of bytes
  %------------------------------

  byte: TYPE = [0 .. 255];

  %% conversion of two bytes to a word
  %% (modbus uses big-endian)
  word: TYPE = [0 .. 65535];

  bytes2word(hi: byte, lo: byte): word = 256 * hi + lo;




  %----------------------------------
  %  Function and subfunction codes
  %----------------------------------

  READ_COILS: byte = 1;

  READ_DISCRETE_INPUTS: byte = 2;

  READ_HOLDING_REGISTERS: byte = 3;
```

```
READ_INPUT_REGISTERS: byte = 4;

WRITE_SINGLE_COIL: byte = 5;

WRITE_SINGLE_REGISTER: byte = 6;

READ_EXCEPTION_STATUS: byte = 7;    % serial line only

DIAGNOSTIC: byte = 8;               % serial line only

GET_COMM_EVENT_COUNTER: byte = 11;  % serial line only

GET_COMM_EVENT_LOG: byte = 12;      % serial line only

WRITE_MULTIPLE_COILS: byte = 15;

WRITE_MULTIPLE_REGISTERS: byte = 16;

REPORT_SLAVE_ID: byte = 17;        % serial line only

READ_FILE_RECORD: byte = 20;

WRITE_FILE_RECORD: byte = 21;

MASK_WRITE_REGISTER: byte = 22;

READ_WRITE_MULTIPLE_REGISTERS: byte = 23;

READ_FIFO_QUEUE: byte = 24;

ENCAPSULATED_INTERFACE_TRANSPORT: byte = 43;


%
% Subfunction codes for diagnostic
% (serial line only)
% These are 16 bit long
%
RETURN_QUERY_DATA: word = 0;

RESTART_COMMUNICATION_OPTION: word = 1;

RETURN_DIAGNOSTIC_REGISTER: word = 2;

CHANGE_ASCII_INPUT_DELIMITER: word = 3;

FORCE_LISTEN_ONLY_MODE: word = 4;

CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER: word = 10;

RETURN_BUS_MESSAGE_COUNT: word = 11;

RETURN_BUS_COMMUNICATION_ERROR_COUNT: word = 12;

RETURN_BUS_EXCEPTION_ERROR_COUNT: word = 13;
```

```
RETURN_SLAVE_MESSAGE_COUNT: word = 14;

RETURN_SLAVE_NO_RESPONSE_COUNT: word = 15;

RETURN_SLAVE_NAK_COUNT: word = 16;

RETURN_SLAVE_BUSY_COUNT: word = 17;

RETURN_BUS_CHARACTER_OVERRUN_COUNT: word = 18;

CLEAR_OVERRUN_COUNTER_AND_FLAG: word = 20;


%
% Modbus Encapsulated Interface types
% (i.e., subfunction  codes for EncapsulatedInterfaceTransport)
%
MEI_CANopen: byte = 13;

MEI_ReadDeviceIdentification: byte = 14;


%-------------------
%  Exception codes
%-------------------

ILLEGAL_FUNCTION: byte = 1;

ILLEGAL_DATA_ADDRESS: byte = 2;

ILLEGAL_DATA_VALUE: byte = 3;

SLAVE_DEVICE_FAILURE: byte = 4;

ACKNOWLEDGE: byte = 5;

SLAVE_DEVICE_BUSY: byte = 6;

MEMORY_PARITY_ERROR: byte = 8;

GATEWAY_PATH_UNAVAILABLE: byte = 10;

GATEWAY_TARGET_DEVICE_FAILED_TO_RESPOND: byte = 11;



%--------------------------
%  Function code checking
%--------------------------

% 0 is the only invalid fcode
invalid_fcode(f: byte): bool = f = 0;


% reserved codes: used by legacy devices but not to be used by others
reserved_fcode(f: byte): bool =
  f=9 OR f=10 OR f=13 OR f=14 OR f=41 OR f=42
      OR f=90 OR f=91 OR f=125 OR f=126 OR f=127;

% user-defined codes: available to the user/unspecified behavior
user_defined_fcode(f: byte): bool =
  (65 <= f AND f <= 72) OR (100 <= f AND f <= 110);

% function codes only for serial-line devices
assigned_serial_line_fcode(f: byte): bool =
   f = READ_EXCEPTION_STATUS OR
   f = DIAGNOSTIC OR
   f = GET_COMM_EVENT_COUNTER OR
   f = GET_COMM_EVENT_LOG OR
   f = REPORT_SLAVE_ID;

% function codes actually defined in the spec and
% applicable to serial-line and non-serial-line devices
assigned_general_fcode(f: byte): bool =
   f = READ_COILS OR
   f = READ_DISCRETE_INPUTS OR
   f = READ_HOLDING_REGISTERS OR
   f = READ_INPUT_REGISTERS OR
   f = WRITE_SINGLE_COIL OR
   f = WRITE_SINGLE_REGISTER OR
   f = WRITE_MULTIPLE_COILS OR
   f = WRITE_MULTIPLE_REGISTERS OR
   f = READ_FILE_RECORD OR
   f = WRITE_FILE_RECORD OR
   f = MASK_WRITE_REGISTER OR
   f = READ_WRITE_MULTIPLE_REGISTERS OR
   f = READ_FIFO_QUEUE OR
   f = ENCAPSULATED_INTERFACE_TRANSPORT;


% assigned codes: union of the 2 previous
assigned_fcode(f: byte): bool =
   assigned_general_fcode(f) OR assigned_serial_line_fcode(f);


% unassigned public codes: not to be used
unassigned_fcode(f: byte): bool =
   1 <= f AND f <= 127 AND
      NOT (user_defined_fcode(f) OR assigned_fcode(f) OR reserved_fcode(f));


% exception code: any code with high-order bit=1
exception_fcode(f: byte): bool = f >= 128;



%---------------------------------------------
%  Subcode checking for diagnostic function
%---------------------------------------------

valid_diagnostic_subcode(f: word): bool =
  f = RETURN_QUERY_DATA OR
  f = RESTART_COMMUNICATION_OPTION OR
  f = RETURN_DIAGNOSTIC_REGISTER OR
  f = CHANGE_ASCII_INPUT_DELIMITER OR
  f = FORCE_LISTEN_ONLY_MODE OR
  f = CLEAR_COUNTERS_AND_DIAGNOSTIC_REGISTER OR
  f = RETURN_BUS_MESSAGE_COUNT OR
  f = RETURN_BUS_COMMUNICATION_ERROR_COUNT OR
  f = RETURN_BUS_EXCEPTION_ERROR_COUNT OR
  f = RETURN_SLAVE_MESSAGE_COUNT OR
  f = RETURN_SLAVE_NO_RESPONSE_COUNT OR
  f = RETURN_SLAVE_NAK_COUNT OR
  f = RETURN_SLAVE_BUSY_COUNT OR
  f = RETURN_BUS_CHARACTER_OVERRUN_COUNT OR
  f = CLEAR_OVERRUN_COUNTER_AND_FLAG;

reserved_diagnostic_subcode(f: word): bool = NOT valid_diagnostic_subcode(f);



%-----------------------
%  Checks on MEI types
%-----------------------

valid_mei_type(f: byte): bool =
   f = MEI_CANopen OR f = MEI_ReadDeviceIdentification;

reserved_mei_type(f: byte): bool = NOT valid_mei_type(f);




%-----------------------------------------------
%  Length constraints for the standard requests
%  - this just gives minimal and maximal length
%  - further checking is required for variable-
%  length requests.
%-----------------------------------------------

acceptable_length(fcode: byte, len: byte): bool =
    (fcode = READ_COILS AND len = 5)
 OR (fcode = READ_DISCRETE_INPUTS AND len = 5)
 OR (fcode = READ_HOLDING_REGISTERS AND len = 5)
 OR (fcode = READ_INPUT_REGISTERS AND len = 5)
 OR (fcode = WRITE_SINGLE_COIL AND len = 5)
 OR (fcode = WRITE_SINGLE_REGISTER AND len = 5)
 OR (fcode = READ_EXCEPTION_STATUS AND len = 1)
 OR (fcode = DIAGNOSTIC AND len >= 5 AND len <= 253)
 OR (fcode = GET_COMM_EVENT_COUNTER AND len = 1)
 OR (fcode = GET_COMM_EVENT_LOG AND len = 1)
 OR (fcode = WRITE_MULTIPLE_COILS AND 6 <= len AND len <= 253)
 OR (fcode = WRITE_MULTIPLE_REGISTERS AND 6 <= len AND len <= 253)
 OR (fcode = REPORT_SLAVE_ID AND len = 1)
 OR (fcode = READ_FILE_RECORD AND 2 <= len AND len <= 245)
 OR (fcode = WRITE_FILE_RECORD AND 2 <= len AND len <= 245)
 OR (fcode = MASK_WRITE_REGISTER AND len = 7)
 OR (fcode = READ_WRITE_MULTIPLE_REGISTERS AND 10 <= len AND len <= 253)
 OR (fcode = READ_FIFO_QUEUE AND len = 3)
 OR (fcode = ENCAPSULATED_INTERFACE_TRANSPORT AND 2 <= len AND len <= 253);
```

```
% Requests that contain only the fcode
single_byte_request(fcode: byte): bool =
    fcode = READ_EXCEPTION_STATUS OR fcode = GET_COMM_EVENT_COUNTER OR
    fcode = GET_COMM_EVENT_LOG OR fcode = REPORT_SLAVE_ID;




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Device-specific data                    %
% - which function the device supports    %
% - which addresses are available         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

supported_fcode(fcode: byte): bool =
   fcode = READ_COILS OR fcode = READ_DISCRETE_INPUTS OR
   fcode = READ_HOLDING_REGISTERS OR fcode = READ_INPUT_REGISTERS OR
   fcode = WRITE_SINGLE_COIL OR fcode = WRITE_SINGLE_REGISTER OR
   fcode = WRITE_MULTIPLE_COILS OR fcode = WRITE_MULTIPLE_REGISTERS OR
   fcode = DIAGNOSTIC OR fcode = MASK_WRITE_REGISTER;


% Address ranges: last address + 1
% can be set to zero if the address space is empty.
coil_address_range: word = 50;                    %% 0 to 49 are valid
discrete_input_address_range: word = 60;          %% 0 to 59 are valid
holding_register_address_range: word = 12;        %% 0 to 11 are valid
input_register_address_range: word = 10;          %% 0 to 9 are valid


% address checking: check whether addresses from a to a + q - 1
% are within the device’s address spaces.
valid_coil_addresses(a: word, q: word): bool =
   0 <= a AND a + q <= coil_address_range;

valid_discrete_input_addresses(a: word, q: word): bool =
   0 <= a AND a + q <= discrete_input_address_range;

valid_holding_register_addresses(a: word, q: word): bool =
   0 <= a AND a + q <= holding_register_address_range;

valid_input_register_addresses(a: word, q: word): bool =
   0 <= a AND a + q <= input_register_address_range;




%------------
% Automaton
%------------

state: TYPE = {
  read_length,
  read_fcode,
  read_first_word_byte1,
  read_first_word_byte2,
  read_second_word_byte1,
  read_second_word_byte2,
  read_third_word_byte1,
  read_third_word_byte2,
  read_fourth_word_byte1,
  read_fourth_word_byte2,
  read_byte_count,
  finish_request,
  done,

  read_mei_type,
  read_rest,

  process_file_request

};


status: TYPE = {
   unknown,
   length_too_short,
   length_too_long,
   fcode_is_invalid,
   fcode_is_reserved,
   fcode_is_user_defined,
   fcode_is_serial_line_only,
   fcode_is_unassigned,
   fcode_is_exception,
   fcode_not_supported,
   diagnostic_subcode_is_reserved,
   restart_communication_option,
   force_listen_only,
   mei_type_is_reserved,
   bad_length_for_fcode,
   bad_length_for_subcode,
   invalid_byte_count,
   byte_count_and_length_are_inconsistent,
   invalid_address,
   invalid_data,
   valid_request
};


%
% Convention used:
% - first byte read defines the packet length
% - then the packet itself is read and checked byte by byte
%   (as many bytes are read as indicated by length)
% - after that, an extra state transition may be performed to update
%   the stat flag. The input byte on this last step is consumed
%   and ignored.
%
modbus: MODULE =
  BEGIN
    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


  INITIALIZATION
    pc = read_length;
    stat = unknown;

    len = 0;
    fcode = 0;
    byte_count = 0;
    first_word = 0;
    second_word = 0;
    third_word = 0;
    fourth_word = 0;

    aux = 0

  TRANSITION
   [ pc = read_length -->
         len’ = b;
         stat’ = IF b < 1
                 THEN length_too_short
                 ELSIF b > 253
                 THEN length_too_long
                 ELSE unknown
                 ENDIF;
         pc’ = IF 1 <= b AND b <= 253 THEN read_fcode ELSE done ENDIF


    [] pc = read_fcode -->
          fcode’ = b;
          stat’ = IF invalid_fcode(b)
                  THEN fcode_is_invalid
                  ELSIF reserved_fcode(b)
                  THEN fcode_is_reserved
                  ELSIF user_defined_fcode(b)
                  THEN fcode_is_user_defined
                  ELSIF unassigned_fcode(b)
                  THEN fcode_is_unassigned
                  ELSIF exception_fcode(b)
                  THEN fcode_is_exception
                  ELSIF NOT supported_fcode(b)
                  THEN fcode_not_supported
                  ELSIF NOT acceptable_length(b, len)
                  THEN bad_length_for_fcode
                  ELSIF single_byte_request(b)
                  THEN valid_request
```

```
              ELSE unknown
              ENDIF;

      pc’ = IF stat’ /= unknown
            THEN done
            ELSIF b = ENCAPSULATED_INTERFACE_TRANSPORT
            THEN read_mei_type
            ELSIF (b = READ_FILE_RECORD OR fcode = WRITE_FILE_RECORD)
            THEN process_file_request
            ELSE read_first_word_byte1
            ENDIF


%% read first word
%% DIAGNOSTIC and READ_FIFO_QUEUE are treated specially
%% all other requests cannot be checked yet: second word is needed
[] 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 to read 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


%% second word
[] pc = read_second_word_byte1 -->
      aux’ = b;
      pc’ = read_second_word_byte2

[] pc = read_second_word_byte2 AND fcode = READ_COILS -->
      second_word’ = 256 * aux + b;
      stat’ = IF second_word’ < 1 OR second_word’ > 2000
              THEN invalid_data
              ELSIF valid_coil_addresses(first_word, second_word’)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’  = done

[] pc = read_second_word_byte2 AND fcode = READ_DISCRETE_INPUTS -->
      second_word’ = 256 * aux + b;
      stat’ = IF second_word’ < 1 OR second_word’ > 2000
              THEN invalid_data
              ELSIF valid_discrete_input_addresses(first_word, second_word’)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

[] pc = read_second_word_byte2 AND fcode = READ_HOLDING_REGISTERS -->
      second_word’ = 256 * aux + b;
      stat’ = IF second_word’ < 1 OR second_word’ > 125
              THEN invalid_data
              ELSIF valid_holding_register_addresses(first_word, second_word’)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

[] pc = read_second_word_byte2 AND fcode = READ_INPUT_REGISTERS -->
      second_word’ = 256 * aux + b;
      stat’ = IF second_word’ < 1 OR second_word’ > 125
              THEN invalid_data
              ELSIF valid_input_register_addresses(first_word, second_word’)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

[] pc = read_second_word_byte2 AND fcode = WRITE_SINGLE_COIL -->
      second_word’ = 256 * aux + b;
      stat’ = IF second_word’ /= 0 AND second_word’ /= 65280
              THEN invalid_data
              ELSIF valid_coil_addresses(first_word, 1)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

[] pc = read_second_word_byte2 AND fcode = WRITE_SINGLE_REGISTER -->
      second_word’ = 256 * aux + b;
      stat’ = IF valid_holding_register_addresses(first_word, 1)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done
```

```
%% Diagnostic request. All subcodes except RETURN_QUERY_DATA
%% Length must be 5
[] pc = read_second_word_byte2 AND fcode = DIAGNOSTIC -->
      second_word’ = 256 * aux + b;
      stat’ = IF len /= 5
              THEN bad_length_for_subcode
              ELSIF first_word = RESTART_COMMUNICATION_OPTION AND
                    (second_word’ = 0 OR second_word’ = 65280)
              THEN restart_communication_option
              ELSIF first_word = FORCE_LISTEN_ONLY_MODE AND second_word’ = 0
              THEN force_listen_only
              ELSIF first_word = CHANGE_ASCII_INPUT_DELIMITER AND b = 0
              THEN valid_request
              ELSIF second_word’ = 0
              THEN valid_request
              ELSE invalid_data
              ENDIF;
      pc’ = done


[] pc = read_second_word_byte2 AND
   (fcode = WRITE_MULTIPLE_COILS OR fcode = WRITE_MULTIPLE_REGISTERS) -->
      second_word’ = 256 * aux + b;
      pc’ = read_byte_count

[] pc = read_second_word_byte2 AND
   (fcode = MASK_WRITE_REGISTER OR fcode = READ_WRITE_MULTIPLE_REGISTERS) -->
      second_word’ = 256 * aux + b;
      pc’ = read_third_word_byte1


%% Third word
[] pc = read_third_word_byte1 -->
      aux’ = b;
      pc’ = read_third_word_byte2

[] pc = read_third_word_byte2 AND fcode = MASK_WRITE_REGISTER -->
      third_word’ = 256 * aux + b;
      stat’ = IF valid_holding_register_addresses(first_word, 1)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

[] pc = read_third_word_byte2 AND fcode = READ_WRITE_MULTIPLE_REGISTERS -->
      third_word’ = 256 * aux + b;
      pc’ = read_fourth_word_byte1

%% Fourth word (command must be READ_WRITE_MULTIPLE_REGISTERS)
[] pc = read_fourth_word_byte1 -->
      aux’ = b;
      pc’ = read_fourth_word_byte2

[] pc = read_fourth_word_byte2 -->
      fourth_word’ = 256 * aux + b;
      pc’ = read_byte_count
```

```
%% read byte count
[] pc = read_byte_count -->
      byte_count’ = b;
      %% set aux’ to the number of extra bytes to read
      aux’ = IF fcode = WRITE_MULTIPLE_COILS OR fcode = WRITE_MULTIPLE_REGISTERS
             THEN len - 6
             ELSE len - 10  %% fcode = READ_WRITE_MULTIPLE_REGISTERS
             ENDIF;
      pc’ = finish_request;


%% read and skip rest of the request
[] pc = finish_request AND aux > 0 -->
      aux’ = aux - 1

%% after the last byte has been read

%% Write multiple coils:
%% len must be byte_count + 6
%% second_word = write quantity = M : must be between 1 and 0x7b0 (i.e., 1968)
%% byte count must be equal to ceil(M/8) (i.e., M <= 8 * byte_count < M + 8)
[] pc = finish_request AND aux = 0 AND fcode = WRITE_MULTIPLE_COILS -->
      stat’ = IF len /= byte_count + 6
              THEN byte_count_and_length_are_inconsistent
              ELSIF second_word < 1 OR second_word < 1968
              THEN invalid_data
              ELSIF second_word > 8 * byte_count OR
                    8 * byte_count >= second_word + 8
              THEN invalid_byte_count
              ELSIF valid_coil_addresses(first_word, second_word)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

%% Write multiple registers:
%% len must be byte_count + 6
%% second_word = write quantity = M : must be between 1 and 0x7b (i.e., 123)
%% byte count must be equal to 2 * M
[] pc = finish_request AND aux = 0 AND fcode = WRITE_MULTIPLE_REGISTERS -->
      stat’ = IF len /= byte_count + 6
              THEN byte_count_and_length_are_inconsistent
              ELSIF second_word < 1 OR second_word > 123
              THEN invalid_data
              ELSIF 2 * byte_count /= second_word
              THEN invalid_byte_count
              ELSIF valid_holding_register_addresses(first_word, second_word)
              THEN valid_request
              ELSE invalid_address
              ENDIF;
      pc’ = done

%% Read write multiple registers:
%% len must be byte count + 10
%% second_word = read_quantity must be between 1 and 0x7d (i.e., 125)
%% fourth_word = write_quantity must be between 1 and 0x79 (i.e., 121)
```

```
    %% byte count must be 2 * write quantity
    [] pc = finish_request AND aux = 0 AND fcode = READ_WRITE_MULTIPLE_REGISTERS -->
          stat’ = IF len /= byte_count + 10
                  THEN byte_count_and_length_are_inconsistent
                  ELSIF second_word < 1 OR second_word > 125 OR
                        fourth_word < 1 OR fourth_word > 121
                  THEN invalid_data
                  ELSIF 2 * byte_count /= fourth_word
                  THEN invalid_byte_count
                  ELSIF valid_holding_register_addresses(first_word, second_word)
                    AND valid_holding_register_addresses(third_word, fourth_word)
                  THEN valid_request
                  ELSE invalid_address
                  ENDIF;
           pc’ = done


    %% MEI
    [] pc = read_mei_type -->
          stat’ = IF valid_mei_type(b)
                  THEN valid_request
                  ELSE mei_type_is_reserved
                  ENDIF;
          aux’ = len - 2;
          pc’ = read_rest

    [] pc = read_rest AND aux > 0 -->
          aux’ = aux - 1

    [] pc = read_rest AND aux = 0 -->
          pc’ = done



    [] ELSE -->
    ]
  END;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Test-case generation   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% To generate a test case of length n that satisfies the
% constraints in lemma test<i>, run
%        sal-bmc -v 4 -d (n+1) test<i>
% or     sal-bmc -v 4 -d <n+1) -s siege test<i>
%
% The counterexample produced by sal-bmc is a test-case
% Note: the option -s siege uses the siege SAT solver, which
% is randomized. Each call gives then a different test case.
%
test1: LEMMA modbus |-
          G(pc = done => stat /= length_too_short);

test2: LEMMA modbus |-
          G(pc = done => stat /= length_too_long);
```

```
test3: LEMMA modbus |-
          G(pc = done => stat /= fcode_is_invalid);

test4: LEMMA modbus |-
          G(pc = done => stat /= fcode_is_reserved);

test5: LEMMA modbus |-
          G(pc = done => stat /= fcode_is_user_defined);

test6: LEMMA modbus |-
          G(pc = done => stat /= fcode_is_unassigned);

test7: LEMMA modbus |-
          G(pc = done => stat /= fcode_is_exception);

test8: LEMMA modbus |-
          G(pc = done => stat /= fcode_not_supported);

%% not possible for current device model (DIAGNOSTIC not supported)
test9: LEMMA modbus |-
          G(pc = done => stat /= diagnostic_subcode_is_reserved);

test10: LEMMA modbus |-
          G(pc = done => stat /= restart_communication_option);

test11: LEMMA modbus |-
          G(pc = done => stat /= force_listen_only);

test12: LEMMA modbus |-
          G(pc = done => stat /= mei_type_is_reserved);

test13: LEMMA modbus |-
          G(pc = done => stat /= bad_length_for_fcode);

test14: LEMMA modbus |-
          G(pc = done => stat /= bad_length_for_subcode);

test15: LEMMA modbus |-
          G(pc = done => stat /= invalid_byte_count);

test16: LEMMA modbus |-
          G(pc = done => stat /= byte_count_and_length_are_inconsistent);

test17: LEMMA modbus |-
          G(pc = done => stat /= invalid_address);

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

test19: LEMMA modbus |-
          G(pc = done => stat /= valid_request);

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

test21: LEMMA modbus |-
```

```
            G(pc = done AND fcode = READ_COILS => stat /= bad_length_for_fcode);

  test22: LEMMA modbus |-
            G(pc = done AND fcode = READ_COILS => stat /= valid_request);

  test23: LEMMA modbus |-
            G(pc = done AND fcode = READ_COILS => stat /= invalid_data);

  test24: LEMMA modbus |-
            G(pc = done AND fcode = READ_COILS => stat /= invalid_address);

END
```