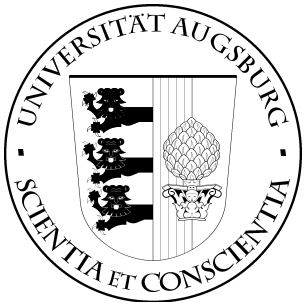
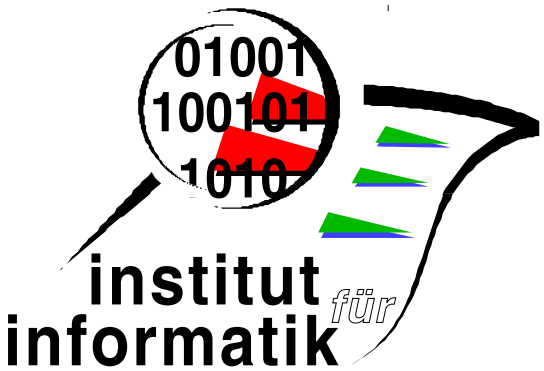
Universität Augsburg



electronic Ticketing
A Smartcard Application Case-Study

Dominik Haneberg

Report 2002-16 Dezember 2002



Institut für Informatik
D-86135 Augsburg

Dominik Haneberg

electronic ticketing
A Smartcard Application Case-Study

Abstract

The electronic ticketing was developed within the scope of the Go!Card project. It is used as a test object for the techniques for the development of secure smartcard applications that are developed in the Go!Card project. Our development method for secure smartcard applications is described in [HRS02].

The electronic ticketing case study deals with an e-commerce scenario for the electronic sale of railway or flight tickets. The customers buy their tickets from a server that transmits the signed and encrypted tickets to the customer, where they are loaded on the customers smartcard. Then the smartcard decrypts and verifies the tickets and stores them. The tickets are checked and obliterated offline by the train's conductor using a portable computer. This report describes the scenario of the electronic ticketing case study, we explain the different functions and discuss desirable security objectives.

This report completely supersedes [Han01] which now is obsolete. The design and the description of the protocols were adjusted to the latest findings of our research and some additional protocols were added.

Contents

List of Figures

Chapter 1

Introduction

1.1 Chipcards

Magnetic stripe cards are used as portable, machine-readable data storage for a while. Best known example are the cards issued by the banks, most people have at least one. The possession of such a card and the knowledge of the appropriate PIN[1] permits access to the account of the card holder.

Over the years some basic disadvantages of magnetic stripe cards became obvious. The storage capacity is very small, the data stored on the card isn't protected because everyone can read it using a magnetic stripe reader. Furthermore magnetic stripe cards aren't taper-poof. External effects like magnetic fields endanger the data on the magnetic stripe. It happens, that automatic teller machines refuse cards, because the stored date is corrupted.

Some of the weaknesses of magnetic stripe cards could be eliminated by a new technology. Chipcards offer a higher storage capacity, they are less vulnerable to external effects and they offer the possibility to restrict the access to the stored data and to prevent manipulation. Chipcards are plastic cards of the same size as common credit cards with an embedded integrated circuit. We distinguish between memory cards which only have a memory chip embedded and microprocessor cards which have a microprocessor embedded which is can execute programs. The technical data and the physical characteristics of chipcards are determined in the ISO-norms 7810, 7813 and 7816.

The simplest available chipcards are so called memory cards. They only have a memory chip embedded, commonly EPROM, EEPROM or FLASH memory.

The more capable chipcards are the so called microprocessor cards or smartcards. Because of their embedded microprocessor, they have wider range of possible applications as the memory cards. The processor cards aren't just passive data carriers, but rather they are able to perform calculations and to process informations. The fundamental advantage of this is, that the secret data that is stored on the card, mustn't leave the card in order to be used. The card carries out all necessary calculations herself and outputs only the result, but not the used secret data. Because the stored secret data can be processed on the card, the card applications can be implemented in a way that prohibits unauthorized access or manipulation of the application data. Therefore smartcards are a tamper-proof storage for secret data. In the meantime many smartcards are available in specialized cryptographic versions, that feature an integrated crypto-coprocessor, that enables the smartcard to exchange encrypted

[1] PIN: Personal Identification Number

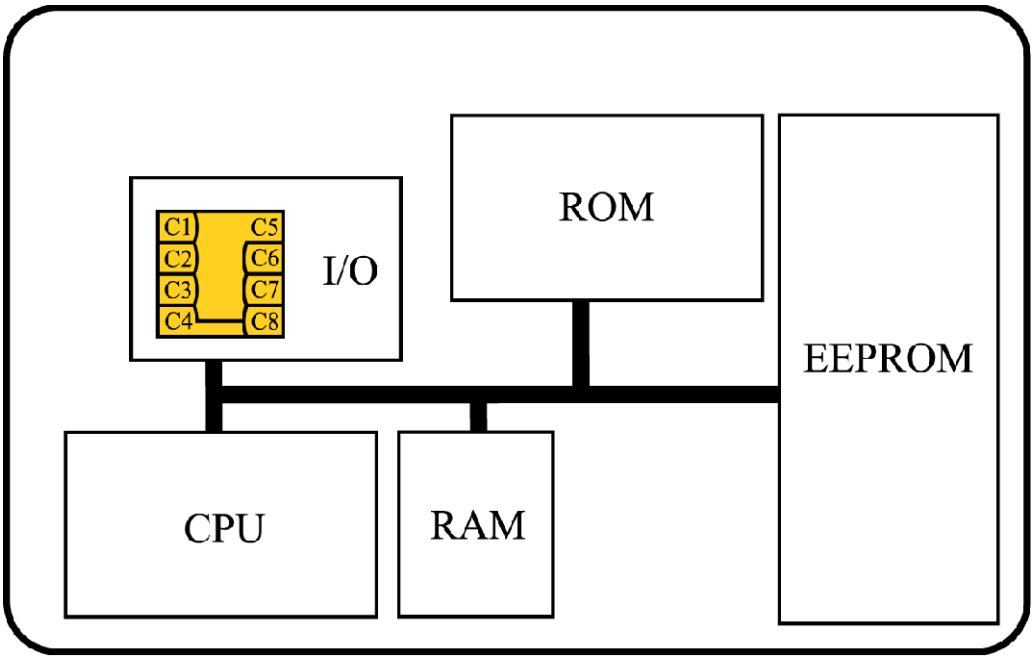


Figure 1.1: Schematic picture of a microprocessor card

data . Best known example of smartcards is the SIM, that is included in all GSM-cell phones. The SIM identifies the user towards the telephone company and enables the encrypted communication between the cell phone and the base station. The latest step in the development are multiapplicative smartcards that allow to install and delete card programs dynamically. Because of that it is possible to supplement a smartcard with additional services at any time. A special case of the smartcars are the so called JavaCards that have the advantage that they can be programmed in the high-level programming language Java.

1.2 The Go!Card Project

The research project "Go!Card — Formal methods for the secure application of Java smartcards" investigates security problems connected to the application of multiapplicative Java smartcards. Our objective is a development method for provably secure smartcard applications and the provision of tools for design of the communication protocols and the development of the card programs. The research project Go!Card is sponsored by the German national research foundation (Deutsche Forschungsgemeinschaft) within the scope of the Schwerpunktprogramm "Sicherheit in der Informations- und Kommunikationstechnik".

There are four crucial points in the development of a smartcard application in which a mistake can lead to security leaks or malfunctions.

1. Application and protocol design
2. Mapping of highlevel data types on the data types supported by the smartcard
3. Correctness of the implementation

4. Coexistence of potentially evil cardlets on one card

1.2.1 Application and protocol design

Composed communication protocols combining the authentication of the participants, the transfer of the application data and the digital signatures or the like, often occur in smartcard applications. Careless design of these protocols can easily lead to vulnerabilities in the system [AN95, Pau98]. Different security issues are possible, for example the accidental disclosure of secret data or the illegitimate duplication of valuables, e. g. electronic coins.

1.2.2 Data mapping

In the communication protocols occur as data types documents, signatures or similar, but the data exchange between card and terminal is done using APDUs[2]. The avoidance of errors in the mapping of the communication protocol data types onto the APDUs is subject of this work package.

1.2.3 Correctness of the implementation

This work package investigates the prevention of implementation errors in the card programs and the securing of a correct implementation of the communication protocols. Therefore we adjust and if necessary enhance well known techniques for program verification to the specifics of JavaCard programs.

1.2.4 Multiapplicativity

The existence of multiple application on one smartcard induces the problem of an appropriate protection of the application data. The card programs have to protect their data against other potentially evil applications. Besides the sealing-off of the card applications, a well-regulated communication between cardlets must be possible. The work package investigates security policies that ensure the desired properties of the data flow.

1.2.5 Synopsis

It is our objective to establish an integrated methodology for the development of secure smartcard applications. This method should cover all the crucial points of the application development and it should accompany the developer from the application design till the implementation.

[2] Application Protocol Data Unit

Chapter 2

The electronic Ticketing case study

To trial our new development method and the verification calculus we test our results on case studies. The central case study of the Go!Card project is the electronic Ticketing application. This is a e-commerce scenario dealing with the sales of railway our flight tickets. The central concept of electronic Ticketing is that the tickets no longer exist in a paper form. The tickets are ordered, issued, stored and checked solely electronic. The customer orders the tickets via a web interface on any PC with smartcard reader or via specialized terminals, which could be set up in railway stations or travel agencies. The order information is transmitted to the server of the issuer, which generates matching tickets and reservations and transmits them over the internet to the smartcard of the customer. The purchased tickets get stored on the smartcard and can be checked offline by the conductor if required. Therefore he uses a portable card reader that reads the ticket data from the smartcard and displays them. Using the same device the conductor obliterates used tickets. Cryptographic methods are used to prevent manipulation and to ensure that only valid tickets get stored on the smartcard.

2.0.6 Scenario

An overall picture of the electronic Ticketing scenario can be seen in figure 2.1. To realize the core services three central components are used. These are:

- The customer's smartcard
- The server of the ticket issuer
- The conductor's checking device

The smartcard contains the card application that handles the encrypted communication with the server of the ticket issuer, stores the tickets and shows and obliterates tickets in interaction with a conductor device. To be able to fulfill its tasks the smartcard must be able to perform cryptographic operations. The use of encryption ensures the confidentiality of the customer data, the use of digital signatures ensures integrity and authenticity of the transferred data. The server of the ticket issuer is activated by a customer who wants to buy a ticket. The server contacts the customer's smartcard and prepares for the transmission of a new ticket. Then he accepts the customer's order and generates a corresponding ticket, lets the user pay the ticket and transfers ticket data to the customer's smartcard.

The conductor's checking device is used in trains to check and obliterate the tickets of the passengers. To do so the conductor's device authenticates itself towards the smartcard,

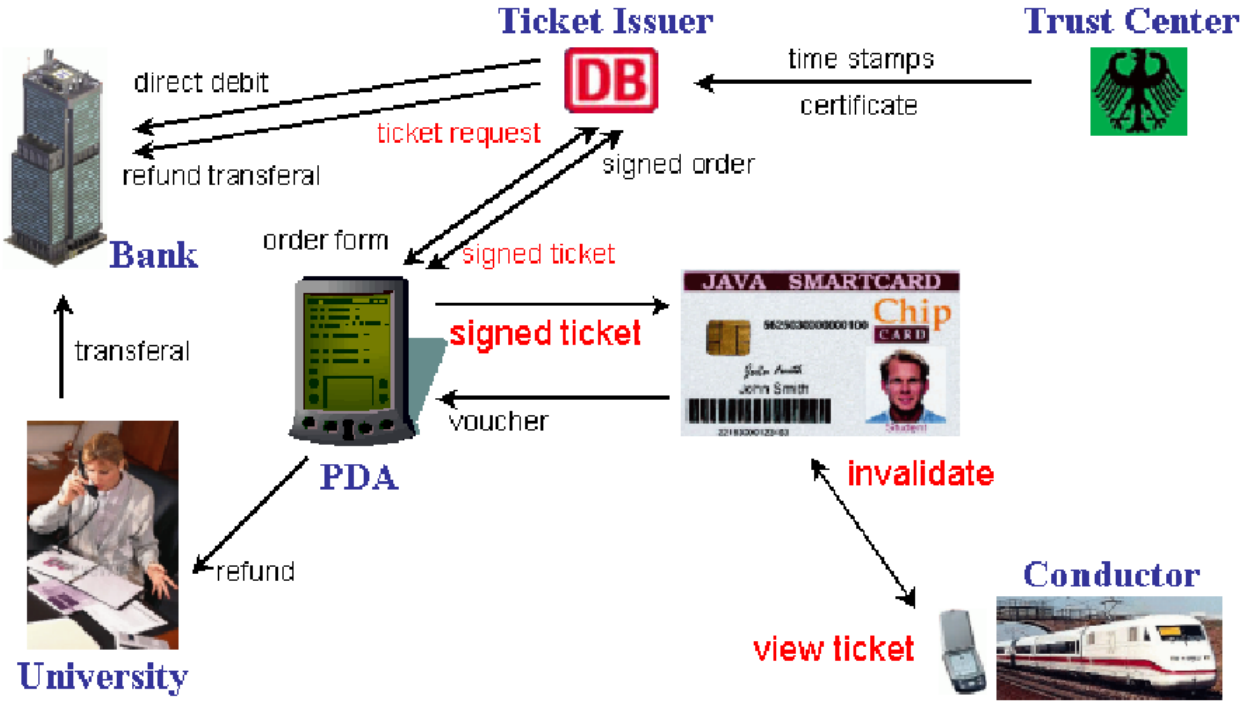


Figure 2.1: Structure of the electronic Ticketing case study

requests the ticket data and shows them to the conductor for verification. If the ticket is okay the checking device orders the smartcard to mark the ticket as obliterated. Thereby it is crucial that the verification of a ticket can be done offline, thus no network connection between the train (conductor) and ticket sales system is needed.

The cardlet also provides the possibility to pass on a ticket from one smartcard to another. Thereby it is important that the ticket is at no time the ticket is in usable condition on both cards simultaneously because this would permit to defraud the ticket issuer. The impossibility of imposture is ensured by using an appropriate communication protocol.

Additionally the application allows it to return unused tickets electronically. When an unused ticket should be returned to the issuer the cardlet transfers the ticket to the server which initiates the refund of the customer. To prevent fraud this function also requires appropriate protocols.

2.0.7 Extensions

The core services of the electronic Ticketing case study can be extended by further functions, for example the automatic refund of tickets bought for business trips. .

Chapter 3

Attributes and methods of the major classes

3.1 The Terminals

The core features of the electronic Ticketing system are realized using three terminals. Purchase and return of tickets are implemented in the **Issuer Terminal**, card management functions (*Delete Ticket*, *Set User PIN*, ...) are consolidated in the **User Terminal**. Obliteration of tickets is the task of the **Conductor Terminal**.

3.1.1 User Terminal

The **User Terminal** is used by the card owner to administrate the card, for example this terminal permits the user to view the currently stored tickets, to delete old tickets and to set an new PIN.

Tasks

This terminal is used for the following functions:

- Change the lock status of a ticket
- Delete a ticket
- Set a new user PIN
- Transfer a ticket
- View the tickets

Attributes

- `dest_card_challenge` This attribute is used to store a challenge received from the destination card of a ticket transfer.
- `dest_card_key` This attribute is used to store the public key of the destination card.

- `dest_card_key_sig` This attribute is used to store the signature of the public key of the destination card.
- `error` This attribute stores the internal error state of the terminal.
- `last_cmd` This attribute is used to store the last command that was sent to the card.
- `lock_status` This attribute is used to store the desired lock status in the *Change Lock Status*-protocol.
- `new_pin` This attribute is used to store the new PIN when a new user PIN should be set.
- `pin` This attribute is used to store the PIN of the card owner.
- `source_card_challenge` This attribute is used to store a challenge received from the source card of a ticket transfer.
- `source_card_key` This attribute is used to store the public key of the source card.
- `source_card_key_sig` This attribute is used to store the signature of the public key of the source card.
- `the_transferred_ticket` This attribute is used to store the ticket that is currently transferred.
- `the_transferred_ticket_sig` This attribute is used to store the signature of the currently transferred ticket.
- `ticket_no` This attribute is used to store the number of the selected ticket, for example the number of the ticket that should be deleted.

3.1.2 Conductor Terminal

The **Conductor Terminal** is used by the conductor to view and to obliterate tickets. It can also be used by the card owner to unlock a locked ticket if necessary.

Tasks

- View the tickets
- Obliterate tickets
- Unlock tickets

Attributes

- `card_authenticated` This attribute stores if a card was successfully authenticated.
- `card_challenge` This attribute is used to store a challenge received from a card.
- `check` This attribute contains a new check that is to be used to obliterate a ticket.
- `error` This attribute stores the internal error state of the terminal.

- `last_cmd` This attribute is used to store the last command that was sent to the card.
- `pin` This attribute is used to store the PIN of the card owner.
- `public_card_key` This attribute is used to store the public key of the card.
- `public_card_key_sig` This attribute is used to store the signature of the public key of the card.
- `public_conductor_key` This attribute is used to store the public key of the conductor terminal.
- `public_conductor_key_sig` This attribute is used to store the signature of the public key of the conductor terminal.
- `secret_conductor_key` This attribute is used to store the secret key of the conductor terminal.
- `terminal_challenge` This attribute is used to store a challenge generated by the terminal.
- `ticket_no` This attribute is used to store the number of the selected ticket.

3.1.3 Issuer Terminal

The **Issuer Terminal** is used to buy new tickets and to return unused ones.

Tasks

- Buy tickets
- Give back tickets

Attributes

- `card_authenticated` This attribute stores if a card was successfully authenticated.
- `card_challenge` This attribute is used to store a challenge received from a card.
- `dest_card_authenticated` Before running the *Recover Transfer*-protocol, this attribute must be true to signal a successful authentication of the destination card.
- `dest_card_key` This attribute is used by the *Recover Transfer*-protocol to store the public key of the destination card of a ticket transfer.
- `dest_ex_trans_ticket` This attribute is set to true during the processing of the *Recover Transfer*-protocol if there is a ticket on the destination card that is marked as in transfer.
- `dest_source_key` While a ticket is transfered, the destination card stores the public key of the source card in the attribute `source_card_key`. The value of this attribute is retrieved by the terminal during the processing of the *Recover Transfer*-protocol and stored in this field.

- `dest_transfer_in_progress` Distinguishes whether there is an unfinished transfer on the destination card or not.
- `error` This attribute stores the internal error state of the terminal.
- `last_cmd` This attribute is used to store the last command that was sent to the card.
- `public_card_key` This attribute is used to store the public key of the card.
- `public_card_key_sig` This attribute is used to store the signature of the public key of the card.
- `public_issuer_key` This attribute is used to store the public key of the ticket issuer.
- `secret_issuer_key` This attribute is used to store the secret key of the ticket issuer.
- `source_card_authenticated` Before running the *Recover Transfer*-protocol, this attribute must be true to signal a successful authentication of the source card.
- `source_card_key` This attribute is used by the *Recover Transfer*-protocol to store the public key of the source card of a ticket transfer.
- `source_dest_key` While a ticket is transfered, the source card stores the public key of the destination card in the attribute `dest_card_key`. The value of this attribute is retrieved by the terminal during the processing of the *Recover Transfer*-protocol and stored in this field.
- `source_ex_trans_ticket` This attribute is set to true during the processing of the *Recover Transfer*-protocol if there is a ticket on the source card that is marked as in transfer.
- `source_transfer_in_progress` Distinguishes whether there is an unfinished transfer on the source card or not.
- `terminal_challenge` This attribute is used to store a challenge generated by the terminal.
- `ticket` This attribute is used to store a new ticket.
- `ticket_status` Used to store the status of a ticket.

3.2 The Cardlet

The cardlet is that part of the software necessary for electronic ticket which resides on the smartcard. It is responsible for realisation of communication with the terminals and for the storage of application data.

3.2.1 Functions

- `anyProtocolRunning()` Returns true if there's currently a protocol running on the card, i. e. if the `nxt_cmd` isn't `any` and otherwise false.
- `incompleteLoad()` Returns true if a ticket load wasn't successfully completed and otherwise false.
- `incompleteTransfer()` Returns true if the card participated in a ticket transfer that wasn't properly completed and otherwise false.
- `pinOk(pin pin)` Returns true if the argument is the correct PIN of the card owner and otherwise false.
- `thisProtocolRunning(cardinstruction ci)` Returns true if the current step of the protocol running is equal to the argument, i. e. if `nxt_cmd` is equal to the value of `ci`. Otherwise the functions returns false.
- `generateChallenge()` Returns a new nonce.

3.2.2 Attributes

- `card_challenge` This attribute is used to store the challenge created by the card.
- `conductor_authenticated` Boolean flag to distinguish whether the conductor was successfully authenticated or not.
- `dest_card_challenge` Contains the challenge generated by the destination card of a ticket transfer.
- `dest_card_key` Contains the public key of the destination card of a ticket transfer.
- `last_ticket` This attribute contains the index of the last ticket that was stored in the ticketstore.
- `max_tickets` Contains the number of tickets that can be simultaneously stored on the card.
- `nxt_cmd` This attribute stores the command that is expected next.
- `nxt_transfer_cmd` This attribute contains the next command in a ticket transfer. It is not cleared by the `select()`-method. The value of this attribute is used to restore the correct value of `nxt_cmd` if a call of `select()` happens while a transfer is under way.
- `pin` Contains the PIN of the card owner.
- `public_card_key` This attribute is used to store the public key of the card.
- `public_card_key_sig` This attribute is used to store the signature of the public key of the card.
- `public_cond_key` This attribute is used to store the public key of the conductor.
- `public_issuer_key` This attribute is used to store the public key of the ticket issuer.

- `secret_card_key` Contains the secret key of the card.
- `source_card_challenge` Contains the challenge generated by the source card of a ticket transfer.
- `source_card_key` This attribute is used to store the public key of the source card of a ticket transfer.
- `terminal_challenge` Contains the challenge created by the terminal.
- `the_tickets` Contains the ticketstore object.
- `the_transferred_ticket` This attribute is used to store the currently transferred ticket on the destination card.
- `ticket_no` Contains the number of the ticket that is currently returned using the *Give Back Ticket*-protocol.
- `transfer_in_progress` Boolean flag to distinguish whether the card is involved in an unfinished transfer or not.
- `user_authenticated` Boolean flag to distinguish whether the card owner was successfully authenticated or not.

3.3 The Ticketstore

The ticketstore is part of the electronic ticketing cardlet. It is responsible for the storage of the ticket data.

3.3.1 Functions

- `addCheck(int ticket_no, check check)` Adds `check` as a new check to the ticket with number `ticket_no`.
- `changeBlockStatus(int ticket_no, int status)` Sets the block status of the ticket with number `ticket_no` to `status`.
- `changeLockStatus(int ticket_no, boolean status)` Sets the lock status of the ticket with number `ticket_no` to `status`.
- `deleteTicket(int ticket_no)` Deletes the ticket indicated by `ticket_no`.
- `deleteTransferredTicket()` Deletes the ticket that is currently in transfer.
- `exTransTicket()` Returns true if the ticketstore contains a ticket that is in transfer and otherwise false.
- `getChecks(int ticket_no)` Returns the checks currently attached to the ticket with number `ticket_no`.
- `getStatus(int ticket_no)` Returns the status of the ticket with number `ticket_no`.
- `getTicket(int ticket_no)` Returns the ticket with number `ticket_no`.

- `getTransferredTicket()` Returns the ticket that is currently marked is in transfer.
- `isLocked(int ticket_no)` Returns true if the ticket with number `ticket_no` is locker and false otherwise.
- `isChecked(int ticket_no)` Returns true if the ticket with number `ticket_no` is oblitterated and false otherwise.
- `number()` Returns the number of tickets currently stored in the ticketstore.
- `storeTicket(ticket ticket)` Stores the ticket `ticket` and returns the position of the new ticket. The ticket is marked as blocked after it is stored in the ticketstore. It must be activated by a subsequent execution of `changeBlockStatus()`.
- `storeTransferred(ticket ticket)` Stores `ticket` and marks it as in transfer.
- `transferredTicketNum()` Returns the number of the ticket currently in transfer.

Chapter 4

The protocols

4.1 Overview

The protocols are all specified as UML[RJB98, OMG99] activity diagrams. Each participant of a protocol is represented by a swimlane. Most often there are two participants, one terminal and one smartcard. Only the protocols that deal with the transfer of tickets have three participants, one terminal and two cards. The activity diagram shows the control flow and the communication of its protocol. The data exchange is represented by `command`- and `response`-objects. The `command`-objects are generated by the terminal and sent to the card, while `response`-objects are generated by the smartcard. We link a note to each object that contains information about the composition of a `command` or `response`. The activities are used to describe changes of the internal state of the terminal and the card. We use branches to express checks of preconditions for protocol steps.

The protocols all have a similar control flow. A protocol run is always initiated by the terminal. After being triggered off by the user, the terminal calls a function of the cardlet. Thereby the terminal can transmit data to the cardlet if necessary. The cardlet performs the requested function and returns data if required. It is not possible for the cardlet to start a protocol run of its own. Wrong data or unsatisfied security preconditions lead to an exception and the termination of the protocol run.

4.2 Add Check

4.2.1 Purpose

This protocol is used to add a check to a ticket on the card. The protocol is used by the conductor's portable card reader to obliterate the ticket for the current trip.

4.2.2 Preconditions

The card must be authenticated, this is done using the *Authenticate Cardlet*-protocol (Section 4.3), and the conductor terminal must have been authenticated using the *Authenticate Conductor*-protocol (Section 4.4). The ticket number which is transferred to the card must be a correct ticketnumber, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.2.3 The protocol

In the first step of the protocol the server tests if there is an internal error. A protocol run is only started if there's no error in the server. If there's no error the server checks of the cardlet was previously authenticated. If this is not the case the execution of the protocol is aborted. Otherwise the server stores the command that will be sent to the card (`last_cmd`) and sends a command to the card consisting of a card instruction (`CMD_addCheck`), the number of the ticket to which the check is to be added (`ticket_no`) and the check (`check`). The first thing the cardlet does after receiving this command is to check if there's curently another protocol running. This is done by checking if `nxt_cmd` is not equal `any`. If this condition isn't satisfied the cardlet generates a `SW_CONDITIONS_NOT_SATISFIED`-exception. If the test is successful the cardlet checks if the Conductor Terminal is properly authenticated. If the terminal isn't authenticated the cardlet produces a `SW_SECURITY_STATUS_NOT_SATISFIED`-exception. Otherwise the cardlet tests if the next precondition is satisfied. This precondition ensures that the ticket number sent by the terminal (`ticket_no`) refer to a ticket that actually exists. If the number is inappropriate the cardlet produces a `SW_RECORD_NOT_FOUND`-exception. If there's a matching ticket, the cardlet stores the new check and informs the terminal of the successful completion of the operation by transmitting `SW_OK`. After receiving the expected response the terminal resets the `last_cmd`-field.

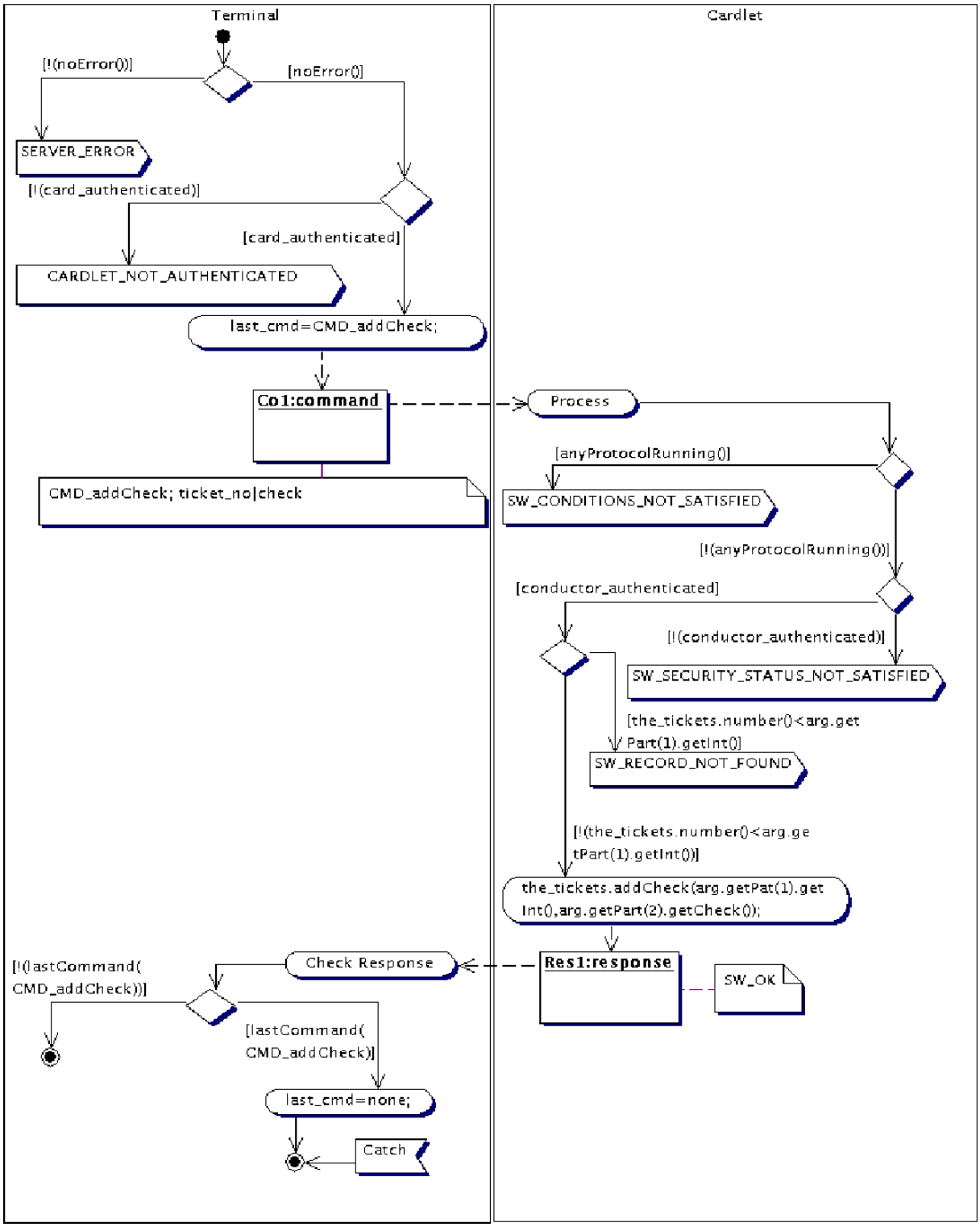


Figure 4.1: *Add Check*-protocol

4.3 Authenticate Cardlet

4.3.1 Purpose

This protocol is used to check the authenticity of a cardlet. Using this protocol a terminal can check if a cardlet is genuine or a fake. This protocol is used by the conductor terminal to check the authenticity of a cardlet to be sure that the tickets on the cardlet are to be considered as authentic.

4.3.2 Preconditions

Before this protocol can be executed the terminal must have acquired and verified the public key of the card. This is done using the *Get Card Key*-protocol (Section 4.9). There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any` and the user must be authenticated, i. e. the correct user PIN must have been entered using the *Authenticate User*-protocol (Section 4.5).

4.3.3 The protocol

As in all other protocols the first step in the *Authenticate Cardlet*-protocol is to test, if the terminal isn't in an error condition. If there's no error the terminal can advance to the next step, to check that the card key and the card key signature were retrieved from the card and that the signature is a valid certificate for the key. The protocol run will only continue if the signature is ok. If these tests were successfully passed, the terminal generates a challenge (a random value of fixed size), stores the challenge in the `terminal_challenge`-field and stores the card instruction that will be sent to the card in `last_cmd`. The next step the terminal performs is to send a command to the cardlet. The command is made up of the card instruction (`CMD_authenticateCard`) and the challenge generated before. After receiving this command the card verifies that all the preconditions of this protocol are fulfilled. First there mustn't be another protocol running, this is checked by testing if `nxt_cmd` is equal to `any` and secondly the user or the conductor must have been authorized. If a precondition isn't satisfied the cardlet generates an appropriate exception. If the test are successful the cardlet stores the received challenge in the field `terminal_challenge` and sends a response to the terminals consisting of `SW_OK` and the signature of the challenge. After receiving the response from the card, the terminal tries to verify the signature. If the signature is correct the terminal sets the field `card_authenticated` to `true` and `last_cmd` to `none`.

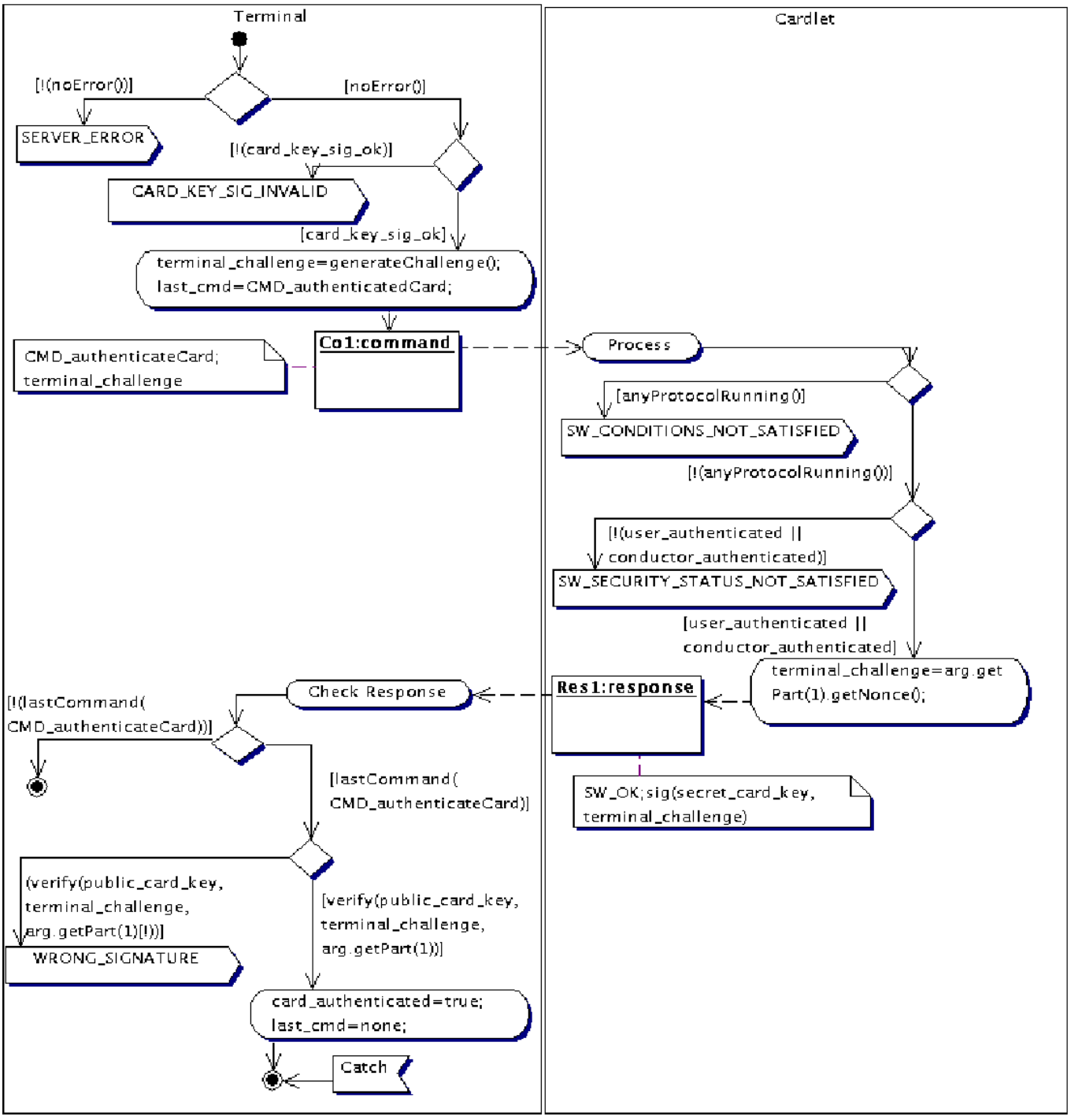


Figure 4.2: *Authenticate Cardlet*-protocol

4.4 Authenticate Conductor

4.4.1 Purpose

This protocol is used to authenticate the conductor terminal. After a successful protocol run the smartcard can be sure to communicate with a genuine conductor terminal.

4.4.2 Preconditions

There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.4.3 The protocol

As usual the protocol starts with the terminal checking if it is ready. If there's no problem, the terminal stores the card instruction that will be sent to the cardlet. Then the command, consisting of the card instruction `CMD_loadCondKey` and the public key of the conductor terminal, is sent to the card. The first action to be performed by the card after receiving the command is to check if there's already a protocol running. If this is the case, an exception is generated. Otherwise the cardlet stores the received key in the field `public_cond_key` and the next card instruction (`CMD_checkCondKeySig`) that must be sent by the terminal in `nxt_cmd`. Thereafter the cardlet send its answer to terminal. The answer consists only of `SW_OK`. After receiving the response form the card, the terminal stores the card instruction that will be sent to the card next (`CMD_checkCondKeySig`) and sends the next command, consisting of the card instruction and the signature of the public key of the conductor (`public_cond_key_sig`), to the card. After receiving this command, the card checks of the card instruction sent by the card is the correct card instruction for the current step in the protocol (`thisProtocolRunning(CMD_checkCondKeySig)`). If the correct card instruction was sent, the cardlet verifies the signature of the conductor key. If the signature is not correct the cardlet creates an `SW_SECURITY_STATUS_NOT_SATISFIED`-exception, otherwise the cardlet generates a new challenge and stores the next expected card instruction. Then a response is sent to the terminal containing the new challenge. The terminal stores the received challenge and the card instruction that will be sent to the card next and sends a new command to the card. This command consists of the card instruction (`CMD_authenticateConductor`) and the signature of the card challenge. The cardlet first verifies that it is in the correct protocol step and, if this is the case, verifies that the signature of the challenge is correct. If the signature could be verified, the cardlet sets the attribute `conductor_authenticated` to `true` and `next_cmd` to `any`. After receiving the answer, the terminal sets `last_cmd` to `none`.

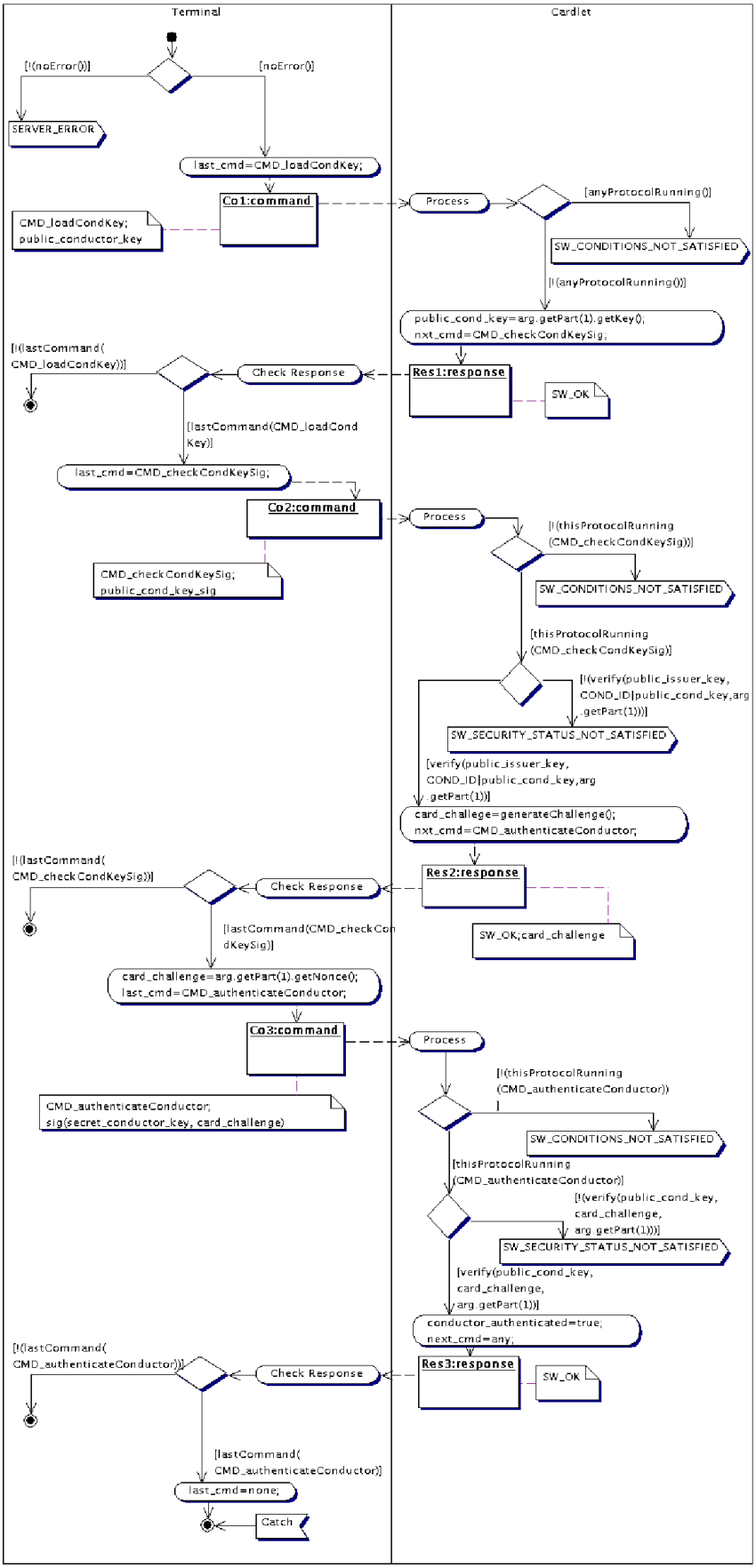


Figure 4.3: *Authenticate Conductor*-protocol

4.5 Authenticate User

4.5.1 Purpose

This protocol is used to verify the authenticity of the card holder. First the card holder enters his PIN. Then the inputed value is transmitted to the card which compares it with the expected value.

4.5.2 Preconditions

There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.5.3 The protocol

First the terminal checks for internal errors. If the terminal is ready the user is prompted for the PIN. The PIN entered by the user is stored in the attribute pin. The `last_cmd`-attribute is set to `CMD_verifyPIN`. The a command is sent to the cardlet consisting of the card instruction `CMD_verifyPIN` and the PIN entered by the user. After receiving this command the cardlet tests if there isn't a protocol running currently. If there's no other protocol, the cardlet verifies if the transmitted PIN is correct. If this is the case, the attribute `user_authenticated` is set to `true`. `SW_OK` is sent as answer to the terminal. If one of the precondition wasn't satisfied, an appropriate exception is thrown. After receiving the answer, the terminal sets `last_cmd` to `none`.

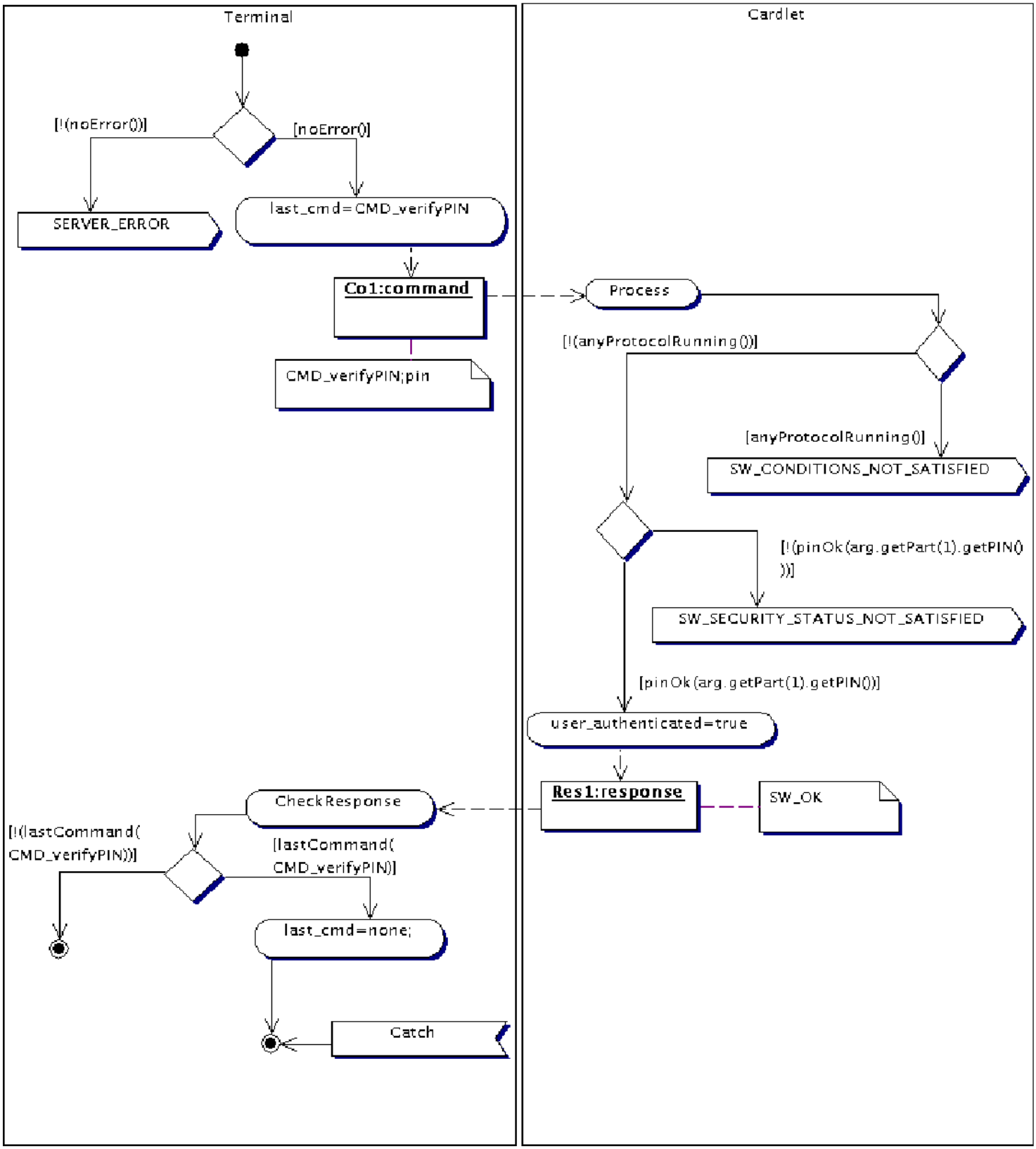


Figure 4.4: *Authenticate User*-protocol

4.6 Change Lock Status

4.6.1 Purpose

This protocol is used to change the lock status of a ticket on a card. Tickets can either be locked or unlocked. The data of locked tickets is only revealed if the user is authenticated, this means that locked tickets aren't visible for the conductor.

4.6.2 Preconditions

The owner of the card must have been successfully authenticated using the *Authenticate User*-protocol (Section 4.5). The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.6.3 The protocol

After checking for internal errors, the terminal asks the user to select the ticket whose lock status is to be changed and the new lock status. The data entered by the user is stored in the attributes `ticket_no` and `lock_status`. The attribute `last_cmd` is set to `CMD_changeLockStatus`. Then the terminal send a command to the cardlet consisting of the card instruction `CMD_changeLockStatus` and `ticket_no` and `lock_status` as additional data. After receiving this command, the cardlet verifies that no other protocol is currently running. If this is the case the cardlet checks if the user was authenticated previously and afterwards it is tested if the first argument refers to an existing ticket. If all checks are successful the lock state of the ticket identified by the first argument is set to the value of the second argument. Terminating the activity of the card `SW_OK` is returned if no errors occurred. After receiving the answer, the terminal sets `last_cmd` to `none`.

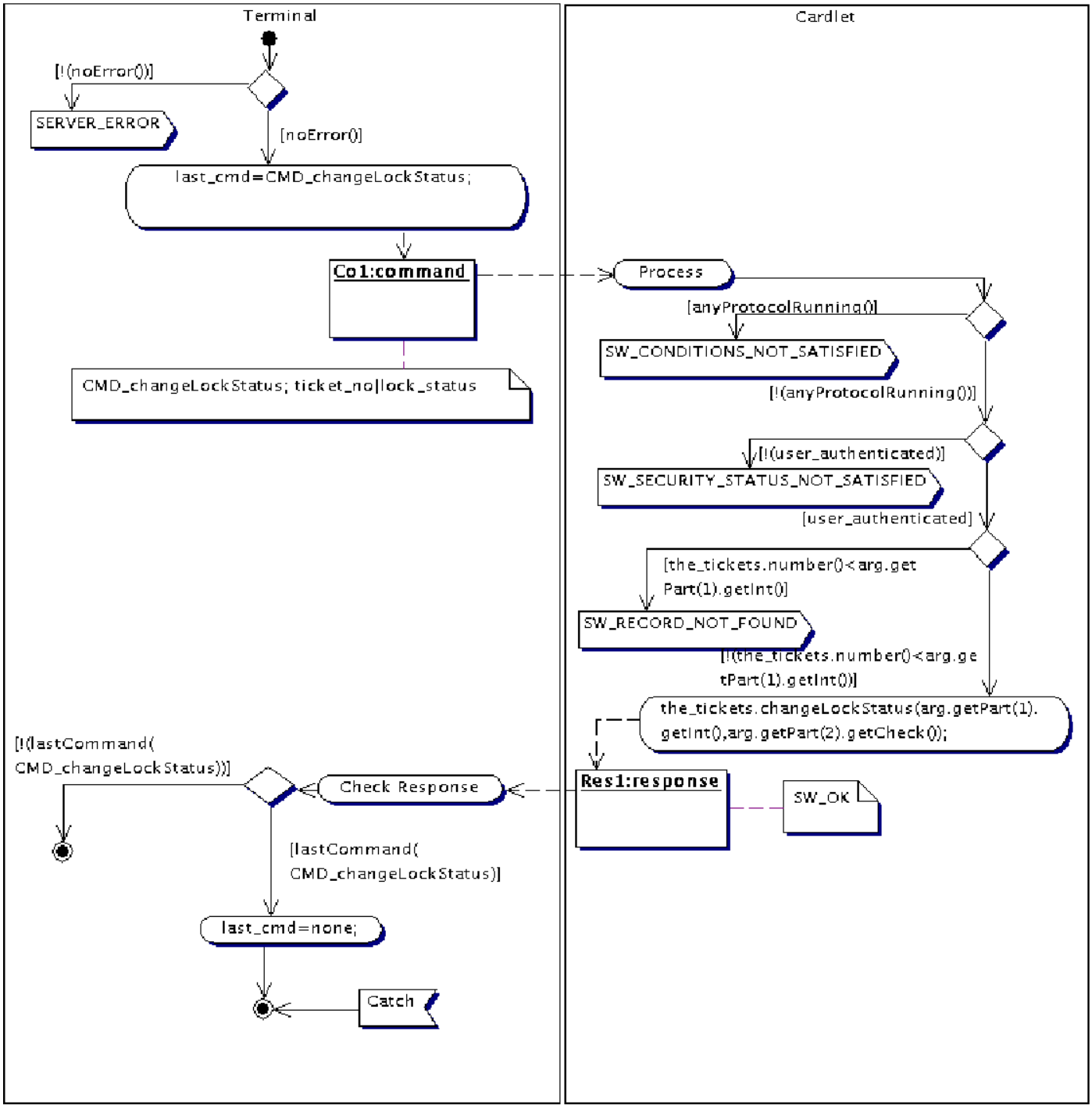


Figure 4.5: *Change Lock Status*-protocol

4.7 Continue Transfer

4.7.1 Purpose

If the transfer of a ticket is performed on a computer with only one card reader, the user must switch between the source card and the destination card at certain points of the protocol. Since removal and later reinsertion of a smartcard into a card reader leads to a reset of the card, which leads to the deletion of the value stored in `nxt_cmd`, the correct value of the field must be restored before the transfer of the ticket can be continued. This is done by the *Continue Transfer*-protocol by copying the value of the `nxt_transfer_cmd`-field into `nxt_cmd`.

4.7.2 Preconditions

This function can be used only if there's an incomplete transfer on the card, i. e. the `transfer_in_progress`-field of the cardlet must be true. Additionally, there mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.7.3 The protocol

If the terminal is ready, `last_cmd` is set to `CMD_continueTransfer` and a command consisting of just this card instruction is sent to the cardlet. After receiving this command the cardlet verifies if there's not another protocol running and if the card actually participated in a ticket transfer that wasn't completed successfully. If all preconditions are satisfied, the attribute `nxt_cmd` is set to the value of `nxt_transfer_cmd` and `SW_OK` is returned to the terminal. After receiving the answer, the terminal sets `last_cmd` to `none`.

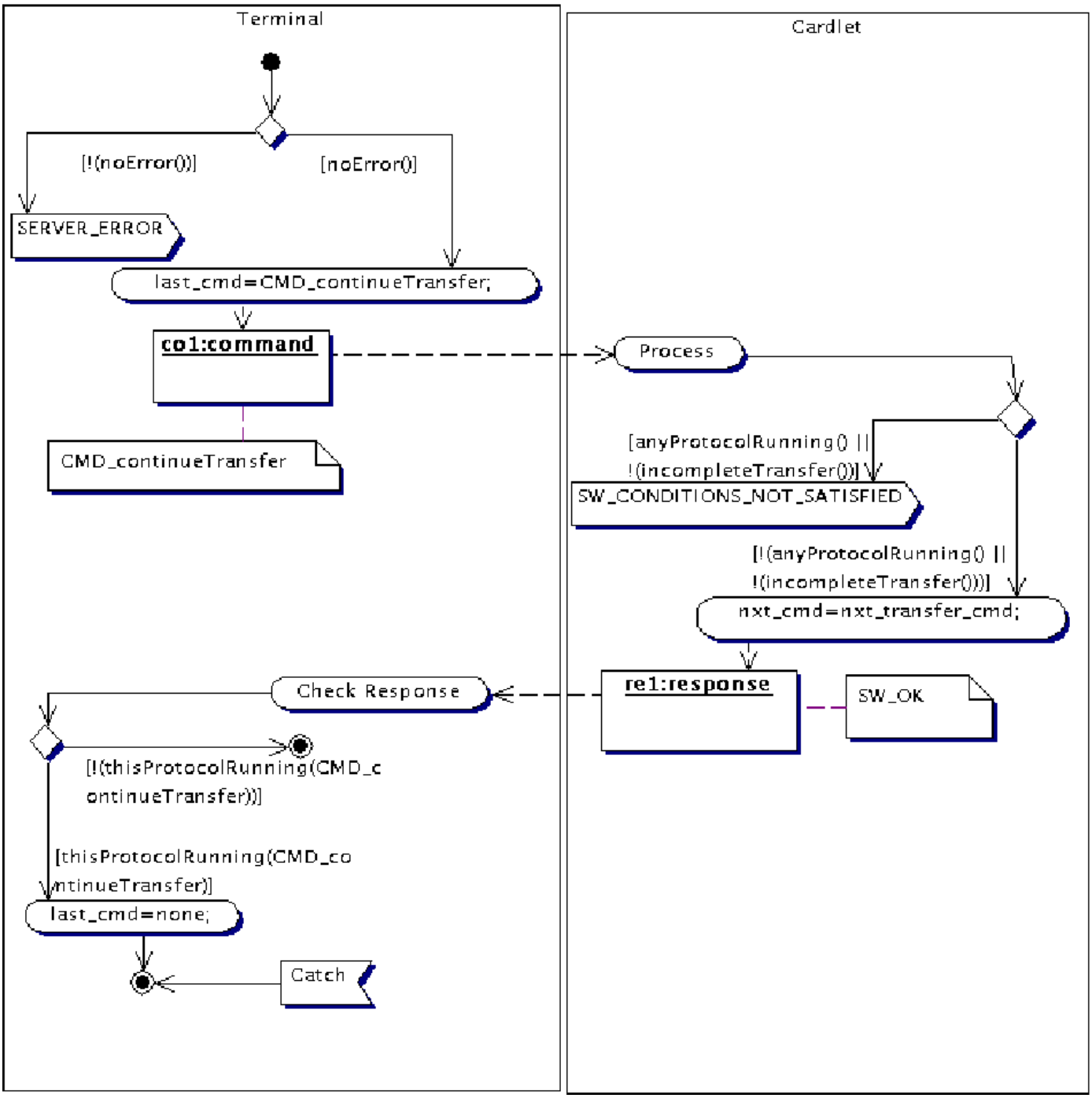


Figure 4.6: *Continue Transfer*-protocol

4.8 Delete Ticket

4.8.1 Purpose

This protocol is used to delete a ticket from a card.

4.8.2 Preconditions

The owner of the card must have been successfully authenticated using the *Authenticate User*-protocol (Section 4.5). The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.8.3 The protocol

First the server ensures that it isn't in an erroneous condition. If the terminal is ready the user can select the ticket that he wants to delete. The user input is stored in `ticket_no`. The attribute `last_cmd` is set to `CMD_deleteTicket` and a command is sent to the card. The command is made up of the card instruction `CMD_deleteTicket` and the ticket number. After receiving the command the cardlet checks that no other protocol is running. If so the cardlet verifies that the user is authenticated and that the data refers to an existing ticket. If the preconditions are satisfied the ticket, identified by the parameter, is deleted ans `SW_OK` is returned to the terminal. After receiving the answer, the terminal sets `last_cmd` to `none`.

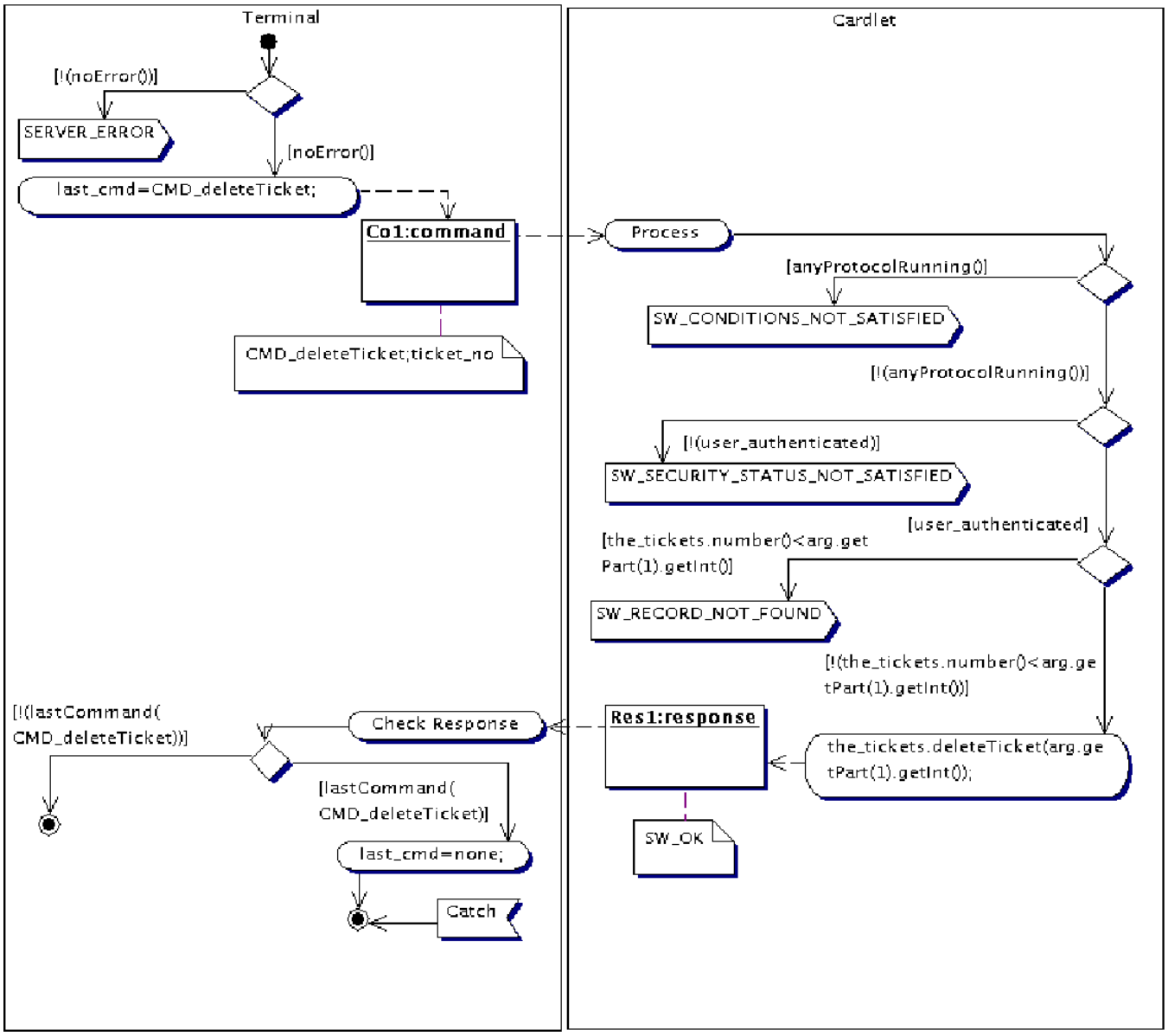


Figure 4.7: *Delete Ticket*-protocol

4.9 Get Card Key

4.9.1 Purpose

This protocol is used by the terminals to determine the public key of a card. After receiving the public card key and its signature from the card, the terminal verifies whether the certificate is ok or not.

4.9.2 Preconditions

There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.9.3 The protocol

After checking that it is ready, the terminal sets `last_cmd` to `CMD_getCardKey` and sends a command containing the card instruction `CMD_getCardKey` to the cardlet. On receiving the command, the cardlet checks if there's another protocol running at the moment. If this is the case an exception is thrown, otherwise `nxt_cmd` is set to `CMD_getCardKeySig`. The cardlets sends a reply to the terminal consisting of `SW_OK` as statusword and the value of `public_card_key` as data. After receiving the reply, the reply data is stored in `public_card_key` and `last_cmd` is set to `CMD_getCardKeySig`. Then the terminal sends the next command, consisting of `CMD_getCardKeySig`, to the cardlet. After receiving this command, the cardlet verifies that the expected card instruction was sent. If this is the case the cardlet sets `nxt_cmd` to `any` and sends a reply made up of the statusword `SW_OK` and `public_card_key_sig` as data, otherwise an exception is thrown. On receiving this reply the terminal tries to verify the signature of the public key of the card. Only if the signature is correct, the terminal continues by storing the received card key signature in `public_card_key_sig`, setting `card_key_sig_ok` to `true` and `last_cmd` to `none`.

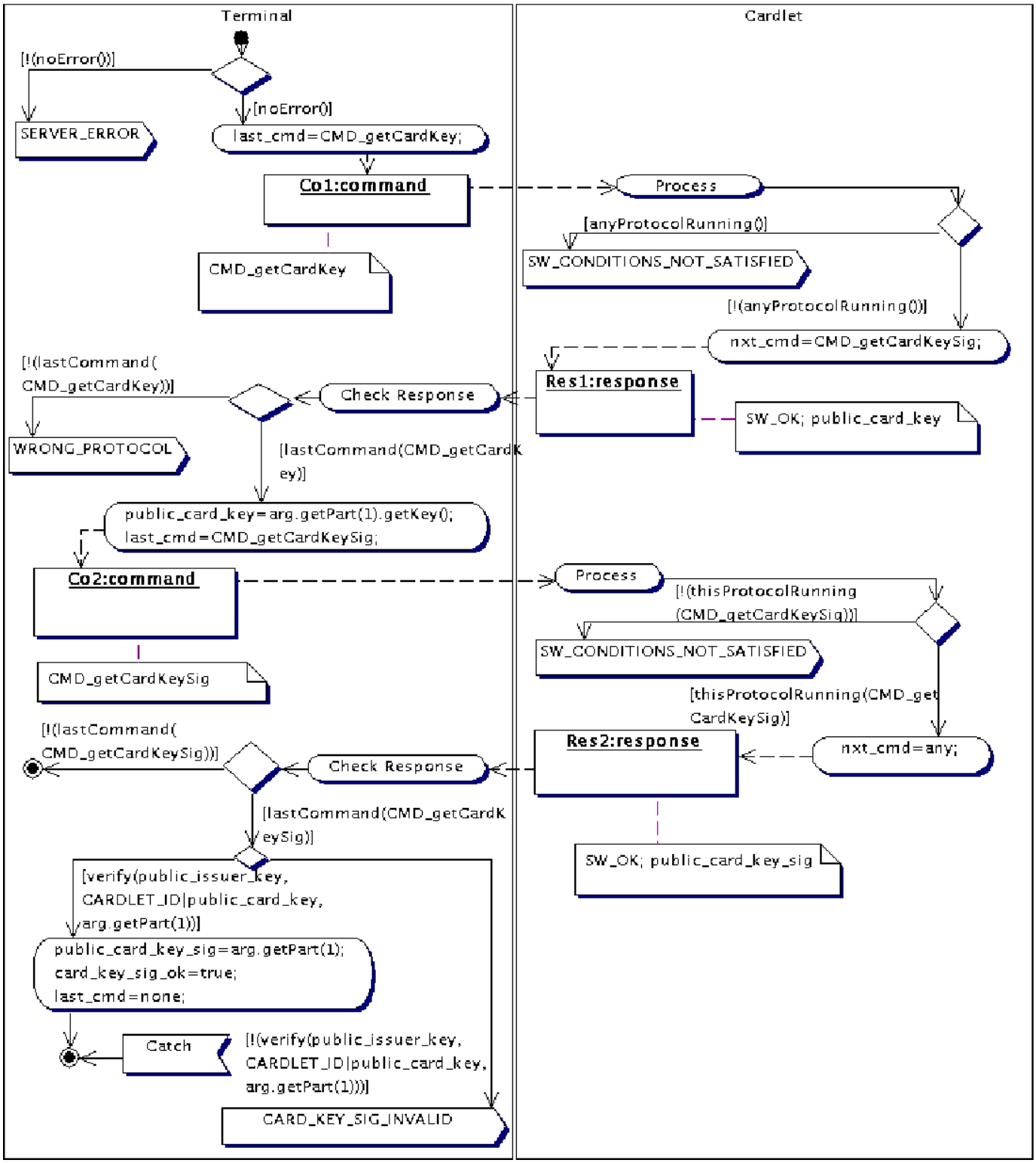


Figure 4.8: *Get Card Key*-protocol

4.10 Give Back Ticket

4.10.1 Purpose

This protocol is used to return unused tickets to the issuer.

4.10.2 Preconditions

Before this protocol can be executed the terminal must have acquired and verified the public key of the card. This is done using the *Get Card Key*-protocol (Section 4.9). The owner of the card must have been successfully authenticated using the *Authenticate User*-protocol (Section 4.5). The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. The ticket that should be returned mustn't be obliterated. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.10.3 The protocol

First the terminal ensures that it is ready to run the protocol. If this is possible the user is requested to select the ticket that should be given back. The user input is stored in `ticket_no`. Then the terminal generates a challenge, stores it in `terminal_challenge` and lastly sets `last_cmd` to `CMD_giveBackTicket`. Then a command is sent to the card consisting of the card instruction `CMD_giveBackTicket` and `terminal_challenge` and `ticket_no` as data. The cardlet receives this command and at first verifies that no other protocol is running at the moment. If this is the case, the other preconditions are checked: Is the user authenticated, does the transferred ticket number refer to an existing ticket and isn't the referenced ticket already obliterated? If one of the preconditions isn't fulfilled an appropriate exception is thrown. Otherwise the cardlets continues with the execution of the protocol. It sets the block status of the ticket, that is identified by the data in the command, to `STATUS_GIVE_BACK`, it sets `nxt_cmd` to `CMD_getDataSig`, `ticket_no` to the value of the second part of the command data and `terminal_challenge` to the value of the first part of the parameters. The the cardlet sends a reply to the terminal containing the status word `SW_OK` and the data of the ticket in question. The ticket data is encrypted for the transmission to the server. After receiving this reply, the server decrypts the ticket data and stores it in the attribute `ticket`. Furthermore `last_cmd` is set to `CMD_getDataSig`. Then the terminal sends another command to the cardlet containing only the card instruction `CMD_getDataSig`. After receiving the command, the cardlets verifies that the contained card instruction is the one expected. If this is the case the cardlet generates a challenge, stores it in `card_challenge` and sets `nxt_cmd` to `CMD_eraseTicket`, otherwise an exception is thrown. The cardlet sends a reply that is made up of the status word `SW_OK` and a data part containing a digital signature of the ticket and the challenge sent by the terminal as first part and `card_challenge` as second part. After receiving this reply the terminal has to verify that the signature is correct and the ticket can be given back. If so the ticket is accepted and the customer is refunded. The data base system ensures that this ticket can't be given back again. The terminal stores the second part of the reply data in `card_challenge` and sets `last_cmd` to `CMD_eraseTicket`. The the terminal sends a command to the cardlet containing the card instruction `CMD_eraseTicket` and a digital signature of the ticket number combined with the challenge generated by the card. On receiving this command the cardlet tests if the expected card instruction is sent. If

this is the case, the signature is verified. If the signature is correct the ticket in question is deleted and `nxt_cmd` is set to `none`.

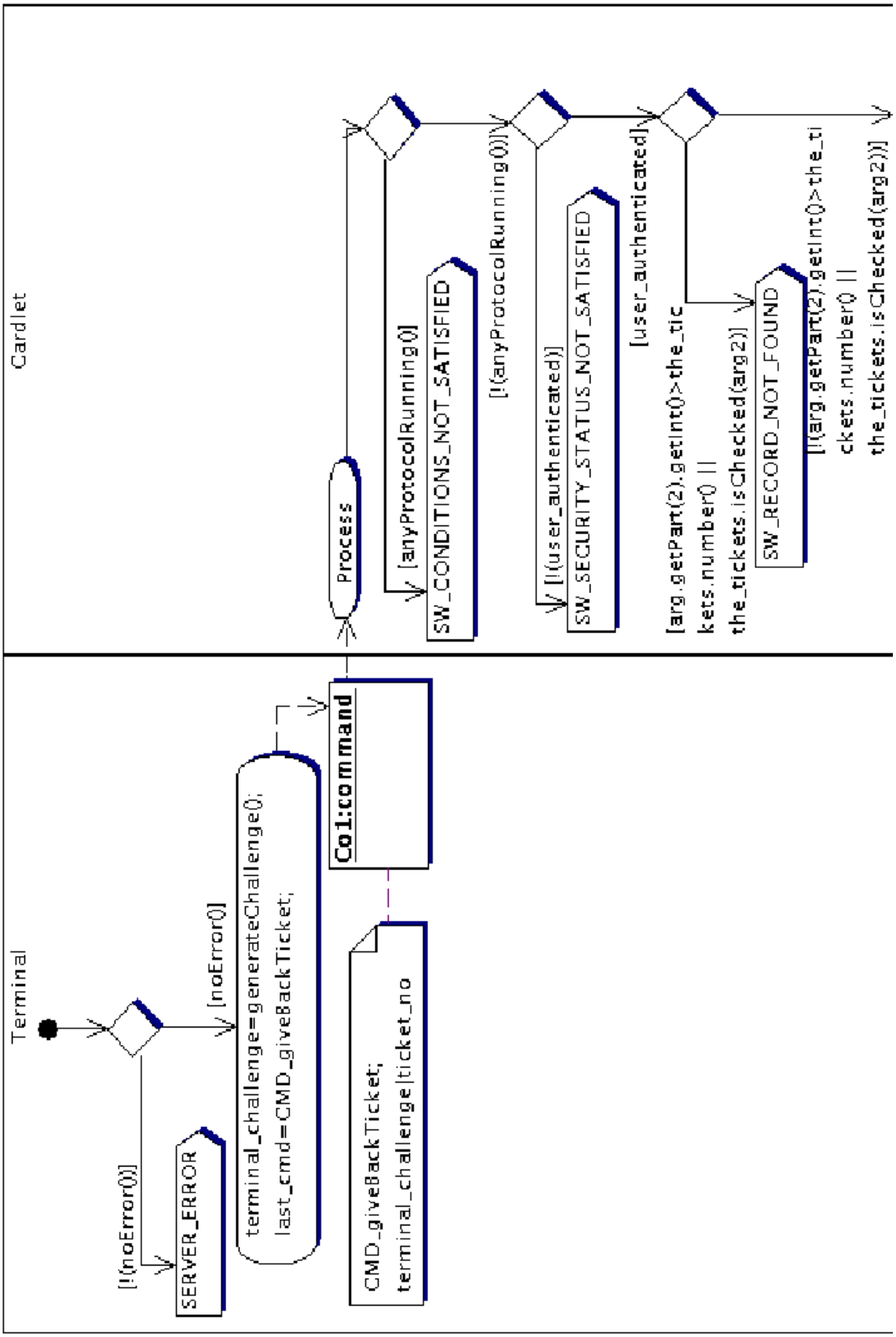


Figure 4.9: *Give Back Ticket*-protocol Part I

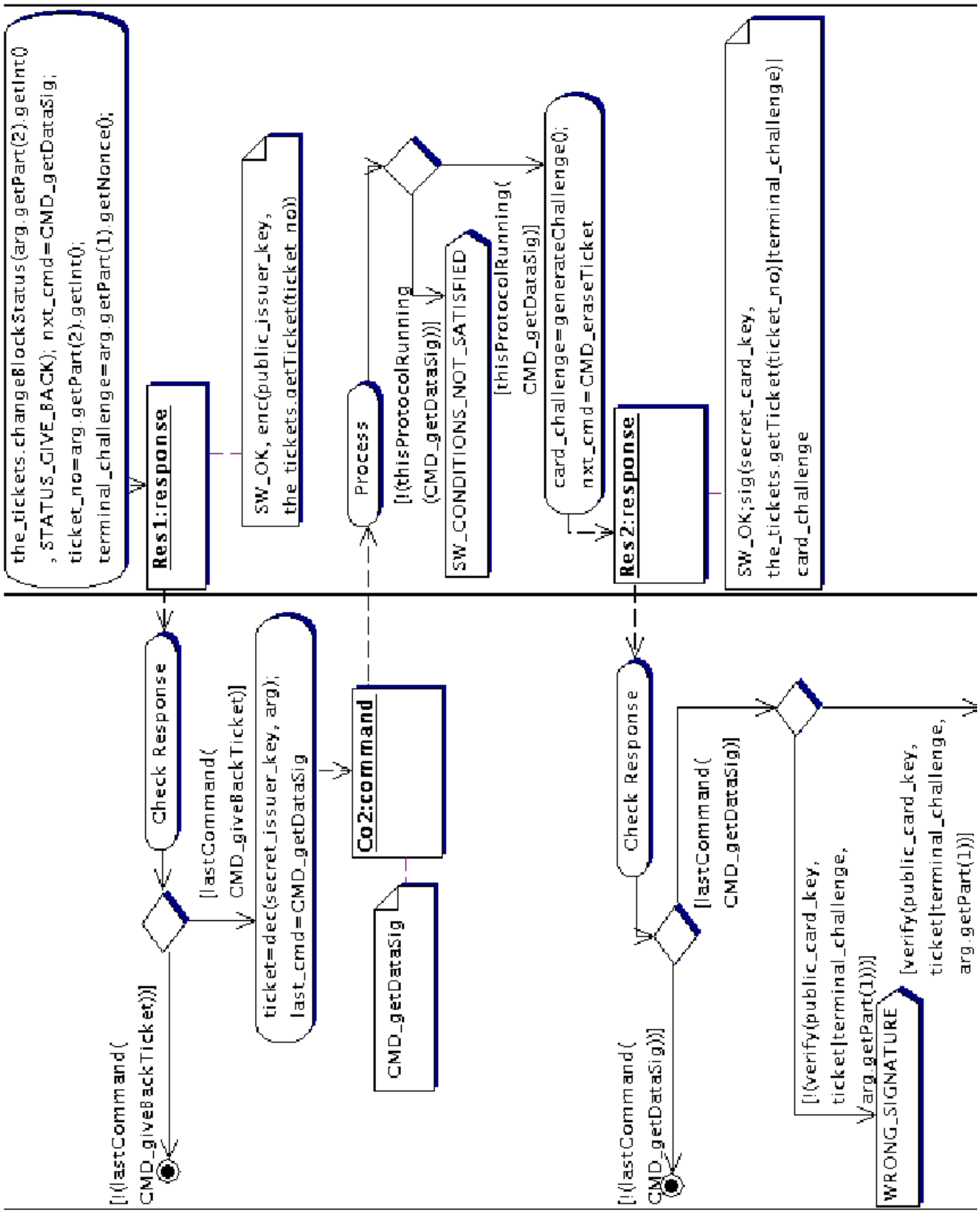


Figure 4.10: *Give Back Ticket*-protocol Part II

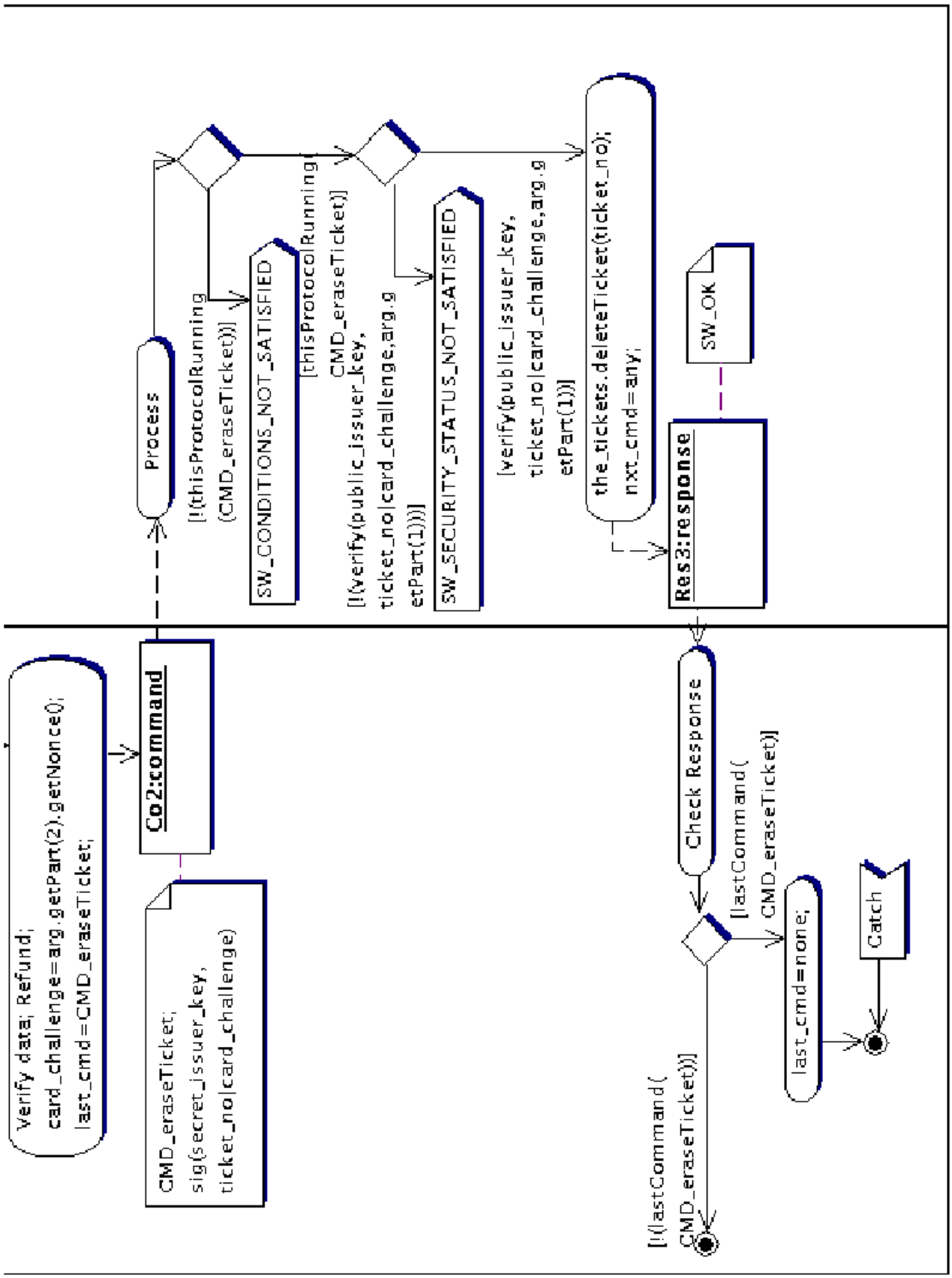


Figure 4.11: *Give Back Ticket*-protocol Part III

4.11 Load Ticket

4.11.1 Purpose

This protocol is used to load a new ticket on a card.

4.11.2 Preconditions

Before this protocol can be executed the terminal must have acquired and verified the public key of the card. This is done using the *Get Card Key*-protocol (Section 4.9). It must be possible to load another ticket on the card, i. e. the number of tickets on the card must be less than `max_tickets`. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.11.3 The protocol

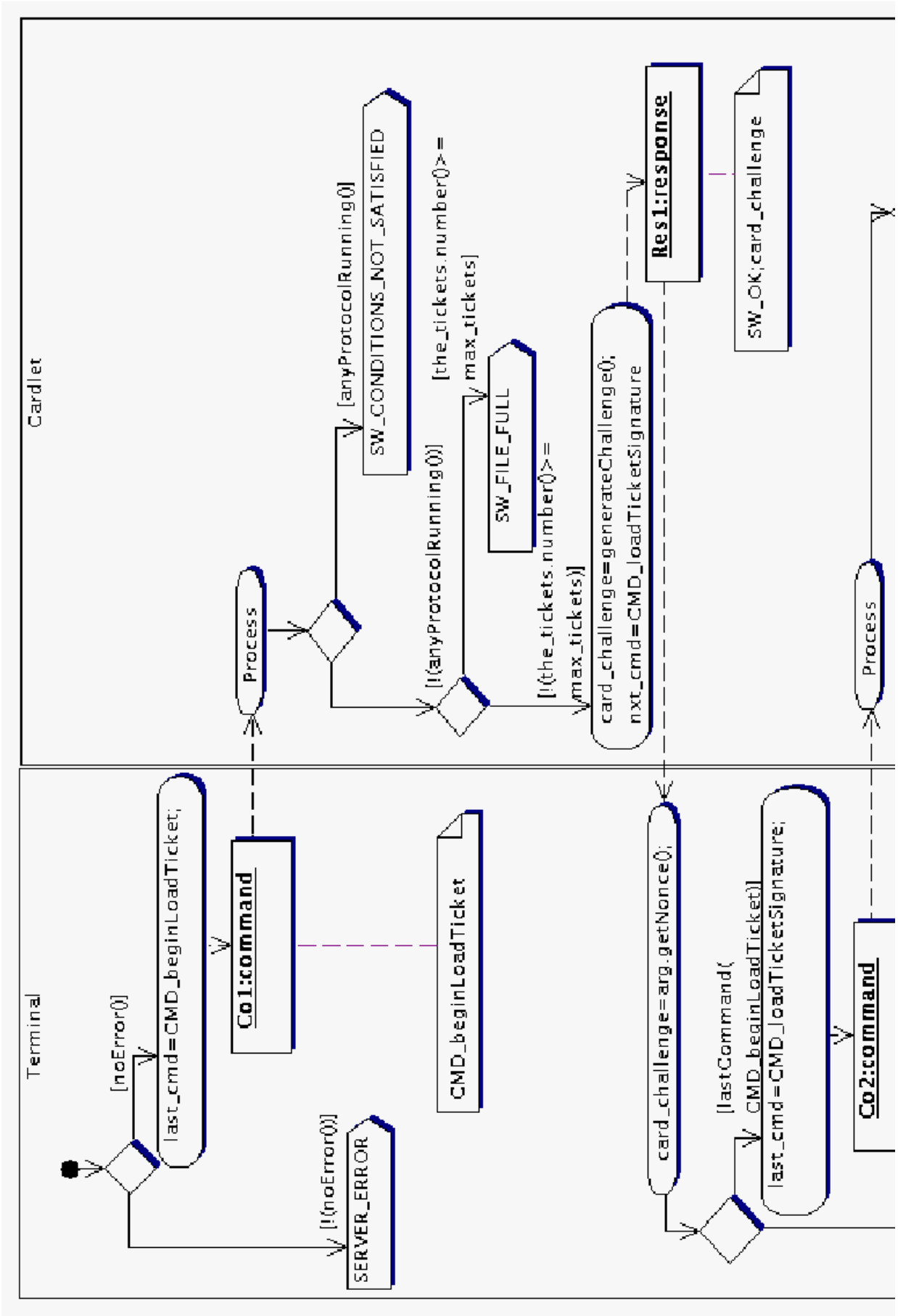


Figure 4.12: *Load Ticket*-protocol Part I

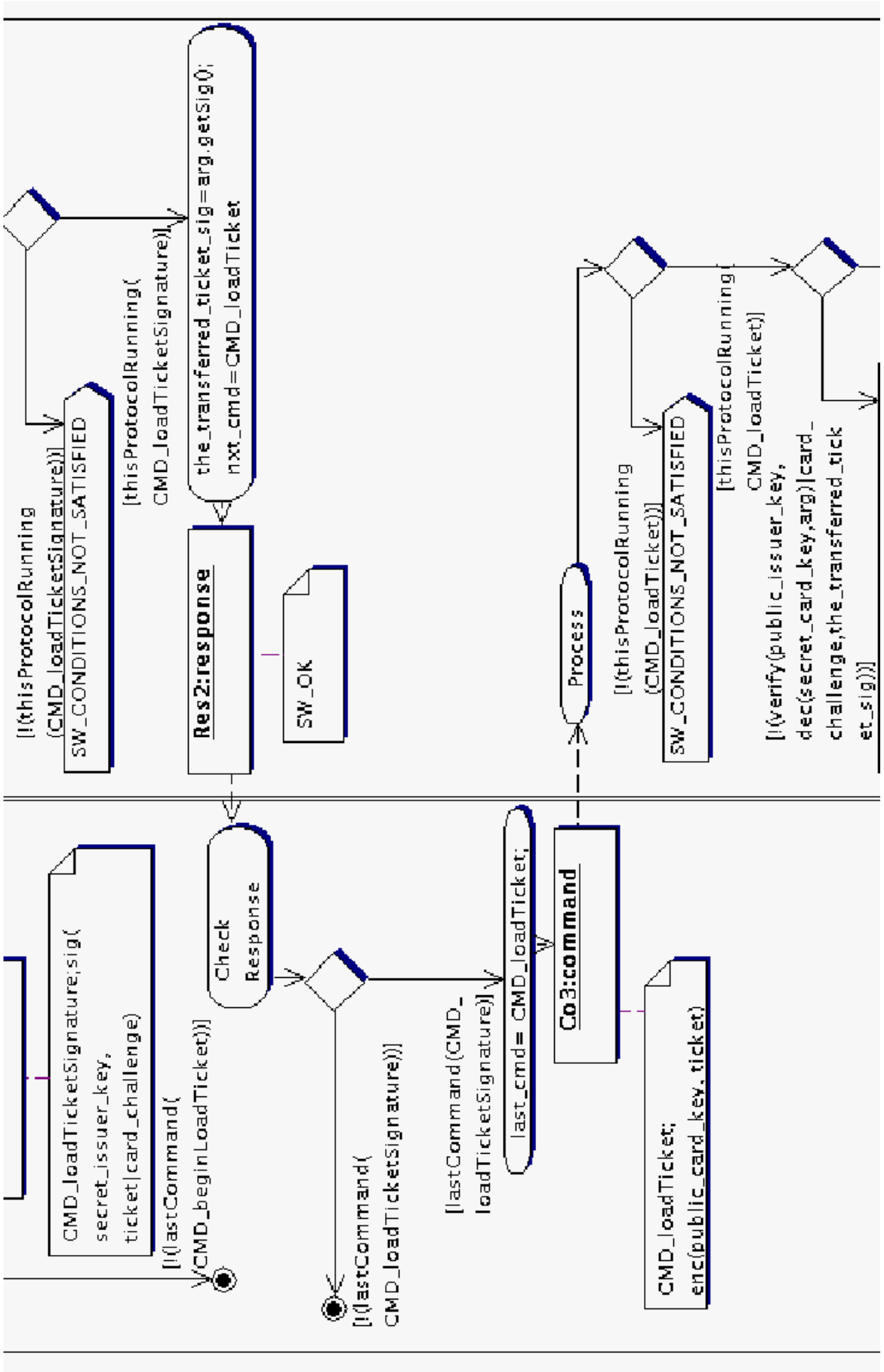


Figure 4.13: *Load Ticket*-protocol Part II

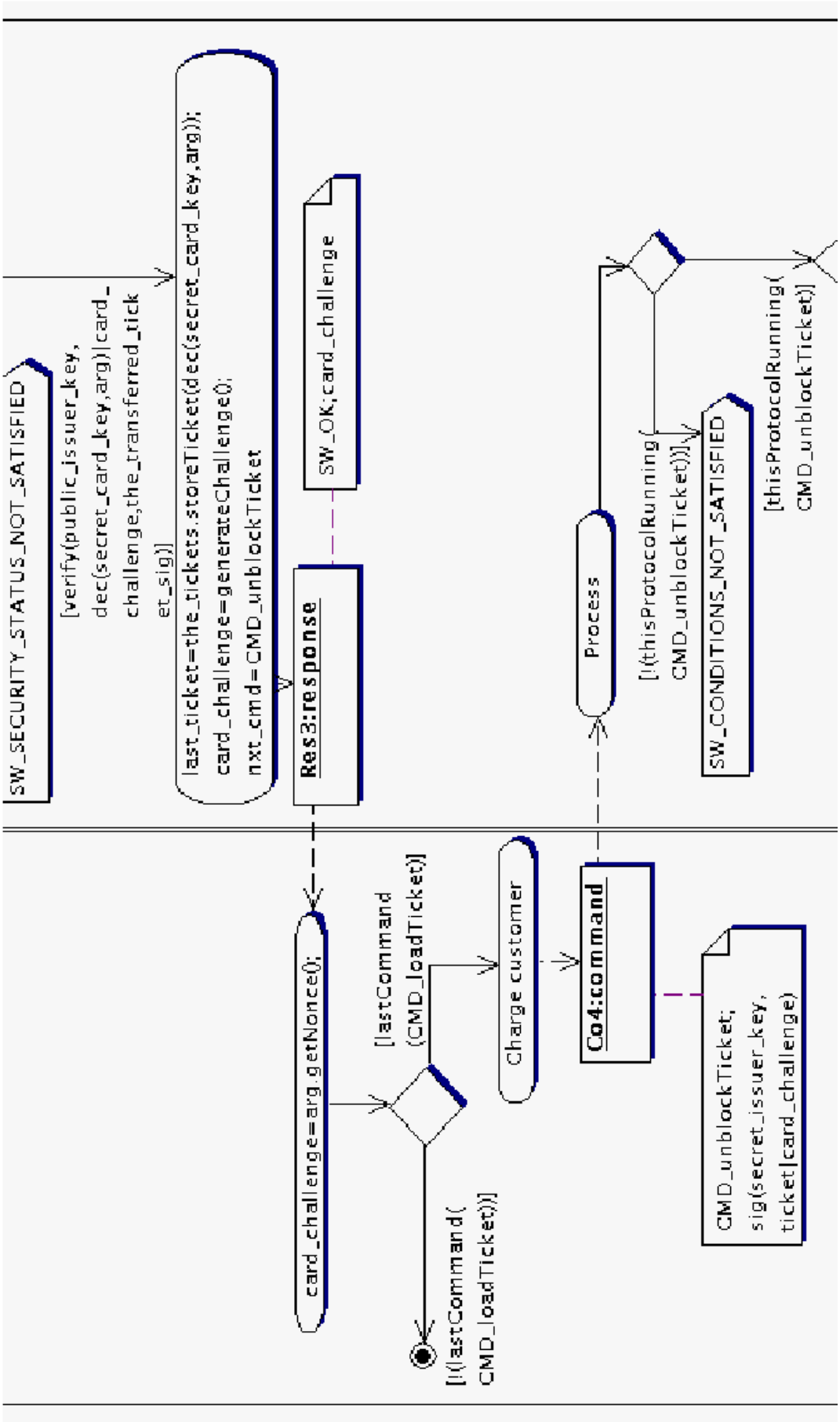


Figure 4.14: *Load Ticket*-protocol Part III

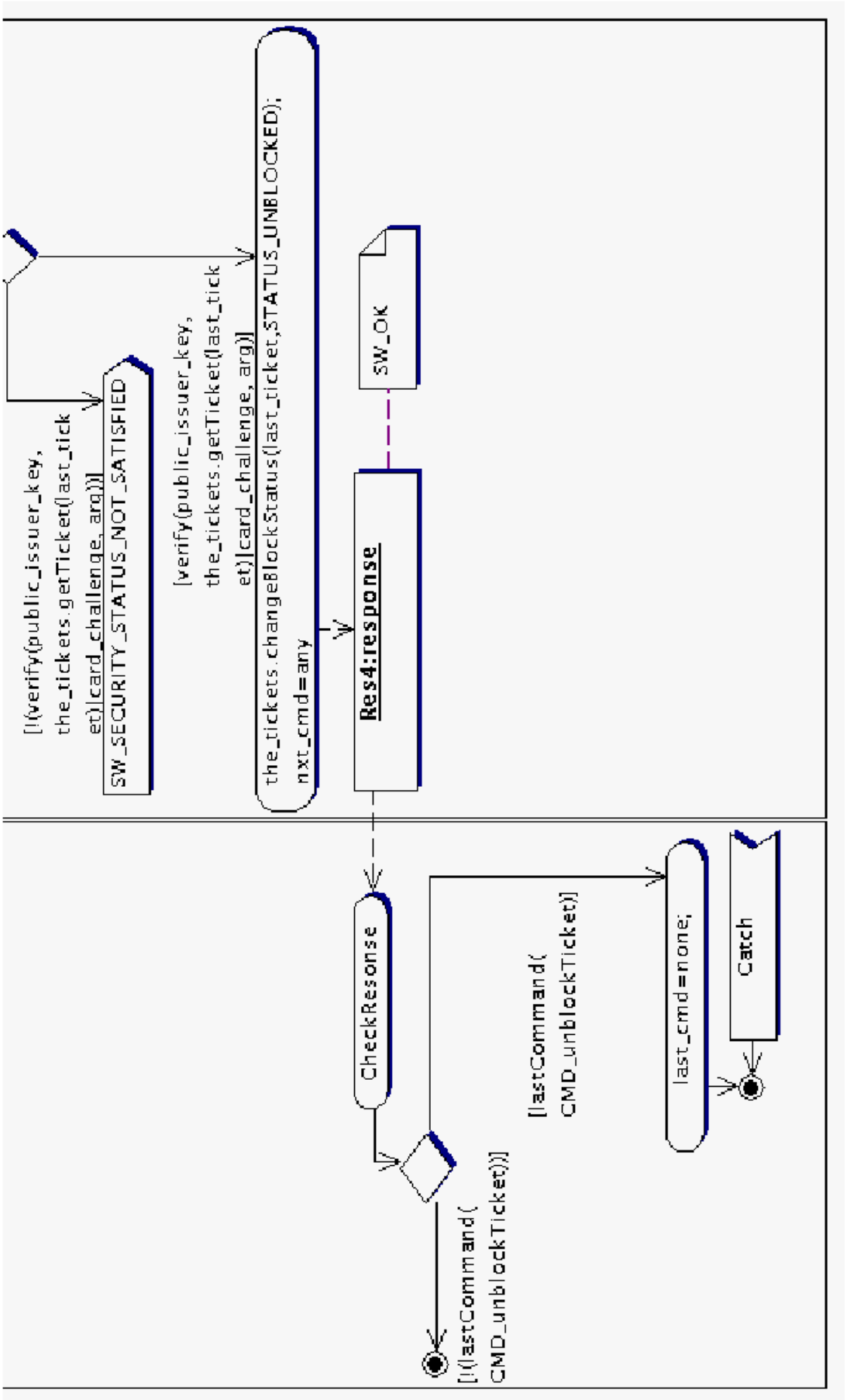


Figure 4.15: *Load Ticket*-protocol Part IV

4.12 Number Of Tickets

4.12.1 Purpose

This protocol is used by the terminals to determine the number of tickets currently stored on a card.

4.12.2 Preconditions

The conductor or the user must have been authenticated using the *Authenticate Conductor*-protocol (Section 4.4) or the *Authenticate User*-protocol (Section 4.5) respectively. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.12.3 The protocol

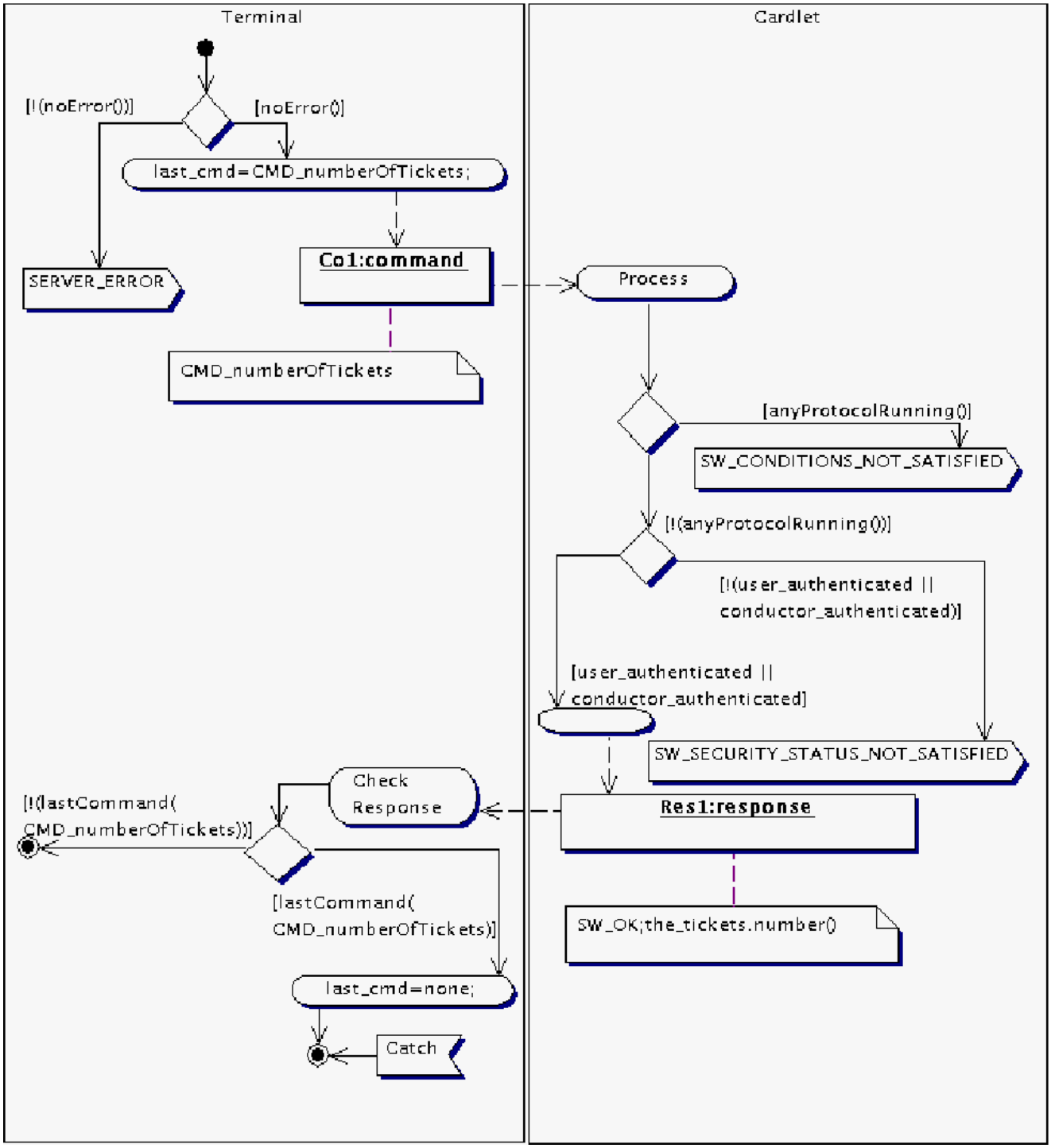


Figure 4.16: *Number Of Tickets*-protocol

4.13 Recover Load

4.13.1 Purpose

This protocol is user to finalize a load operation that wasn't complete properly. Incomplete loads can happen for instance due to communication problems with the server of the ticket issuer.

4.13.2 Preconditions

There must be a ticket on the card whose load wasn't completed properly. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.13.3 The protocol

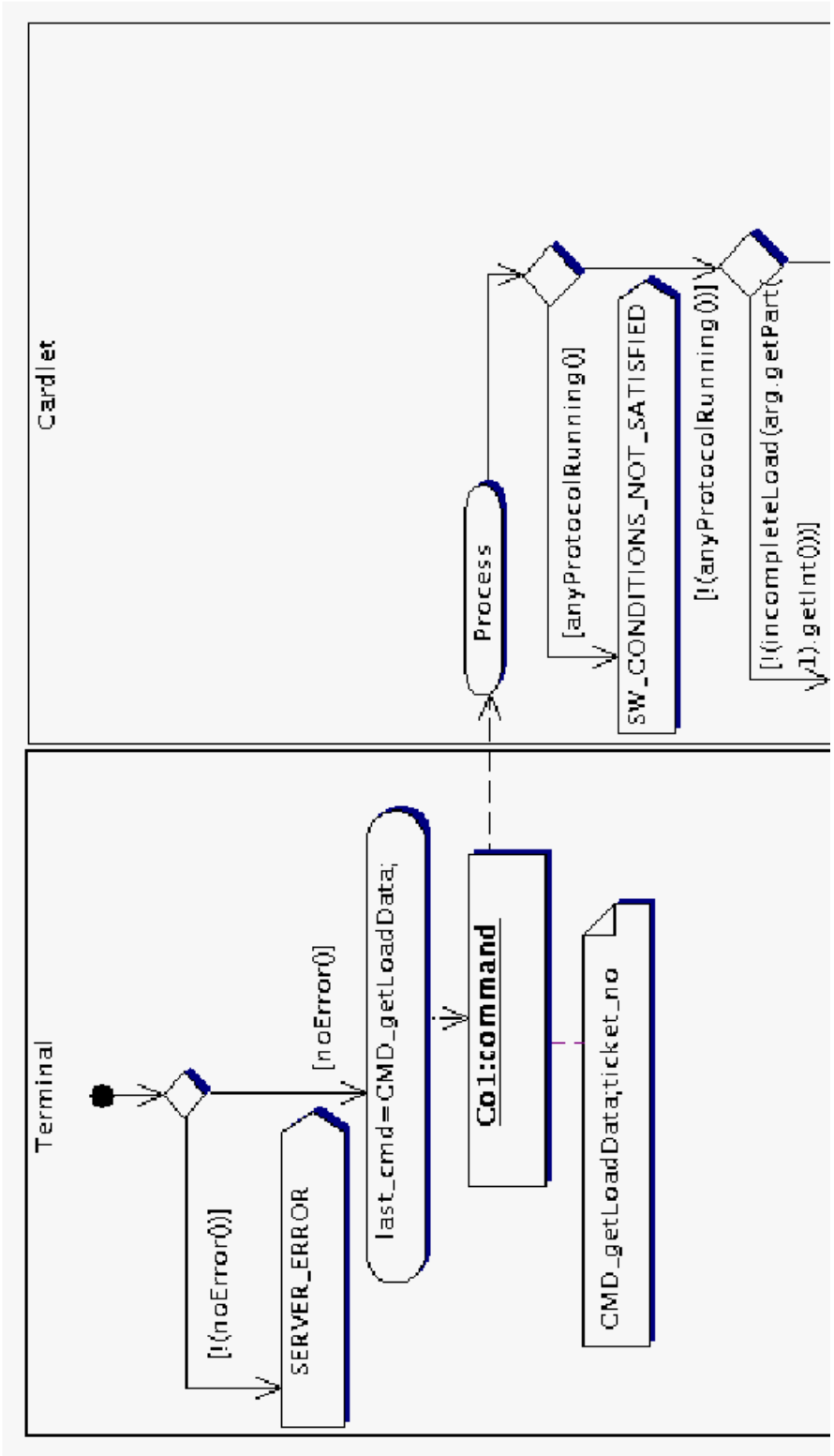


Figure 4.17: *Recover Load*-protocol Part I

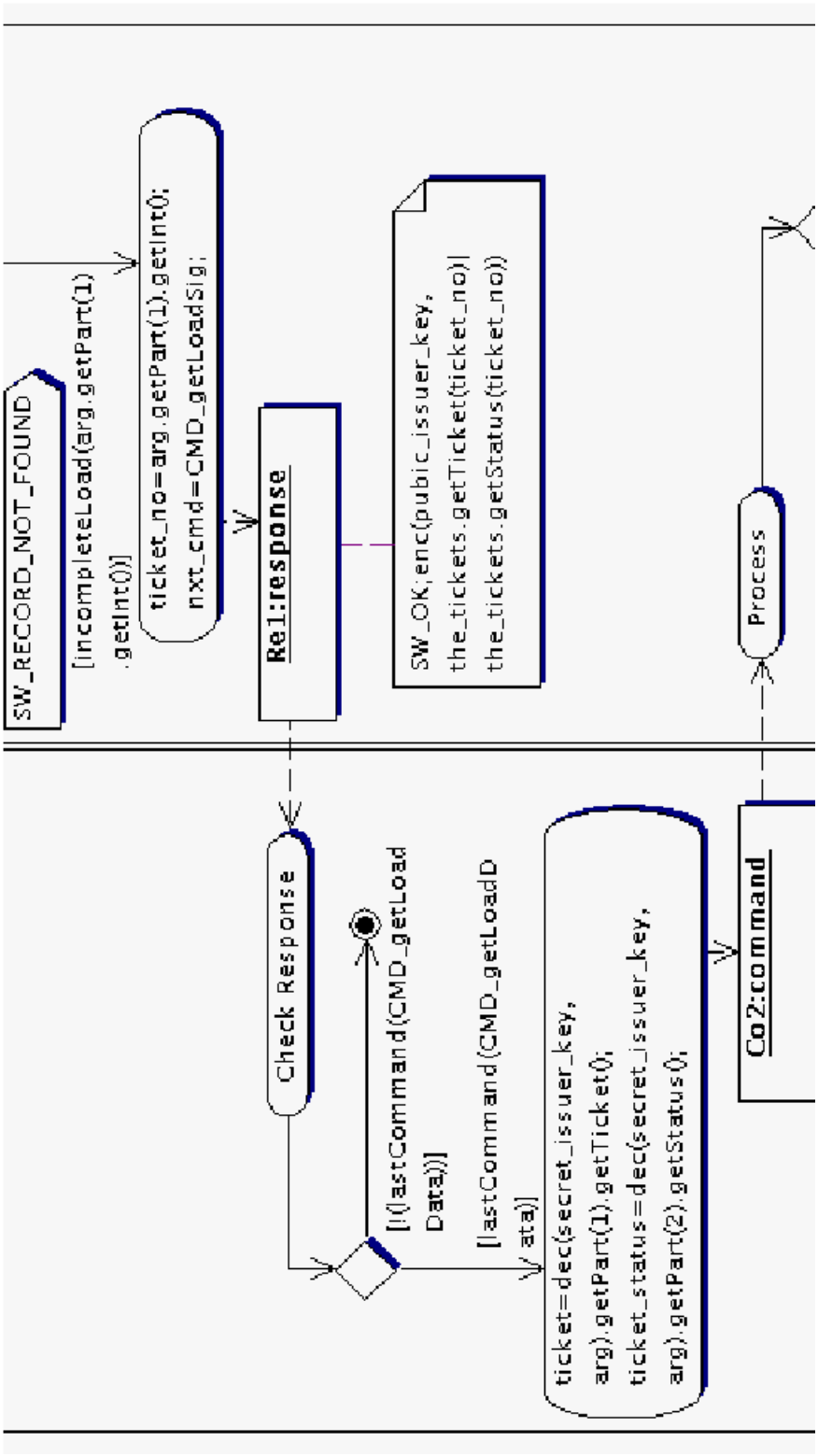


Figure 4.18: *Recover Load*-protocol Part II

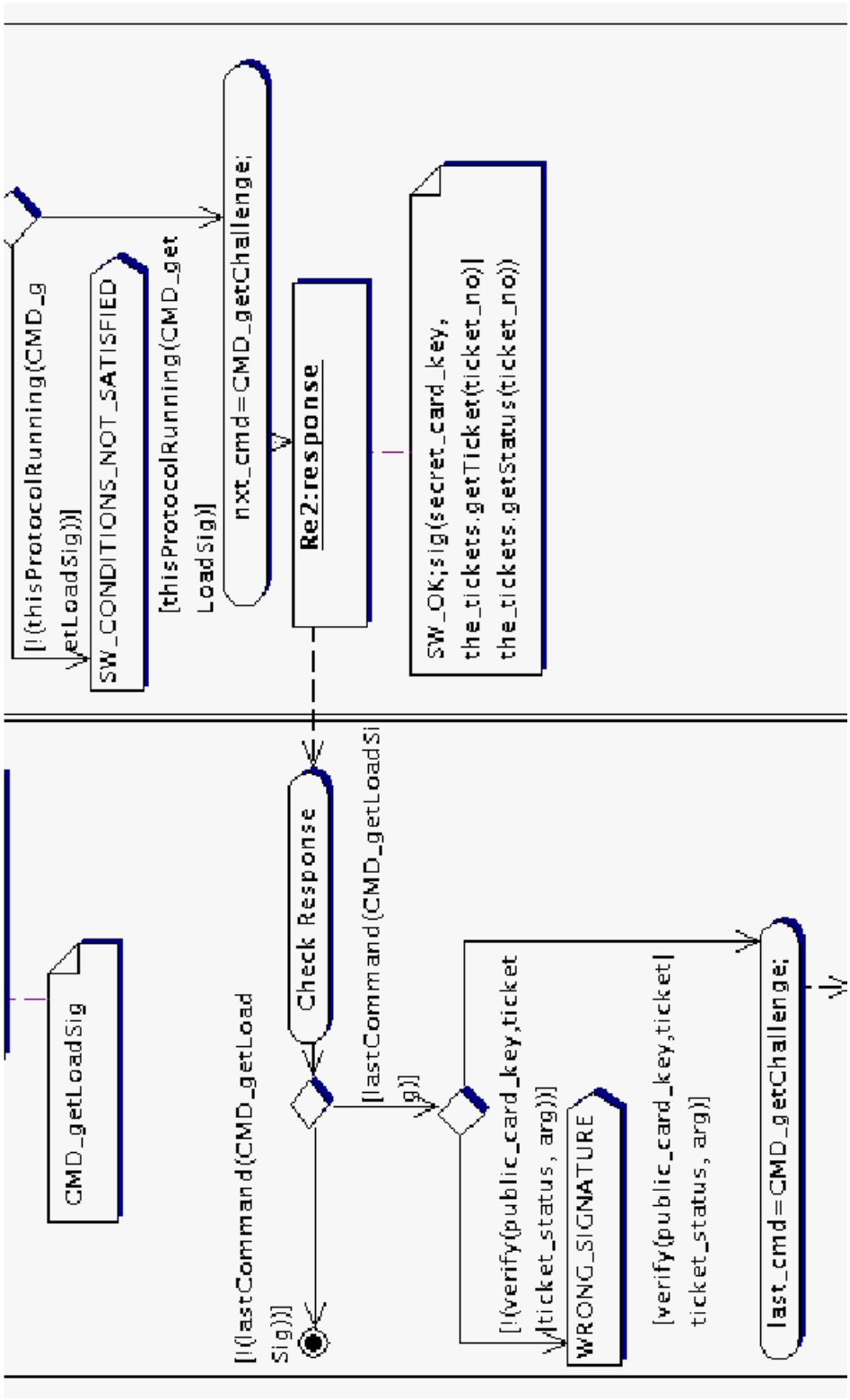


Figure 4.19: *Recover Load*-protocol Part III

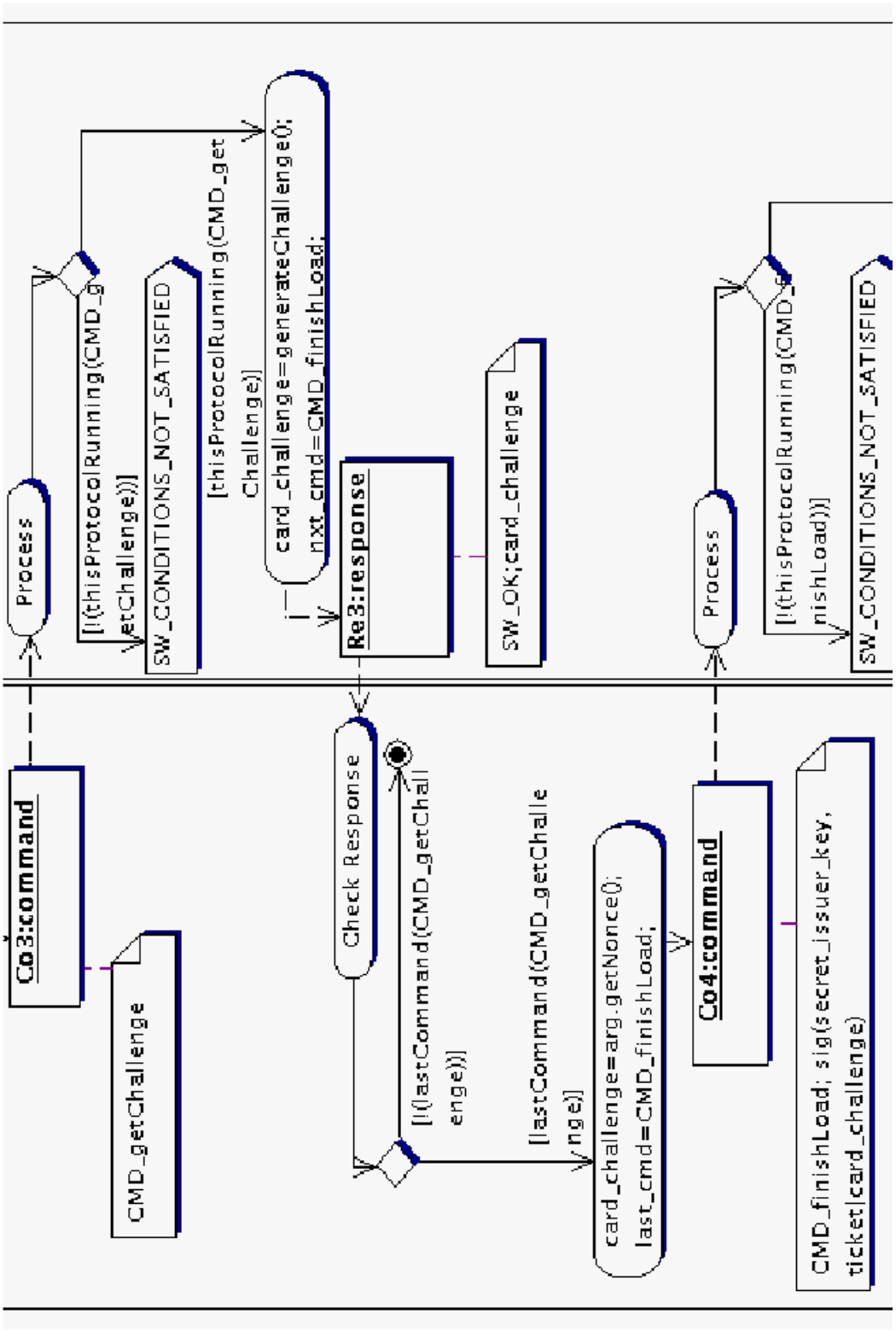


Figure 4.20: *Recover Load*-protocol Part IV

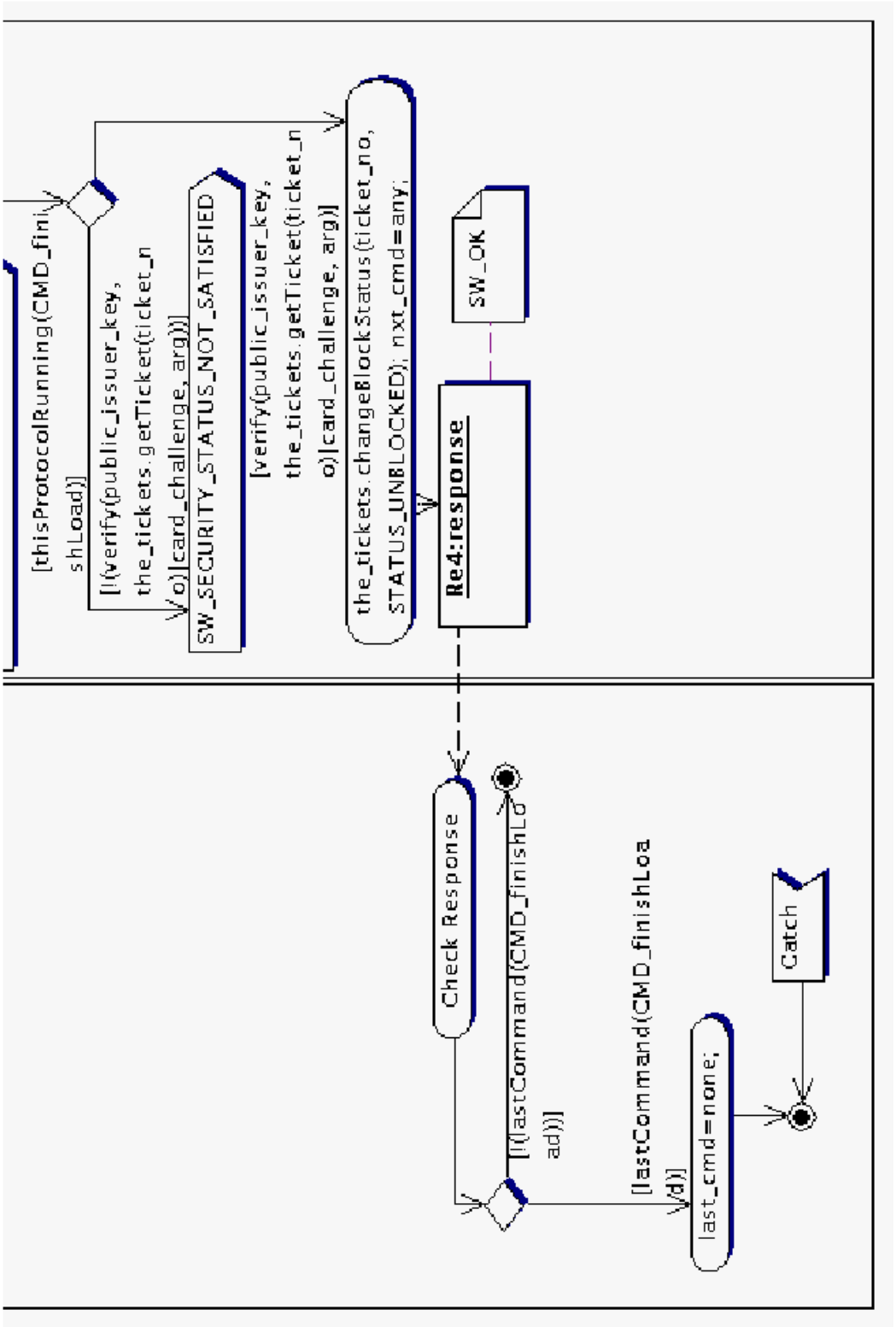


Figure 4.21: *Recover Load*-protocol Part V

4.14 Recover Transfer

4.14.1 Purpose

While a ticket is transferred the ticket in question is marked accordingly. Tickets that are marked as in transfer are unusable. If the ticket transfer is interrupted the ticket remains unusable until this protocol is run and the source card and the destination card are restored to consistent internal states.

4.14.2 Preconditions

Both cards, the source and the destination card, must have been authenticated properly using *Authenticate Cardlet*-protocol (Section 4.3). On both cards the `nxt_cmd`-attribute must be any. The two cards must be the source and the destination cards that were actually used in the erroneous transfer. The source card must be placed in the source card terminal, the destination card in the destination card terminal.

4.14.3 The protocol

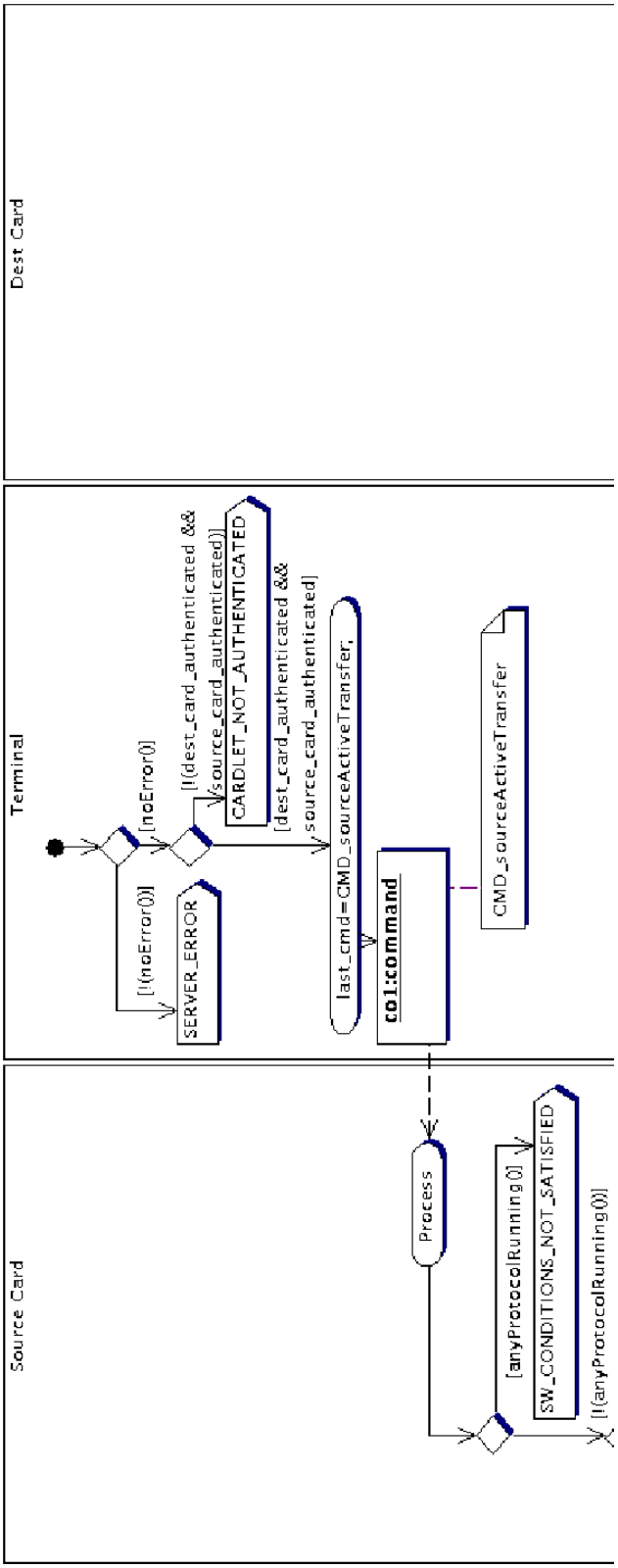


Figure 4.22: *Recover Transfer*-protocol Part I

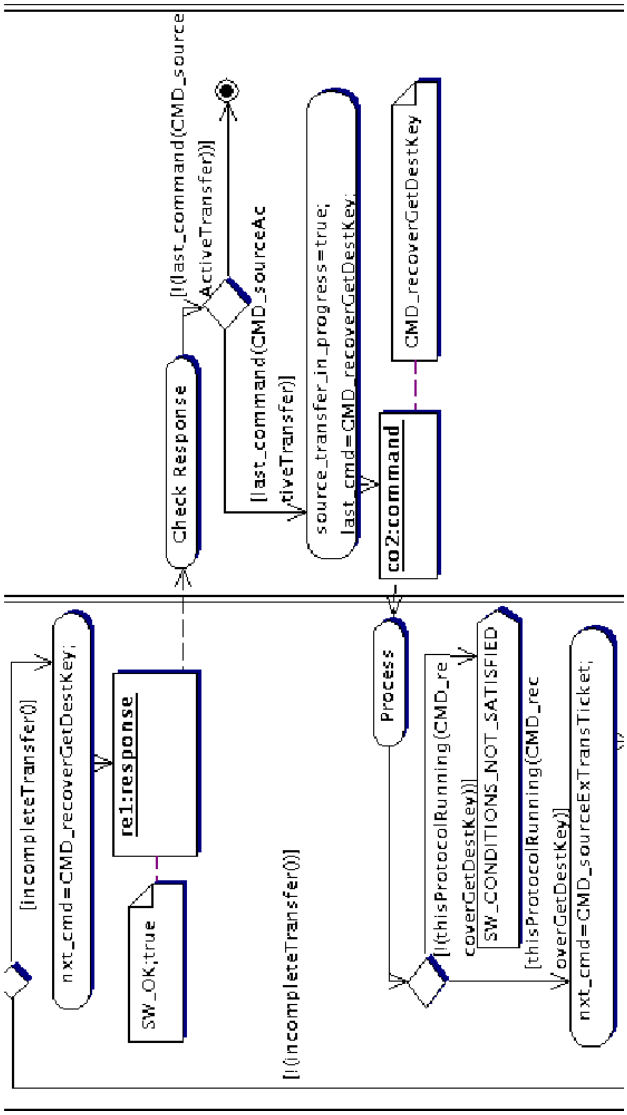


Figure 4.23: *Recover Transfer*-protocol Part II

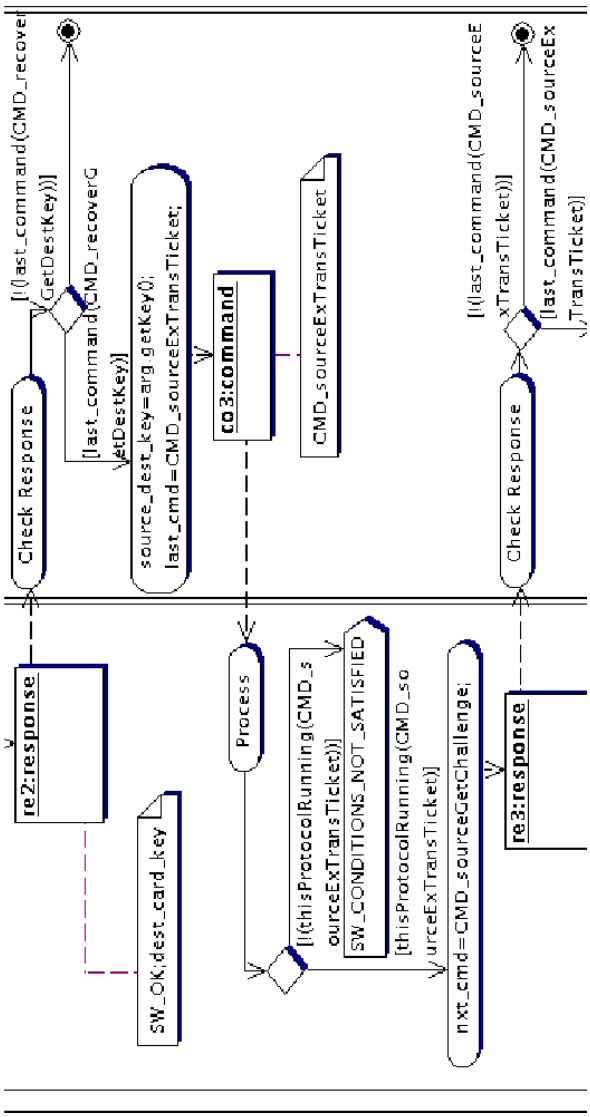


Figure 4.24: *Recover Transfer*-protocol Part III

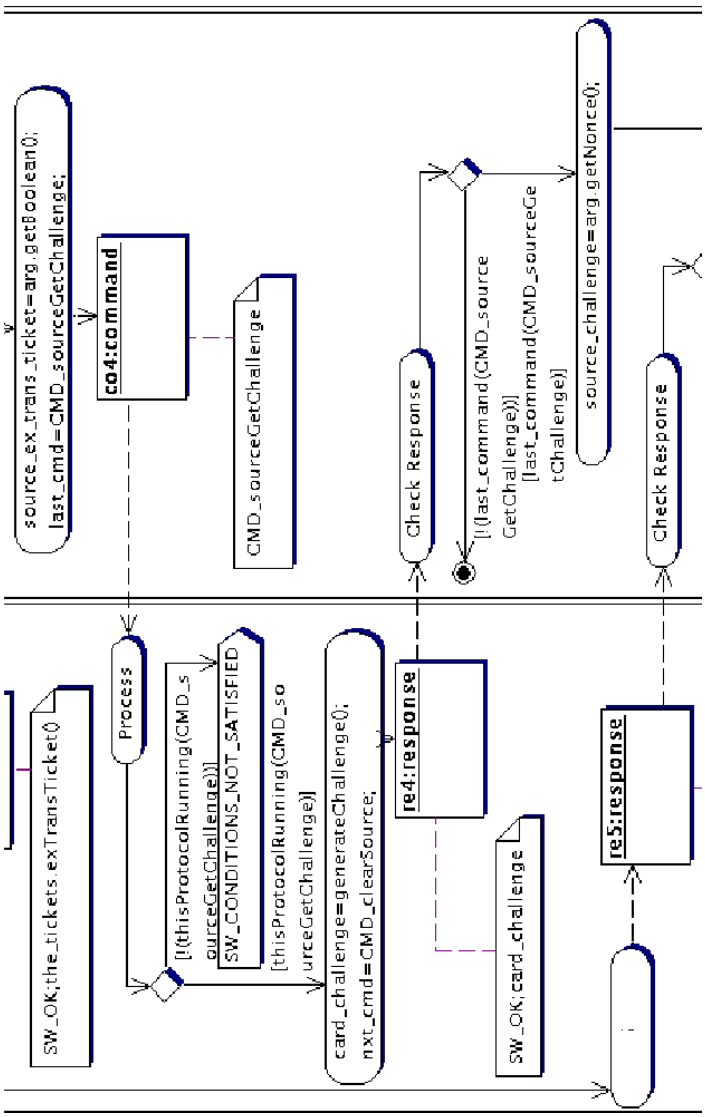


Figure 4.25: *Recover Transfer*-protocol Part IV

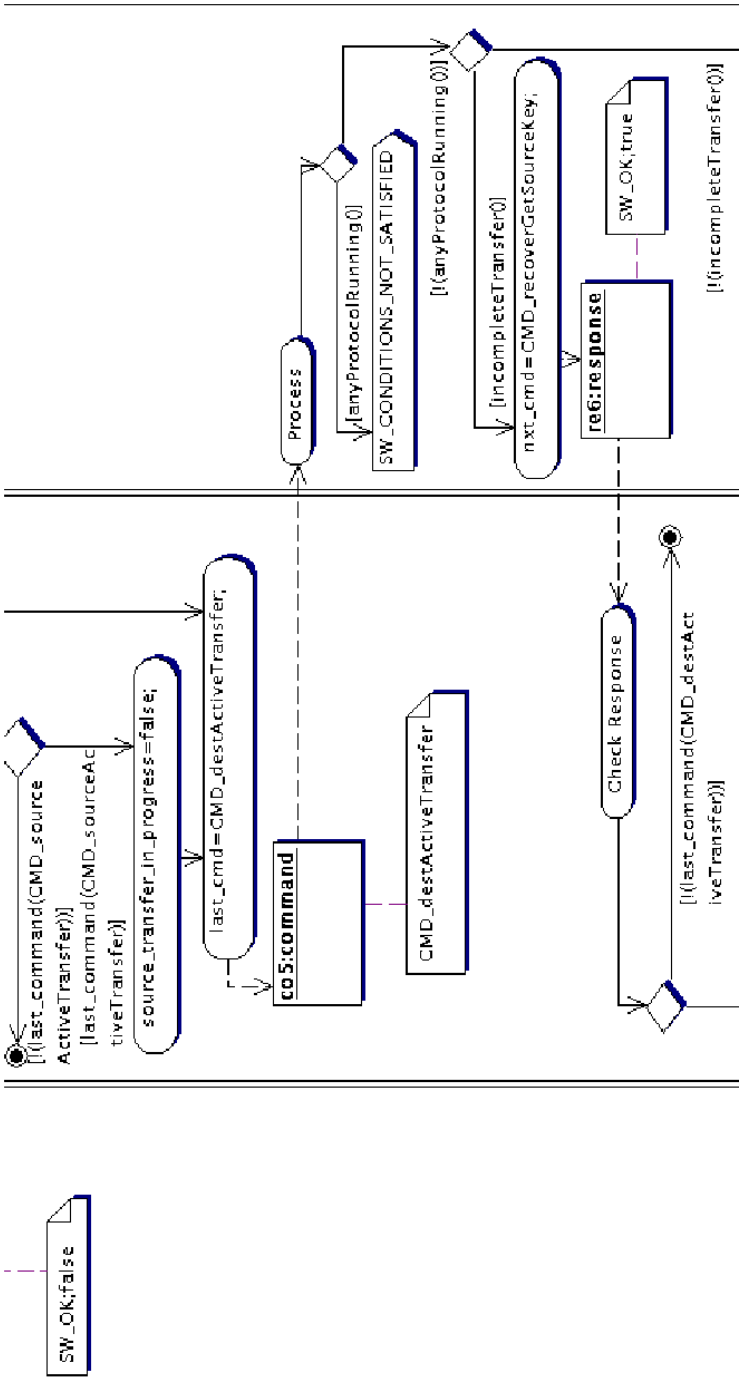


Figure 4.26: *Recover Transfer*-protocol Part V

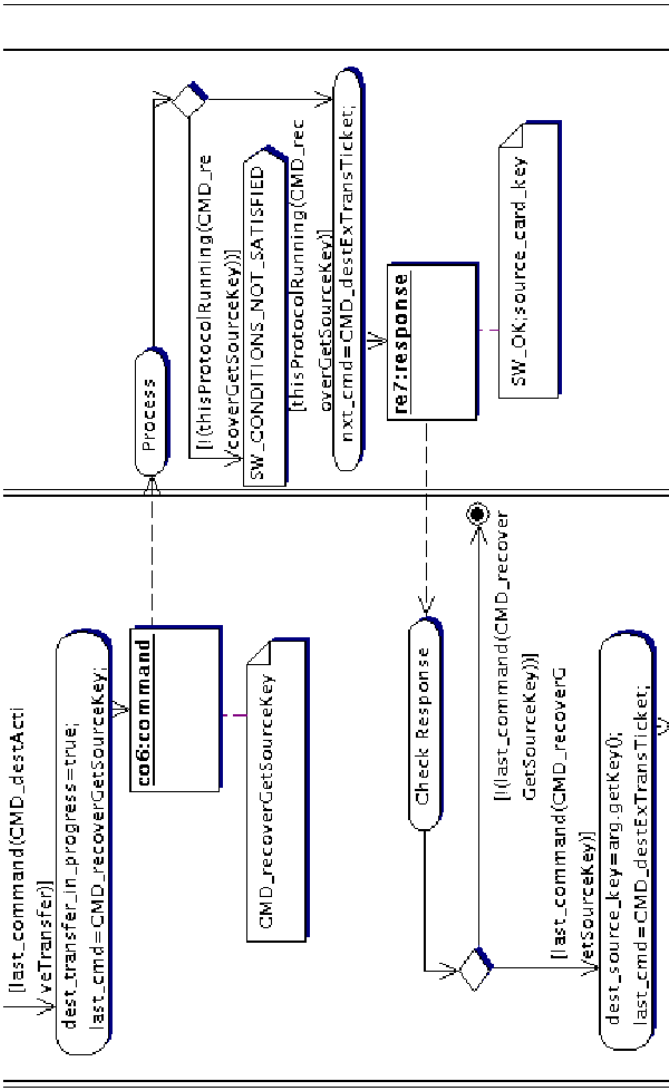


Figure 4.27: *Recover Transfer*-protocol Part VI

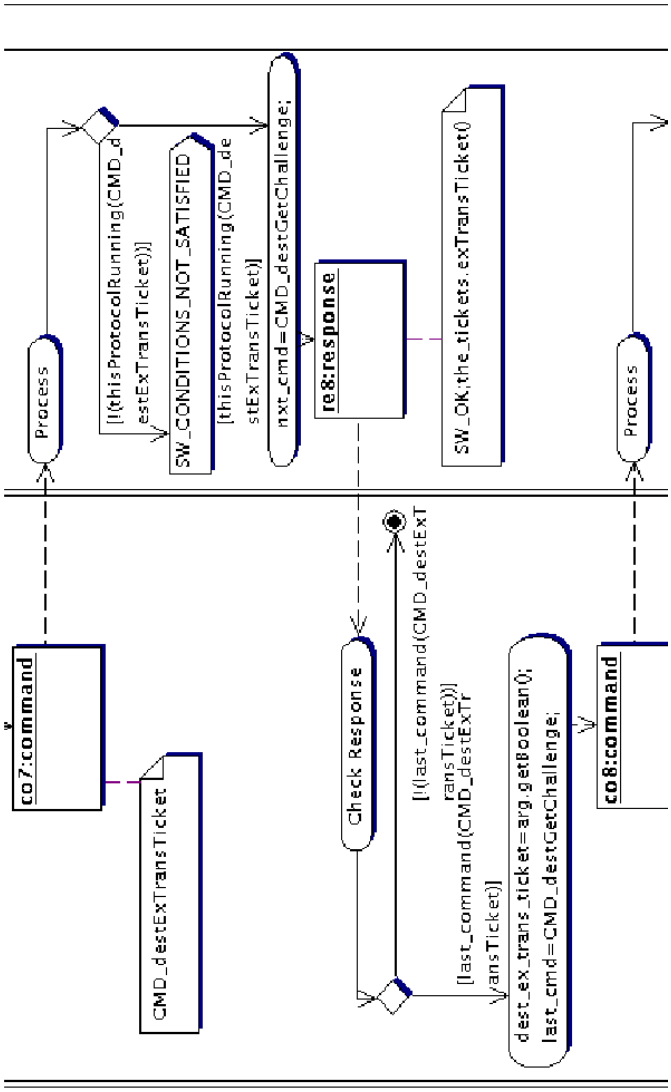


Figure 4.28: *Recover Transfer*-protocol Part VII

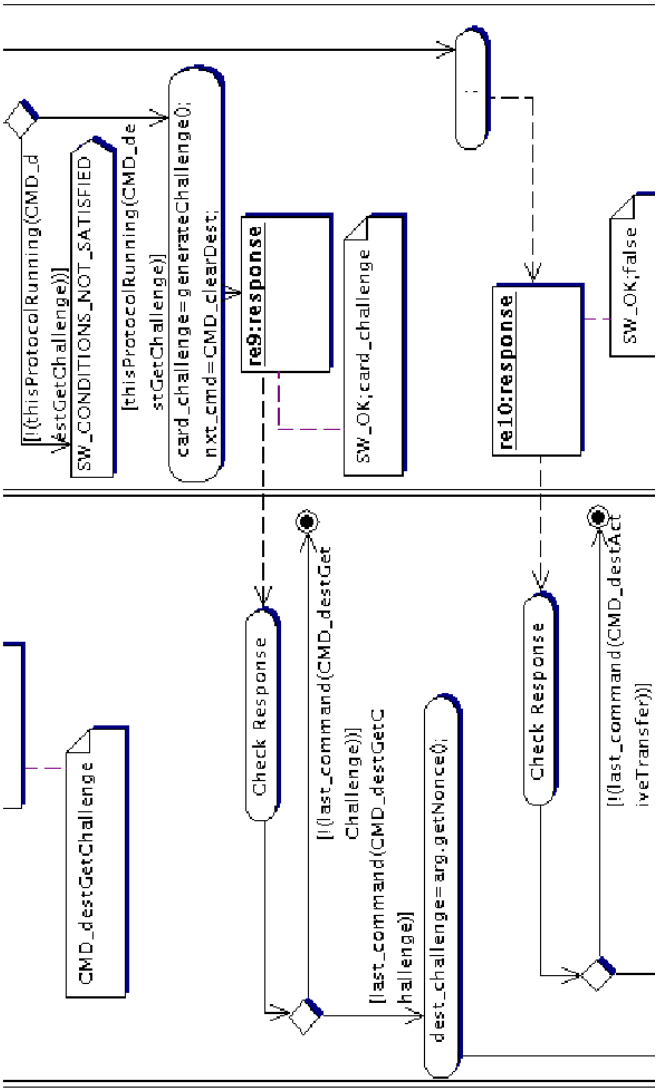


Figure 4.29: *Recover Transfer*-protocol Part VIII

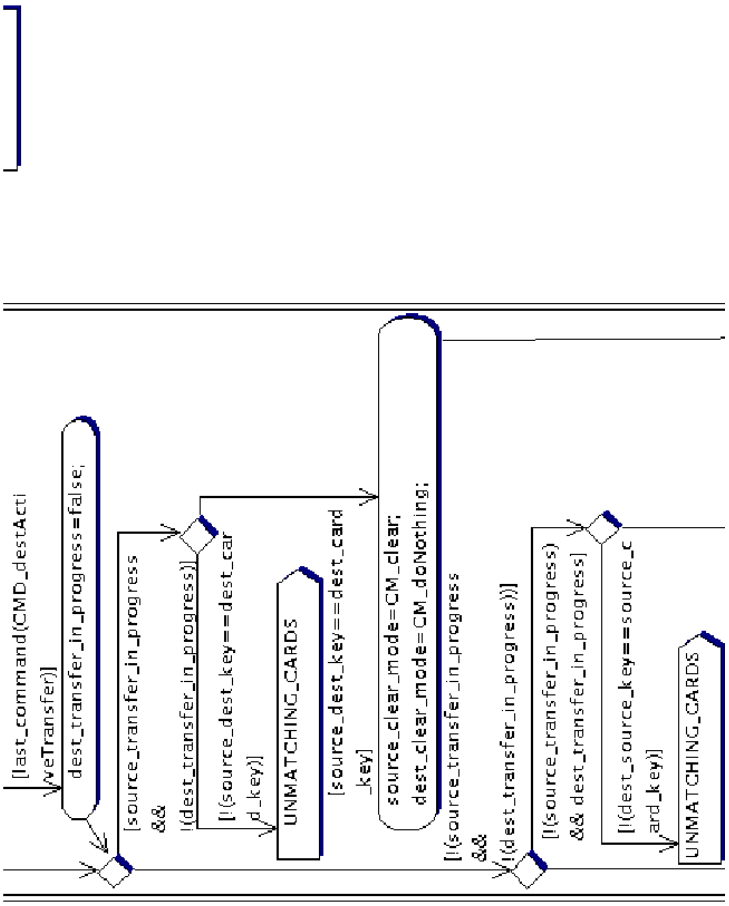


Figure 4.30: *Recover Transfer*-protocol Part IX

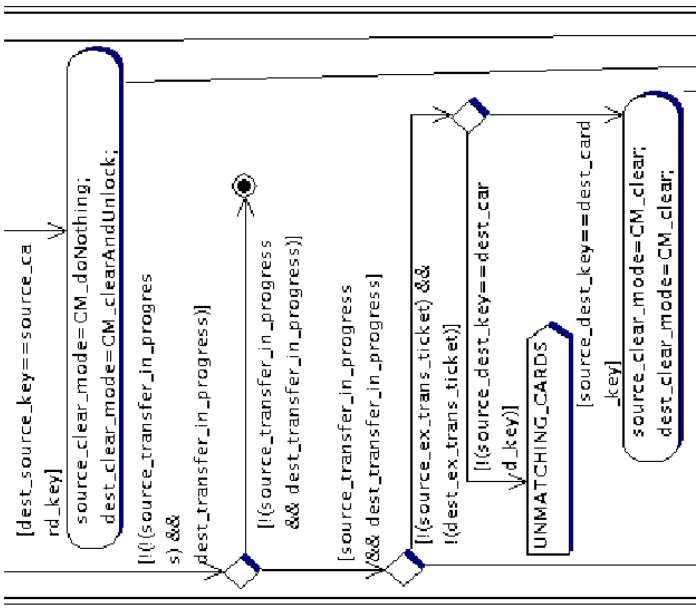


Figure 4.31: *Recover Transfer*-protocol Part X

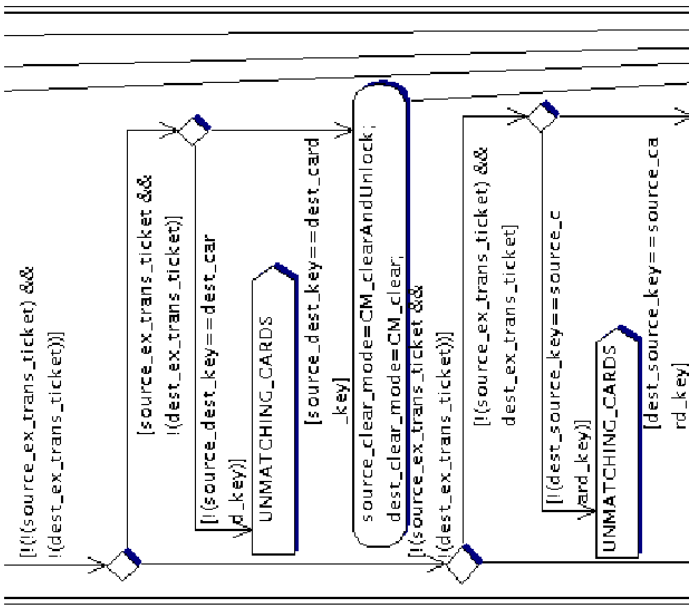


Figure 4.32: *Recover Transfer*-protocol Part XI

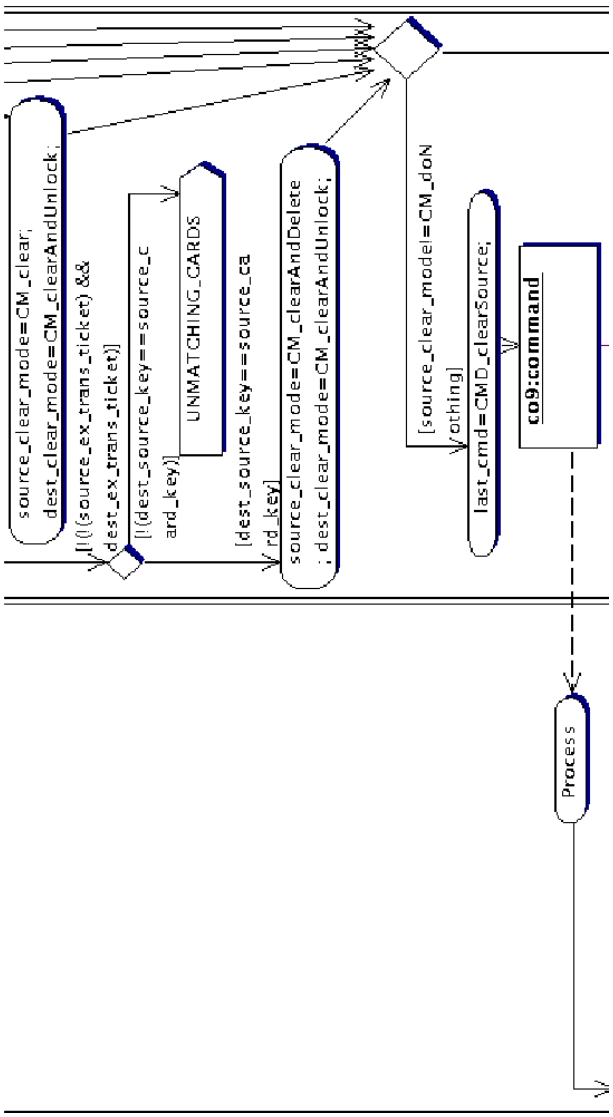


Figure 4.33: *Recover Transfer*-protocol Part XII

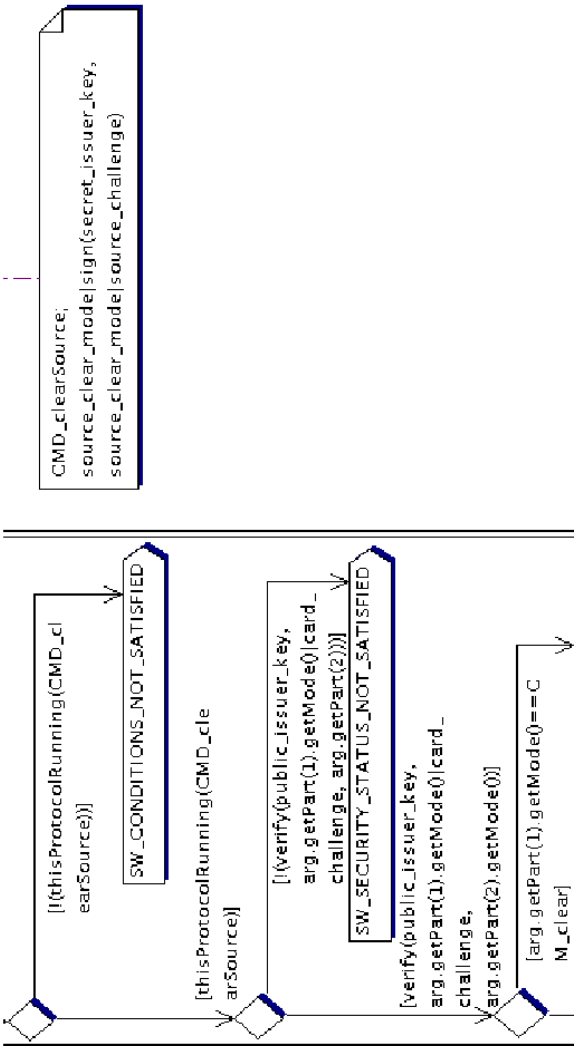


Figure 4.34: *Recover Transfer*-protocol Part XIII

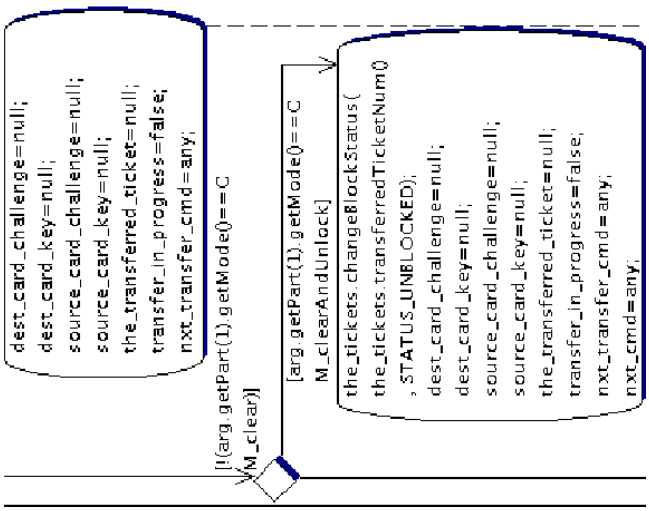


Figure 4.35: *Recover Transfer*-protocol Part XIV

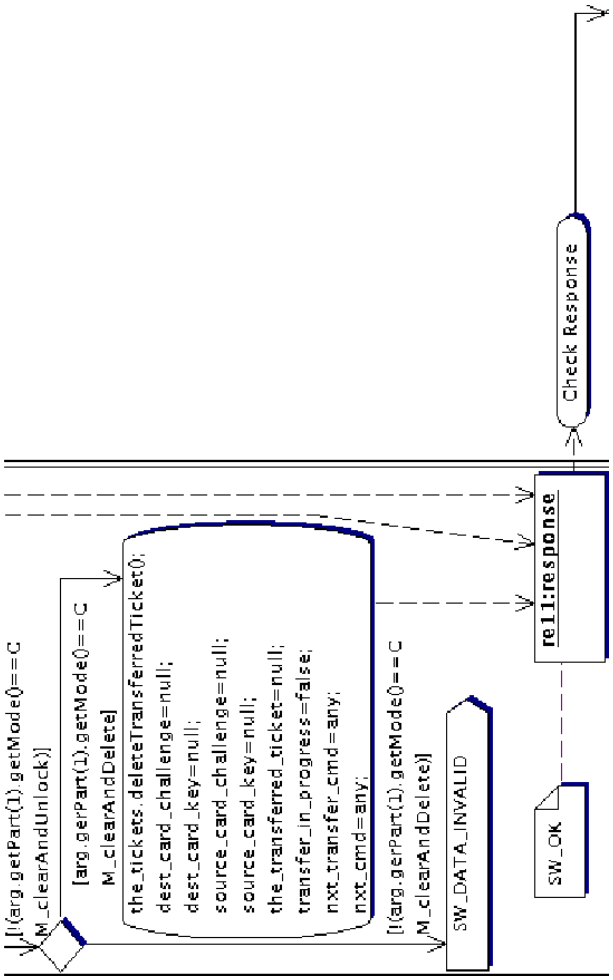


Figure 4.36: *Recover Transfer*-protocol Part XV

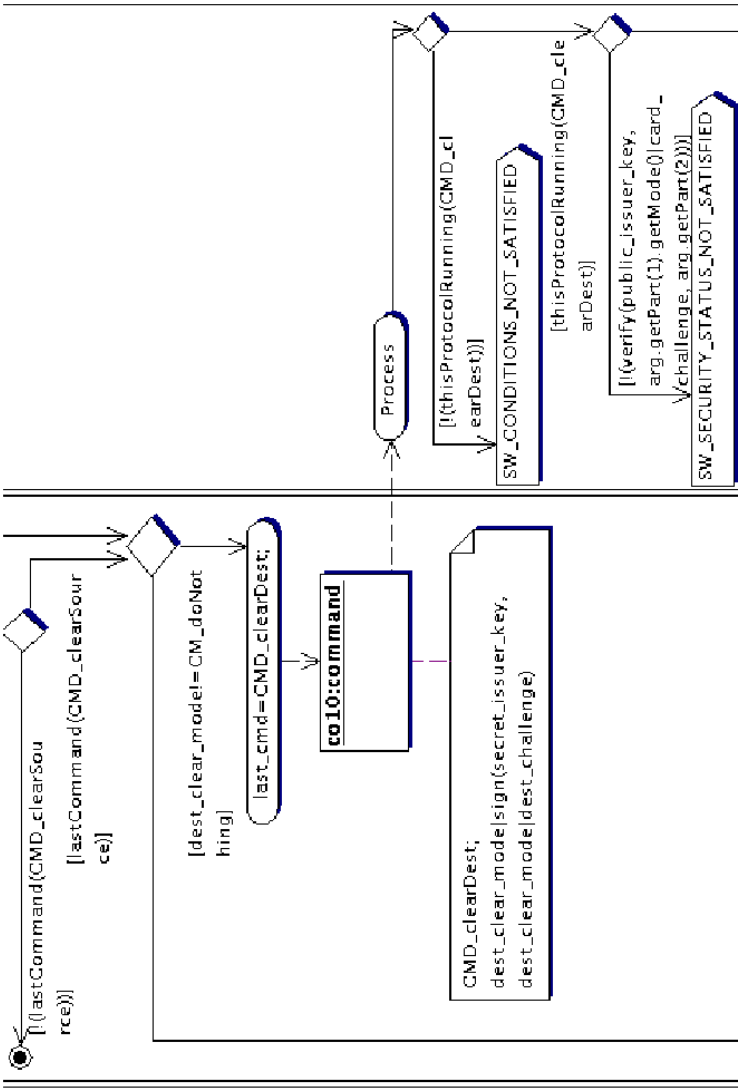


Figure 4.37: *Recover Transfer*-protocol Part XVI

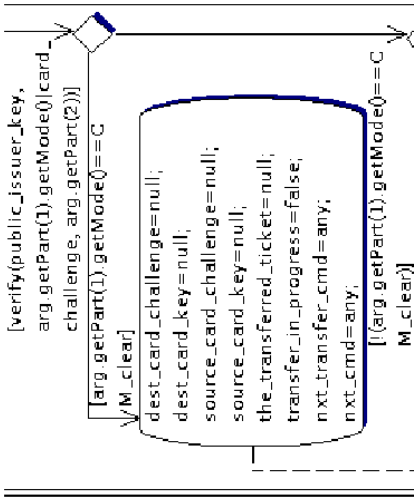


Figure 4.38: *Recover Transfer*-protocol Part XVII

Figure 4.39: *Recover Transfer*-protocol Part XVIII

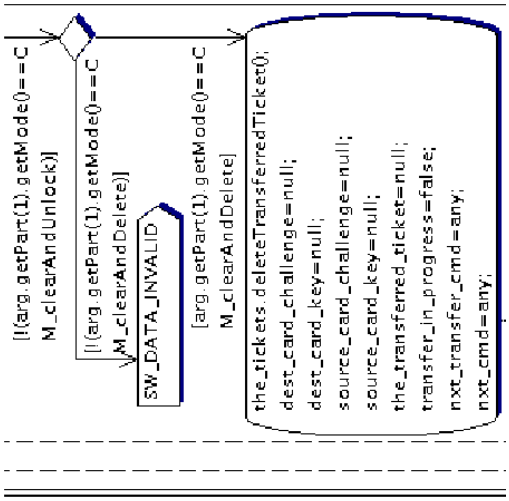


Figure 4.40: *Recover Transfer*-protocol Part XIX

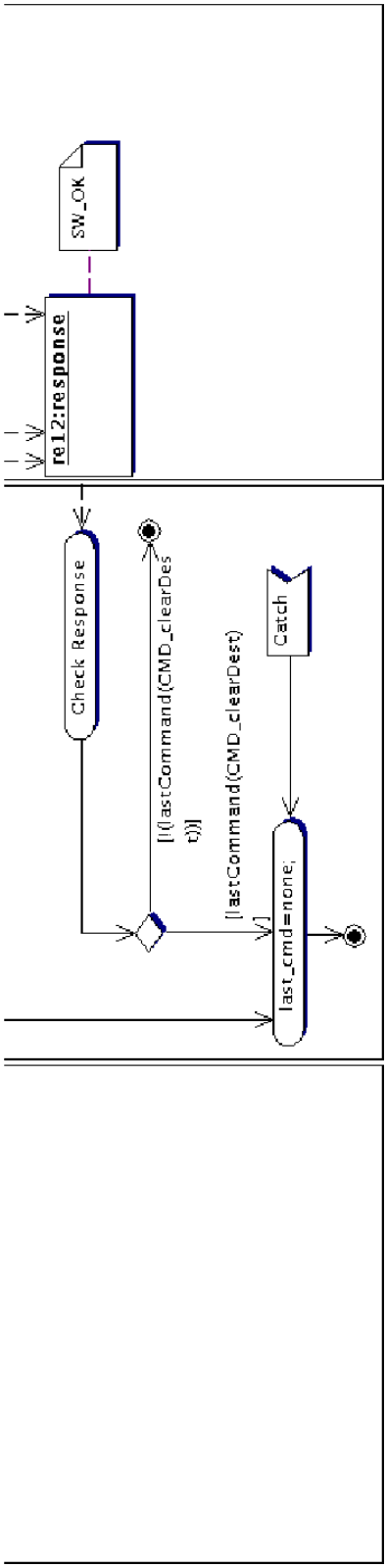


Figure 4.41: *Recover Transfer*-protocol Part XX

4.15 Select

4.15.1 Purpose

This protocol describes the actions performed by the `select`-method. The `select`-method of a cardlet is called by the card runtime environment when the card receives an APDU containing a select command for the corresponding cardlet.

4.15.2 Preconditions

None.

4.15.3 The protocol

The `select`-method resets protocol-specific attributes of the cardlet and tries to resume an incomplete ticket transfer if necessary.

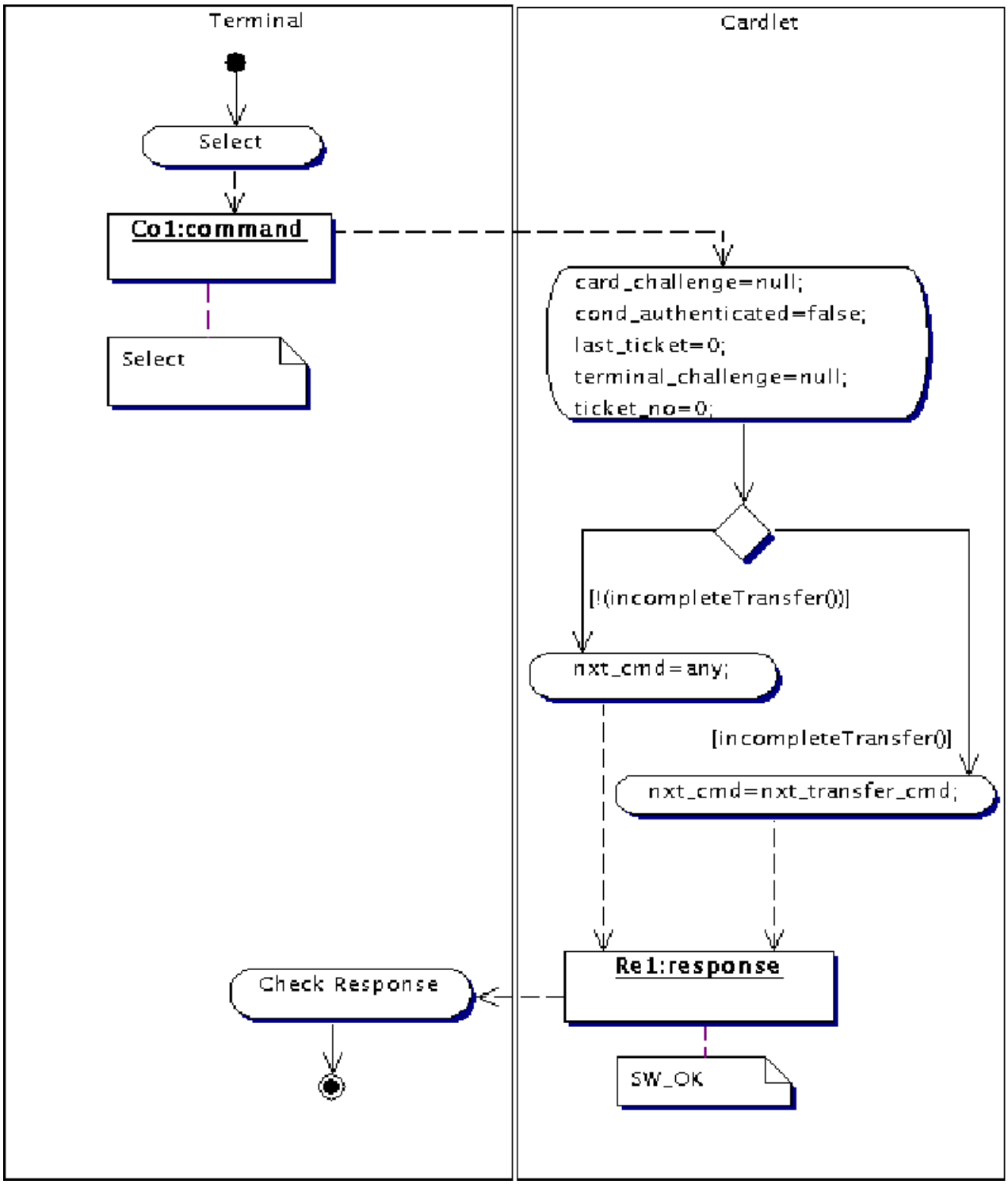


Figure 4.42: *Select*-protocol

4.16 Set User PIN

4.16.1 Purpose

This function is used by the card owner to set a new PIN. After a successful authentication of the card holder the card accepts a new user PIN and stores it.

4.16.2 Preconditions

The owner of the card must have been successfully authenticated using the *Authenticate User*-protocol (Section 4.5). There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.16.3 The protocol

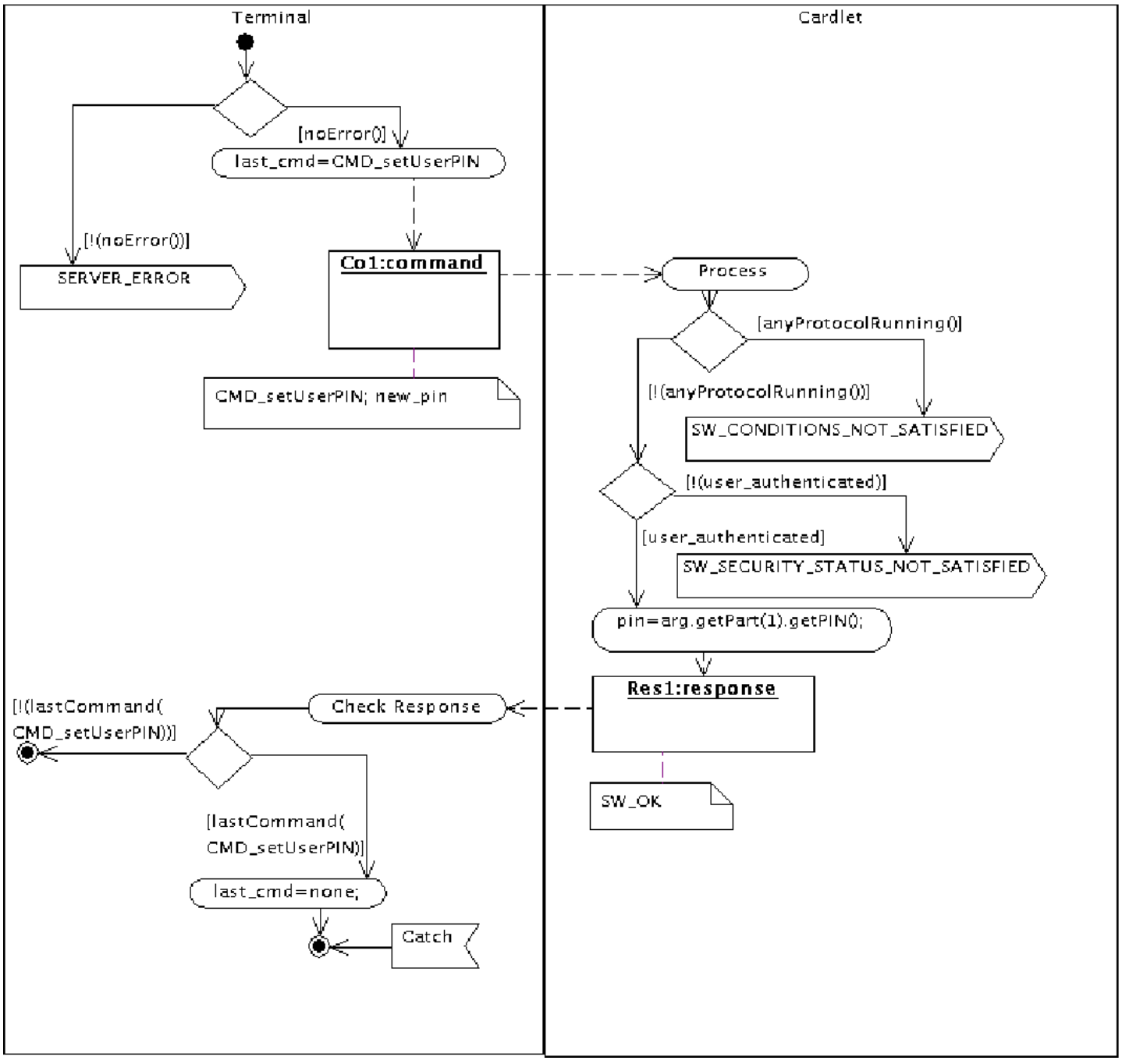


Figure 4.43: *Set User PIN*-protocol

4.17 Transfer Ticket

4.17.1 Purpose

This tickets is used to transfer a ticket form one card to another. This is useful for example if a group of people travels together and one person buys the tickets for all participants and hands them one later.

4.17.2 Preconditions

Before this protocol can be executed the terminal must have acquired the public key and the public key signature of the destination card and the source card. This is done using the *Get Card Key*-protocol (Section 4.9). The keys must be stored in the attributes `dest_card_key` and `source_card_key` respectively. The signatures must be stored in `dest_card_key_sig` and `source_card_key_sig`. It must be possible to load another ticket on the destination card, i. e. the number of tickets on the destination card must be less than `max_tickets`. On both cards the `nxt_cmd`-field of the cardlet must be `any`, i. e. none of the cards should currently run another protocol. Both cards mustn't be in an incomplete transfer, i. e. a transfer must be completed before a new one can be started. The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. The ticket that is to be transferred mustn't be obliterated. The owner of the source card must be authenticated.

4.17.3 Additional information

If this protocol is performed on a computer with only one card reader, it is necessary to switch between destination and source card. After removing and subsequently reinserting a card the protocol cannot be continued at once, due to the resetting of the `nxt_cmd`-field. To restore the proper value of `nxt_cmd` the *Continue Transfer*-protocol must be applied.

4.17.4 The protocol

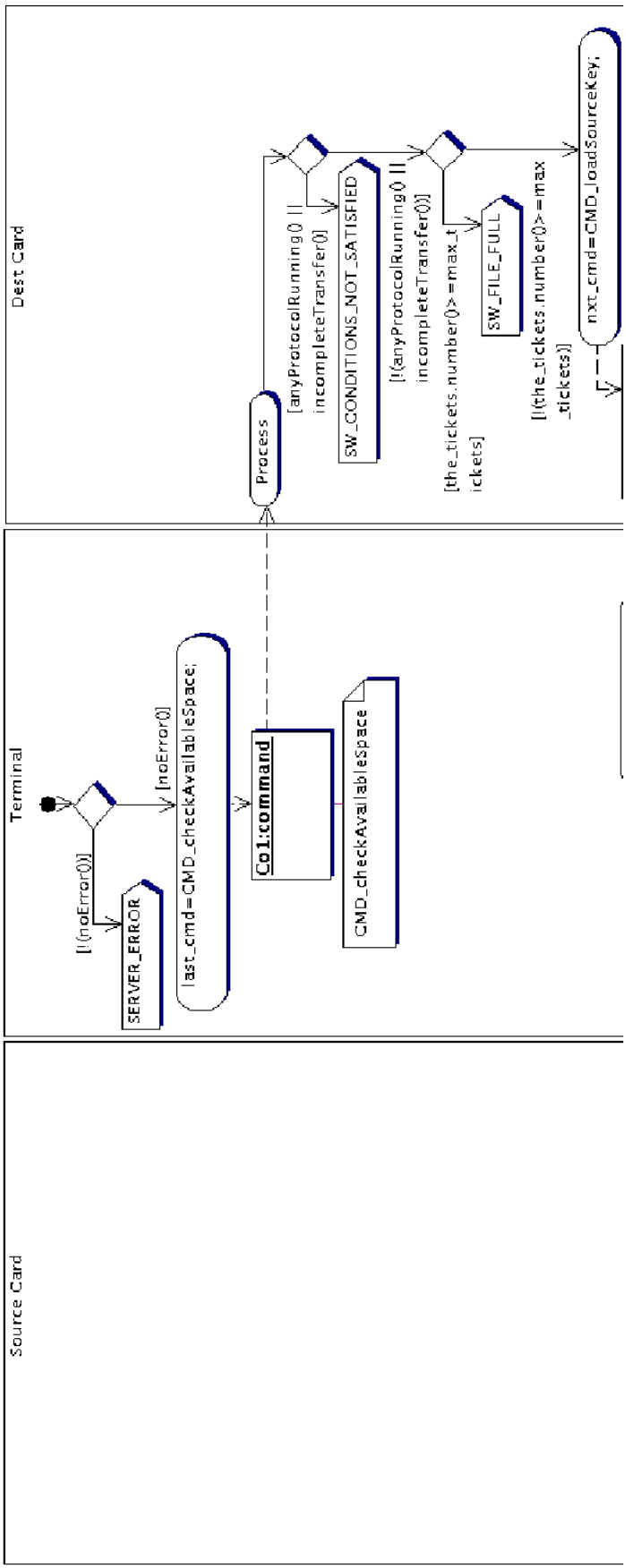


Figure 4.44: *Transfer Ticket*-protocol Part I

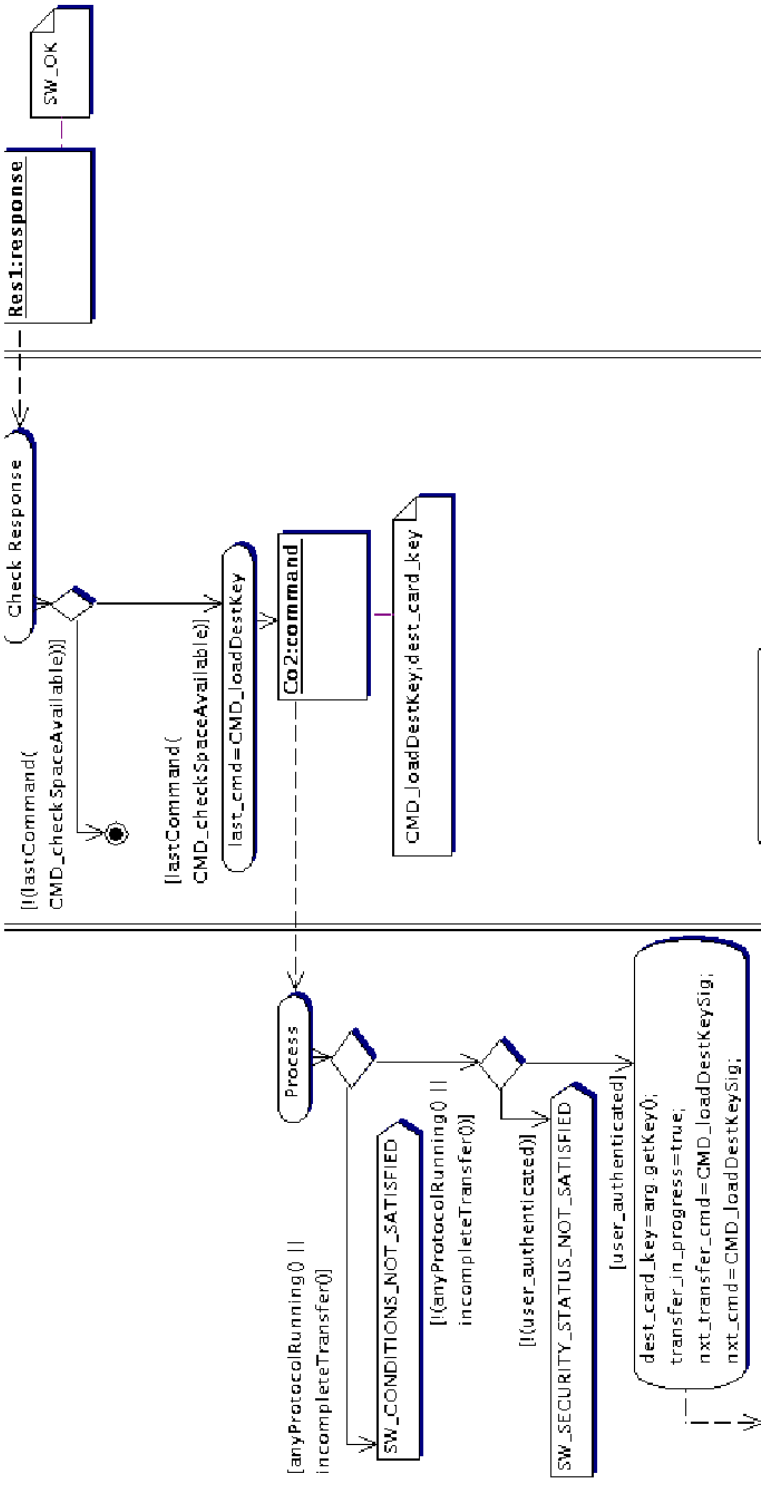


Figure 4.45: *Transfer Ticket*-protocol Part II

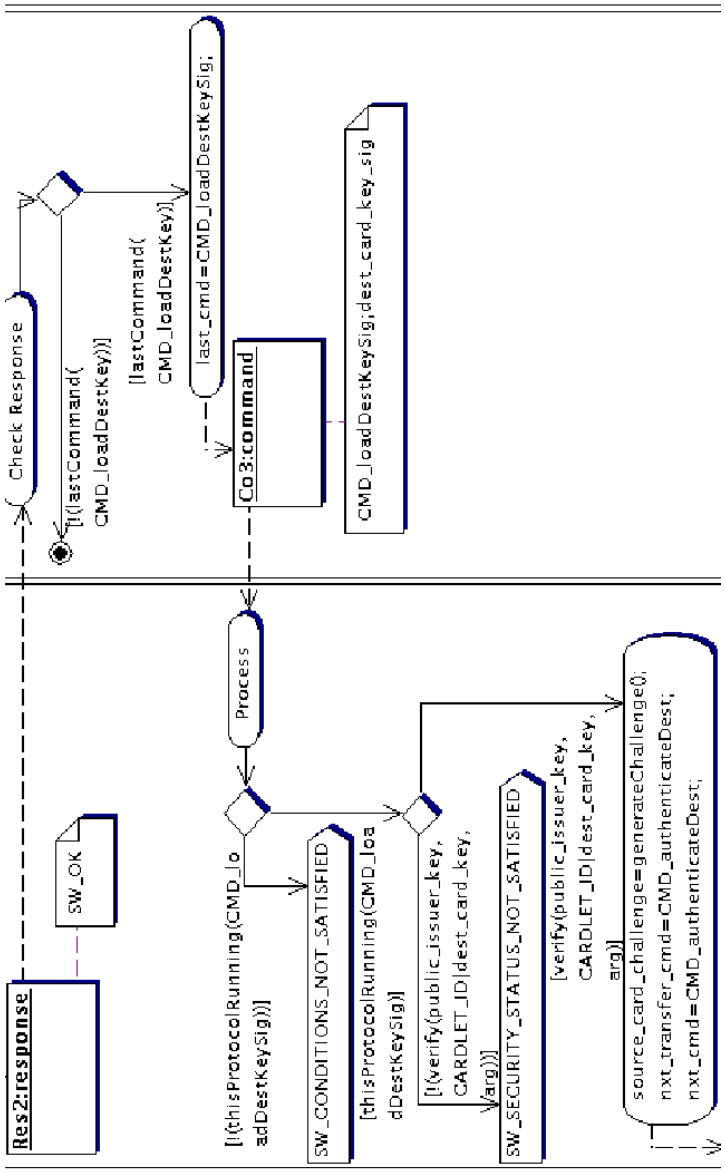


Figure 4.46: *Transfer Ticket*-protocol Part III

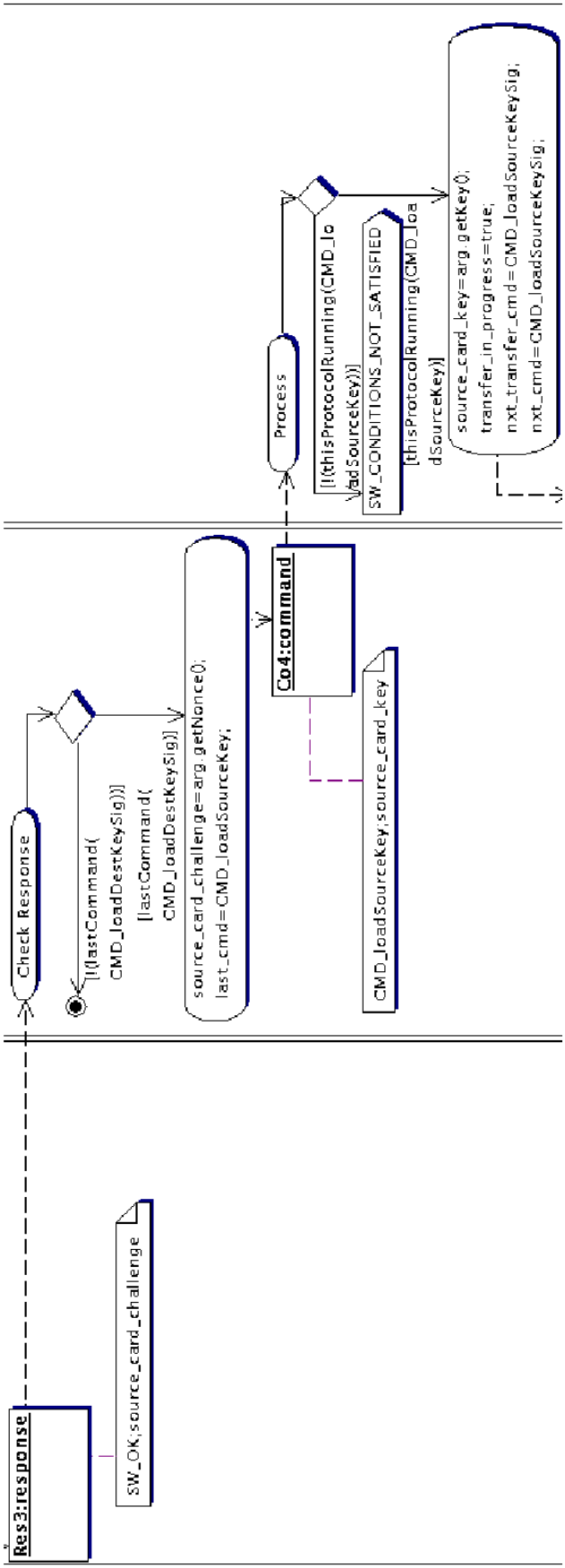


Figure 4.47: *Transfer Ticket*-protocol Part IV

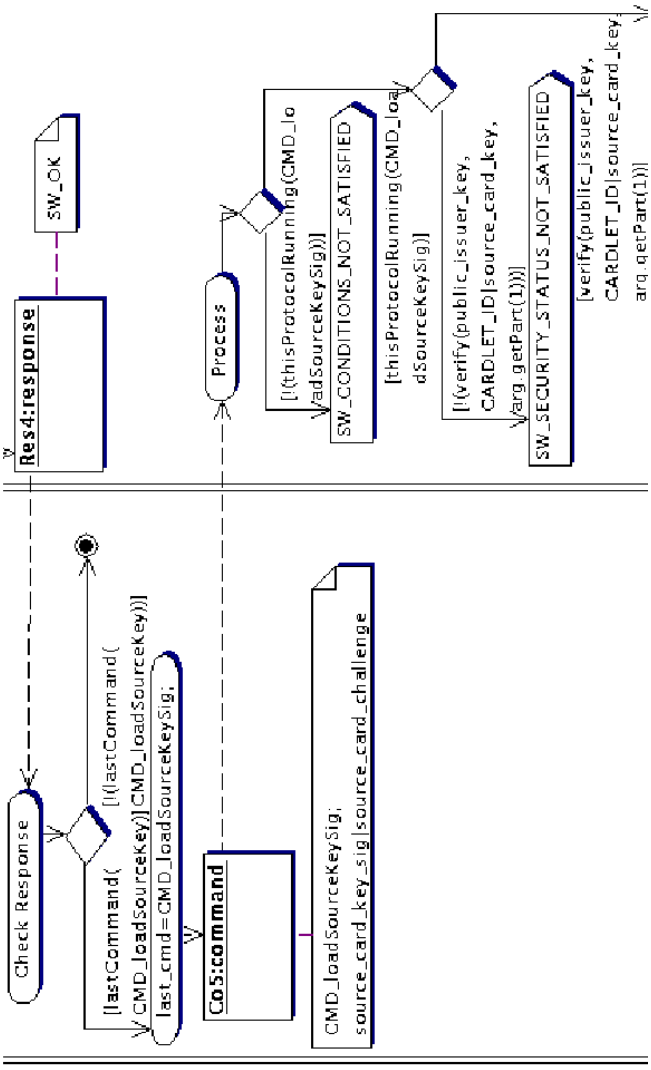


Figure 4.48: *Transfer Ticket*-protocol Part V

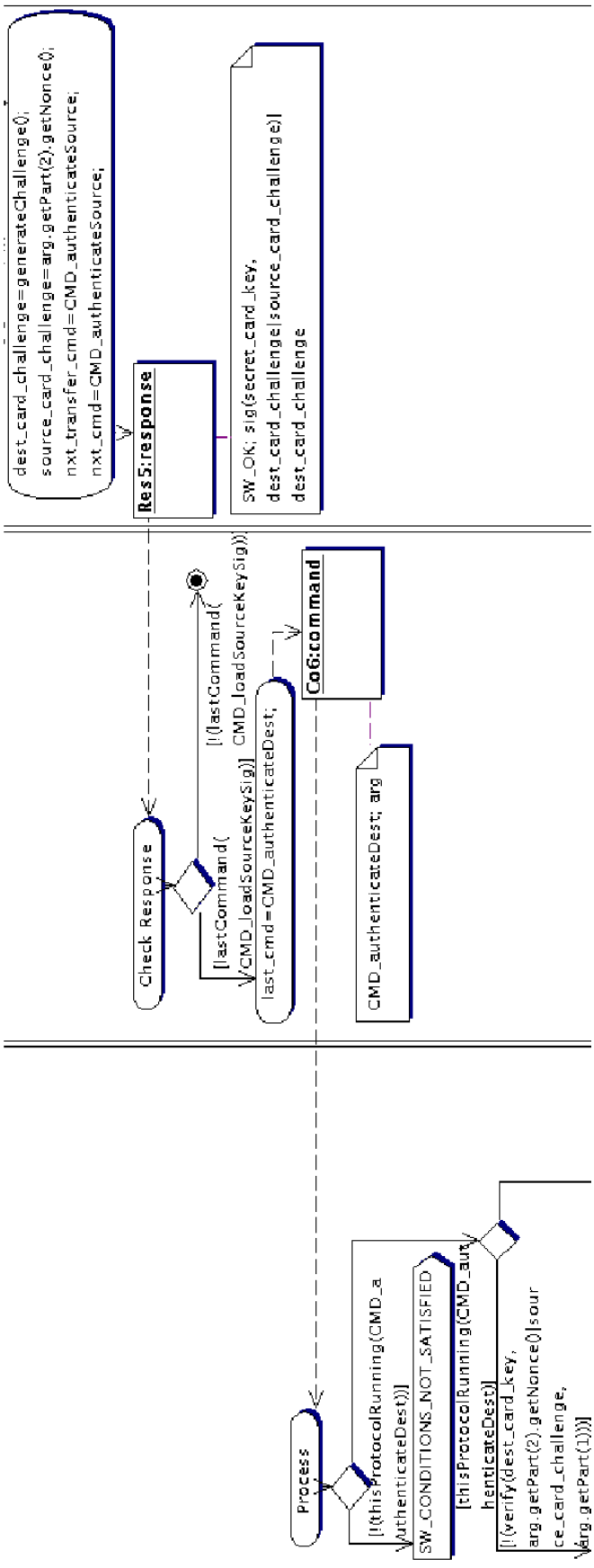


Figure 4.49: *Transfer Ticket*-protocol Part VI

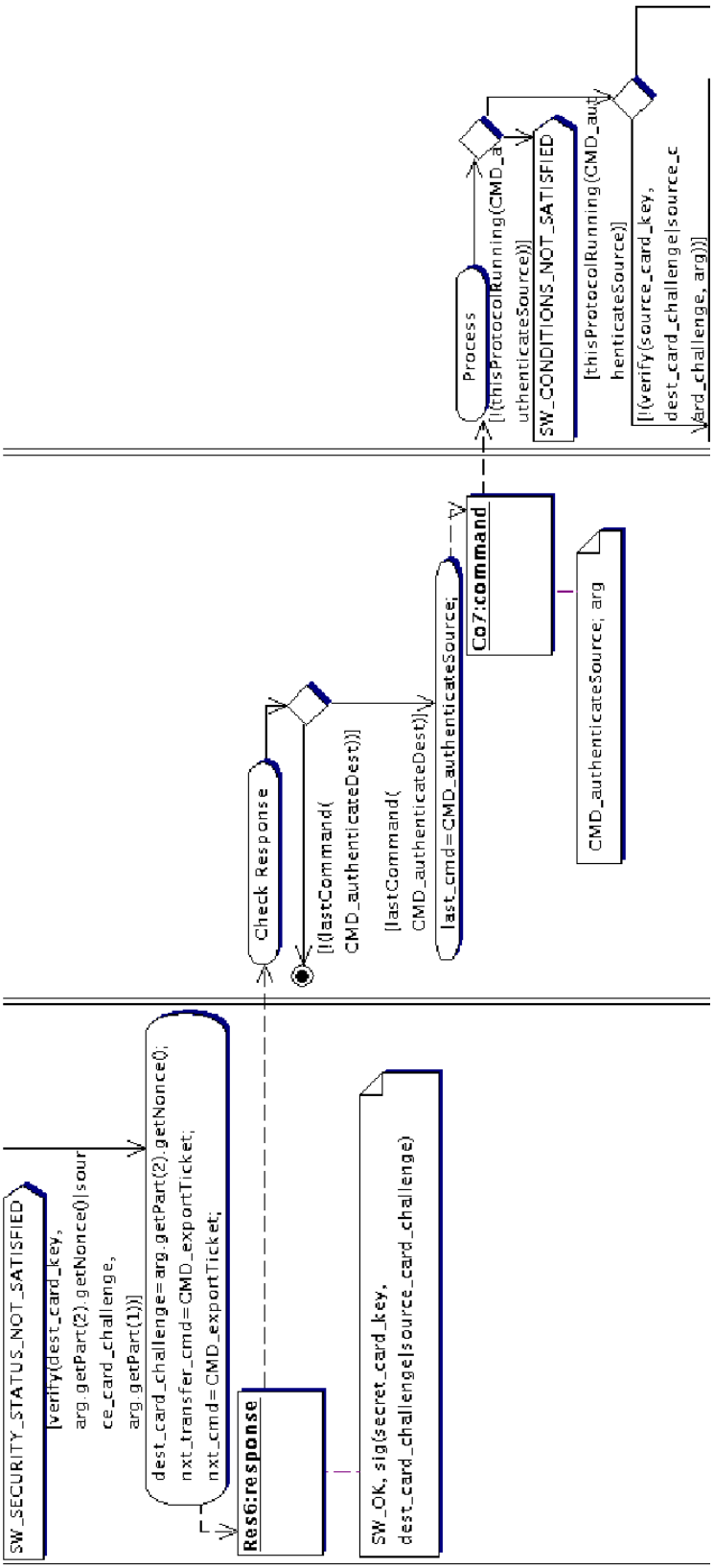


Figure 4.50: *Transfer Ticket*-protocol Part VII

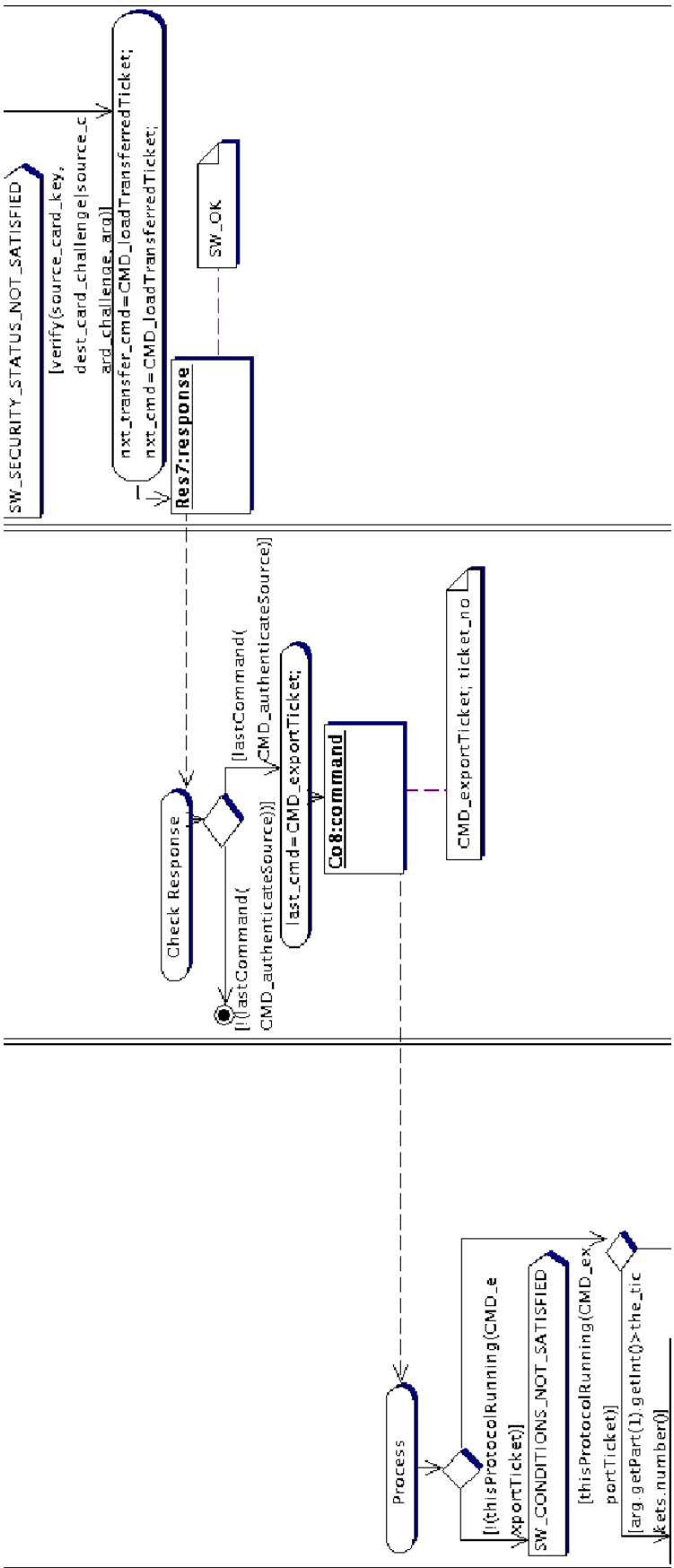


Figure 4.51: *Transfer Ticket*-protocol Part VIII

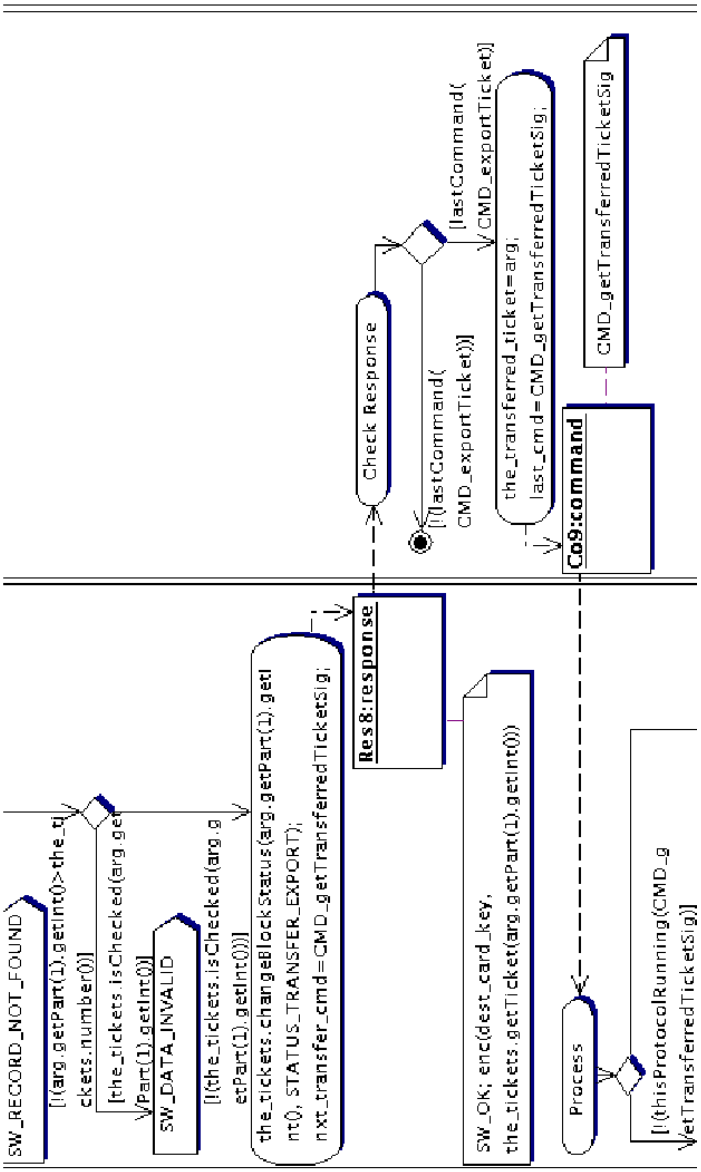


Figure 4.52: *Transfer Ticket*-protocol Part IX

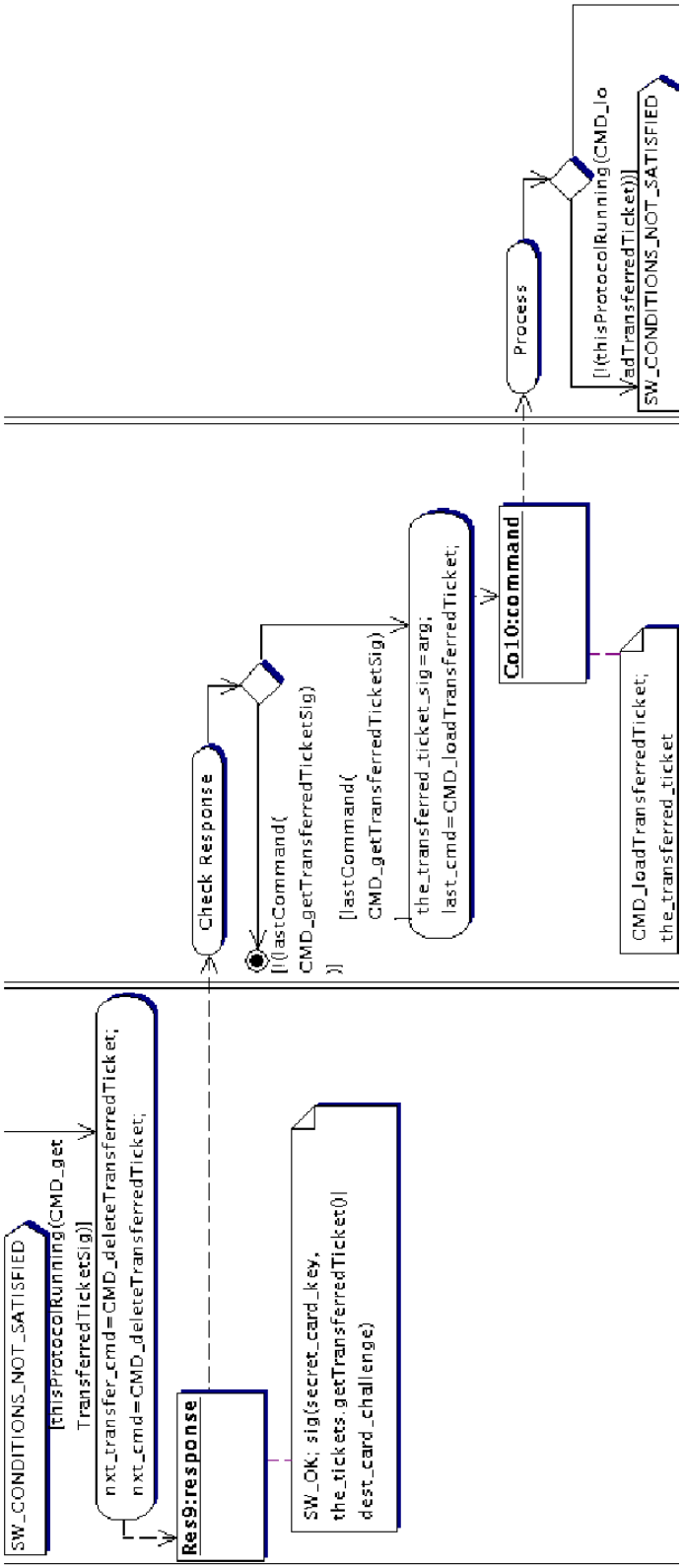


Figure 4.53: *Transfer Ticket*-protocol Part X

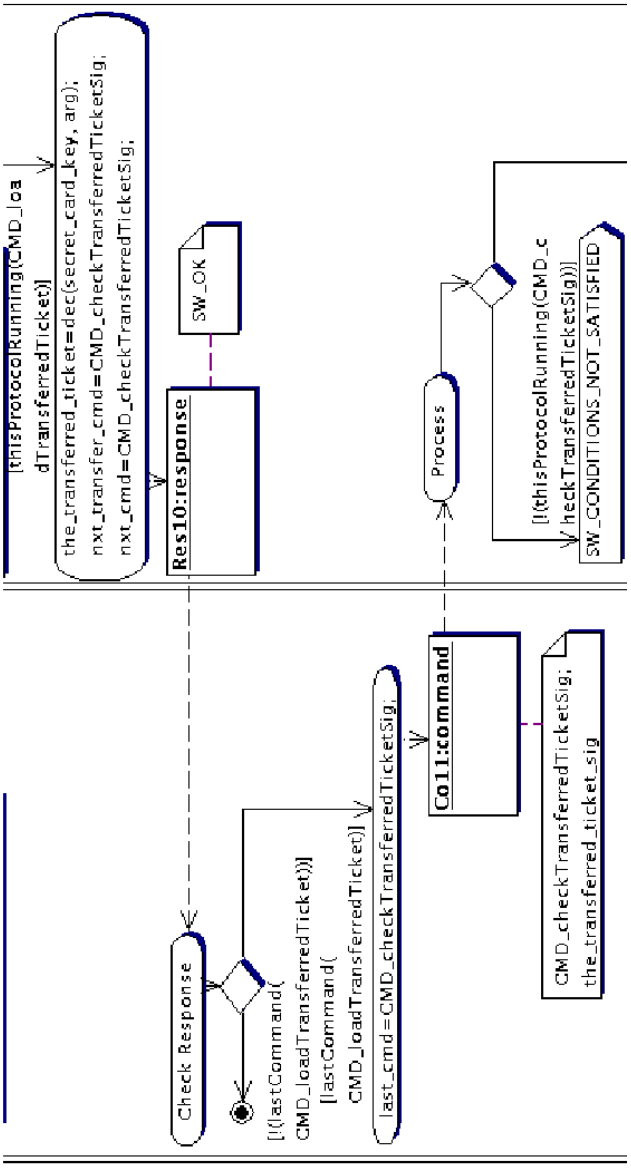


Figure 4.54: *Transfer Ticket*-protocol Part XI

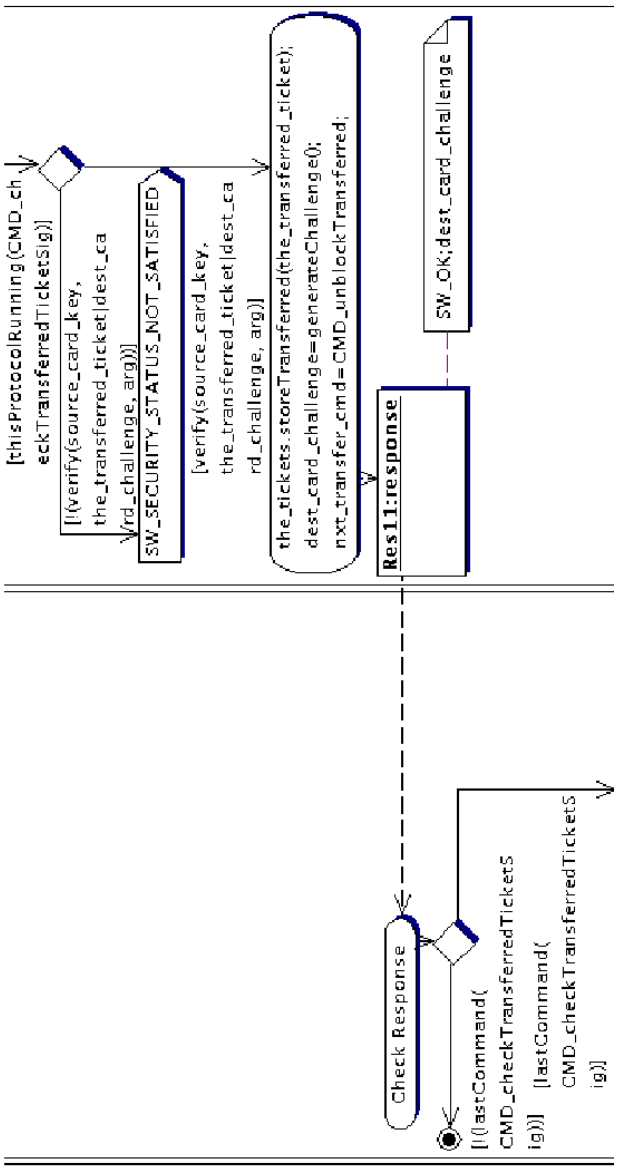


Figure 4.55: *Transfer Ticket*-protocol Part XII

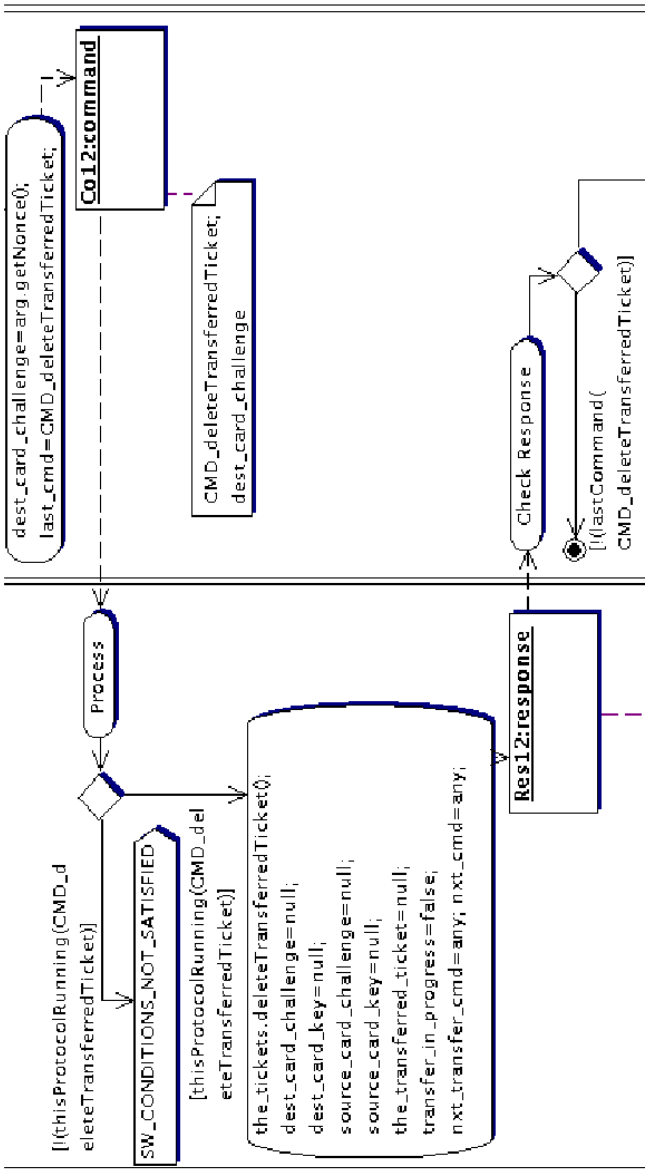


Figure 4.56: *Transfer Ticket*-protocol Part XIII

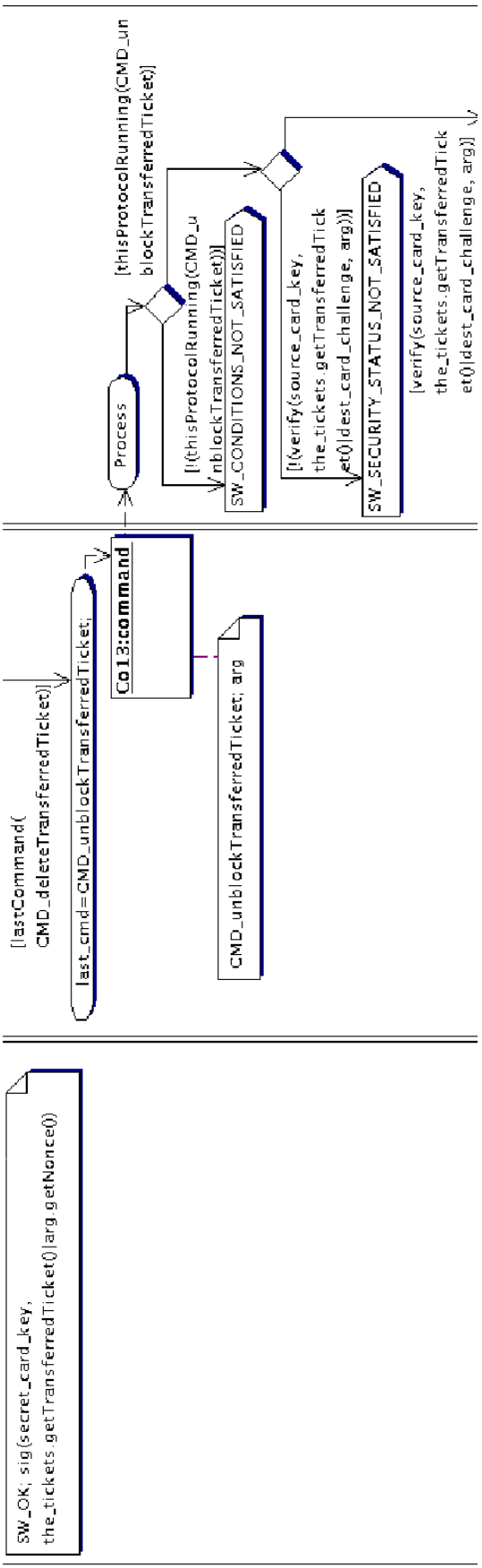


Figure 4.57: *Transfer Ticket*-protocol Part XIV

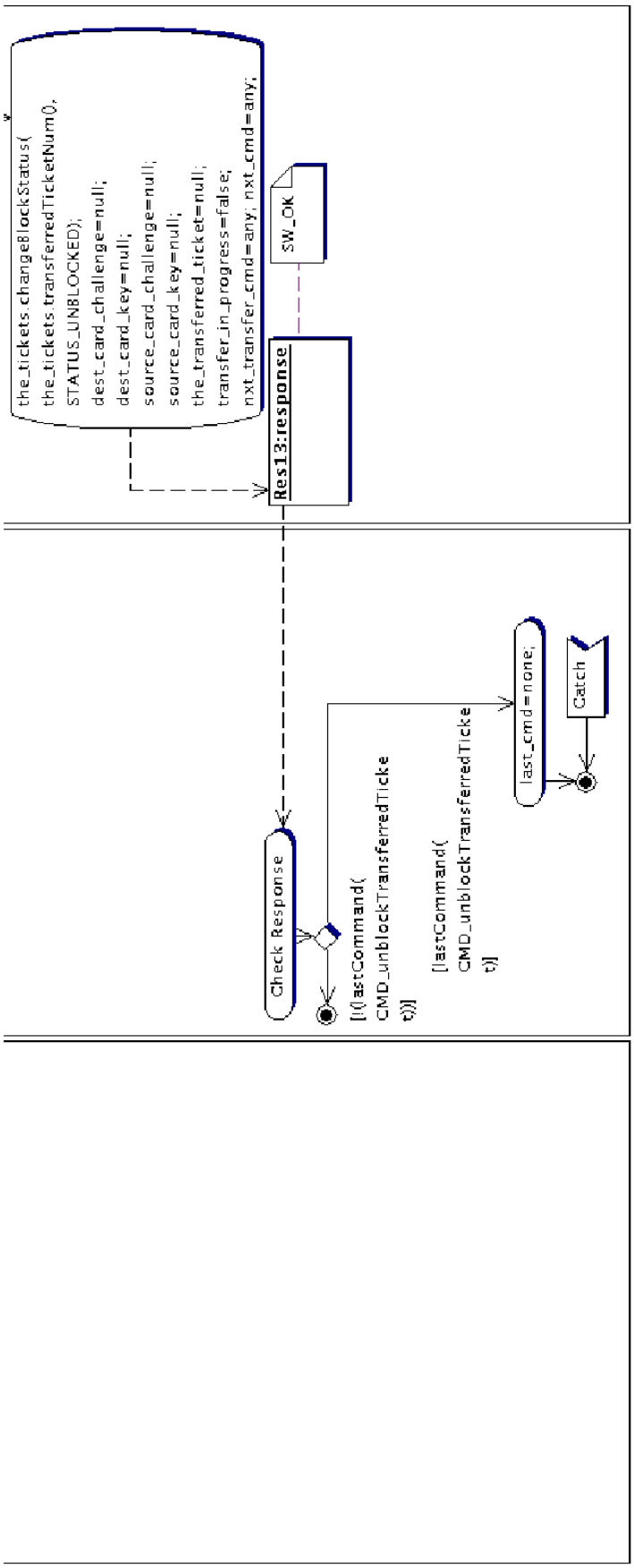


Figure 4.58: *Transfer Ticket*-protocol Part XV

4.18 View Checks

4.18.1 Purpose

This protocol is used by the terminals to get the checks of a ticket that is stored on the card.

4.18.2 Preconditions

The conductor or the user must have been authenticated using the *Authenticate Conductor*-protocol (Section 4.4) or the *Authenticate User*-protocol (Section 4.5) respectively. The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. If the user isn't authenticated the cardlet will only reveal the checks of tickets that aren't locked. If the checks of a locked ticket are requested and the user isn't authenticated, the card will answer with `SW_DATA_INVALID`. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.18.3 The protocol

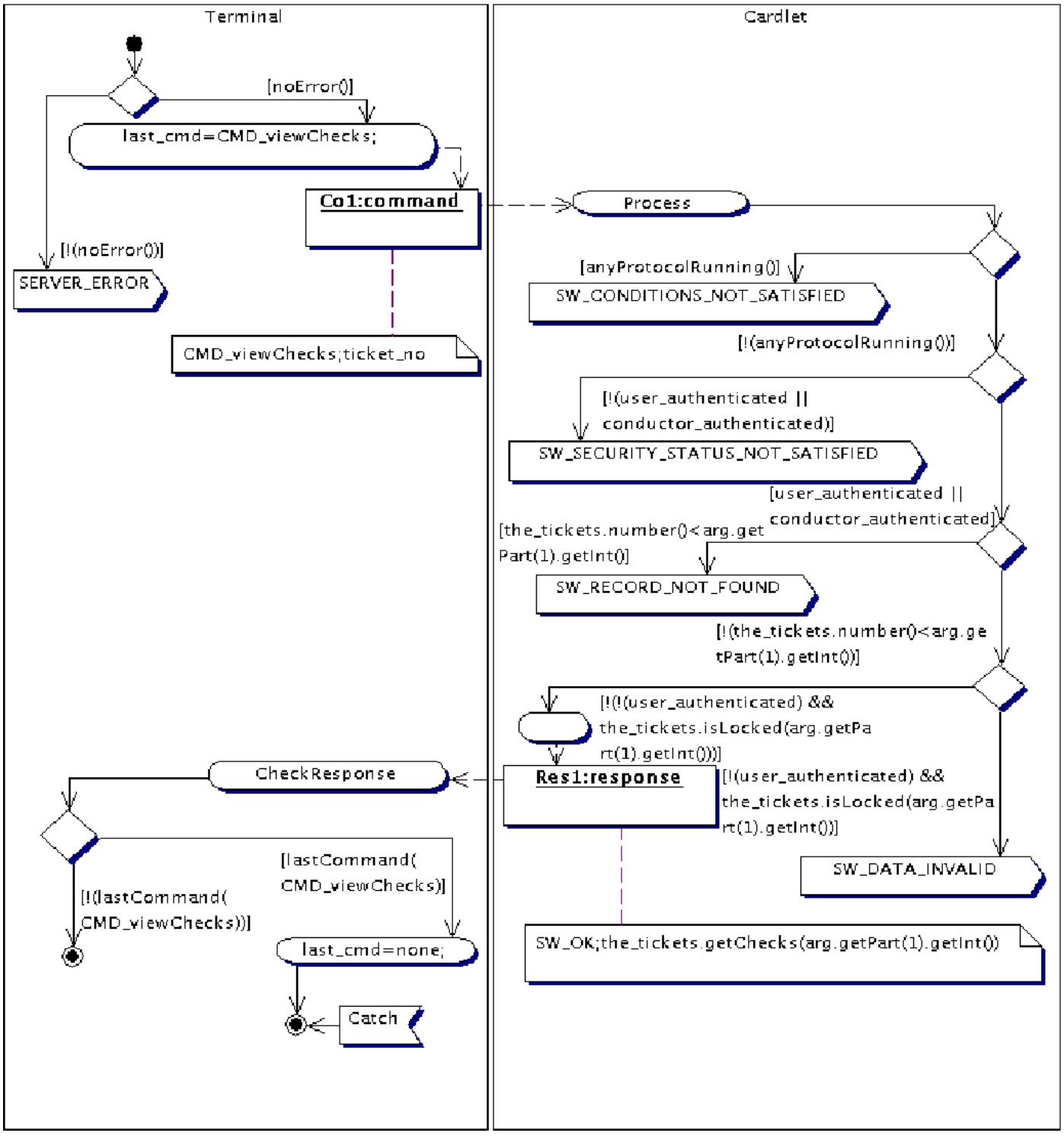


Figure 4.59: *View Checks*-protocol

4.19 View Ticket

4.19.1 Purpose

This protocol is used by the terminals to get the data and the status of a ticket that is stored on the card.

4.19.2 Preconditions

The conductor or the user must have been authenticated using the *Authenticate Conductor*-protocol (Section 4.4) or the *Authenticate User*-protocol (Section 4.5) respectively. The ticket number which is transferred to the card must be a correct ticket number, i. e. the transmitted number must be less or equal to the number of tickets currently stored on the card. If the user isn't authenticated the cardlet will only reveal tickets that aren't locked. If a locked ticket is requested and the user isn't authenticated, the card will answer with `SW_DATA_INVALID`. There mustn't be any other protocol running on the card, i. e. the `nxt_cmd`-field of the cardlet must be `any`.

4.19.3 The protocol

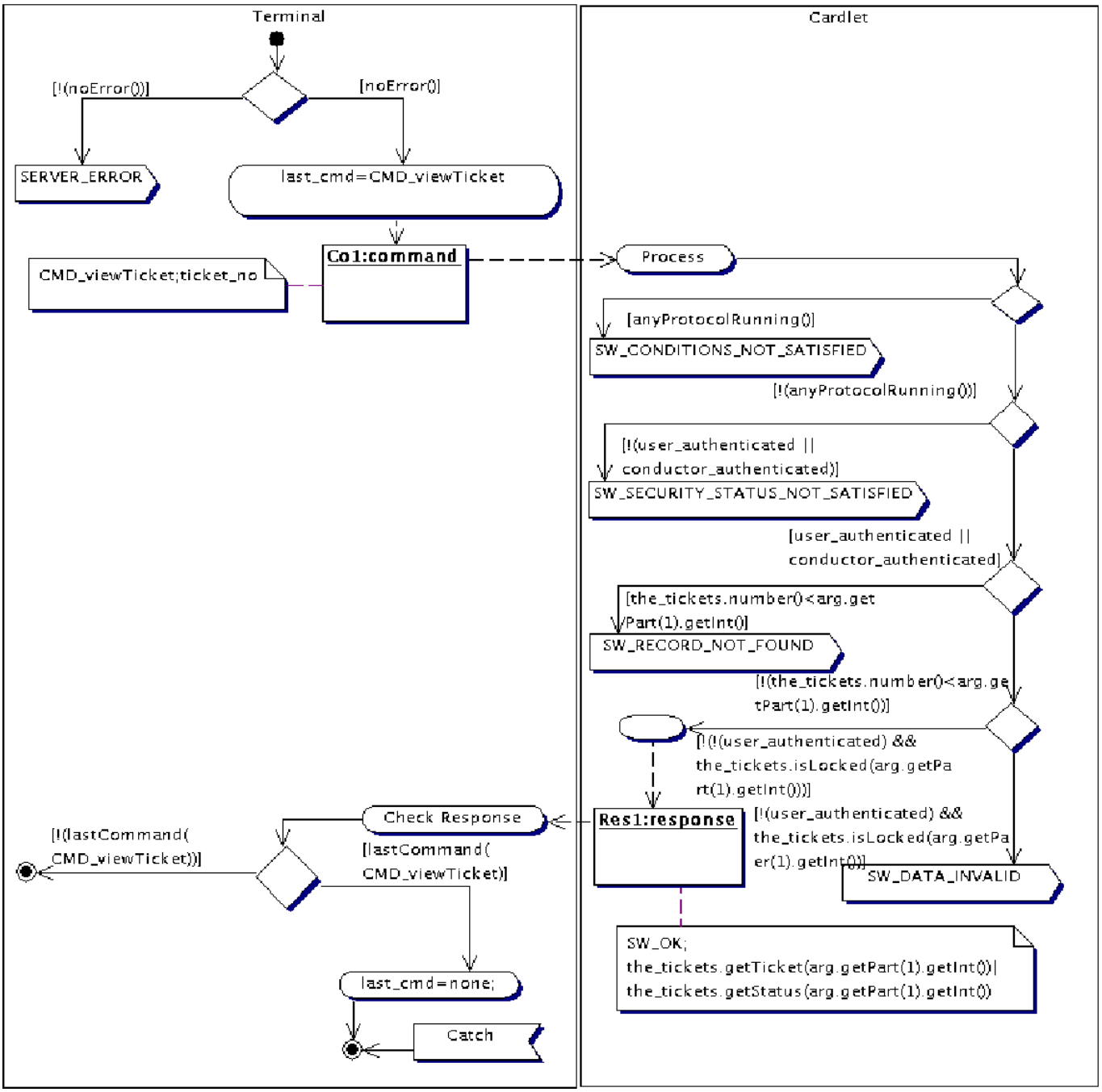


Figure 4.60: *View Ticket*-protocol

Bibliography

[AN95] R. Anderson and R. Needham. Programming satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*. Springer LNCS 1000, 1995.

[Han01] D. Haneberg. electronic ticketing – a case-study. Technical Report 9, Institut für Informatik, Universität Augsburg, http://www.informatik.uni-augsburg.de/forschung/techBerichte/reports/2001-9.ps.gz, 2001.

[HRS02] D. Haneberg, W. Reif, and K. Stenzel. A Method for Secure Smartcard Applications. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, Proceedings AMAST 2002*. Springer LNCS 2422, 2002.

[OMG99] The Object Management Group (OMG). *OMG Unified Modeling Language Specification*, 1999. http://www.omg.org/technology/uml.

[Pau98] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.