



The United Nations  
University

**UNU/IIST**

International Institute for  
Software Technology

---

# Formal Development of A Toll Way Control System

---

Mia Indrika

31 August 1995

## UNU/IIST

UNU/IIST enables developing countries to attain self-reliance in software technology by: (i) their own development of high integrity computing systems, (ii) highest level post-graduate university teaching, (iii) international level research, and, through the above, (iv) use of as sophisticated software as reasonable.

UNU/IIST contributes through: (a) advanced, joint industry-university advanced development projects in which rigorous techniques supported by semantics-based tools are applied in case studies to large scale software developments, (b) own and joint university and academy institute research in which new techniques for application domain and computing platform modeling, requirements capture, software engineering and programming are being investigated, (c) advanced, post-graduate and post-doctoral level courses which typically teach Design Calculi oriented software development techniques, (d) events [panels, task forces, workshops and symposia], and (e) dissemination.

Application-wise, the advanced development projects presently focus on software to support large-scale infrastructure systems such as railways, manufacturing industries, health care systems, etc., and are thus aligned with UN and International Aid System concerns. The research projects parallel and support the advanced development projects.

At present, the technical focus of UNU/IIST in all of the above is on applying, teaching, researching, and disseminating Design Calculi oriented techniques and tools for trustworthy software development. UNU/IIST currently emphasizes techniques that permit proper development steps and interfaces. UNU/IIST also endeavours to promulgate sound project and product management principles.

UNU/IIST's primary dissemination strategy is to act as a clearing house for reports from research and technology centres in industrial countries to industries and academic institutions in developing countries. At present more than 175 institutions worldwide contribute to UNU/IIST's report collection while UNU/IIST at the same time subscribes to more than 125 international scientific and technical journals. Information on reports received (and produced) and on journal articles is to be disseminated regularly to developing country centres — which are then free to order a reasonable number of report and article copies from UNU/IIST.

Dines Bjørner, Director

UNU/IIST Reports are either *R*esearch, *T*echnical, *C*ompendia or *A*dministrative reports:

$\mathcal{R}$  Research Report •  $\mathcal{T}$  Technical Report •  $\mathcal{C}$  Compendium •  $\mathcal{A}$  Administrative Report



The United Nations  
University

**UNU/IIST**

**International Institute for  
Software Technology**

P.O. Box 3058  
Macau

---

# Formal Development of A Toll Way Control System

---

**Mia Indrika**

## **Abstract**

In this report we present development of A Toll Way Control System by step-wise refinement. We also report the significant changes which have been made during the development. This work is carried out using the RAISE specification language and its associated method. Familiarity with RAISE or at least an ability to read it, is assumed

Mia Indrika was a fellow of UNU/IIST during the period 7 February 1995 to 31 August 1995, on leave from the Faculty of Computer Science, University of Indonesia, where she is a lecturer. Her current research interest is Formal Development of Software. E-mail: mia@cs.ui.ac.id

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What is being presented . . . . .	1
1.2	Motivation . . . . .	1
<b>2</b>	<b>The System under Consideration</b>	<b>2</b>
2.1	A Toll Way System . . . . .	2
2.2	A Toll Way Control System . . . . .	3
2.2.1	Functional Requirements . . . . .	3
2.2.2	Physical component requirements . . . . .	3
2.2.3	System components . . . . .	5
<b>3</b>	<b>Development</b>	<b>7</b>
3.1	Initial specifications: Properties . . . . .	8
3.1.1	Description . . . . .	8
3.1.2	Validation . . . . .	14
3.2	Second specification: Booth . . . . .	15
3.2.1	Description . . . . .	15
3.2.2	Validation . . . . .	19
3.2.3	Verification . . . . .	20
3.3	Third specification: Components . . . . .	20
3.3.1	Description . . . . .	20
3.3.2	Validation . . . . .	25
3.3.3	Verification . . . . .	26
3.4	Final specification: Imperative Concurrent . . . . .	27
3.4.1	Description . . . . .	27
3.4.2	Validation . . . . .	32
3.4.3	Verification . . . . .	32
<b>4</b>	<b>From Design to Implementation</b>	<b>32</b>
4.1	Concurrency aspects . . . . .	33
4.2	Data and function aspects . . . . .	33
<b>5</b>	<b>Conclusion</b>	<b>34</b>
5.1	Summary . . . . .	34
5.2	Future Work . . . . .	35
<b>6</b>	<b>Acknowledgements</b>	<b>35</b>
<b>A</b>	<b>Specifications</b>	<b>36</b>
A.1	Initial Specification: Properties . . . . .	36
A.2	Second Specification: Booth . . . . .	39
A.3	Third Specification: Components . . . . .	44
A.4	Final Specification: Imperative Concurrent . . . . .	56

**B Justifications**

**67**

# 1 Introduction

## 1.1 What is being presented

In this report we present a step-wise development of a Toll Way Computing System — from an abstract applicative description to an imperative concurrent description. More precisely, in section 3.1 we present an abstract specification, and in section 3.2 we describe the development to the next level which is more concrete. In section 3.3 a decomposition of the specification is presented. Finally, we present an imperative concurrent specification in section 3.4. The development takes several iterations before a specification is settled upon. In this report we also present the significant changes which have been made during the development.

A brief description of the system is given in the section 2. For simplicity, the specification developed here only covers basic features of the toll way system. Also, in this specification we do not address concerns such as hardware malfunctions.

## 1.2 Motivation

The development as presented serves:

- To understand an existing systems in precise formal terms. Of course, in the short time and with the limited resources available, we do not claim to formally describe exactly the “real” system.
- To let the development techniques serve as a basis for new software development.
- To show that formal methods can be applied to real applications. That is, to help convince industry of the benefits that can be gained by using formal methods.
- As a case-study to serve as a basis for asking and solving interesting computing science questions such as:
  - One set of such questions pertains to the actual development: Is it correct?
  - Another set of questions arise as novel form of implementations are discovered. For example, what may appear as a global state (i.e. the total of all plazas) may be decomposed into a set of processes, and many exciting decomposition rules may be discovered. An example such as the Toll Way System offers the chance for the computing scientist to discover such new ideas.
- As a more complete case-study in the future to serve as a basis for student lecture notes in a software development course.

## 2 The System under Consideration

In this section we first give a synopsis-like informal description at an abstract level of a toll way system. This description is followed by a rather more concrete informal description of a toll way computing system. Our development technique is to provide software for the control of the latter. This we will do by providing a careful step-wise development from a formal description of the abstract level to the formal description of the latter.

### 2.1 A Toll Way System

Before a description of a toll way control system is given, we first explain the concept of a toll way system briefly. A toll way consists of a set of entry plazas, a set of exit plazas and a fare schedule. Cars may enter only at entry plazas and leave only at exit plazas. A fare schedule associates a fee structure to each pair of entry and exit plaza between which a car can travel. A fee structure associate a fee to each vehicle type and time period of day. (We could instead of the above single fare schedule, associate a set of fare schedules with a toll way system, namely one for each exit plaza. An exit plaza fare schedule associate a fee structure to each entry plaza from which a car may come. The two kinds of fare schedules are, however, isomorphic).

An entry plaza is connected with an exit plaza by a road. A car thus arrives at the toll way at one of its entry plazas from which it will travel along a road leading to an exit plaza where the car driver will pay a toll according to the fare schedule.

To familiarize us with the notion of how roads relates to a toll way system, an example of the roads are shown by figure 1. We have labeled the entry plazas by  $y_1, y_2$  and exit plazas by  $x_1, x_2, x_3$ . From the figure we can see five roads:  $r_1 : (y_1, x_1)$ ,  $r_2 : (y_1, x_2)$ ,  $r_3 : (y_1, x_3)$ ,  $r_4 : (y_2, x_1)$ , and  $r_5 : (y_2, x_2)$ . So, if a car arrives at entry plaza  $y_2$ , it may travel along road  $r_4$  or  $r_5$

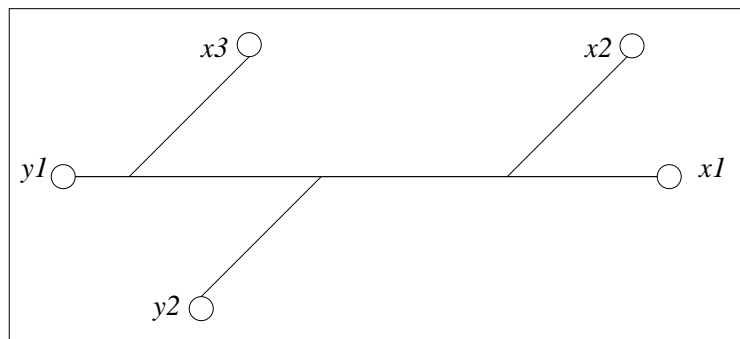


Figure 1. Roads

## 2.2 A Toll Way Control System

The toll way control system which is considered here is the system designed by the Toll Way Company of Indonesia. Below we present the functional requirements followed by a brief description of the system.

### 2.2.1 Functional Requirements

Based on the document [6] we derive the main objectives of the system, which are

1. to collect correct toll fees which are in accordance to a fare schedule
2. to manage the monies correctly, securing particularly that the monies received by an exit plaza equals the monies delivered by the car drivers leaving at that exit plaza
3. to record information about all cars coming through any plaza
4. to control cars entering and leaving the toll way (cars must follow specified entry and exit procedures)
5. to give service to the car driver; by service we mean providing facilities to the car driver passing through a plaza, e.g. by giving a card, a receipt, a sign that indicates a booth is open, etc

The following informal description of the system is based on the informal specification given in the Indonesian Language [6] and flow charts [7]. Each plaza has one or more booths through which cars enter or leave a plaza. When a car arrives at an entry plaza, arrival information is created and maintained by the entry plaza; a card which contains the arrival information is given to car driver. At an exit plaza, the user returns the card and pays a toll which is calculated based on which entry plaza the car comes from and on the category of car; then exit information is created. This exit information is printed on the card and otherwise maintained by the exit plaza.

Furthermore, at every plaza, there are supervisors and operators. A supervisor's duty is to supervise operators including opening and closing booths. An operator is placed in every open booth, operates the equipment and gives service to car driver.

### 2.2.2 Physical component requirements

In this section we explain the physical component requirements in term of how the Indonesian toll way system is implemented.

1. prevent cheating committed by operators
2. record transactions without “significant” error
3. continuous operation with least possibility of damage
4. mistakes in device handling should not result in fatal damage
5. components and their sub-systems must adhere to existing industrial standards and open system architecture
6. components must be modular and use standard bus, the connection between modules must use an existing standard interface
7. devices should be able to be operated 24 hours per day continuously
8. Indonesian products should be use as far as possible
9. software for the components must be portable

Both functional and physical requirements give us direction to the development. Since we will develop the system in terms of software aspects, we will consider only the software requirements.

Requirements 1, 2, 4, and 9 involve software problems. These four requirements are not clearly described, they are too vague, for example, we do not know exactly what “cheating” and “portable” are. Nevertheless, we can give an interpretation to these requirements.

In requirement 9, “portable” to us means that the software must run on many computing platforms. Requirements for portability would not be meaningful if the range of possible computing platforms is not given. We delegate the responsibility of portability to the implementor.

Requirement 4 concerns mistakes in device handling. In this work we only define what we consider to be “normal” cases. Nevertheless, handling mistakes can likewise be formally specified.

We think that “cheating” in requirement 1 involves money (fraud). An operator can commit fraud in many possible ways, for example:

- overcharge the car driver, operator keeps the difference for him/herself
- keeps the payment of car driver for him/herself without recording it
- steals booth’s money

Overcharging can be prevented by making the driver aware of the fee schedule toll, for example by issuing of receipt or displaying the toll. One possibility to avoid the second is by calculating and recording the toll automatically with minimum interaction of an operator. The third is more difficult to prevent, we propose possible solutions such as recording the identifier of an operator

and recording all transactions. In general, requirement 1 states that the monies received by an exit plaza equals the monies delivered by car drivers leaving that plaza.

Requirement 2 requires that all transactions are to be recorded correctly.

### 2.2.3 System components

Based on the above physical requirements, the Toll Way Company decided to design the system as composed of the following physical components (electro-mechanical devices):

1. Toll Collector's Terminal (TCT)
2. Badge Card Terminal (BCT)
3. Duty Card Terminal (DCT)
4. Pre-Paid Card Terminal (PPCT)
5. Integrated Optical Beam and Loop Coil Sensor (IOL)
6. Plaza Computer System (PCS)
7. Plaza Computer Administration (PCA)
8. Printer (LPR)
9. Barrier (LCB)
10. Upper Traffic Light (UPL)
11. Under Traffic Light (UNL)
12. Fare Display (FD)
13. Violation Alarm (VA)
14. Vehicle Class Display (VCD)

Figure 2 describes physical connection between all components. Each box represents a component and a line between two boxes means there is a data exchange between the two components and/or control from one to another component.

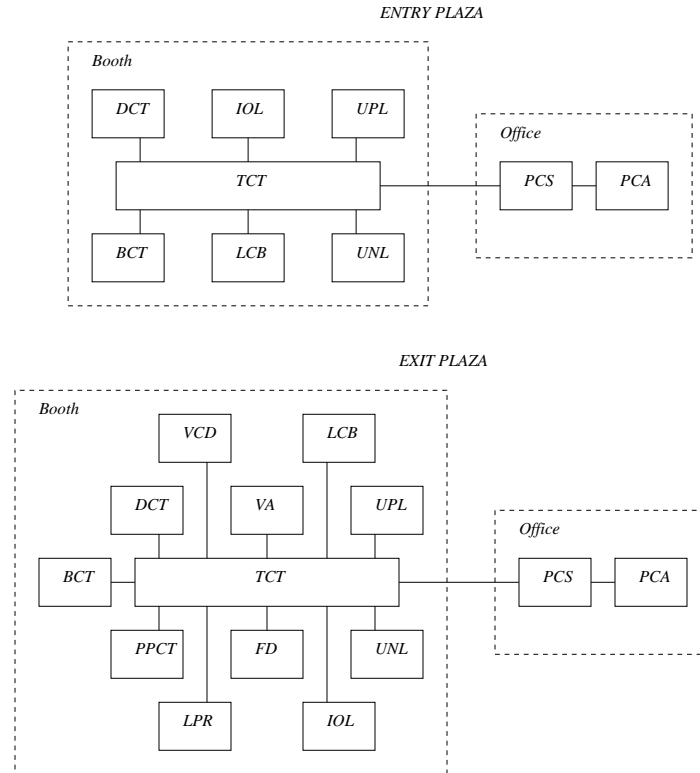


Figure 2. Physical connection between components

The function of each component is described in the following.

**TCT** is a main device in a booth. Its main functions are collecting transaction data and controlling other components.

**BCT** is a motorized card reader. Its functions are reading operator and supervisor identification cards.

**LPR** prints receipt and report.

**DCT** At entry plaza, DCT produces a card. It encodes, on the card, information given by TCT concerning when and from where the car entered toll way and category of car. At exit plaza, DCT decodes and reads information from a card, sends the information to TCT, then encodes exit information from TCT on the card.

**PPCT** is a motorized card reader. PPCT has functions to read the encoded balance from Pre-Paid Card, encode the new balance given by TCT and print it on the card. Pre-Paid Card is a magnetic card that has value; it can be used for payment.

**IOL** is a sensor for detecting an incoming car. It is placed at a gate lane in the booth area, so that any incoming car will reach the sensor. IOL will send a signal to TCT whenever it detects a car.

**FD** displays the toll that a user has to pay.

**UPL** is controlled manually via TCT by an operator. It gives a sign to the user whether a booth is open. Green light indicates a booth is open and ready for service, red light indicates a booth is closed.

**UNL** is controlled manually via TCT by an operator. Green light indicates that a transaction with a car is finished, hence the car can leave, and red light otherwise.

**VIA** gives sign if there is a transaction violation, that is, there is no prior transaction but a car is passing IOL.

**VCD** displays category of car (indicated by color).

**LCB** is a barrier; it is open when a booth lane is open, and it is closed when a booth lane is closed.

**PCS** provides information needed by a booth to open the booth such as fare schedule, time, etc. PCS also receives transaction records from TCT and stores them. If a report is requested, PCS will send the transaction data to PCA.

**PCA** receives transaction data from PCS. Request for a transaction report can be done from PCA.

### 3 Development

This section presents the specification in four step-wise development levels using the RAISE Specification Language (RSL) [5].

RAISE has a method [4] for developing such a system in a sequence of steps. One begins with a suitable abstract specification which is intended as a model of what the system is really to offer rather than implementation details. In each step, the developer decides which design decisions are to be made in order to eventually produce an efficient executable implementation, so the sequence of steps serves to produce such an implementation. Subsequent steps need be verified with a previous step and validated against requirements.

This approach is suitable for development of systems which has many components interacting with each other. If we start from the detailed level, that is, a specification that involves all the system components, it will be difficult to justify whether that specification is a model of the system as we intend it. We will follow the above method in developing the toll way control system.

The outline of the specific development for this case is roughly:

1. formulating an abstract specification by producing suitable types, function and axioms that capture the essential properties and requirements of the toll way system

2. developing a more concrete specification by introducing in this case the concept of booths, and by designing more concrete types and giving more concrete definitions of functions
3. developing an even more concrete specification by introducing several of the physical components, and also decomposing the specification such that its sub modules each model a physical component
4. introducing concurrency since the real system has concurrent nature; plaza office system and booth systems are running concurrently

We will describe each step of the development in the next sub-sections. Each sub-section will give (i) the aims of the development, (ii) specification interwoven with narrative descriptions, (iii) discussion of the choice of design decisions and other significant problems, and (iv) verification — checking that the specification is an implementation of the previous one — and validation — checking that the specification meets the requirement.

As mentioned earlier, for simplicity, the specification developed here only covers basic features of the toll way control system.

### 3.1 Initial specifications: Properties

#### 3.1.1 Description

At this first level, we formulate a suitable abstract specification that captures the essential properties of the toll way system; it is the basis for the future development.

Writing an initial specification — an abstract one — is a hard task. Our experience was that during the development of a more detailed specification we found that the initial one was not appropriate, so we had to backtrack and write a new one. Below we present and comment on both the initial and final top level abstractions.

In the first attempt, we viewed a toll way system as containing:

**Plazas** which delimit a road in a toll way and contain booths.

**Cars** with drivers possessing money.

**Money** from car drivers delivered to (a booth of) a plaza. The amount paid is a function of entry and exit plaza, category of the car, and time period of day.

**Records** are created as a car enters an entry plaza or leaves an exit plaza.

From those entities we define signatures of main functions which express properties of the system.

```

type Plaza, Records, Car, Money = Int

value
  car_money: Car → Money,

  booths_money: Plaza → Money,

  charge: (Car × Plaza) × Plaza → Money,

  records: Plaza → Records

```

For every car entering and leaving a plaza, a record is created and added to the records at the plaza.

```

value
  enter_plaza: Car × Plaza  $\xrightarrow{\sim}$  Car × Plaza,

  leave_plaza: (Car × Plaza) × Plaza  $\xrightarrow{\sim}$  Car × Plaza,

  add_rec_entry: Car × Plaza × Records  $\xrightarrow{\sim}$  Records,

  add_rec_exit: (Car × Plaza) × Plaza × Records  $\xrightarrow{\sim}$  Records

```

Of course, a car can only enter or leave a plaza if the plaza contains open booths, hence partiality of function is required.

```

value
  has_open_booth: Plaza → Bool,

  can_enter_plaza: Car × Plaza → Bool
  can_enter_plaza(c, entry)  $\equiv$  has_open_booth(entry),

  can_leave_plaza: (Car × Plaza) × Plaza → Bool
  can_leave_plaza((c, entry), exit)  $\equiv$  has_open_booth(exit)

```

The above does not constitute a sufficiently complete description of the toll way system. We now express properties of the systems in the form of axioms:

- When a car enters then it delivers no money.

**axiom**

```
[money_enter]
  ∀ c : Car, entry : Plaza •
    let (c', entry') = enter_plaza(c, entry) in
      car_money(c') = car_money(c) ∧
      booths_money(entry') = booths_money(entry)
    end ≡
      true
  pre can_enter_plaza(c, entry)
```

- When a car leaves plaza then it delivers money.

**axiom**

```
[money_leave]
  ∀ c : Car, entry, exit : Plaza •
    let (c', p') = leave_plaza((c, entry), exit) in
      let pay = charge(c, entry, exit) in
        car_money(c') = car_money(c) - pay ∧
        booths_money(p') = booths_money(exit) + pay
      end
    end ≡
      true
  pre can_leave_plaza((c, entry), exit)
```

- When a car enters or leaves, records at the plaza are updated. This is done by adding record of the car to plaza's record.

```
[records_enter]
  ∀ c : Car, entry : Plaza •
    let (c', entry') = enter_plaza(c, entry) in
      records(entry') = add_rec_entry(c, entry, records(entry))
    end ≡
      true
  pre can_enter_plaza(c, entry),
```

```
[records_leave]
  ∀ c : Car, entry, exit : Plaza •
    let (c', p') = leave_plaza((c, entry), exit) in
      records(p') = add_rec_exit((c, entry), exit, records(exit))
    end ≡
      true
  pre can_leave_plaza((c, entry), exit)
```

In this first attempt initial specification, we do not see *card*. The reason for being that, we planned to introduce *card* when we arrive at a more detailed specification, as a component of

the *booth*. We initially didn't consider the fact that the card will be given to users at an entry plaza and the users will carry it along until they arrive at an exit plaza where the card is going to be returned. If we introduce the card in the booth system we can not express that a car has a card when it arrives at an exit booth.

Actually, we could solve this problem by considering card as an attribute of car, but it doesn't look natural, since a car has not had a card before it enters a toll way.

After type *Card* is added, the signatures of the new initial specification are as follows. Note that now we can calculate toll (function *charge*) and get entry information using data from the card; this is what actually is done in a real system.

```

type Plaza, Records, Car, Card, Money = Int

value
  enter_plaza : Car × Plaza  $\xrightarrow{\sim}$  (Car × Card) × Plaza,

  leave_plaza : (Car × Card) × Plaza  $\xrightarrow{\sim}$  (Car × Card) × Plaza,

  booths_money : Plaza → Money,

  add_rec_entry : Car × Plaza × Records  $\xrightarrow{\sim}$  Records,

  add_rec_exit : (Car × Card) × Plaza × Records  $\xrightarrow{\sim}$  Records,

  records : Plaza → Records,

  charge : (Car × Card) × Plaza → Money,

  has_open_booth : Plaza → Bool,

  can_enter_plaza : Car × Plaza → Bool
  can_enter_plaza(c, entry)  $\equiv$  has_open_booth(entry),

  can_leave_plaza : (Car × Card) × Plaza → Bool
  can_leave_plaza((c, dc), exit)  $\equiv$  has_open_booth(exit)

```

Another problem concerns the properties of the system. In the *money\_leave* axiom, we express that when a car leaves, the money funds of the car will decrease and the money funds of the plaza will increase by the amount of toll. In real life, however, sometimes a car “escapes” without paying the toll. To consider this situation we changed the *money\_leave* axiom.

In the first attempt we solved it by adding a function *trans\_status*. Function *trans\_status* explains the transaction status of a car; *no\_trans* means there is no transaction, that is, a car “escapes”

without giving back the card and paying the toll at an exit plaza, or without getting a card at an entry plaza.

```

type Trans_Status == no_trans | trans

value trans_status : Car → Trans_Status

axiom
  [money_leave]
  ∀ c : Car, entry, exit : Plaza •
    let (c', p') = leave_plaza((c, entry), exit) in
      if trans_status(c) = no_trans then
        car_money(c') = car_money(c) ∧
        booths_money(p') = booths_money(exit)
      else
        let pay = charge(c, entry, exit) in
          car_money(c') = car_money(c) - pay ∧
          booths_money(p') = booths_money(exit) + pay
        end
      end
    end ≡
      true
    pre can_leave_plaza((c, entry), exit)

```

When we arrived at the detailed level, we found that it was still not appropriate. We had difficulty in separating records of legally passing cars and of illegal ones. An illegal passing record is created when a car passes a sensor but there is no prior transaction. A legal passing record is created after transaction with a car is finished. Both legal and illegal passing records are sent to the plaza as soon as the sensor detects a car.

To solve this problem we decided to have separate function for legal passing and illegal passing, both at entry plaza and at exit plaza. In addition, we also add *plaza\_type* and concern about *time*. The signatures of the new (final) abstract specification are therefore as follows.

```

type
  Plaza,
  Records,
  Car,
  Card,
  Money = Int,
  Time = Nat,
  Plaza_type == entry | exit | both

```

**value**

legal\_enter\_plaza :

$$\text{Car} \times \text{Plaza} \times \text{Time} \xrightarrow{\sim} (\text{Car} \times \text{Card}) \times \text{Plaza},$$

illegal\_enter\_plaza :

$$\text{Car} \times \text{Plaza} \times \text{Time} \xrightarrow{\sim} (\text{Car} \times \text{Card}) \times \text{Plaza},$$

legal\_leave\_plaza :

$$\begin{aligned} &(\text{Car} \times \text{Card}) \times \text{Plaza} \times \text{Time} \xrightarrow{\sim} \\ &(\text{Car} \times \text{Card}) \times \text{Plaza}, \end{aligned}$$

illegal\_leave\_plaza :

$$\begin{aligned} &(\text{Car} \times \text{Card}) \times \text{Plaza} \times \text{Time} \xrightarrow{\sim} \\ &(\text{Car} \times \text{Card}) \times \text{Plaza}, \end{aligned}$$

add\_legal\_entry\_rec :

$$\text{Car} \times \text{Plaza} \times \text{Time} \times \text{Records} \xrightarrow{\sim} \text{Records},$$

add\_illegal\_entry\_rec :

$$\text{Car} \times \text{Plaza} \times \text{Time} \times \text{Records} \xrightarrow{\sim} \text{Records},$$

add\_legal\_exit\_rec :

$$(\text{Car} \times \text{Card}) \times \text{Plaza} \times \text{Time} \times \text{Records} \xrightarrow{\sim} \text{Records},$$

add\_illegal\_exit\_rec :

$$(\text{Car} \times \text{Card}) \times \text{Plaza} \times \text{Time} \times \text{Records} \xrightarrow{\sim} \text{Records},$$

records : Plaza  $\rightarrow$  Records,

money : Plaza  $\rightarrow$  Money,

plaza\_type : Plaza  $\rightarrow$  Plaza\_type,

booths\_money : Plaza  $\rightarrow$  Money,

charge : (Car  $\times$  Card)  $\times$  Plaza  $\times$  Time  $\rightarrow$  Money,

car\_money : Car  $\rightarrow$  Money,

has\_open\_booth : Plaza  $\rightarrow$  **Bool**,

can\_enter\_plaza : Car  $\times$  Plaza  $\rightarrow$  **Bool**

can\_enter\_plaza(c, p)  $\equiv$

has\_open\_booth(p)  $\wedge$

(plaza\_type(p) = entry  $\vee$  plaza\_type(p) = both),

```

can_leave_plaza : (Car × Card) × Plaza → Bool
can_leave_plaza((c, dc), p) ≡
  has_open_booth(p) ∧
  (plaza_type(p) = exit ∨ plaza_type(p) = both)

```

Furthermore, now we have axioms concerning money and records for each passing either at an entry or at an exit plaza. For example, *money\_legal\_leave* and *records\_legal\_enter* axioms are as follows.

```

[money_legal_leave]
∀ c : T.Car, dc : T.Card, p : Plaza, t : T.Time •
  let
    ((c', dc'), p') = legal_leave_plaza((c, dc), p, t),
    pay = charge((c, dc), p, t)
  in
    T.car_money(c') = T.car_money(c) - pay ∧
    booths_money(p') = booths_money(p) + pay ∧
    money(p') = money(p)
  end ≡
  true
  pre can_leave_plaza((c, dc), p),

[records_legal_enter]
∀ c : T.Car, p : Plaza, t : T.Time •
  let ((c', dc), p') = legal_enter_plaza(c, p, t) in
    records(p') = add_legal_entry_rec(c, p, t, records(p))
  end ≡
  true
  pre can_enter_plaza(c, p)

```

The complete initial specification can be found in appendix A.1. We decided to put only the type of interest *Plaza* and main functions in the *A-TW-0* module and put other types and functions in a subsidiary type module called *TYPES*. We then made a global object *T* from *TYPES*:

```
object T : TYPES
```

### 3.1.2 Validation

Validation means checking whether a specification meets the requirements. In section 2.2 we have a list of functional requirements and physical component requirements. Below we first

check each functional requirement then some physical requirements and we consider whether they are reflected in this initial specification or whether they will be dealt in later steps.

## Functional Requirements

The checking of requirements 1 (collect correct toll fee), 4 (control cars entering and leaving), and 5 (give service to the car driver) is deferred to later step.

Requirement 2, that is, correct management of monies, is covered by axioms *money\_legal\_enter*, *money\_illegal\_enter*, *money\_legal\_leave*, and *money\_illegal\_leave*.

Requirement 3, that is, recording of information about all cars coming through the plaza, is covered by axioms *records\_legal\_enter*, *records\_illegal\_enter*, *records\_legal\_leave*, and *records\_illegal\_leave*. Requirements 1, 2, 4, and 5 will give direction in deciding what other information should be recorded.

In the case that the requirements can not be seen immediately from the specification, we can formulate them as a theorem and justify it.

## Physical Requirements

Based on the reason we have explained earlier we consider only two requirements, requirements 1 (no cheating by operators) and 2 (all transactions recorded). We will later record amounts of money in the records (with new kinds of record entry). We therefore defer checking requirements 1 and 2.

## 3.2 Second specification: Booth

### 3.2.1 Description

At this level, we describe the system in more detail by introducing the concept of *booth* since a car passes a plaza through one of its open booths. In addition to the types in previous specification, we have types *Booth* to denote a single booth and *Booths* to denote a finite collection of open booths.

It is natural to use a map to express *Booths* as a collection of *Booth* since each booth has a unique index (*BI*). To state that the number of booths is finite, an axiom *booths\_finite* is needed. We also need to ensure that given an identification from domain of *Booths*, it will give a unique booth (*booth\_convergent* axiom).

```

type
  Booth,
  /* Booth's index */
  BI,
  Booths = BI  $\overrightarrow{m}$  Booth

axiom
  /* BI is finite */
  [booths_finite] card { i | i : BI } post true,

  /* Booths is convergent for arguments in domain */
  [booths_convergent]
   $\forall i : BI, bs : Booths \bullet i \in \mathbf{dom} \ bs \Rightarrow (bs(i) \mathbf{post true})$ 

```

Furthermore, functions related to *Booth* and *Booths* are added. The following function *booths* returns the collection of the open booths of a plaza.

```

value
  booths : Plaza  $\rightarrow$  Booths

```

A car enters or leaves a plaza through one of its open booths, provided there is at least one open booth at that plaza. As in initial specification, we have also separate function for legal passing and illegal passing.

```

value
  legal_enter_booth :
    T.Car  $\times$  Plaza  $\times$  T.Time  $\xrightarrow{\sim}$  T.Car  $\times$  Booth  $\times$  BI,

  illegal_enter_booth :
    T.Car  $\times$  Plaza  $\times$  T.Time  $\xrightarrow{\sim}$  T.Car  $\times$  Booth  $\times$  BI,

  legal_leave_booth :
    (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\xrightarrow{\sim}$  T.Car  $\times$  Booth  $\times$  BI,

  illegal_leave_booth :
    (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\xrightarrow{\sim}$  T.Car  $\times$  Booth  $\times$  BI,

  can_enter_booth : Car  $\times$  Plaza  $\rightarrow$  Bool
  can_enter_booth(c, p)  $\equiv$ 
    has_open_booth(p)  $\wedge$ 
    (plaza_type(p) = T.entry  $\vee$  plaza_type(p) = T.both),

```

$$\begin{aligned} \text{can\_leave\_booth} &: (\text{Car} \times \text{Card}) \times \text{Plaza} \rightarrow \mathbf{Bool} \\ \text{can\_leave\_booth}((c, dc), p) &\equiv \\ &\text{has\_open\_booth}(p) \wedge \\ &(\text{plaza\_type}(p) = \text{T.exit} \vee \text{plaza\_type}(p) = \text{T.both}) \end{aligned}$$

Now, we give definition to the function *has\_open\_booth* which has not been defined yet in the previous level.

**value**

$$\begin{aligned} \text{has\_open\_booth} &: \text{Plaza} \rightarrow \mathbf{Bool} \\ \text{has\_open\_booth}(p) &\equiv \mathbf{dom} \text{booths}(p) \neq \{\} \end{aligned}$$

Booth keeps the monies delivered by car drivers. The function *booth\_money* extracts the money that a booth has. When a booth is closed, the booth's money is handed over to the plaza and added to plaza's money (*money*). We also give an explicit definition to function *booths\_money* from the previous level. This function extracts the money of all open booths.

**value**

$$\begin{aligned} \text{booth\_money} &: \text{Booth} \rightarrow \text{Money}, \\ \\ \text{sum\_money} &: \text{Booths} \rightarrow \text{Money}, \\ \\ \text{booths\_money} &: \text{Plaza} \rightarrow \text{Money} \\ \text{booths\_money}(p) &\equiv \text{sum\_money}(\text{booths}(p)), \\ \\ \text{money} &: \text{Plaza} \rightarrow \text{Money} \end{aligned}$$

**axiom**

$$\begin{aligned} &[\text{sum\_money\_ax}] \\ &\forall p : \text{Plaza}, b : \text{Booth}, bs : \text{Booths}, i : \text{BI} \bullet \\ &\quad bs = \text{booths}(p) \wedge i \in \mathbf{dom} \text{booths}(p) \Rightarrow \\ &\quad ( \\ &\quad \quad \text{sum\_money}(bs \uparrow [i \mapsto b]) \equiv \\ &\quad \quad \quad \text{sum\_money}(bs) - \text{booth\_money}(bs(i)) + \text{booth\_money}(b) \\ &\quad ) \end{aligned}$$

Function *sum\_money* counts up the monies of all open booths. We do not give an explicit definition to *sum\_money*, we give instead an implicit one in the form of an axiom.

Properties of the system are described in more detail by giving the properties of booth, plaza and the relation between them.

- Properties of the booth mainly concern money. When leaving, money from a car driver is delivered to the booth first and later handed over to the plaza.

**axiom**

[legal\_leave\_booth\_ax]

$$\forall$$

$$\begin{aligned} &c, c' : \text{T.Car}, \\ &dc, dc' : \text{T.Card}, \\ &p : \text{Plaza}, \\ &b : \text{Booth}, \\ &i : \text{BI}, \\ &t : \text{T.Time} \end{aligned}$$

•

$$\begin{aligned} &\text{can\_leave\_booth}((c, dc), p) \wedge \\ &((c', dc'), b, i) = \text{legal\_leave\_booth}((c, dc), p, t) \Rightarrow \\ &\text{T.car\_money}(c') = \text{T.car\_money}(c) - \text{charge}((c, dc), p, t) \wedge \\ &i \in \mathbf{dom} \text{booths}(p) \wedge \\ &\text{booth\_money}(b) = \\ &\text{booth\_money}((\text{booths}(p))(i)) + \text{charge}((c, dc), p, t), \end{aligned}$$

[illegal\_leave\_booth\_ax]

$$\forall$$

$$\begin{aligned} &c, c' : \text{T.Car}, \\ &dc, dc' : \text{T.Card}, \\ &p : \text{Plaza}, \\ &b : \text{Booth}, \\ &i : \text{BI}, \\ &t : \text{T.Time} \end{aligned}$$

•

$$\begin{aligned} &\text{can\_leave\_booth}((c, dc), p) \wedge \\ &((c', dc'), b, i) = \text{illegal\_leave\_booth}((c, dc), p, t) \Rightarrow \\ &\text{T.car\_money}(c') = \text{T.car\_money}(c) \wedge \\ &i \in \mathbf{dom} \text{booths}(p) \wedge \\ &\text{booth\_money}(b) = \text{booth\_money}((\text{booths}(p))(i)) \end{aligned}$$

These axioms describe the status of booth's money in different situations, namely car leaves a booth legally/illegally. Enter axioms are similar, but no money changes hands.

- Properties of plaza mainly concern records. As a car passes a plaza through one of its open booths, information about car is sent directly to the plaza office from the booth and records at the plaza office are extended with car's record. E.g. for legal entry, the axiom is as follows.

**axiom**

[legal\_enter\_plaza\_ax]

$$\forall$$

$$\begin{aligned}
& c, c', c'' : \text{T.Car}, \\
& dc, dc' : \text{T.Card}, \\
& p, p' : \text{Plaza}, \\
& b : \text{Booth}, \\
& i : \text{BI}, \\
& t : \text{T.Time} \\
& \bullet \\
& \text{can\_enter\_plaza}(c, p) \wedge \\
& ((c', dc), p') = \text{legal\_enter\_plaza}(c, p, t) \Rightarrow \\
& ((c'', dc'), b, i) = \text{legal\_enter\_booth}(c, p, t) \Rightarrow \\
& i \in \mathbf{dom} \text{booths}(p) \wedge \\
& c' = c'' \wedge \\
& dc = dc' \wedge \\
& \text{booths}(p') = \text{booths}(p) \uparrow [i \mapsto b] \wedge \\
& \text{money}(p') = \text{money}(p) \wedge \\
& \text{records}(p') = \text{add\_legal\_entry\_rec}(c, p, t, \text{records}(p)),
\end{aligned}$$

The only significant changes have been made during development of this level were with respect to the changes we made in initial specification. In the Appendix A.2, complete specification of this level can be found.

### 3.2.2 Validation

As we did in the previous step we check whether the requirements are reflected in this level.

### Functional Requirements

Requirements 2 (manage the monies correctly) and 3 (record all car information) are covered by the axioms concerning booth and plaza. Checking for requirements 1 (collect correct toll fees), 4 (control cars entering and leaving) and 5 (give service to the car driver) is deferred to the next level.

### Physical Requirements

Both checking for requirements 1 (no cheating by operators) and 2 (all transaction recorded) is deferred to the next level.

### 3.2.3 Verification

Verification means checking whether the development step is correct, that is, all the signatures of the previous level are in this new specification and also the axioms of the previous level are true. We first formulate a development relation  $A\_TW\_0\_1$  which asserts that  $A\_TW\_1$  implements  $A\_TW\_0$  and then justify it:

**justification** A\_TW\_0\_1\_J of A\_TW\_0\_1:  $\vdash A\_TW\_1 \preceq A\_TW\_0$  **end**

Proof obligations — the conditions should be justified — for this development relation which are generated by RAISE tool can be found in Appendix B.

## 3.3 Third specification: Components

### 3.3.1 Description

At this level, we develop more concrete specification by introducing several physical components. We decompose the specification into plaza office system ( $PLAZA\_BODY$  module) and booth system ( $BOOTH$  module), then decompose  $BOOTH$  module into system components modules:  $BCT$ ,  $DCT$ ,  $PPCT$ , and  $IOL$ . In addition, we also describe the system in more detail by giving more concrete types, explicit definitions to the functions and describing how the system and its sub-systems go together. We do not model other physical components such as LPR, FD, UPL, UNL, VIA, VCD, LCB; as they are trivial and can easily be modeled.

In relation to the system components described in section 2.2,  $PLAZA\_BODY$  can be seen as plaza office with PCS (Plaza Computer System) and PCA (Plaza Computer Administration), with  $BOOTH$  can be seen as TCT (Toll Collector's Terminal).

As in the previous level we have also a subsidiary module called  $TYPES1$  to put all the types that we want to use across modules together with useful functions on these types and instantiate it as a global object  $T1$ . Besides giving more concrete type to the previous types, in this type module we introduce type  $PP\_Card$  for pre-paid card that can be used for payment and also other types needed.

Again, the only significant changes we made during development of this level were with respect to the changes in initial specification.

### TYPES1 Module

$TYPES1$  is an extension (and hence an implementation) of  $TYPES$ ; it gives more concrete types

and provides new types and functions we need.

We could say that each car has a unique identification and carries with it an instrument for payment which could be in the form of either pre-paid card, cash or both. These instruments can be used by car drivers to pay a toll. Note that not all car drivers have a pre-paid card. To represent this condition, we use variant type *PP\_Card* containing constant *nil* — means no pre-paid card — constructor *ppc* for generating value of *PP\_Card*, and destructor *val* to extract the value. To cover two kinds of payment methods, that is, by pre-paid card first or by cash first, we also use variant type *Payment* containing two constants *card\_first* and *cash\_first*. Furthermore, function *charge* to calculate a toll is added. Toll is computed based on car id, entry plaza the car comes from, exit plaza and fare schedule.

**type**

Car\_id,

PP\_Card == nil | ppc(val : Money),

Car :: car\_id : Car\_id cash : Money pp\_card : PP\_Card

Payment == card\_first | cash\_first,

Opt\_plaza\_id == nil\_pid | mk\_pid(Plaza\_id)

**value**

car\_money : Car → Money

car\_money(c) ≡

cash(c) + if pp\_card(c) = nil then 0 else val(pp\_card(c)) end,

charge : (Car\_id × Opt\_plaza\_id) × Plaza\_id × Fares × Time → Money

Note that function *car\_money* is now defined explicitly. Money fund of a car driver is sum of cash money and value of pre-paid card.

Initially a card is empty, then entry and exit information will be printed on it later. We use combination of record and variant type as follows to represent this situation.

**type**

Card :: entry\_info : D\_Card exit\_info : D\_Card,

D\_Card == empty | mk\_info(pass\_info : Card\_Info),

Card\_Info ::

car\_id : Car\_id

plaza\_id : Opt\_plaza\_id

booth\_id : Booth\_id

```

    time : Opt_time,

    Opt_plaza_id == nil_pid | mk_pid(Plaza_id),

    Opt_time == nil_time | mk_time(Time)

```

Using variant types *Opt\_plaza\_id* and *Opt\_time* as shown above is a convenient way to cover the situation where we do not know *plaza\_id* and *time*; this will happen when a car runs away.

Records is a list of entry record, exit record, open booth record, or close booth record.

```

type
  Records = Record*,
  Record = Car_Entry_Rec | Car_Exit_Rec | Booth_Open_Rec | Booth_Close_Rec,

  Car_Entry_Rec = Legal_Entry_Rec | Illegal_Entry_Rec,
  Legal_Entry_Rec ::
    car_id : Car_id time : Time p_id : Plaza_id b_id : Booth_id,
  Illegal_Entry_Rec :: time : Time p_id : Plaza_id b_id : Booth_id,

  Car_Exit_Rec = Legal_Exit_Rec | Illegal_Exit_Rec,
  Legal_Exit_Rec ::
    car_id : Car_id
    time : Time
    p_id_enter : Opt_plaza_id
    p_id_leave : Plaza_id
    b_id : Booth_id
    car_pay : Money,
  Illegal_Exit_Rec :: time : Time p_id : Plaza_id b_id : Booth_id,

  Open_Booth_Rec :: booth_id : Booth_id open_info : Open_Info,
  Open_Info :: time : Time sup_id : ID session_no : Nat opr_id : ID,

  Close_Booth_Rec ::
    booth_id : Booth_id close_info : Close_Info booth_money : Money,
  Close_Info :: time : Time sup_id : ID

```

An entry record is either a legal entry record or an illegal entry record. An illegal entry record is created when there is no transaction, that is, when a car runs away. And so is an illegal exit record.

An opening record is created when a booth is open. It contains booth id and other information such as time, supervisor id, session number and operator id. Whereas a closing record contains booth id, total money, time and supervisor id.

When we design the entries of the records, we consider the requirements mentioned in the previous section. We record all the information which is needed to meet the requirements. For example, in order to be able to verify the money of a booth, we record each toll that a car driver should pay when the car leaves a booth legally.

Supervisor and operator each one have an identification card.

```

type
  Sup :: b_card : B_Card,
  Opr :: b_Card : B_Card,
  B_Card :: id : ID

```

### PLAZA\_BODY Module

This module contains more explicit functions definitions related to the plaza system and its interaction with the booth system.

Type *Plaza* now becomes more concrete, as a record containing plaza id, plaza type, fare schedule, collection of open booths, records and money. Functions *booths*, *records*, *money* and *plaza\_type* from the previous level now become fields of record *Plaza*.

Type *Booth* becomes more concrete too. It is therefore natural to introduce a module for booth system — *BOOTH* module — and put the concrete type of *Booth* in it. In *PLAZA\_BODY* module we instantiate *BOOTH* as *B*.

```

object B : BOOTH

type
  /* Booth's index */
  BI,
  /* Booth */
  Booth = B.Booth,
  /* Collection of open booths */
  Booths = BI  $\overrightarrow{m}$  Booth,

  /* Plaza */
  Plaza ::
    plaza_id : T1.Plaza_id
    plaza_type : T1.Plaza_type
    fares : T1.Fares
    booths : Booths
    records : T1.Records

```

```
plaza_money : T1.Money
```

**axiom**

```
/* BI is finite */
[booths_finite] card { i | i : BI } post true,

/* Booths is convergent for arguments in domain */
[booths_convergent]  $\forall i : BI, bs : Booths \bullet$ 
   $i \in \mathbf{dom} bs \Rightarrow (bs(i) \mathbf{post true})$ 
```

New functions *open\_booth* and *close\_booth* are introduced and defined explicitly. Those functions are needed here, because plaza is the one who decides which booth will be opened and closed by a supervisor. Opening and closing records of a booth are then sent to the plaza. Explicit definitions for the functions related to a plaza that still underspecified in the previous level such as *legal\_enter\_plaza*, *illegal\_enter\_plaza*, *legal\_leave\_plaza*, and *illegal\_leave\_plaza* are given by relating them to booth. The details can be seen in Appendix A.3.

## BOOTH Module

As mentioned above, functions in the *PLAZA\_BODY* module are defined by relating them to the booth system. This module thus contains functions that are needed by the plaza and defines the functions explicitly by relating them to other components, such as BCT, DCT, PPCT and IOL.

We know that a booth will send the records to plaza office when the booth is open or closed and when the car either enters or leaves toll way legally or illegally. Besides that, a booth keeps the money delivered by the user until the booth is closed. Therefore, we define functions *open\_booth*, *close\_booth*, *legal\_enter\_booth*, *illegal\_enter\_booth*, *legal\_leave\_booth* and *illegal\_leave\_booth* to describe these behaviors. We define these functions in term of the appropriate functions of component modules.

A complete specification of *BOOTH* can be found in Appendix A.3.

## Component Modules: BCT, DCT, PPCT, IOL

*BCT* module describes what BCT does. BCT takes part when a booth is open or closed, that is, when a supervisor or an operator inserts their identification cards. BCT will then read the card.

When a car arrives at an entry plaza, DCT produces a card which contains entry information of the car. DCT reads the card and adds exit information to it when a car arrives at an exit

plaza. These are described in functions *enter\_booth* and *leave\_booth* in *DCT* module.

PPCT is a pre-paid card reader, so it takes part when a car leaves a plaza and pays a toll with a pre-paid card. PPCT will read the balance and print the rest amount of the money after it is computed by TCT. Functions *get\_balance* and *change\_balance* in the *PPCT* module tell about these.

When a car leaves a booth, it will pass the sensor (IOL). Whenever the sensor detects a car has passed by but no transaction, such as payment or giving the card back to an operator, has been done before, it means the car leaves the booth illegally. This situation is modeled as follows. Initially the sensor is in *wait* state and it will be changed to *expect* state if a transaction is finished. So, if the sensor is in *wait* state, but it detects a car, that means the car leaves the booth illegally. Function *pass\_sensor* in *IOL* module will return passing status of the car, *illegal* or *ok*.

Complete specifications of *BCT*, *DCT*, *PPCT* and *IOL* modules are in Appendix A.3. Each module describes only assumption about the hardware.

### 3.3.2 Validation

#### Functional requirements

Requirement 1, to collect toll fees which are in accordance to a fare schedule, is covered by function *charge* in *TYPES1* module.

Functions *legal\_enter\_booth*, *illegal\_enter\_booth*, *legal\_leave\_booth* and *illegal\_leave\_booth* in the *BOOTH* module refer to the requirement 2, that is, correct management of monies.

Functions *legal\_enter\_plaza*, *illegal\_enter\_plaza*, *legal\_leave\_plaza* and *illegal\_leave\_plaza* in the *PLAZA\_BODY* module, *legal\_enter\_booth*, *illegal\_enter\_booth*, *legal\_leave\_booth* and *illegal\_leave\_booth* in the *BOOTH* module refer to requirement 3 (record all car information), 4 (control cars entering and leaving) and 5 (give service to the car driver). These functions describe the behavior of the system when a car passes through a booth of a plaza: what information is recorded, what procedures should be followed by a car driver and what facilities are provided to car drivers. So these requirements are met.

#### Physical requirements

From the previous section we gather some possibilities to meet requirement 1 (no cheating by operators). We will check whether the development of this specification meets the requirement.

To prevent an operator from keeping the payment without recording it, a toll is computed automatically by the booth system soon after an operator inserts the card of a car driver, information concerning payment then is stored automatically as *Legal\_Exit\_Rec* record. This record will be sent to the plaza after the car has passed a sensor. This behavior is modeled in function *legal\_leave\_plaza* in the *PLAZA\_BODY* module and *legal\_leave\_booth* in the *BOOTH* module.

In function *open\_booth* in the *BOOTH* module we can see that when a booth is opened an *Open\_Booth\_Rec* record is created; it contains amongst other things the id of operator. Functions *legal\_leave\_booth* and *illegal\_leave\_booth* describes that all transactions at an exit plaza are also recorded. By having this information, we can prevent an operator from stealing money.

Overcharging a car driver can be prevented by giving a receipt to car driver or displaying a toll. In this specification we did not model either printing a receipt or displaying a toll, but it can easily be inserted during translation.

Requirement 2, that is, all transactions are recorded correctly, is covered by functions *legal\_enter\_plaza*, *illegal\_enter\_plaza*, *legal\_leave\_plaza*, *illegal\_leave\_plaza* in the *PLAZA\_BODY* module and functions *legal\_enter\_booth*, *illegal\_enter\_booth*, *legal\_leave\_booth*, *illegal\_leave\_booth* in the *BOOTH* module.

### 3.3.3 Verification

We would like also to state and justify a development relation between *A\_TW\_1* and *PLAZA\_BODY*. We could not state this as

$$A\_TW\_1 \preceq PLAZA\_BODY$$

since *A\_TW\_1* hides functions that are not defined in *PLAZA\_BODY*. What we do instead is to show that an extension of *PLAZA\_BODY* named *PLAZA* — which defines all the hidden functions — implements *A\_TW\_1*. The complete *PLAZA* module can be found in Appendix C.3.

Then we formulate a development relation *A\_TW\_1\_PLAZA* which asserts that *PLAZA* implements *A\_TW\_1* and then justify it:

**justification** *A\_TW\_1\_PLAZA\_J* of *A\_TW\_1\_PLAZA* :  $\vdash PLAZA \preceq A\_TW\_1$  **end**

Appendix B gives the proof obligations for this development relation.

### 3.4 Final specification: Imperative Concurrent

#### 3.4.1 Description

In real life, plaza office system, booth systems and component systems run concurrently and communicate with each other, so it is natural to write a concurrent specification. From the previous level we have six applicative modules. We follow the RAISE method [4] to transform them into imperative concurrent ones.

RSL provides means for specifying concurrent systems by providing parallel combinators and communication primitives. A communication takes place through a channel. In general, an RSL expression that accesses a channel denotes a process. Process interaction in RSL is based on synchronous communication with message passing. RSL processes communicate using unidirectional channels, where the receiving process need not be known by the sending process; only the channel name is used [4]. During the communication, values can be passed in both directions by using the usual parameter passing mechanism and using access modes to channels: **in**, **out**, or **in** and **out**. In the remainder of this section we briefly explain how to transform the last specification into imperative concurrent one.

#### C\_PLAZA Module

*C\_PLAZA* module is imperative concurrent version of *PLAZA\_BODY*. It is developed as follows.

- Define objects  $BS[i : BI]$ . Each one is an instance of *C\_BOOTH*, the imperative concurrent version of *BOOTH*.

```

object
  /* Booths */
  BS[i : BI] : C_BOOTH

```

- Define variables. Fields of *Plaza* become variables, type *Plaza* is removed.

```

object
  /* Variables */
  V :
    class
      variable
        plaza_id : CT.Plaza_id,
        plaza_type : CT.Plaza_type,
        fares : CT.Fares,
        open_booths : BI-set,
        records : CT.Record*,

```

```

    plaza_money : CT.Money
end

```

- Define a function *main*:

```

value main : Unit → in CH.any out CH.any write V.any Unit

```

This function will define main process of *C\_PLAZA* module; it may contain several communications, as well as assignments to and read from variables. Therefore, we give this function accesses to input, output on channels and write to variables.

- Define an initial function. This function will be called once to start the concurrent process. We decided to have *Plaza\_id*, *Plaza\_type* and *Fares* as parameters. Plaza id, plaza type and fare list are needed to start a plaza; different plazas have different plaza ids, types and fare lists.

The body of *init* is the parallel composition of the initial process of each booth in parallel with initial assignment to the variables of the *C\_PLAZA* module followed by a call of *main*.

```

value
  init :
    CT.Plaza_id × CT.Plaza_type × CT.Fares →
    in CH.any out CH.any write V.any Unit
  init(p_id, p_type, f) ≡
    || { BS[i].init() | i : BI }
    ||
    (
      V.plaza_id := p_id ;
      V.plaza_type := p_type ;
      V.fares := f ;
      V.open_booths := {} ;
      V.records := ⟨ ⟩ ;
      V.plaza_money := 0 ;
      main()
    )

```

- Define interface function signatures. The interface function will do channel communication, so that a user need only to call this function instead of writing the details of channel communication. From each function that has type *Plaza* in its signature, define the signature of corresponding interface function by removing type *Plaza* and inserting access descriptor **in CH.any out CH.any**. For a partial function, make it total first by defining special value to indicate failures. A total function does not diverge or deadlock before either waiting to communicate or terminating. Functions *legal\_enter\_plaza*, *illegal\_enter\_plaza*, *legal\_leave\_plaza*, *illegal\_leave\_plaza*, *open\_booth*, and *close\_booth* become total.

**value**

```

legal_enter_plaza :
  CT.Car × CT.Time →
    in CH.any out CH.any CT.Legal_enter_plaza_result,

illegal_enter_plaza :
  CT.Car × CT.Time →
    in CH.any out CH.any CT.Illegal_enter_plaza_result,

legal_leave_plaza :
  (CT.Car × CT.Card) × CT.Time →
    in CH.any out CH.any CT.Legal_leave_plaza_result,

illegal_leave_plaza :
  (CT.Car × CT.Card) × CT.Time →
    in CH.any out CH.any CT.Illegal_leave_plaza_result,

open_booth :
  CT.Sup × CT.Opr × BI × CT.Time →
    in CH.any out CH.any CT.Plaza_open_booth_result

close_booth :
  CT.Sup × BI × CT.Time →
    in CH.any out CH.any CT.Plaza_close_booth_result

```

Types *Legal\_enter\_plaza\_result*, *Illegal\_enter\_plaza\_result*, *Legal\_leave\_plaza\_result*, *Illegal\_leave\_plaza\_result*, *Plaza\_open\_booth\_result*, and *Plaza\_close\_booth\_result* are defined to make the partial functions total. These types are placed in *C\_TYPES* module which is an extension of *TYPES1* module.

**type**

```

Legal_enter_plaza_result == fail | ok(car : Car, dcard : Card),
Illegal_enter_plaza_result == fail | ok(car : Car, dcard : Card),
Legal_leave_plaza_result == fail | ok(car : Car, dcard : Card),
Illegal_leave_plaza_result == fail | ok(car : Car, dcard : Card),
Plaza_open_booth_result == fail | ok,
Plaza_close_booth_result == fail | ok,
Open_booth_result == fail | ok(rec : Open_Booth_Rec)

```

- Define channels and their signatures according to interface functions. If the function has parameter non **Unit** and also result type non **Unit**, there will be two channels. The signature of each channel is generated from the signature of its corresponding interface function.

**object**

```

/* Channels */
CH :
  class
    channel
      enter_plaza : CT.Car,
      enter_plaza_res : CT.Enter_plaza_result,
      leave_plaza : CT.Car × CT.Card,
      leave_plaza_res : CT.Leave_plaza_result,
      open_booth : CT.Sup × BI,
      open_booth_res : CT.Plaza_open_booth_result,
      close_booth : CT.Sup × BI,
      close_booth_res : CT.Plaza_close_booth_result
    end

```

- Define interface function bodies. The body of each interface function is simply an output of its argument on the corresponding channel and followed by an input on the corresponding channel. For example, the body of interface function *open\_booth* is

```

open_booth(sup, opr, i, t) ≡
  CH.open_booth ! (sup, opr, i, t); CH.open_booth_res?

```

- Define the body of the main function. The body of *main* is a **while true** loop containing an external choice between expressions, one expression for each interface function.

In general an expression starts with an input on a channel, and the body of the expression is obtained by transforming the body of corresponding function in the applicative version to an imperative one and then outputting the result on a channel. Below we give only an expression for function *open\_booth*.

```

value
  main() ≡
    while true do
      let (sup, opr, i, t) = CH.open_booth? in
        if i ∉ V.open_booths then
          let
            b_id = give_booth_id(i),
            b_type = give_booth_type(i),
            res =
              BS[i].open_booth
                (sup, opr, V.plaza_id, b_id, b_type, V.fares, t)
          in
            if res ≠ CT.fail then
              V.open_booths := V.open_booths ∪ {i} ;
              V.records := V.records ^ ⟨CT.rec(res)⟩ ;
              CH.open_booth_res ! CT.ok

```

```

        else
            CH.open_booth_res ! CT.fail
        end
    end
else
    CH.open_booth_res ! CT.fail
end
end
[]
...
end

```

- Remove the convergence axiom of partial functions that now become total. Transform other axioms to imperative ones.

Complete *C\_PLAZA* module and *C\_TYPES* module can be found in Appendix A.4.

### C\_BOOTH Module

An imperative concurrent version of *BOOTH* is developed in the same way as *C\_PLAZA*. The method is explained briefly as follows. Instantiate the concurrent component modules, that are, imperative concurrent version of *BCT*, *DCT*, *PPCT* and *IOL*. Fields of record *Booth* becomes variables, type *Booth* is then removed. Define interface functions, channels and function *main* in the same method as explained before. Initial function *init* will then be the parallel composition of the initial process of the component modules in parallel with a call of *main*.

#### value

```

init : Unit → in CH.any out CH.any write V.any Unit
init() ≡
    BC.init() || DC.init() || PC.init() || I.init() || main()

```

*BC*, *DC*, *PC* and *I* are instances of *C\_BCT*, *C\_DCT*, *C\_PPCT* and *C\_IOL* respectively. Appendix A.4 contains the complete specification of the *C\_BOOTH* module.

### Component Modules: C\_BCT, C\_DCT, C\_PPCT, C\_IOL

*BCT*, *DCT*, *PPCT* and *IOL* modules do not have a type of interest, so in their concurrent version there is no variable. *BCT*, *DCT* and *PPCT* only read or print information on a card, without saving any information; all information is given and recorded in *TCT*. *IOL* only gives a signal whenever it detects a car passing through.

In *C\_BCT*, *C\_DCT* and *C\_PPCT* modules we define interface functions from the function concerning *B\_Card*, *Card* and *PP\_Card* respectively. In the *C\_IOL* module, interface functions are defined from the function concerning *Sensor\_State*. Channels, function *main* and initial function *init* are then defined in the same way as before.

Complete specification of these modules can be found in Appendix A.4.

### 3.4.2 Validation

The imperative concurrent modules are developed by applying a method for transition from sequential applicative counterparts. The result still behaves like the applicative one. Also, all the features in the sequential applicative module are in the imperative concurrent one. So the previous validation need not be repeated.

### 3.4.3 Verification

The development step was from sequential applicative to imperative concurrent. We do not formulate the development relation which asserts that *C\_PLAZA* implements *PLAZA*, since the method for developing from applicative module to an imperative module preserves the properties of the previous module; it changes the style only. Thus, what we need is to check informally whether the method for this transition has been followed correctly.

There is a meta-theorem on the relation between applicative specification and their imperative and concurrent counterparts. We could conservatively extend an imperative or concurrent specification which had been obtained by the transformation method such that we could justify that the extension is an implementation of its corresponding applicative specification. Since the extension is conservative, that is every property of the extension is a property of the original one, then implicitly, the imperative or the concurrent specification is also an implementation of its applicative counterpart. See the RAISE method [4] for full details of this technique.

## 4 From Design to Implementation

The aim of this section is to give an idea how to implement the RAISE specifications we have. RSL is a general specification language and not specially designed for any particular programming language. We decided that the toll way control system is to be implemented using the C programming language.

Before going further discussing how to implement our final specifications, we first present three issues to consider when looking at both RSL and implementation language [4].

First of all, there are RSL constructs that are very far from those in C and other programming languages. These constructs include axioms, post conditions, quantifications and other features that are typically used at the more abstract level of RSL specification. Hence, all these constructs should be eliminated on the way to the final specification which is subject to implementation.

Secondly, there are RSL constructs that can be translated into the programming language in a reasonably straightforward manner. This includes most of the algorithmic parts of RSL.

Thirdly, there might be concepts in the relevant programming language for which there are no immediately corresponding constructs in RSL but which we would like to use. Some can be modeled by special RSL modules. Others, like exceptions and their handling, we can construct a model of or alternatively we can introduce them during coding.

In the remainder of this section, two kinds of aspects related to translating the final specifications into code, namely concurrency aspects and data and function aspects, will be briefly discussed.

#### 4.1 Concurrency aspects

The concurrency between modules in final specification represents the communication between electro-mechanical device processes. RSL processes communicate through unidirectional channels and are based on synchronous communication with message passing. Some of the channels in the specifications represent standard communication protocols. Others represent physical interactions, and sometimes combinations of physical interaction and communication protocol. For example, channel *open\_booth* in *C\_BOOTH* represents a communication protocol — between plaza office and booth — and physical interactions — the interactions of a supervisor and an operator with a booth.

Some processes are models of hardware or models of physical actions, and these will not be translated. We can write low level communication routines to implement interface functions for hardware calls. Since *C\_BCT*, *C\_PPCT*, *C\_DCT* and *C\_IOL* are models of hardware, they represent assumption about the hardware; they are not translated. In the remaining section we will consider only *C\_BOOTH* and *C\_PLAZA* modules.

In each module, its main process involves external choice. External choice serves to specify a choice between different kinds of communication. This external choice can be translated into an event driven condition statement.

#### 4.2 Data and function aspects

As we stated before, we will implement the system using C language. Data types, such as record and sequence, are easy to translate. An RSL record can be translated into the C structures and sequences can easily be translated, say, into linked list or arrays.

The sequential part in a *let* expression can be translated using local variables. Furthermore, each expression inside the external choice expression visualizes the behavior of the system when certain conditions hold. However, not all expressions can be translated. But the expressions capture the idea of the system and from that we will know what are required in coding. Thus, it will provide us with guidelines during translation.

## 5 Conclusion

### 5.1 Summary

We have presented a step-wise development of toll way system using the RAISE method and specification language and shown the significant changes that has been made during the development. Our experience with this case study is that even though initially it was difficult to achieve the right initial specification and to make a model that describes the system, writing a formal description led us to understand the system better. It is evident, as shown in this work, that formal method can be applied to real applications.

The use of the RAISE specification language aids in making description more concise and clear. We had difficulty in understanding the documents describing the system in natural language and using the flow charts since we found many ambiguities and unclear explanations. A formal description can be used as formal documentation and as a basis for writing natural language documentation. Verification using support tool of RAISE helps to detect errors, such as implementation errors and inconsistency errors; and it results in having greater assurance of the system design. Furthermore, we can use the formal descriptions as a basis for code development.

We could say that this work is a re-engineering of an existing system. Re-engineering is, according to [8], the use of an existing information of a system to alter or reconstitute the existing system in an effort to improve its overall quality. Hence this work is only a partial re-engineering. We could also say that this work is a kind of reverse engineering in the sense that from an existing system we derive an abstract specification which accords with the definition in [8], but in fact we worked from the informal description of an existing system rather than from the implementation. What we do have is a good basis for further development of the system.

RAISE supports only *invent-and-verify* strategy for the refinement. Decomposing a specification was not found to be a trivial task, for example choosing the right formalization in the more abstract level so that a proper formalization for the component module could be arrived at is not easy. Even though the capability of decomposing could be achieved when more experience has been gained, it would be desirable if more guidelines on how to decompose a specification or possibly decomposition rules are provided. As the RAISE method undergoes further research, we expect such guidelines to emanate.

For a deeply nested specification with many interacting components, proving an implementation

relation is often very tedious. Guidelines on technique for proving such an implementation relation, possibly compositional proof technique, are needed.

At last, this case study can be used as an example in a software development course. It is hoped that this case study will give the students a realistic view of the benefits using formal method. Of course, other realistic case studies covering different domain problems are also needed. Furthermore, we found that RSL can be chosen as a formal language for this course since RSL is an expressive and flexible language: one language can be used for specifying different abstraction levels and RSL is not designed for a specific domain problem.

## 5.2 Future Work

We would like to see this report in a larger context, namely that of an overall software system for the support of Indonesian Toll Ways, and eventually the entire Road System. Such a system could contain additional features: traffic management (monitoring and controlling traffic to avoid congestion), road maintenance (collecting statistics for preventive maintenance of, for example, road surfaces), etc. For such a larger system, it requires domain analysis, that is, to better understand an application domain in which the software is to serve and to better understand the expectation of a software [1]. An example of domain analysis applied to a railway system can be seen in [2].

## 6 Acknowledgements

This report represents one, a major, aspect of the author's work while at UNU/IIST. The author was supported in this work by Chris George and Dines Bjørner. The author gratefully acknowledges their support.

## References

- [1] D. Bjørner, C. W. George, and S. Prehn. Domain analysis — a prerequisite for requirements capture. Technical Report 37, UNU/IIST, Macau, March 1995.
- [2] D. Bjørner, Dong Yu Lin, and S. Prehn. Domain analysis: A case study of railway station management. Technical Report 23, UNU/IIST, Macau, November 1994. Presented at KICS'94: The Kunming (Yunnan, PRC) International CASE Symposium, November 1994.
- [3] Chris W. George and S. Prehn. The RAISE Justification Handbook. LaCoS Report DOC/7, Computer Resources International A/S, 1992.
- [4] The RAISE Development Group. *The RAISE Development Method*. BCS Practitioner Series. Prentice Hall, London, 1995.
- [5] The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, London, 1992.
- [6] P.T. Jasa Marga. *Dokumen Pengadaan dan Pemasangan Peralatan Tol Cabang Jakarta Cikampek: Spesifikasi Teknis*. Jakarta, Indonesia, July 1993. This document gives a narrative technical specification of Toll Way components.
- [7] P.T. Jasa Marga. *Spesifikasi Rinci Rancangan Perangkat Lunak Sistem Peralatan Tol - TCT Sistem Tertutup*. Jakarta, Indonesia, rev. 03 edition, October 1993. This document gives specification of Toll Way software system using flow charts.
- [8] Roger S. Pressman. *Software Engineering A Practitioner's Approach*. Computer Science Series. McGraw-Hill International Editions, Singapore, 3rd edition, 1992.

## A Specifications

This section gives detailed descriptions of the development of a toll way control system by providing specifications for each step, starting from an abstract applicative description to an imperative concurrent one.

### A.1 Initial Specification: Properties

#### TYPES Module

```

scheme
  TYPES =
  class
    type Time = Nat, Money = Int, Records, Car, Card, Plaza_type == entry | exit | both

```

```

value
  car_money : Car → Money
end

```

```

object
  T : TYPES

```

## A\_TW\_0 Module

```

scheme
  A_TW_0 =
  hide
    add_legal_entry_rec, add_illegal_entry_rec, add_legal_exit_rec, add_illegal_exit_rec, booths_money
  in
    class
      type Plaza

    value
      legal_enter_plaza : T.Car × Plaza × T.Time  $\xrightarrow{\sim}$  (T.Car × T.Card) × Plaza,
      illegal_enter_plaza : T.Car × Plaza × T.Time  $\xrightarrow{\sim}$  (T.Car × T.Card) × Plaza,
      legal_leave_plaza : (T.Car × T.Card) × Plaza × T.Time  $\xrightarrow{\sim}$  (T.Car × T.Card) × Plaza,
      illegal_leave_plaza : (T.Car × T.Card) × Plaza × T.Time  $\xrightarrow{\sim}$  (T.Car × T.Card) × Plaza,
      add_legal_entry_rec : T.Car × Plaza × T.Time × T.Records  $\xrightarrow{\sim}$  T.Records,
      add_illegal_entry_rec : T.Car × Plaza × T.Time × T.Records  $\xrightarrow{\sim}$  T.Records,
      add_legal_exit_rec : (T.Car × T.Card) × Plaza × T.Time × T.Records  $\xrightarrow{\sim}$  T.Records,
      add_illegal_exit_rec : (T.Car × T.Card) × Plaza × T.Time × T.Records  $\xrightarrow{\sim}$  T.Records,
      records : Plaza → T.Records,
      money : Plaza → T.Money,
      plaza_type : Plaza → T.Plaza_type,
      booths_money : Plaza → T.Money,
      charge : (T.Car × T.Card) × Plaza × T.Time → T.Money,
      has_open_booth : Plaza → Bool,
      can_enter_plaza : T.Car × Plaza → Bool
      can_enter_plaza(c, p)  $\equiv$ 
        has_open_booth(p)  $\wedge$  (plaza_type(p) = T.entry  $\vee$  plaza_type(p) = T.both),

```

$\text{can\_leave\_plaza} : (\text{T.Car} \times \text{T.Card}) \times \text{Plaza} \rightarrow \mathbf{Bool}$   
 $\text{can\_leave\_plaza}((c, dc), p) \equiv$   
 $\text{has\_open\_booth}(p) \wedge (\text{plaza\_type}(p) = \text{T.exit} \vee \text{plaza\_type}(p) = \text{T.both})$

**axiom**

[money\_legal\_enter]

$\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
**let**  $((c', dc), p') = \text{legal\_enter\_plaza}(c, p, t)$  **in**  
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) \wedge$   
 $\text{booths\_money}(p') = \text{booths\_money}(p) \wedge \text{money}(p') = \text{money}(p)$   
**end**  $\equiv$   
**true**  
**pre**  $\text{can\_enter\_plaza}(c, p),$

[money\_illegal\_enter]

$\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
**let**  $((c', dc), p') = \text{illegal\_enter\_plaza}(c, p, t)$  **in**  
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) \wedge$   
 $\text{booths\_money}(p') = \text{booths\_money}(p) \wedge \text{money}(p') = \text{money}(p)$   
**end**  $\equiv$   
**true**  
**pre**  $\text{can\_enter\_plaza}(c, p),$

[money\_legal\_leave]

$\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
**let**  $((c', dc'), p') = \text{legal\_leave\_plaza}((c, dc), p, t), \text{pay} = \text{charge}((c, dc), p, t)$  **in**  
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) - \text{pay} \wedge$   
 $\text{booths\_money}(p') = \text{booths\_money}(p) + \text{pay} \wedge \text{money}(p') = \text{money}(p)$   
**end**  $\equiv$   
**true**  
**pre**  $\text{can\_leave\_plaza}((c, dc), p),$

[money\_illegal\_leave]

$\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
**let**  $((c', dc'), p') = \text{illegal\_leave\_plaza}((c, dc), p, t)$  **in**  
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) \wedge$   
 $\text{booths\_money}(p') = \text{booths\_money}(p) \wedge \text{money}(p') = \text{money}(p)$   
**end**  $\equiv$   
**true**  
**pre**  $\text{can\_leave\_plaza}((c, dc), p),$

[records\_legal\_enter]

$\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
**let**  $((c', dc), p') = \text{legal\_enter\_plaza}(c, p, t)$  **in**  
 $\text{records}(p') = \text{add\_legal\_entry\_rec}(c, p, t, \text{records}(p))$   
**end**  $\equiv$   
**true**  
**pre**  $\text{can\_enter\_plaza}(c, p),$

[records\_illegal\_enter]

```

    ∀ c : T.Car, p : Plaza, t : T.Time •
      let ((c', dc), p') = illegal_enter_plaza(c, p, t) in
        records(p') = add_illegal_entry_rec(c, p, t, records(p))
      end ≡
      true
      pre can_enter_plaza(c, p),

  [records_legal_leave]
  ∀ c : T.Car, dc : T.Card, p : Plaza, t : T.Time •
    let ((c', dc), p') = legal_leave_plaza((c, dc), p, t) in
      records(p') = add_legal_exit_rec((c, dc), p, t, records(p))
    end ≡
    true
    pre can_leave_plaza((c, dc), p),

  [records_illegal_leave]
  ∀ c : T.Car, dc : T.Card, p : Plaza, t : T.Time •
    let ((c', dc), p') = illegal_leave_plaza((c, dc), p, t) in
      records(p') = add_illegal_exit_rec((c, dc), p, t, records(p))
    end ≡
    true
    pre can_leave_plaza((c, dc), p)
end

```

Note that all the functions that are not needed in the detailed level are hidden.

## A.2 Second Specification: Booth

### TYPES1 Module

```

scheme
TYPES1 =
  class
  type
    Time = Nat,
    Money = Int,
    Car_id,
    Plaza_id,
    Booth_id,
    Fares,
    ID,
    PP_Card == nil_ppc | ppc(val : Money),
    Car :: car_id : Car_id cash : Money pp_card : PP_Card,
    Card :: entry_info : D_Card exit_info : D_Card,
    D_Card == empty | mk_info(pass_info : Card_Info),
    Card_Info :: car_id : Car_id plaza_id : Opt_plaza_id booth_id : Booth_id time : Opt_time,
    Comps_Status == ok | fail,
    B_Card :: id : ID,

```

```

Sup :: b_card : B_Card,
Opr :: b_card : B_Card,
Records = Record*,
Record = Car_Entry_Rec | Car_Exit_Rec | Open_Booth_Rec | Close_Booth_Rec,
Car_Entry_Rec = Legal_Entry_Rec | Illegal_Entry_Rec,
Legal_Entry_Rec :: car_id : Car_id time : Time p_id : Plaza_id b_id : Booth_id,
Illegal_Entry_Rec :: time : Time p_id : Plaza_id b_id : Booth_id,
Car_Exit_Rec = Legal_Exit_Rec | Illegal_Exit_Rec,
Legal_Exit_Rec ::
  car_id : Car_id
  entry_time : Opt_time
  p_id_enter : Opt_plaza_id
  exit_time : Time
  p_id_leave : Plaza_id
  b_id : Booth_id
  car_pay : Money,
Illegal_Exit_Rec :: time : Time p_id : Plaza_id b_id : Booth_id,
Open_Booth_Rec :: booth_id : Booth_id open_info : Open_Info,
Close_Booth_Rec :: booth_id : Booth_id close_info : Close_Info booth_money : Money,
Open_Info :: time : Time sup_id : ID session_no : Nat opr_id : ID,
Close_Info :: time : Time sup_id : ID,
Opt_plaza_id == nil_pid | mk_pid(Plaza_id),
Opt_time == nil_time | mk_time(Time),
Passing_status == illegal | ok,
Payment == card_first | cash_first,
Sensor_state == expect | waiting,
Plaza_type == entry | exit | both,
Booth_type == entry | exit

```

**value**

```

car_money : Car → Money
car_money(c) ≡ cash(c) + if pp_card(c) = nil_ppc then 0 else val(pp_card(c)) end,

```

```

charge : Car_id × Opt_plaza_id × Opt_time × Plaza_id × Time × Fares → Money,

```

```

check_comps : Booth_id → Comps_Status,

```

```

get_session_no : Sup × Booth_id → Nat,

```

```

payment_method : Car → Payment,

```

```

add_counter : Nat → Nat

```

```

end

```

**object**

```

T1 : TYPES1

```

## A\_TW\_1 Module

**scheme**

A\_TW\_1 =

**hide**

legal\_enter\_booth, illegal\_enter\_booth, legal\_leave\_booth, illegal\_leave\_booth, can\_enter\_booth,  
can\_leave\_booth, add\_legal\_entry\_rec, add\_illegal\_entry\_rec, add\_legal\_exit\_rec,  
add\_illegal\_exit\_rec, booths\_money, sum\_money

**in**

**class**

**type** Plaza, Booth, BI, Booths = BI  $\rightsquigarrow$  Booth

**axiom**

*/\* BI is finite \*/*

[booths\_finite] **card** { i | i : BI } **post true**,

*/\* Booths is convergent for arguments in domain \*/*

[booths\_convergent]  $\forall i : BI, bs : Booths \bullet i \in \mathbf{dom} bs \Rightarrow (bs(i) \mathbf{post true})$

**value**

legal\_enter\_plaza : T.Car  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Plaza,

illegal\_enter\_plaza : T.Car  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Plaza,

legal\_leave\_plaza : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Plaza,

illegal\_leave\_plaza : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Plaza,

legal\_enter\_booth : T.Car  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Booth  $\times$  BI,

illegal\_enter\_booth : T.Car  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Booth  $\times$  BI,

legal\_leave\_booth :

(T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Booth  $\times$  BI,

illegal\_leave\_booth :

(T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\rightsquigarrow$  (T.Car  $\times$  T.Card)  $\times$  Booth  $\times$  BI,

add\_legal\_entry\_rec : T.Car  $\times$  Plaza  $\times$  T.Time  $\times$  T.Records  $\rightsquigarrow$  T.Records,

add\_illegal\_entry\_rec : T.Car  $\times$  Plaza  $\times$  T.Time  $\times$  T.Records  $\rightsquigarrow$  T.Records,

add\_legal\_exit\_rec : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\times$  T.Records  $\rightsquigarrow$  T.Records,

add\_illegal\_exit\_rec : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\times$  T.Records  $\rightsquigarrow$  T.Records,

booths : Plaza  $\rightarrow$  Booths,

records : Plaza  $\rightarrow$  T.Records,

money : Plaza  $\rightarrow$  T.Money,

plaza\_type : Plaza  $\rightarrow$  T.Plaza\_type,

booths\_money : Plaza  $\rightarrow$  T.Money  
 booths\_money(p)  $\equiv$  sum\_money(booths(p)),

booth\_money : Booth  $\rightarrow$  T.Money,

sum\_money : Booths  $\rightarrow$  T.Money,

charge : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\times$  T.Time  $\rightarrow$  T.Money,

has\_open\_booth : Plaza  $\rightarrow$  **Bool**  
 has\_open\_booth(p)  $\equiv$  **dom** booths(p)  $\neq$  {},

can\_enter\_booth : T.Car  $\times$  Plaza  $\rightarrow$  **Bool**  
 can\_enter\_booth(c, p)  $\equiv$   
 has\_open\_booth(p)  $\wedge$  (plaza\_type(p) = T.entry  $\vee$  plaza\_type(p) = T.both),

can\_leave\_booth : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\rightarrow$  **Bool**  
 can\_leave\_booth((c, dc), p)  $\equiv$   
 has\_open\_booth(p)  $\wedge$  (plaza\_type(p) = T.exit  $\vee$  plaza\_type(p) = T.both),

can\_enter\_plaza : T.Car  $\times$  Plaza  $\rightarrow$  **Bool**  
 can\_enter\_plaza(c, p)  $\equiv$   
 has\_open\_booth(p)  $\wedge$  (plaza\_type(p) = T.entry  $\vee$  plaza\_type(p) = T.both),

can\_leave\_plaza : (T.Car  $\times$  T.Card)  $\times$  Plaza  $\rightarrow$  **Bool**  
 can\_leave\_plaza((c, dc), p)  $\equiv$   
 has\_open\_booth(p)  $\wedge$  (plaza\_type(p) = T.exit  $\vee$  plaza\_type(p) = T.both)

#### axiom

[legal\_enter\_booth\_conv]  
 $\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 can\_enter\_booth(c, p)  $\Rightarrow$  (legal\_enter\_booth(c, p, t) **post true**),

[illegal\_enter\_booth\_conv]  
 $\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 can\_enter\_booth(c, p)  $\Rightarrow$  (illegal\_enter\_booth(c, p, t) **post true**),

[legal\_enter\_booth\_ax]  
 $\forall c, c' : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 can\_enter\_booth(c, p)  $\wedge$  ((c', dc), b, i) = legal\_enter\_booth(c, p, t)  $\Rightarrow$   
 T.car\_money(c') = T.car\_money(c)  $\wedge$   
 i  $\in$  **dom** booths(p)  $\wedge$  booth\_money(b) = booth\_money((booths(p))(i)),

[illegal\_enter\_booth\_ax]  
 $\forall c, c' : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 can\_enter\_booth(c, p)  $\wedge$  ((c', dc), b, i) = illegal\_enter\_booth(c, p, t)  $\Rightarrow$   
 T.car\_money(c') = T.car\_money(c)  $\wedge$   
 i  $\in$  **dom** booths(p)  $\wedge$  booth\_money(b) = booth\_money((booths(p))(i)),

[legal\_leave\_booth\_conv]  
 $\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \Rightarrow (\text{legal\_leave\_booth}((c, dc), p, t) \text{ post true}),$

[illegal\_leave\_booth\_conv]  
 $\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \Rightarrow (\text{illegal\_leave\_booth}((c, dc), p, t) \text{ post true}),$

[legal\_leave\_booth\_ax]  
 $\forall c, c' : \text{T.Car}, dc, dc' : \text{T.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \wedge ((c', dc'), b, i) = \text{legal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) - \text{charge}((c, dc), p, t) \wedge$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $\text{booth\_money}(b) = \text{booth\_money}((\text{booths}(p))(i)) + \text{charge}((c, dc), p, t),$

[illegal\_leave\_booth\_ax]  
 $\forall c, c' : \text{T.Car}, dc, dc' : \text{T.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \wedge ((c', dc'), b, i) = \text{illegal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $\text{T.car\_money}(c') = \text{T.car\_money}(c) \wedge$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge \text{booth\_money}(b) = \text{booth\_money}((\text{booths}(p))(i)),$

[legal\_enter\_plaza\_conv]  
 $\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \Rightarrow (\text{legal\_enter\_plaza}(c, p, t) \text{ post true}),$

[illegal\_enter\_plaza\_conv]  
 $\forall c : \text{T.Car}, p : \text{Plaza}, t : \text{T.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \Rightarrow (\text{illegal\_enter\_plaza}(c, p, t) \text{ post true}),$

[legal\_enter\_plaza\_ax]  
 $\forall c, c', c'' : \text{T.Car}, dc, dc' : \text{T.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \wedge ((c', dc), p') = \text{legal\_enter\_plaza}(c, p, t) \Rightarrow$   
 $((c'', dc'), b, i) = \text{legal\_enter\_booth}(c, p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc = dc' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \uparrow [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_legal\_entry\_rec}(c, p, t, \text{records}(p)),$

[illegal\_enter\_plaza\_ax]  
 $\forall c, c', c'' : \text{T.Car}, dc, dc' : \text{T.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \wedge ((c', dc), p') = \text{illegal\_enter\_plaza}(c, p, t) \Rightarrow$   
 $((c'', dc'), b, i) = \text{illegal\_enter\_booth}(c, p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc = dc' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \uparrow [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_illegal\_entry\_rec}(c, p, t, \text{records}(p)),$

[legal\_leave\_plaza\_conv]

```

     $\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$ 
      can_leave_plaza((c, dc), p)  $\Rightarrow$  (legal_leave_plaza((c, dc), p, t) post true),

  [illegal_leave_plaza_conv]
     $\forall c : \text{T.Car}, dc : \text{T.Card}, p : \text{Plaza}, t : \text{T.Time} \bullet$ 
      can_leave_plaza((c, dc), p)  $\Rightarrow$  (illegal_leave_plaza((c, dc), p, t) post true),

  [legal_leave_plaza_ax]
     $\forall c, c', c'' : \text{T.Car}, dc, dc', dc'' : \text{T.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$ 
      can_leave_plaza((c, dc), p)  $\wedge$  ((c', dc'), p') = legal_leave_plaza((c, dc), p, t)  $\Rightarrow$ 
      ((c'', dc''), b, i) = legal_leave_booth((c, dc), p, t)  $\Rightarrow$ 
       $i \in \mathbf{dom} \text{booths}(p) \wedge$ 
       $c' = c'' \wedge$ 
       $dc' = dc'' \wedge$ 
       $\text{booths}(p') = \text{booths}(p) \uparrow [i \mapsto b] \wedge$ 
       $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_legal\_exit\_rec}((c, dc), p, t, \text{records}(p)),$ 

  [illegal_leave_plaza_ax]
     $\forall c, c', c'' : \text{T.Car}, dc, dc', dc'' : \text{T.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T.Time} \bullet$ 
      can_leave_plaza((c, dc), p)  $\wedge$  ((c', dc'), p') = illegal_leave_plaza((c, dc), p, t)  $\Rightarrow$ 
      ((c'', dc''), b, i) = illegal_leave_booth((c, dc), p, t)  $\Rightarrow$ 
       $i \in \mathbf{dom} \text{booths}(p) \wedge$ 
       $c' = c'' \wedge$ 
       $dc' = dc'' \wedge$ 
       $\text{booths}(p') = \text{booths}(p) \uparrow [i \mapsto b] \wedge$ 
       $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_illegal\_exit\_rec}((c, dc), p, t, \text{records}(p)),$ 

  [sum_money_ax]
     $\forall p : \text{Plaza}, b : \text{Booth}, bs : \text{Booths}, i : \text{BI} \bullet$ 
       $bs = \text{booths}(p) \wedge i \in \mathbf{dom} \text{booths}(p) \Rightarrow$ 
       $(\text{sum\_money}(bs \uparrow [i \mapsto b]) \equiv \text{sum\_money}(bs) - \text{booth\_money}(bs(i)) + \text{booth\_money}(b))$ 
end

```

We also hide all functions that are not needed in the next level.

### A.3 Third Specification: Components

#### PLAZA\_BODY Module

```

scheme
  PLAZA_BODY =
  class
    object
      /* Booth */
      B : BOOTH

  type
    /* Booth's index */

```

```

BI,
/* Booth */
Booth = B.Booth,
/* open booth set */
Booths = BI  $\overline{m}$  Booth,
/* Plaza */
Plaza ::
  plaza_id : T1.Plaza_id
  plaza_type : T1.Plaza_type
  fares : T1.Fares
  booths : Booths
  records : T1.Records
  money : T1.Money

```

**axiom**

```

/* BI is finite */
[booths_finite] card { i | i : BI } post true,

/* Booths is convergent for arguments in domain */
[booths_convergent]  $\forall i : BI, bs : Booths \bullet i \in \mathbf{dom} \ bs \Rightarrow (bs(i) \mathbf{post true})$ 

```

**value**

```

open_booth : T1.Sup  $\times$  T1.Opr  $\times$  Plaza  $\times$  BI  $\times$  T1.Time  $\xrightarrow{\sim}$  Plaza
open_booth(sup, opr, p, i, t)  $\equiv$ 
  let b_id = give_booth_id(p, i) in
    if B.can_open_booth(b_id) then
      let
        b_type = give_booth_type(p, i),
        (b, r) = B.open_booth(sup, opr, plaza_id(p), b_id, b_type, fares(p), t),
        bs' = booths(p)  $\cup$  [i  $\mapsto$  b],
        rs' = records(p)  $\hat{\ } \langle r \rangle$ 
      in
        mk_Plaza(plaza_id(p), plaza_type(p), fares(p), bs', rs', money(p))
    end
  else
    p
  end
end
pre can_open_booth(p, i),

close_booth : T1.Sup  $\times$  Plaza  $\times$  BI  $\times$  T1.Time  $\xrightarrow{\sim}$  Plaza
close_booth(sup, p, i, t)  $\equiv$ 
  let
    b = (booths(p))(i),
    bs' = booths(p)  $\setminus$  {i},
    (r, m) = B.close_booth(sup, b, t),
    rs' = records(p)  $\hat{\ } \langle r \rangle$ 
  in
    mk_Plaza(plaza_id(p), plaza_type(p), fares(p), bs', rs', money(p) + m)
  end
pre can_close_booth(p, i),

```

legal\_enter\_plaza : T1.Car  $\times$  Plaza  $\times$  T1.Time  $\xrightarrow{\sim}$  (T1.Car  $\times$  T1.Card)  $\times$  Plaza  
 legal\_enter\_plaza(c, p, t)  $\equiv$

**let**  
   i = choose\_booth(c, p),  
   ((c', dc), b', r) = B.legal\_enter\_booth(c, (booths(p))(i), t),  
   bs' = booths(p)  $\dagger$  [i  $\mapsto$  b'],  
   rs' = records(p)  $\hat{\ } \langle r \rangle$   
**in**  
 ((c', dc), mk\_Plaza(plaza\_id(p), plaza\_type(p), fares(p), bs', rs', money(p)))  
**end**  
**pre** can\_enter\_plaza(c, p),

illegal\_enter\_plaza : T1.Car  $\times$  Plaza  $\times$  T1.Time  $\xrightarrow{\sim}$  (T1.Car  $\times$  T1.Card)  $\times$  Plaza  
 illegal\_enter\_plaza(c, p, t)  $\equiv$

**let**  
   i = choose\_booth(c, p),  
   ((c', dc), b', r) = B.illegal\_enter\_booth(c, (booths(p))(i), t),  
   bs' = booths(p)  $\dagger$  [i  $\mapsto$  b'],  
   rs' = records(p)  $\hat{\ } \langle r \rangle$   
**in**  
 ((c', dc), mk\_Plaza(plaza\_id(p), plaza\_type(p), fares(p), bs', rs', money(p)))  
**end**  
**pre** can\_enter\_plaza(c, p),

legal\_leave\_plaza : (T1.Car  $\times$  T1.Card)  $\times$  Plaza  $\times$  T1.Time  $\xrightarrow{\sim}$  (T1.Car  $\times$  T1.Card)  $\times$  Plaza  
 legal\_leave\_plaza((c, dc), p, t)  $\equiv$

**let**  
   i = choose\_booth(c, p),  
   ((c', dc'), b', r) = B.legal\_leave\_booth((c, dc), (booths(p))(i), t),  
   bs' = booths(p)  $\dagger$  [i  $\mapsto$  b'],  
   rs' = records(p)  $\hat{\ } \langle r \rangle$   
**in**  
 ((c', dc'), mk\_Plaza(plaza\_id(p), plaza\_type(p), fares(p), bs', rs', money(p)))  
**end**  
**pre** can\_leave\_plaza((c, dc), p),

illegal\_leave\_plaza : (T1.Car  $\times$  T1.Card)  $\times$  Plaza  $\times$  T1.Time  $\xrightarrow{\sim}$  (T1.Car  $\times$  T1.Card)  $\times$  Plaza  
 illegal\_leave\_plaza((c, dc), p, t)  $\equiv$

**let**  
   i = choose\_booth(c, p),  
   ((c', dc'), b', r) = B.illegal\_leave\_booth((c, dc), (booths(p))(i), t),  
   bs' = booths(p)  $\dagger$  [i  $\mapsto$  b'],  
   rs' = records(p)  $\hat{\ } \langle r \rangle$   
**in**  
 ((c', dc'), mk\_Plaza(plaza\_id(p), plaza\_type(p), fares(p), bs', rs', money(p)))  
**end**  
**pre** can\_leave\_plaza((c, dc), p),

choose\_booth : T1.Car  $\times$  Plaza  $\xrightarrow{\sim}$  BI,

give\_booth\_id : Plaza × BI  $\xrightarrow{\sim}$  T1.Booth\_id,

give\_booth\_type : Plaza × BI  $\xrightarrow{\sim}$  T1.Booth\_type,

charge : (T1.Car × T1.Card) × Plaza × T1.Time → T1.Money

charge((c, dc), p, t) ≡

**let**

entry\_id = T1.plaza\_id(T1.pass\_info(T1.entry\_info(dc))),

entry\_time = T1.time(T1.pass\_info(T1.entry\_info(dc)))

**in**

T1.charge(T1.car\_id(c), entry\_id, entry\_time, plaza\_id(p), t, fares(p))

**end,**

has\_open\_booth : Plaza → **Bool**

has\_open\_booth(p) ≡ **dom** booths(p) ≠ {},

can\_enter\_plaza : T1.Car × Plaza → **Bool**

can\_enter\_plaza(c, p) ≡

has\_open\_booth(p) ∧ (plaza\_type(p) = T1.entry ∨ plaza\_type(p) = T1.both),

can\_leave\_plaza : (T1.Car × T1.Card) × Plaza → **Bool**

can\_leave\_plaza((c, dc), p) ≡

has\_open\_booth(p) ∧ (plaza\_type(p) = T1.exit ∨ plaza\_type(p) = T1.both),

can\_open\_booth : Plaza × BI → **Bool**

can\_open\_booth(p, i) ≡ i ∉ **dom** booths(p),

can\_close\_booth : Plaza × BI → **Bool**

can\_close\_booth(p, i) ≡ i ∈ **dom** booths(p),

booth\_money : Booth → T1.Money

booth\_money(b) ≡ B.money(b)

**axiom**

[give\_booth\_id\_conv]

∀ p : Plaza, i : BI • i ∉ **dom** booths(p) ⇒ (give\_booth\_id(p, i) **post true**),

[give\_booth\_type\_conv]

∀ p : Plaza, i : BI • i ∉ **dom** booths(p) ⇒ (give\_booth\_type(p, i) **post true**),

[choose\_booth\_conv]

∀ c : T1.Car, p : Plaza • has\_open\_booth(p) ⇒ (choose\_booth(c, p) **post true**),

[choose\_booth\_ax]

∀ c : T1.Car, p : Plaza • has\_open\_booth(p) ⇒ choose\_booth(c, p) ∈ **dom** booths(p),

[legal\_enter\_plaza\_conv]

∀ c : T1.Car, p : Plaza, t : T1.Time •

can\_enter\_plaza(c, p) ⇒ (legal\_enter\_plaza(c, p, t) **post true**),

[illegal\_enter\_plaza\_conv]

```

    ∀ c : T1.Car, p : Plaza, t : T1.Time •
      can_enter_plaza(c, p) ⇒ (illegal_enter_plaza(c, p, t) post true),

  [legal_plaza_conv]
  ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time •
    can_leave_plaza((c, dc), p) ⇒ (legal_leave_plaza((c, dc), p, t) post true),

  [illegal_plaza_conv]
  ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time •
    can_leave_plaza((c, dc), p) ⇒ (illegal_leave_plaza((c, dc), p, t) post true)
end

```

## PLAZA Module

```

scheme
PLAZA =
  extend PLAZA_BODY with
  class
  value
    can_enter_booth : T1.Car × Plaza → Bool
    can_enter_booth(c, p) ≡
      has_open_booth(p) ∧ (plaza_type(p) = T1.entry ∨ plaza_type(p) = T1.both),

    can_leave_booth : (T1.Car × T1.Card) × Plaza → Bool
    can_leave_booth((c, dc), p) ≡
      has_open_booth(p) ∧ (plaza_type(p) = T1.exit ∨ plaza_type(p) = T1.both),

    legal_enter_booth : T1.Car × Plaza × T1.Time  $\rightsquigarrow$  (T1.Car × T1.Card) × Booth × BI
    legal_enter_booth(c, p, t) ≡
      let i = choose_booth(c, p), ((c', dc), b', r) = B.legal_enter_booth(c, (booths(p))(i), t) in
        ((c', dc), b', i)
      end
      pre can_enter_booth(c, p),

    illegal_enter_booth : T1.Car × Plaza × T1.Time  $\rightsquigarrow$  (T1.Car × T1.Card) × Booth × BI
    illegal_enter_booth(c, p, t) ≡
      let i = choose_booth(c, p), ((c', dc), b', r) = B.illegal_enter_booth(c, (booths(p))(i), t) in
        ((c', dc), b', i)
      end
      pre can_enter_booth(c, p),

    legal_leave_booth :
      (T1.Car × T1.Card) × Plaza × T1.Time  $\rightsquigarrow$  (T1.Car × T1.Card) × Booth × BI
    legal_leave_booth((c, dc), p, t) ≡
      let
        i = choose_booth(c, p), ((c', dc'), b', r) = B.legal_leave_booth((c, dc), (booths(p))(i), t)
      in
        ((c', dc'), b', i)
      end

```

**pre** can\_leave\_booth((c, dc), p),

illegal\_leave\_booth :

$(T1.Car \times T1.Card) \times Plaza \times T1.Time \xrightarrow{\sim} (T1.Car \times T1.Card) \times Booth \times BI$

illegal\_leave\_booth((c, dc), p, t)  $\equiv$

**let**

i = choose\_booth(c, p), ((c', dc'), b', r) = B.illegal\_leave\_booth((c, dc), (booths(p))(i), t)

**in**

((c', dc'), b', i)

**end**

**pre** can\_leave\_booth((c, dc), p),

add\_legal\_entry\_rec :  $T1.Car \times Plaza \times T1.Time \times T1.Records \xrightarrow{\sim} T1.Records$

add\_legal\_entry\_rec(c, p, t, r)  $\equiv$

**let** i = choose\_booth(c, p), ((c', dc), b', r) = B.legal\_enter\_booth(c, (booths(p))(i), t) **in**  
records(p)  $\hat{\ } \langle r \rangle$

**end**

**pre** can\_enter\_plaza(c, p),

add\_illegal\_entry\_rec :  $T1.Car \times Plaza \times T1.Time \times T1.Records \xrightarrow{\sim} T1.Records$

add\_illegal\_entry\_rec(c, p, t, r)  $\equiv$

**let** i = choose\_booth(c, p), ((c', dc), b', r) = B.illegal\_enter\_booth(c, (booths(p))(i), t) **in**  
records(p)  $\hat{\ } \langle r \rangle$

**end**

**pre** can\_enter\_plaza(c, p),

add\_legal\_exit\_rec :  $(T1.Car \times T1.Card) \times Plaza \times T1.Time \times T1.Records \xrightarrow{\sim} T1.Records$

add\_legal\_exit\_rec((c, dc), p, t, r)  $\equiv$

**let**

i = choose\_booth(c, p), ((c', dc'), b', r) = B.legal\_leave\_booth((c, dc), (booths(p))(i), t)

**in**

records(p)  $\hat{\ } \langle r \rangle$

**end**

**pre** can\_leave\_plaza((c, dc), p),

add\_illegal\_exit\_rec :  $(T1.Car \times T1.Card) \times Plaza \times T1.Time \times T1.Records \xrightarrow{\sim} T1.Records$

add\_illegal\_exit\_rec((c, dc), p, t, r)  $\equiv$

**let**

i = choose\_booth(c, p), ((c', dc'), b', r) = B.illegal\_leave\_booth((c, dc), (booths(p))(i), t)

**in**

records(p)  $\hat{\ } \langle r \rangle$

**end**

**pre** can\_leave\_plaza((c, dc), p),

booths\_money : Plaza  $\rightarrow$  T1.Money

booths\_money(p)  $\equiv$  sum\_money(booths(p)),

sum\_money : Booths  $\rightarrow$  T1.Money

**axiom**

[legal\_enter\_booth\_conv]

```

    ∀ c : T1.Car, p : Plaza, t : T1.Time •
      can_enter_booth(c, p) ⇒ (legal_enter_booth(c, p, t) post true),

  [illegal_enter_booth_conv]
    ∀ c : T1.Car, p : Plaza, t : T1.Time •
      can_enter_booth(c, p) ⇒ (illegal_enter_booth(c, p, t) post true),

  [legal_leave_booth_conv]
    ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time •
      can_leave_booth((c, dc), p) ⇒ (legal_leave_booth((c, dc), p, t) post true),

  [illegal_leave_booth_conv]
    ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time •
      can_leave_booth((c, dc), p) ⇒ (illegal_leave_booth((c, dc), p, t) post true),

  [legal_entry_rec_conv]
    ∀ c : T1.Car, p : Plaza, t : T1.Time, r : T1.Records •
      can_enter_plaza(c, p) ⇒ (add_legal_entry_rec(c, p, t, r) post true),

  [illegal_entry_rec_conv]
    ∀ c : T1.Car, p : Plaza, t : T1.Time, r : T1.Records •
      can_enter_plaza(c, p) ⇒ (add_illegal_entry_rec(c, p, t, r) post true),

  [legal_exit_rec_conv]
    ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time, r : T1.Records •
      can_leave_plaza((c, dc), p) ⇒ (add_legal_exit_rec((c, dc), p, t, r) post true),

  [illegal_exit_rec_conv]
    ∀ c : T1.Car, dc : T1.Card, p : Plaza, t : T1.Time, r : T1.Records •
      can_leave_plaza((c, dc), p) ⇒ (add_illegal_exit_rec((c, dc), p, t, r) post true),

  [sum_money_ax]
    ∀ p : Plaza, b : Booth, bs : Booths, i : BI •
      bs = booths(p) ∧ i ∈ dom booths(p) ⇒
      (sum_money(bs † [i ↦ b]) ≡ sum_money(bs) - booth_money(bs(i)) + booth_money(b))
end

```

## BOOTH Module

```

scheme
BOOTH =
  class
  object
    /* Badge Card Terminal */
    BC : BCT,
    /* Duty Card Terminal */
    DC : DCT,
    /* Pre-Paid Card Terminal */
    PC : PPCT,

```

```

/* Car Sensor */
I : IOL

```

**type**

```

/* Booth */
Booth ::
  plaza_id : T1.Plaza_id
  booth_id : T1.Booth_id
  booth_type : T1.Booth_type
  fares : T1.Fares
  money : T1.Money
  sensor : T1.Sensor_state
  counter : Nat

```

**value**

```

legal_enter_booth :
  T1.Car × Booth × T1.Time → (T1.Car × T1.Card) × Booth × T1.Car_Entry_Rec
legal_enter_booth(c, b, t) ≡
  let
    dc = DC.legal_enter_booth(c, plaza_id(b), booth_id(b), t),
    b' =
      mk_Booth
        (plaza_id(b), booth_id(b), booth_type(b), fares(b), money(b), T1.expect, counter(b)),
    (b'', r) = entry_pass_sensor(c, b', t)
  in
    ((c, dc), b'', r)
end,

```

```

illegal_enter_booth :
  T1.Car × Booth × T1.Time → (T1.Car × T1.Card) × Booth × T1.Car_Entry_Rec
illegal_enter_booth(c, b, t) ≡
  let dc = T1.mk_Card(T1.empty, T1.empty), (b', r) = entry_pass_sensor(c, b, t) in
    ((c, dc), b', r)
end,

```

```

legal_leave_booth :
  (T1.Car × T1.Card) × Booth × T1.Time →
  (T1.Car × T1.Card) × Booth × T1.Car_Exit_Rec
legal_leave_booth((c, dc), b, t) ≡
  if T1.entry_info(dc) = T1.empty then
    let
      entry_id = T1.nil_pid,
      entry_time = T1.nil_time,
      car_pay = T1.charge(T1.car_id(c), entry_id, entry_time, plaza_id(b), t, fares(b)),
      pm = T1.payment_method(c),
      (cash', ppc') = payment((T1.cash(c), T1.pp_card(c)), pm, car_pay),
      c' = T1.mk_Car(T1.car_id(c), cash', ppc'),
      b' =
        mk_Booth
          (
            plaza_id(b),

```

```

        booth_id(b),
        booth_type(b),
        fares(b),
        money(b) + car_pay,
        T1.expect,
        counter(b)
    ),
    (b'', r) = exit_pass_sensor(c, entry_id, entry_time, car_pay, b', t)
in
    ((c', dc), b'', r)
end
else
let
    (entry_id, entry_time, dc') = DC.legal_leave_booth((c, dc), plaza_id(b), booth_id(b), t),
    car_pay = T1.charge(T1.car_id(c), entry_id, entry_time, plaza_id(b), t, fares(b)),
    pm = T1.payment_method(c),
    (cash', ppc') = payment((T1.cash(c), T1.pp_card(c)), pm, car_pay),
    c' = T1.mk_Car(T1.car_id(c), cash', ppc'),
    b' =
        mk_Booth
        (
            plaza_id(b),
            booth_id(b),
            booth_type(b),
            fares(b),
            money(b) + car_pay,
            T1.expect,
            counter(b)
        ),
    (b'', r) = exit_pass_sensor(c, entry_id, entry_time, car_pay, b', t)
in
    ((c', dc'), b'', r)
end
end,

```

illegal\_leave\_booth :

$(T1.Car \times T1.Card) \times Booth \times T1.Time \rightarrow$   
 $(T1.Car \times T1.Card) \times Booth \times T1.Car_Exit_Rec$

illegal\_leave\_booth((c, dc), b, t)  $\equiv$

```

let
    entry_id = T1.nil_pid,
    entry_time = T1.nil_time,
    car_pay = 0,
    (b', r) = exit_pass_sensor(c, entry_id, entry_time, car_pay, b, t)
in
    ((c, dc), b', r)
end,

```

open\_booth :

$T1.Sup \times T1.Opr \times T1.Plaza\_id \times T1.Booth\_id \times T1.Booth\_type \times T1.Fares \times T1.Time$   
 $\xrightarrow{\sim}$

```

    Booth × T1.Open_Booth_Rec
open_booth(sup, opr, p_id, b_id, b_type, f, t) ≡
  let
    sup_id = BC.read_card(T1.b_card(sup)),
    session = T1.get_session_no(sup, b_id),
    opr_id = BC.read_card(T1.b_card(opr)),
    o_info = T1.mk_Open_Info(t, sup_id, session, opr_id),
    b = mk_Booth(p_id, b_id, b_type, f, 0, T1.waiting, 0),
    r = T1.mk_Open_Booth_Rec(b_id, o_info)
  in
    (b, r)
  end
pre can_open_booth(b_id),

close_booth : T1.Sup × Booth × T1.Time → T1.Close_Booth_Rec × T1.Money
close_booth(sup, b, t) ≡
  let
    b_id = booth_id(b),
    m = money(b),
    sup_id = BC.read_card(T1.b_card(sup)),
    c_info = T1.mk_Close_Info(t, sup_id),
    r = T1.mk_Close_Booth_Rec(b_id, c_info, m)
  in
    (r, m)
  end,

can_open_booth : T1.Booth_id → Bool
can_open_booth(b_id) ≡ T1.check_comps(b_id) = T1.ok,

entry_pass_sensor : T1.Car × Booth × T1.Time → Booth × T1.Car_Entry_Rec
entry_pass_sensor(c, b, t) ≡
  let
    ps = I.pass_sensor(c, sensor(b)),
    ct' = T1.add_counter(counter(b)),
    b' =
      mk_Booth(plaza_id(b), booth_id(b), booth_type(b), fares(b), money(b), T1.waiting, ct')
  in
    if ps = T1.illegal then
      (b', T1.mk_Illegal_Entry_Rec(t, plaza_id(b), booth_id(b)))
    else
      (b', T1.mk_Legal_Entry_Rec(T1.car_id(c), t, plaza_id(b), booth_id(b)))
    end
  end,

exit_pass_sensor :
  T1.Car × T1.Opt_plaza_id × T1.Opt_time × T1.Money × Booth × T1.Time →
  Booth × T1.Car_Exit_Rec
exit_pass_sensor(c, entry_id, entry_time, car_pay, b, t) ≡
  let
    ps = I.pass_sensor(c, sensor(b)),
    ct' = T1.add_counter(counter(b)),

```

```

    b' =
      mk_Booth(plaza_id(b), booth_id(b), booth_type(b), fares(b), money(b), T1.waiting, ct')
  in
    if ps = T1.illegal then
      (b', T1.mk_Illegal_Exit_Rec(t, plaza_id(b), booth_id(b)))
    else
      (
        b',
        T1.mk_Legal_Exit_Rec
          (T1.car_id(c), entry_time, entry_id, t, plaza_id(b), booth_id(b), car_pay)
      )
    end
  end,

payment :
  (T1.Money × T1.PP_Card) × T1.Payment × T1.Money → (T1.Money × T1.PP_Card)
payment((cash, ppc), pm, pay) ≡
  case pm of
    T1.card_first →
      let ppc_val = PC.get_balance(ppc) in
        if ppc_val ≥ pay then
          let ppc' = PC.change_balance(ppc, ppc_val - pay) in (cash, ppc') end
        else
          let ppc' = PC.change_balance(ppc, 0), cash' = (cash + ppc_val - pay) in
            (cash', ppc')
          end
        end
      end,
    T1.cash_first →
      if cash ≥ pay then
        (cash - pay, ppc)
      else
        let
          ppc_val = PC.get_balance(ppc),
          ppc' = PC.change_balance(ppc, ppc_val + cash - pay)
        in
          (0, ppc')
        end
      end
  end
end

axiom
[open_booth_conv]
∀
  sup : T1.Sup,
  opr : T1.Opr,
  p_id : T1.Plaza_id,
  b_id : T1.Booth_id,
  b_type : T1.Booth_type,
  f : T1.Fares,
  t : T1.Time

```

```

    •
    can_open_booth(b_id) ⇒ (open_booth(sup, opr, p_id, b_id, b_type, f, t) post true)
  end

```

## Other Component Modules

**scheme**

BCT =

**class**

**value**

read\_card : T1.B\_Card → T1.ID

read\_card(bc) ≡ T1.id(bc)

**end**

**scheme**

DCT =

**class**

**value**

legal\_enter\_booth : T1.Car × T1.Plaza\_id × T1.Booth\_id × T1.Time → T1.Card

legal\_enter\_booth(c, p\_id, b\_id, t) ≡

**let**

info = T1.mk\_Card\_Info(T1.car\_id(c), T1.mk\_pid(p\_id), b\_id, T1.mk\_time(t)),

dc = T1.mk\_Card(T1.mk\_info(info), T1.empty)

**in**

dc

**end,**

legal\_leave\_booth :

(T1.Car × T1.Card) × T1.Plaza\_id × T1.Booth\_id × T1.Time →

T1.Opt\_plaza\_id × T1.Opt\_time × T1.Card

legal\_leave\_booth((c, dc), p\_id, b\_id, t) ≡

**let**

entry\_id = T1.plaza\_id(T1.pass\_info(T1.entry\_info(dc))),

entry\_time = T1.time(T1.pass\_info(T1.entry\_info(dc))),

info = T1.mk\_Card\_Info(T1.car\_id(c), T1.mk\_pid(p\_id), b\_id, T1.mk\_time(t)),

dc' = T1.mk\_Card(T1.entry\_info(dc), T1.mk\_info(info))

**in**

(entry\_id, entry\_time, dc')

**end**

**end**

**scheme**

PPCT =

**class**

**value**

get\_balance : T1.PP\_Card → T1.Money

get\_balance(ppc) ≡ T1.val(ppc),

change\_balance : T1.PP\_Card × T1.Money → T1.PP\_Card

```

    change_balance(ppc, v) ≡ T1.ppc(v)
  end

scheme
  IOL =
    class
      value
        pass_sensor : T1.Car × T1.Sensor_state → T1.Passing_status
        pass_sensor(c, ss) ≡ if ss = T1.expect then T1.ok else T1.illegal end
    end

```

## A.4 Final Specification: Imperative Concurrent

### C\_TYPES Module

```

scheme
  C_TYPES =
    extend TYPES1 with
      class
        type
          Legal_enter_plaza_result == fail | ok(car : Car, dcard : Card),
          Illegal_enter_plaza_result == fail | ok(car : Car, dcard : Card),
          Legal_leave_plaza_result == fail | ok(car : Car, dcard : Card),
          Illegal_leave_plaza_result == fail | ok(car : Car, dcard : Card),
          Plaza_open_booth_result == fail | ok,
          Plaza_close_booth_result == fail | ok,
          Open_booth_result == fail | ok(rec : Open_Booth_Rec)
        end

      object
        CT : C_TYPES
      end

```

### C\_PLAZA Module

```

scheme
  C_PLAZA =
    hide CH, V, BS, main in
      class
        object
          /* Booths */
          BS[i : BI] : C_BOOTH,
          /* Variables */
          V :
            class
              variable
                plaza_id : CT.Plaza_id,

```

```

    plaza_type : CT.Plaza_type,
    fares : CT.Fares,
    open_booths : BI-set,
    records : CT.Record*,
    plaza_money : CT.Money
  end,
/* Channels */
CH :
  class
    channel
      legal_enter_plaza : CT.Car × CT.Time,
      legal_enter_plaza_res : CT.Legal_enter_plaza_result,
      illegal_enter_plaza : CT.Car × CT.Time,
      illegal_enter_plaza_res : CT.Illegal_enter_plaza_result,
      legal_leave_plaza : (CT.Car × CT.Card) × CT.Time,
      legal_leave_plaza_res : CT.Legal_leave_plaza_result,
      illegal_leave_plaza : (CT.Car × CT.Card) × CT.Time,
      illegal_leave_plaza_res : CT.Illegal_leave_plaza_result,
      open_booth : CT.Sup × CT.Opr × BI × CT.Time,
      open_booth_res : CT.Plaza_open_booth_result,
      close_booth : CT.Sup × BI × CT.Time,
      close_booth_res : CT.Plaza_close_booth_result
    end
  end

type BI

axiom
/* BI is finite */
[booths_finite] card { i | i : BI } post true

value
legal_enter_plaza :
  CT.Car × CT.Time → in CH.any out CH.any CT.Legal_enter_plaza_result
legal_enter_plaza(c, t) ≡ CH.legal_enter_plaza ! (c, t) ; CH.legal_enter_plaza_res?,

illegal_enter_plaza :
  CT.Car × CT.Time → in CH.any out CH.any CT.Illegal_enter_plaza_result
illegal_enter_plaza(c, t) ≡ CH.illegal_enter_plaza ! (c, t) ; CH.illegal_enter_plaza_res?,

legal_leave_plaza :
  (CT.Car × CT.Card) × CT.Time → in CH.any out CH.any CT.Legal_leave_plaza_result
legal_leave_plaza(c, dc) ≡ CH.legal_leave_plaza ! (c, dc) ; CH.legal_leave_plaza_res?,

illegal_leave_plaza :
  (CT.Car × CT.Card) × CT.Time → in CH.any out CH.any CT.Illegal_leave_plaza_result
illegal_leave_plaza(c, dc) ≡ CH.legal_leave_plaza ! (c, dc) ; CH.illegal_leave_plaza_res?,

open_booth :
  CT.Sup × CT.Opr × BI × CT.Time →
  in CH.any out CH.any CT.Plaza_open_booth_result
open_booth(sup, opr, i, t) ≡ CH.open_booth ! (sup, opr, i, t) ; CH.open_booth_res?,

```

```

close_booth :
  CT.Sup × BI × CT.Time → in CH.any out CH.any CT.Plaza_close_booth_result
close_booth(sup, i, t) ≡ CH.close_booth ! (sup, i, t) ; CH.close_booth_res?,

choose_booth : CT.Car  $\xrightarrow{\sim}$  BI,

give_booth_id : BI  $\xrightarrow{\sim}$  CT.Booth_id,

give_booth_type : BI  $\xrightarrow{\sim}$  CT.Booth_type,

/* initialization */
init :
  CT.Plaza_id × CT.Plaza_type × CT.Fares → in CH.any out CH.any write V.any Unit
init(p_id, p_type, f) ≡
  || { BS[i].init() | i : BI }
  ||
  (
    V.plaza_id := p_id ;
    V.plaza_type := p_type ;
    V.fares := f ; V.open_booths := {} ; V.records := ⟨⟩ ; V.plaza_money := 0 ; main()
  ),

/* main */
main : Unit → in CH.any out CH.any write V.any Unit
main() ≡
  while true do
    let (sup, opr, i, t) = CH.open_booth? in
      if i ∉ V.open_booths then
        let
          b_id = give_booth_id(i),
          b_type = give_booth_type(i),
          res = BS[i].open_booth(sup, opr, V.plaza_id, b_id, b_type, V.fares, t)
        in
          if res ≠ CT.fail then
            V.open_booths := V.open_booths ∪ {i} ;
            V.records := V.records ^ ⟨CT.rec(res)⟩ ; CH.open_booth_res ! CT.ok
          else
            CH.open_booth_res ! CT.fail
          end
        end
      else
        CH.open_booth_res ! CT.fail
      end
    end
  []
  let (c, t) = CH.legal_enter_plaza?, i = choose_booth(c) in
    if i ∈ V.open_booths then
      let ((c', dc), r) = BS[i].legal_enter_booth(c, t) in
        V.records := V.records ^ ⟨r⟩ ; CH.legal_enter_plaza_res ! CT.ok(c', dc)
      end

```

```

    else
      CH.legal_enter_plaza_res ! CT.fail
    end
  end
end
[]
let ((c, dc), t) = CH.legal_leave_plaza?, i = choose_booth(c) in
  if i ∈ V.open_booths then
    let ((c', dc'), r) = BS[i].legal_leave_booth((c, dc), t) in
      V.records := V.records ^ ⟨r⟩ ; CH.legal_leave_plaza_res ! CT.ok(c', dc')
    end
  else
    CH.legal_leave_plaza_res ! CT.fail
  end
end
[]
let (sup, i, t) = CH.close_booth? in
  if i ∈ V.open_booths then
    let (r, m) = BS[i].close_booth(sup, t) in
      V.open_booths := V.open_booths \ {i} ;
      V.records := V.records ^ ⟨r⟩ ;
      V.plaza_money := V.plaza_money + m ; CH.close_booth_res ! CT.ok
    end
  else
    CH.close_booth_res ! CT.fail
  end
end
end
end
end

axiom
[ give_booth_id_conv ] ∀ i : BI • i ∉ V.open_booths ⇒ (give_booth_id(i) post true),

[ choose_booth_ax ] ∀ c : CT.Car • V.open_booths ≠ {} ⇒ choose_booth(c) ∈ V.open_booths,

[ choose_booth_conv ] ∀ c : CT.Car • V.open_booths ≠ {} ⇒ (choose_booth(c) post true)
end

```

## C\_BOOTH Module

```

scheme
C_BOOTH =
  hide CH, BC, DC, PC, I, main in
  class
  object
    /* Badge Card Terminal */
    BC : C_BCT,
    /* Duty Card Terminal */
    DC : C_DCT,
    /* Pre-Paid Card Terminal */
    PC : C_PPCT,

```

```

/* Car Sensor */
I : C_IOL,
/* Variables */
V :
  class
    variable
      booth_id : CT.Booth_id,
      plaza_id : CT.Plaza_id,
      booth_type : CT.Booth_type,
      fares : CT.Fares,
      money : CT.Money,
      sensor : CT.Sensor_state,
      counter : Nat
    end,
/* Channels */
CH :
  class
    channel
      legal_enter_booth : CT.Car × CT.Time,
      legal_enter_booth_res : (CT.Car × CT.Card) × CT.Car_Entry_Rec,
      illegal_enter_booth : CT.Car × CT.Time,
      illegal_enter_booth_res : (CT.Car × CT.Card) × CT.Car_Entry_Rec,
      legal_leave_booth : (CT.Car × CT.Card) × CT.Time,
      legal_leave_booth_res : (CT.Car × CT.Card) × CT.Car_Exit_Rec,
      illegal_leave_booth : (CT.Car × CT.Card) × CT.Time,
      illegal_leave_booth_res : (CT.Car × CT.Card) × CT.Car_Exit_Rec,
      open_booth :
        CT.Sup × CT.Opr × CT.Plaza_id × CT.Booth_id × CT.Booth_type × CT.Fares ×
        CT.Time,
      open_booth_res : CT.Open_booth_result,
      close_booth : CT.Sup × CT.Time,
      close_booth_res : CT.Close_Booth_Rec × CT.Money,
      entry_pass_sensor : CT.Car × CT.Time,
      entry_pass_sensor_res : CT.Car_Entry_Rec,
      exit_pass_sensor :
        CT.Car × CT.Opt_plaza_id × CT.Opt_time × CT.Money × CT.Time,
      exit_pass_sensor_res : CT.Car_Exit_Rec
    end
  end
value
legal_enter_booth :
  CT.Car × CT.Time →
  in CH.any out CH.any (CT.Car × CT.Card) × CT.Car_Entry_Rec
legal_enter_booth(c, t) ≡ CH.legal_enter_booth ! (c, t) ; CH.legal_enter_booth_res?,

illegal_enter_booth :
  CT.Car × CT.Time →
  in CH.any out CH.any (CT.Car × CT.Card) × CT.Car_Entry_Rec
illegal_enter_booth(c, t) ≡ CH.illegal_enter_booth ! (c, t) ; CH.illegal_enter_booth_res?,

legal_leave_booth :

```

```

    (CT.Car × CT.Card) × CT.Time →
    in CH.any out CH.any (CT.Car × CT.Card) × CT.Car_Exit_Rec
legal_leave_booth((c, dc), t) ≡ CH.legal_leave_booth ! ((c, dc), t) ; CH.legal_leave_booth_res?,

illegal_leave_booth :
    (CT.Car × CT.Card) × CT.Time →
    in CH.any out CH.any (CT.Car × CT.Card) × CT.Car_Exit_Rec
illegal_leave_booth((c, dc), t) ≡
    CH.illegal_leave_booth ! ((c, dc), t) ; CH.illegal_leave_booth_res?,

open_booth :
    CT.Sup × CT.Opr × CT.Plaza_id × CT.Booth_id × CT.Booth_type × CT.Fares ×
    CT.Time
    →
    in CH.any out CH.any CT.Open_booth_result
open_booth(sup, opr, p_id, b_id, b_type, f, t) ≡
    CH.open_booth ! (sup, opr, p_id, b_id, b_type, f, t) ; CH.open_booth_res?,

close_booth :
    CT.Sup × CT.Time → in CH.any out CH.any CT.Close_Booth_Rec × CT.Money
close_booth(sup, t) ≡ CH.close_booth ! (sup, t) ; CH.close_booth_res?,

entry_pass_sensor : CT.Car × CT.Time → in CH.any out CH.any CT.Car_Entry_Rec
entry_pass_sensor(c, t) ≡ CH.entry_pass_sensor ! (c, t) ; CH.entry_pass_sensor_res?,

exit_pass_sensor :
    CT.Car × CT.Opt_plaza_id × CT.Opt_time × CT.Money × CT.Time →
    in CH.any out CH.any CT.Car_Exit_Rec
exit_pass_sensor(c, entry_id, entry_time, car_pay, t) ≡
    CH.exit_pass_sensor ! (c, entry_id, entry_time, car_pay, t) ; CH.exit_pass_sensor_res?,

get_session_no : CT.Sup × CT.Booth_id → Nat,

/* initialization */
init : Unit → in CH.any out CH.any write V.any Unit
init() ≡ BC.init() || DC.init() || PC.init() || I.init() || (V.sensor := CT.waiting ; main()),

/* main */
main : Unit → in CH.any out CH.any write V.any Unit
main() ≡
    while true do
        let (sup, opr, p_id, b_id, b_type, f, t) = CH.open_booth? in
            if CT.check_comps(b_id) = CT.ok then
                let
                    sup_id = BC.read_card(CT.b_card(sup)),
                    session = CT.get_session_no(sup, b_id),
                    opr_id = BC.read_card(CT.b_card(opr)),
                    o_info = CT.mk_Open_Info(t, sup_id, session, opr_id),
                    r = CT.mk_Open_Booth_Rec(b_id, o_info)
                in
                    V.plaza_id := p_id ;

```

```

    V.booth_id := b_id ;
    V.booth_type := b_type ;
    V.fares := f ;
    V.money := 0 ;
    V.sensor := CT.waiting ; V.counter := 0 ; CH.open_booth_res ! CT.ok(r)
  end
else
  CH.open_booth_res ! CT.fail
end
end
[]
let
  (c, t) = CH.legal_enter_booth?,
  dc = DC.legal_enter_booth(c, V.plaza_id, V.booth_id, t)
in
  V.sensor := CT.waiting ; CH.legal_enter_booth_res ! ((c, dc), entry_pass_sensor(c, t))
end
[]
let (c, t) = CH.illegal_enter_booth?, dc = CT.mk_Card(CT.empty, CT.empty) in
  CH.illegal_enter_booth_res ! ((c, dc), entry_pass_sensor(c, t))
end
[]
let ((c, dc), t) = CH.legal_leave_booth? in
  if CT.entry_info(dc) = CT.empty then
    let
      entry_id = CT.nil_pid,
      entry_time = CT.nil_time,
      car_pay = CT.charge(CT.car_id(c), entry_id, entry_time, V.plaza_id, t, V.fares),
      pm = CT.payment_method(c),
      (cash', ppc') = payment((CT.cash(c), CT.pp_card(c)), pm, car_pay),
      c' = CT.mk_Car(CT.car_id(c), cash', ppc')
    in
      V.money := V.money + car_pay ;
      V.sensor := CT.expect ;
      CH.legal_leave_booth_res !
        ((c', dc), exit_pass_sensor(c, entry_id, entry_time, car_pay, t))
    end
  else
    let
      (entry_id, entry_time, dc') =
        DC.legal_leave_booth((c, dc), V.plaza_id, V.booth_id, t),
      car_pay = CT.charge(CT.car_id(c), entry_id, entry_time, V.plaza_id, t, V.fares),
      pm = CT.payment_method(c),
      (cash', ppc') = payment((CT.cash(c), CT.pp_card(c)), pm, car_pay),
      c' = CT.mk_Car(CT.car_id(c), cash', ppc')
    in
      V.money := V.money + car_pay ;
      V.sensor := CT.expect ;
      CH.legal_leave_booth_res !
        ((c', dc'), exit_pass_sensor(c, entry_id, entry_time, car_pay, t))
    end
  end
end

```

```

    end
  end
  []
  let
    ((c, dc), t) = CH.illegal_leave_booth?,
    entry_id = CT.nil_pid,
    entry_time = CT.nil_time,
    car_pay = 0
  in
    CH.illegal_leave_booth_res !
    ((c, dc), exit_pass_sensor(c, entry_id, entry_time, car_pay, t))
  end
  []
  let
    (sup, t) = CH.close_booth?,
    sup_id = BC.read_card(CT.b_card(sup)),
    c_info = CT.mk_Close_Info(t, sup_id),
    r = CT.mk_Close_Booth_Rec(V.booth_id, c_info, V.money)
  in
    CH.close_booth_res ! (r, V.money)
  end
  []
  let
    (c, t) = CH.entry_pass_sensor?,
    ps = I.pass_sensor(c, V.sensor),
    ct' = CT.add_counter(V.counter)
  in
    V.sensor := CT.waiting ;
    V.counter := ct' ;
    if ps = CT.illegal then
      CH.entry_pass_sensor_res ! CT.mk_Illegal_Entry_Rec(t, V.plaza_id, V.booth_id)
    else
      CH.entry_pass_sensor_res !
      CT.mk_Legal_Entry_Rec(CT.car_id(c), t, V.plaza_id, V.booth_id)
    end
  end
  end
  []
  let
    (c, entry_id, entry_time, car_pay, t) = CH.exit_pass_sensor?,
    ps = I.pass_sensor(c, V.sensor),
    ct' = CT.add_counter(V.counter)
  in
    V.sensor := CT.waiting ;
    V.counter := ct' ;
    if ps = CT.illegal then
      CH.exit_pass_sensor_res ! CT.mk_Illegal_Exit_Rec(t, V.plaza_id, V.booth_id)
    else
      CH.exit_pass_sensor_res !
      CT.mk_Legal_Exit_Rec
      (CT.car_id(c), entry_time, entry_id, t, V.plaza_id, V.booth_id, car_pay)
    end
  end
end

```

```

    end
  end,

  payment :
    (CT.Money × CT.PP_Card) × CT.Payment × CT.Money →
    in CH.any out CH.any CT.Money × CT.PP_Card
  payment((cash, ppc), pm, pay) ≡
  case pm of
    CT.card_first →
      let ppc_val = PC.get_balance(ppc) in
      if ppc_val ≥ pay then
        let ppc' = PC.change_balance(ppc, ppc_val - pay) in (cash, ppc') end
      else
        let ppc' = PC.change_balance(ppc, 0), cash' = (cash + ppc_val - pay) in
          (cash', ppc')
        end
      end
    end,
    CT.cash_first →
      if cash ≥ pay then
        (cash - pay, ppc)
      else
        let
          ppc_val = PC.get_balance(ppc),
          ppc' = PC.change_balance(ppc, ppc_val + cash - pay)
        in
          (0, ppc')
        end
      end
    end
  end
end
end

```

## Other Component Modules

```

scheme
C_BCT =
  hide CH, main in
  class
  object
    /* Channels */
    CH : class channel read_card : CT.B_Card, read_card_res : CT.ID end

  value
    read_card : CT.B_Card → in CH.any out CH.any CT.ID
    read_card(bc) ≡ CH.read_card ! bc ; CH.read_card_res?,

    /* initialization */
    init : Unit → in CH.any out CH.any Unit
    init() ≡ main(),

```

```

/* main */
main : Unit → in CH.any out CH.any Unit
main() ≡ while true do let bc = CH.read_card? in CH.read_card_res ! CT.id(bc) end end
end

```

**scheme**

C\_DCT =

hide CH, main in

class

object

/\* Channels \*/

CH :

class

channel

legal\_enter\_booth : CT.Car × CT.Plaza\_id × CT.Booth\_id × CT.Time,

legal\_enter\_booth\_res : CT.Card,

legal\_leave\_booth : (CT.Car × CT.Card) × CT.Plaza\_id × CT.Booth\_id × CT.Time,

legal\_leave\_booth\_res : CT.Opt\_plaza\_id × CT.Opt\_time × CT.Card

end

value

legal\_enter\_booth :

CT.Car × CT.Plaza\_id × CT.Booth\_id × CT.Time → in CH.any out CH.any CT.Card

legal\_enter\_booth(c, p\_id, b\_id, t) ≡

CH.legal\_enter\_booth ! (c, p\_id, b\_id, t) ; CH.legal\_enter\_booth\_res?,

legal\_leave\_booth :

(CT.Car × CT.Card) × CT.Plaza\_id × CT.Booth\_id × CT.Time →

in CH.any out CH.any CT.Opt\_plaza\_id × CT.Opt\_time × CT.Card

legal\_leave\_booth((c, dc), p\_id, b\_id, t) ≡

CH.legal\_leave\_booth ! ((c, dc), p\_id, b\_id, t) ; CH.legal\_leave\_booth\_res?,

/\* initialization \*/

init : Unit → in CH.any out CH.any Unit

init() ≡ main(),

/\* main \*/

main : Unit → in CH.any out CH.any Unit

main() ≡

while true do

let

(c, p\_id, b\_id, t) = CH.legal\_enter\_booth?,

info = CT.mk\_Card\_Info(CT.car\_id(c), CT.mk\_pid(p\_id), b\_id, CT.mk\_time(t)),

dc = CT.mk\_Card(CT.mk\_info(info), CT.empty)

in

CH.legal\_enter\_booth\_res ! dc

end

□

let

((c, dc), p\_id, b\_id, t) = CH.legal\_leave\_booth?,

```

    entry_id = CT.plaza_id(CT.pass_info(CT.entry_info(dc))),
    entry_time = CT.time(CT.pass_info(CT.entry_info(dc))),
    info = CT.mk_Card_Info(CT.car_id(c), CT.mk_pid(p_id), b_id, CT.mk_time(t)),
    dc' = CT.mk_Card(CT.entry_info(dc), CT.mk_info(info))
  in
    CH.legal_leave_booth_res ! (entry_id, entry_time, dc')
  end
end
end

scheme
C_PPCT =
  hide CH, main in
    class
      object
        /* Channels */
        CH :
          class
            channel
              get_balance : CT.PP_Card,
              get_balance_res : CT.Money,
              change_balance : CT.PP_Card × CT.Money,
              change_balance_res : CT.PP_Card
            end
          end
        value
          get_balance : CT.PP_Card → in CH.any out CH.any CT.Money
          get_balance(ppc) ≡ CH.get_balance ! ppc ; CH.get_balance_res?,

          change_balance : CT.PP_Card × CT.Money → in CH.any out CH.any CT.PP_Card
          change_balance(ppc, v) ≡ CH.change_balance ! (ppc, v) ; CH.change_balance_res?,

          /* initialization */
          init : Unit → in CH.any out CH.any Unit
          init() ≡ main(),

          /* main */
          main : Unit → in CH.any out CH.any Unit
          main() ≡
            while true do
              let ppc = CH.get_balance? in CH.get_balance_res ! CT.val(ppc) end
              []
              let (ppc, v) = CH.change_balance? in CH.change_balance_res ! CT.ppc(v) end
            end
          end
        end
      end
    end

scheme
C_IOL =
  hide CH, main in
    class

```

```

object
  /* Channels */
  CH :
    class
      channel pass_sensor : CT.Car × CT.Sensor_state, pass_sensor_res : CT.Passing_status
    end

value
  pass_sensor : CT.Car × CT.Sensor_state → in CH.any out CH.any CT.Passing_status
  pass_sensor(c, ss) ≡ CH.pass_sensor ! (c, ss) ; CH.pass_sensor_res?,

  /* initialization */
  init : Unit → in CH.any out CH.any Unit
  init() ≡ main(),

  /* main */
  main : Unit → in CH.any out CH.any Unit
  main() ≡
    let (c, ss) = CH.pass_sensor? in
      if ss = CT.expect then
        CH.pass_sensor_res ! CT.ok
      else
        CH.pass_sensor_res ! CT.illegal
      end
    end
end

```

## B Justifications

This section contains generated proof obligations for development relation of step 2 and step 3. These proof obligations could be justified using RAISE justification editor. Detailed explanation of justification is given in [4] and [3].

### Development relation: Step 2

```

theory A_TW_0_1 :
  axiom ⊢ A_TW_1 ≼ A_TW_0
end

```

```

justification A_TW_0_1_J of A_TW_0_1 :
  ⊢ ⊢ A_TW_1 ≼ A_TW_0 ⊣
  implementation_relation_expansion_inf :
  • ⊢ ⊢ ∀ (c, p) : T.Car × Plaza •
    can_enter_plaza(c, p) ≡
      has_open_booth(p) ∧ (plaza_type(p) = T.entry ∨ plaza_type(p) = T.both) ⊣

```

- <:argument:>*
- $\lfloor \forall ((c, dc), p) : (T.Car \times T.Card) \times Plaza \bullet$   
 $can\_leave\_plaza((c, dc), p) \equiv$   
 $has\_open\_booth(p) \wedge (plaza\_type(p) = T.exit \vee plaza\_type(p) = T.both) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc), p') = legal\_enter\_plaza(c, p, t) \mathbf{in}$   
 $T.car\_money(c') = T.car\_money(c) \wedge$   
 $booths\_money(p') = booths\_money(p) \wedge money(p') = money(p)$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_enter\_plaza(c, p) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc), p') = illegal\_enter\_plaza(c, p, t) \mathbf{in}$   
 $T.car\_money(c') = T.car\_money(c) \wedge$   
 $booths\_money(p') = booths\_money(p) \wedge money(p') = money(p)$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_enter\_plaza(c, p) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, dc : T.Card, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc'), p') = legal\_leave\_plaza((c, dc), p, t), pay = charge((c, dc), p, t) \mathbf{in}$   
 $T.car\_money(c') = T.car\_money(c) - pay \wedge$   
 $booths\_money(p') = booths\_money(p) + pay \wedge money(p') = money(p)$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_leave\_plaza((c, dc), p) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, dc : T.Card, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc'), p') = illegal\_leave\_plaza((c, dc), p, t) \mathbf{in}$   
 $T.car\_money(c') = T.car\_money(c) \wedge$   
 $booths\_money(p') = booths\_money(p) \wedge money(p') = money(p)$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_leave\_plaza((c, dc), p) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc), p') = legal\_enter\_plaza(c, p, t) \mathbf{in}$   
 $records(p') = add\_legal\_entry\_rec(c, p, t, records(p))$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_enter\_plaza(c, p) \rfloor$
- <:argument:>*
- $\lfloor \forall c : T.Car, p : Plaza, t : T.Time \bullet$   
 $let ((c', dc), p') = illegal\_enter\_plaza(c, p, t) \mathbf{in}$   
 $records(p') = add\_illegal\_entry\_rec(c, p, t, records(p))$   
 $\mathbf{end} \equiv$   
 $\mathbf{true}$   
 $\mathbf{pre} \ can\_enter\_plaza(c, p) \rfloor$
- <:argument:>*

- $\ulcorner \forall c : T.Car, dc : T.Card, p : Plaza, t : T.Time \bullet$   
 $\text{let } ((c', dc), p') = \text{legal\_leave\_plaza}((c, dc), p, t) \text{ in}$   
 $\text{records}(p') = \text{add\_legal\_exit\_rec}((c, dc), p, t, \text{records}(p))$   
 $\text{end} \equiv$   
 $\text{true}$   
 $\text{pre can\_leave\_plaza}((c, dc), p) \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \forall c : T.Car, dc : T.Card, p : Plaza, t : T.Time \bullet$   
 $\text{let } ((c', dc), p') = \text{illegal\_leave\_plaza}((c, dc), p, t) \text{ in}$   
 $\text{records}(p') = \text{add\_illegal\_exit\_rec}((c, dc), p, t, \text{records}(p))$   
 $\text{end} \equiv$   
 $\text{true}$   
 $\text{pre can\_leave\_plaza}((c, dc), p) \urcorner$   
 $\langle :argument: \rangle$

end

### Development relation: Step 3

**devt\_relation** A\_TW\_1\_PLAZA ( PLAZA for A\_TW\_11 ) :  
 $\vdash \text{PLAZA} \preceq \text{A\_TW\_11}$

**justification** A\_TW\_1\_PLAZA\_J of A\_TW\_1\_PLAZA :

$\ulcorner \vdash \text{PLAZA} \preceq \text{A\_TW\_11} \urcorner$   
implementation\_relation\_expansion\_inf :

- $\ulcorner \text{card } \{ i \mid i : BI \} \text{ post true} \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \forall i : BI, bs : Booths \bullet i \in \text{dom } bs \Rightarrow (bs(i) \text{ post true}) \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \text{let } x = \text{add\_legal\_entry\_rec} \text{ in}$   
 $\forall y : T1.Car \times Plaza \times T1.Time \times T1.Records \bullet x(y) \text{ post true}$   
 $\text{end} \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \text{let } x = \text{add\_illegal\_entry\_rec} \text{ in}$   
 $\forall y : T1.Car \times Plaza \times T1.Time \times T1.Records \bullet x(y) \text{ post true}$   
 $\text{end} \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \text{let } x = \text{add\_legal\_exit\_rec} \text{ in}$   
 $\forall y : (T1.Car \times T1.Card) \times Plaza \times T1.Time \times T1.Records \bullet x(y) \text{ post true}$   
 $\text{end} \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \text{let } x = \text{add\_illegal\_exit\_rec} \text{ in}$   
 $\forall y : (T1.Car \times T1.Card) \times Plaza \times T1.Time \times T1.Records \bullet x(y) \text{ post true}$   
 $\text{end} \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \forall p : Plaza \bullet \text{booths\_money}(p) \equiv \text{sum\_money}(\text{booths}(p)) \urcorner$   
 $\langle :argument: \rangle$
- $\ulcorner \forall p : Plaza \bullet \text{has\_open\_booth}(p) \equiv \text{dom } \text{booths}(p) \neq \{ \} \urcorner$   
 $\langle :argument: \rangle$

- $\ulcorner \forall (c, p) : \text{T1.Car} \times \text{Plaza} \bullet$   
 $\text{can\_enter\_booth}(c, p) \equiv$   
 $\text{has\_open\_booth}(p) \wedge (\text{plaza\_type}(p) = \text{T1.entry} \vee \text{plaza\_type}(p) = \text{T1.both}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall ((c, dc), p) : (\text{T1.Car} \times \text{T1.Card}) \times \text{Plaza} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \equiv$   
 $\text{has\_open\_booth}(p) \wedge (\text{plaza\_type}(p) = \text{T1.exit} \vee \text{plaza\_type}(p) = \text{T1.both}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall (c, p) : \text{T1.Car} \times \text{Plaza} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \equiv$   
 $\text{has\_open\_booth}(p) \wedge (\text{plaza\_type}(p) = \text{T1.entry} \vee \text{plaza\_type}(p) = \text{T1.both}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall ((c, dc), p) : (\text{T1.Car} \times \text{T1.Card}) \times \text{Plaza} \bullet$   
 $\text{can\_leave\_plaza}(c, dc), p) \equiv$   
 $\text{has\_open\_booth}(p) \wedge (\text{plaza\_type}(p) = \text{T1.exit} \vee \text{plaza\_type}(p) = \text{T1.both}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c : \text{T1.Car}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_booth}(c, p) \Rightarrow (\text{legal\_enter\_booth}(c, p, t) \text{ post true}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c : \text{T1.Car}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_booth}(c, p) \Rightarrow (\text{illegal\_enter\_booth}(c, p, t) \text{ post true}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c, c' : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_booth}(c, p) \wedge ((c', dc), b, i) = \text{legal\_enter\_booth}(c, p, t) \Rightarrow$   
 $\text{T1.car\_money}(c') = \text{T1.car\_money}(c) \wedge$   
 $i \in \text{dom booths}(p) \wedge \text{booth\_money}(b) = \text{booth\_money}(\text{booths}(p)(i)) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c, c' : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_booth}(c, p) \wedge ((c', dc), b, i) = \text{illegal\_enter\_booth}(c, p, t) \Rightarrow$   
 $\text{T1.car\_money}(c') = \text{T1.car\_money}(c) \wedge$   
 $i \in \text{dom booths}(p) \wedge \text{booth\_money}(b) = \text{booth\_money}(\text{booths}(p)(i)) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \Rightarrow (\text{legal\_leave\_booth}((c, dc), p, t) \text{ post true}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \Rightarrow (\text{illegal\_leave\_booth}((c, dc), p, t) \text{ post true}) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c, c' : \text{T1.Car}, dc, dc' : \text{T1.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \wedge ((c', dc'), b, i) = \text{legal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $\text{T1.car\_money}(c') = \text{T1.car\_money}(c) - \text{charge}((c, dc), p, t) \wedge$   
 $i \in \text{dom booths}(p) \wedge$   
 $\text{booth\_money}(b) = \text{booth\_money}(\text{booths}(p)(i)) + \text{charge}((c, dc), p, t) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c, c' : \text{T1.Car}, dc, dc' : \text{T1.Card}, p : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_booth}((c, dc), p) \wedge ((c', dc'), b, i) = \text{illegal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $\text{T1.car\_money}(c') = \text{T1.car\_money}(c) \wedge$   
 $i \in \text{dom booths}(p) \wedge \text{booth\_money}(b) = \text{booth\_money}(\text{booths}(p)(i)) \urcorner$   
 $\langle : \text{argument} : \rangle$
- $\ulcorner \forall c : \text{T1.Car}, p : \text{Plaza}, t : \text{T1.Time} \bullet$

- $\text{can\_enter\_plaza}(c, p) \Rightarrow (\text{legal\_enter\_plaza}(c, p, t) \text{ post true}) \lrcorner$
- <:argument:>
- $\lrcorner \forall c : \text{T1.Car}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \Rightarrow (\text{illegal\_enter\_plaza}(c, p, t) \text{ post true}) \lrcorner$
- <:argument:>
- $\lrcorner \forall c, c', c'' : \text{T1.Car}, dc, dc' : \text{T1.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \wedge ((c', dc), p') = \text{legal\_enter\_plaza}(c, p, t) \Rightarrow$   
 $((c'', dc'), b, i) = \text{legal\_enter\_booth}(c, p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc = dc' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \dagger [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_legal\_entry\_rec}(c, p, t, \text{records}(p)) \lrcorner$
- <:argument:>
- $\lrcorner \forall c, c', c'' : \text{T1.Car}, dc, dc' : \text{T1.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_enter\_plaza}(c, p) \wedge ((c', dc), p') = \text{illegal\_enter\_plaza}(c, p, t) \Rightarrow$   
 $((c'', dc'), b, i) = \text{illegal\_enter\_booth}(c, p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc = dc' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \dagger [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_illegal\_entry\_rec}(c, p, t, \text{records}(p)) \lrcorner$
- <:argument:>
- $\lrcorner \forall c : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_plaza}((c, dc), p) \Rightarrow (\text{legal\_leave\_plaza}((c, dc), p, t) \text{ post true}) \lrcorner$
- <:argument:>
- $\lrcorner \forall c : \text{T1.Car}, dc : \text{T1.Card}, p : \text{Plaza}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_plaza}((c, dc), p) \Rightarrow (\text{illegal\_leave\_plaza}((c, dc), p, t) \text{ post true}) \lrcorner$
- <:argument:>
- $\lrcorner \forall c, c', c'' : \text{T1.Car}, dc, dc', dc'' : \text{T1.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_plaza}((c, dc), p) \wedge ((c', dc'), p') = \text{legal\_leave\_plaza}((c, dc), p, t) \Rightarrow$   
 $((c'', dc''), b, i) = \text{legal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc' = dc'' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \dagger [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_legal\_exit\_rec}((c, dc), p, t, \text{records}(p)) \lrcorner$
- <:argument:>
- $\lrcorner \forall c, c', c'' : \text{T1.Car}, dc, dc', dc'' : \text{T1.Card}, p, p' : \text{Plaza}, b : \text{Booth}, i : \text{BI}, t : \text{T1.Time} \bullet$   
 $\text{can\_leave\_plaza}((c, dc), p) \wedge ((c', dc'), p') = \text{illegal\_leave\_plaza}((c, dc), p, t) \Rightarrow$   
 $((c'', dc''), b, i) = \text{illegal\_leave\_booth}((c, dc), p, t) \Rightarrow$   
 $i \in \mathbf{dom} \text{booths}(p) \wedge$   
 $c' = c'' \wedge$   
 $dc' = dc'' \wedge$   
 $\text{booths}(p') = \text{booths}(p) \dagger [i \mapsto b] \wedge$   
 $\text{money}(p') = \text{money}(p) \wedge \text{records}(p') = \text{add\_illegal\_exit\_rec}((c, dc), p, t, \text{records}(p)) \lrcorner$
- <:argument:>
- $\lrcorner \forall p : \text{Plaza}, b : \text{Booth}, bs : \text{Booths}, i : \text{BI} \bullet$   
 $bs = \text{booths}(p) \wedge i \in \mathbf{dom} \text{booths}(p) \Rightarrow$   
 $(\text{sum\_money}(bs \dagger [i \mapsto b]) \equiv \text{sum\_money}(bs) - \text{booth\_money}(bs(i)) + \text{booth\_money}(b)) \lrcorner$

<:argument:>  
end