Department of Computing Science and Mathematics
University of Stirling

Using a formal user-centred model to build a formal system-centred model

Robert G. Clark and Ana M. D. Moreira

Technical Report CSM-140

March 1997

Department of Computing Science and Mathematics
University of Stirling

Using a formal user-centred model to build a formal system-centred model

Robert G. Clark and Ana M. D. Moreira

Department of Computing Science and Mathematics, University of Stirling
Stirling FK9 4LA, Scotland
and
Department of Informatics, Faculty of Science and Technology
New University of Lisbon, Portugal

Email rgc@compsci.stirling.ac.uk | amm@di.fct.unl.pt

Technical Report CSM-140

March 1997

Contents

List of Figures

Abstract

We have been investigating the process of constructing an executable, formal and object-oriented specification from a set of informal requirements. This is not an easy task as there is a wide gap between the structure and notation of a formal specification and the requirements from which it is to be derived. It also cannot be a formal process.

As informal requirements are usually expressed in terms of the behaviour which the environment expects from the system, we propose that the construction of a formal and executable user-centred model should precede the construction of a formal object-oriented specification. By prototyping the user-centred model, we can both validate it with respect to the requirements and show up inconsistencies in the requirements. The user-centred model can then be used to support the construction of a formal system-centred model, i.e. the object-oriented specification.

When both models are expressed in the same executable formal language, the informal task of validating the object-oriented specification with respect to the requirements can be replaced by verifying that it is equivalent to the user-centred model. We already have the ROOA (Rigorous Object-Oriented Analysis) method, which proposes a process to build a formal system-centred model. Now we are proposing a process to build a user-centred model. As an example of this approach, we show its use within the ROOA method.

Chapter 1

Introduction

1.1 Background

As we believe that there are many benefits in a formal approach to software development, we have been investigating how formality can be introduced into the object-oriented analysis process. The first result of this study was the Rigorous Object-Oriented Analysis (ROOA) method [13, 14, 15, 16, 17].

The ROOA method provides a process for the systematic construction of an executable, formal and object-oriented specification from a set of informal requirements. The aim is to give a complete and accurate description of the static, dynamic and functional aspects of a problem in terms of entities from the problem domain. This specification can be analysed for syntactic or static semantic errors, can form the basis for formal reasoning and can be used as the starting point in a formal development trajectory.

In ROOA, the object-oriented specification is expressed in the formal description technique LOTOS [3]. As LOTOS specifications are executable, rapid prototyping can be used to ensure completeness and consistency and for validation with respect to the requirements. The systematic creation of a formal model, combined with rapid prototyping, shows up inconsistencies, contradictions, ambiguities and omissions in the requirements sufficiently early in the development that feedback can be given to the requirements capture process.

1.2 Formal specifications from informal requirements

A major objective of the ROOA method is to provide a systematic process whereby an initial formal requirements specification can be developed from a set of informal requirements. This cannot be a formal process. It is also far from simple as there is a wide gulf between the structure and notation of the specification and of the requirements from which it is to be derived. In this report, we propose a means of narrowing this gap.

Informal requirements are usually expressed, and are most easily understood, in terms of the behaviour which the environment expects from the system. We therefore propose that the construction of a formal and executable *user-centred* specification should precede the construction of a formal object-oriented specification. As a user-centred specification (or model) specifies the system in terms of the behaviour expected by the environment, it is closer to the requirements and is therefore easier to create than a *system-centred* object-oriented specification. The gap between the informal requirements and the initial formal specification has therefore been narrowed. When the user-centred specification is expressed in LOTOS, prototyping can be used to validate it with respect to the informal requirements.

The formal user-centred model can then be used to help in the construction of the formal object-oriented specification, i.e. the *system-centred model.* When both models are expressed in the same executable formal language, they can be composed, and the composition executed, to increase our confidence that the object-oriented specification exhibits the expected behaviour.

We therefore construct a formal user-centred model, validate it with respect to the requirements and then use the model as an aid in the construction of a formal object-oriented specification. Once we have an initial formal specification, it is possible, at least in theory, to *verify* that it is equivalent to subsequent specifications. Therefore instead of validating that the object-oriented specification satisfies the informal requirements, we can verify that it is equivalent to the user-centred model.

Although our examples describe how a formal user-centred model can be integrated into the ROOA method, we believe that the approach of initially producing a user-centred model before proceeding to the construction of a system-centred model has general applicability and can be used with other methods and with different formal languages.

In Chapter 2, we describe the process to be followed in the building of a formal user-centred model and how it can be used as a step in the creation of a system-centred model. We then describe how we can verify that the two models are equivalent and how this approach can be integrated into the ROOA method.

In Chapter 3, we apply the method to a case study and create a user-centred model which is validated with respect to the requirements. It is then used in the creation of a ROOA system-centred model and we examine how the two models can be shown to be equivalent.

Chapter 2

The systematic creation of formal object-oriented specifications

2.1 Introduction

In this chapter, we first define a step by step process to be followed in the construction of a formal and executable user-centred model. We show how prototyping this model can both help our understanding of, and show up inconsistencies in, the informal requirements. Prototyping is also used to validate the user-centred model with respect to the requirements. We then show how the user-centred model can be used as an intermediate step in the building of a formal object-oriented specification.

As an example of the approach, we describe how the construction of a formal user-centred model can be integrated with ROOA method. As both the user-centred and ROOA models are specified using the same executable formal language, we can compose the two models and prototype the resulting specification to increase our confidence that the ROOA model exhibits the correct behaviour. We then describe how verifying that the user-centred and ROOA models are equivalent can replace the task of validating the ROOA model.

2.2 The two formal models

The *user-centred model* is concerned with the interaction of the environment with the system as seen from the viewpoint of the environment. The environment includes human users, hardware devices and other software components which we refer to collectively as *agents*. The user-centred model is a set of *agent views* where each agent view describes, from the viewpoint of an agent, a way in which the system is to be used. An agent view is concerned with the role being played. For example, a particular human being can play different roles and therefore take part in more than one agent view while several different people can play a similar role and therefore take part in different instances of the same agent view.

To summarise, the focus in the user-centred model is on the way agents use the system, not on the system itself.

The *system-centred model* is concerned with modelling an idealised view of the system and its interaction with the environment. It is the formal requirements specification of the system from which the design and eventual implementation will be developed.

2.3 Advantages of a user-centred model

The idea of constructing two models is not new. The major difference in our approach is that we propose the construction of two *formal* models both of which are expressed in the same *executable* specification language. As the user-centred model is concerned with events, i.e. interactions between the environment and the system, it is best expressed in an executable event-oriented formal language such as LOTOS [3].

A formal user-centred model provides the following benefits:

- It allows a formal model to be produced earlier in the development. This enables us to reason about the system behaviour and the requirements before the formal object-oriented specification exists.
- It is closer to the requirements and hence more easily understood and checked for validity by clients.
- It helps identify areas in which the requirements are vague or inconsistent. This will support the formulation of questions so that the human agents can clarify their requirements.
- It helps identify ambiguities and contradictions in the views of different agents.
- As each event in the user-centred model will have a direct counterpart in the object-oriented specification, the user-centred model can act as an intermediate step in the construction of the formal object-oriented specification.
- It identifies the services to be provided by the object-oriented specification, i.e. the external events in which the system takes part. The identification of the required services helps in the identification of objects in the object-oriented specification.
- It helps us determine the boundary between the environment and the system.
- The formal step of verifying that the user-centred and object-oriented specification are equivalent can replace the informal step of validating the object-oriented specification with respect to the requirements.

2.4 Related work

The creation of a formal user-centred model has also been proposed by Hsia *et al.* [10]. However, they do not:

- use an executable specification language,
- deal with how the development of a user-centred model can be integrated into the development of a formal system-centred model,
- deal with object-oriented development.

In the requirements analysis phase of the Object-Oriented Software Engineering (OOSE) method, Jacobson [11] proposes the construction of a requirements model and an analysis model which correspond, in some respects, to our user-centred and system-centred models.

The OOSE requirements model is composed of a use case model and a simplified object model called a domain object model. A use case describes possible complete sequences of events which occur in response to some action initiated by an agent. Unlike our agent views, use cases include both interactions of an agent with the system and the consequent interactions between objects in the domain object model. The use case model is the complete set of use cases.

Once a requirements model has been created, the next stage in OOSE is structuring the information. This results in the analysis model which can be used as the base for the subsequent design stage. The OOSE models are not specified in a formal language although Regnell *et al.* have shown how this can be achieved for the use case model [20].

Use cases have recently been introduced into Rumbaugh's OMT [22]. In Object Behaviour Analysis (OBA), *use scenarios* play a similar role to use cases [21].

Glinz has shown how scenarios can be specified using Statecharts [8]. Although he proposes that the specified scenarios can be integrated to provide a complete specification, he does not demonstrate how this can be done when scenarios interact with one another. He also states that future work will involve specifying both the user and system aspects of a problem in a single Statechart specification, but does not show how this can be done or provide a systematic development process.

LOTOS was designed to be used in the specification of OSI systems. With such systems, a *service specification* is produced initially to describe the system in terms of its external behaviour. This is then followed by a lower level *protocol specification* which includes internal structure [5, 25]. Although there may appear to be a similarity between a service specification and our user-centred model, the focus is different. Our user-centred model is concerned with the roles played by the agents which constitute the environment and their view of the system while a service specification can be regarded as a very high-level system-centred model.

Amyot *et al.* [1] use LOTOS to formalise what they call *Timethreads*, i.e. execution paths through the system. However, their approach is more concerned with the system-centred rather than with the user-centred model.

There is a similarity between our approach and work on specification via viewpoints [12, 18]. As with agent views, viewpoints promote a separation of concerns. However, as different viewpoints are developed by different participants using different notations, the integration of viewpoints is difficult. Agent views are more restricted than viewpoints as they are only concerned with the direct interactions between the environment and the system. A major problem in understanding and capturing user requirements is being overwhelmed by their complexity which is why we believe that restricting our attention to external events and a single specification language is a virtue.

2.5 Formalising agent views

When an agent interacts with a system there will often be a set of possible behaviours which can take place. Consider, for example, the situation in which we lift a phone and dial a number. The call may, or may not, be answered. An agent view is the set of such possible behaviours and it fully defines the agent's interaction with the system. An agent view can be represented by a tree which shows the possible alternative event sequences. We refer to a particular path through the tree, i.e. a sequence of external events, as an *agent scenario*.

We can regard the definition of an agent view to be a class template which requires possible alternative series of services from the system. However, although an agent view defines behaviour and encapsulates a state, each instance is not really an object as it does not have an object identifier. The construction of an agent view does help us identify the objects which will provide the services required by the agent. Hence, a simple object model is constructed in conjunction with the agent views.

In the early stages of analysis, it is important that we do not get bogged down in detail. This is especially important when an executable formal model is being produced as it will necessarily contain more detail than an informal model. That is why we formalise agent views rather than use cases; formalising complete use cases would introduce too much detail because they deal with internal system interactions as well as with external behaviour and it is the external behaviour with which we are concerned at this stage. Also, use cases do not provide a clear distinction between the user-centred and system-centred models.

Our formal user-centred model therefore consists of a set of consistent and validated agent views which describe all aspects of how the system is to be used. Each agent view interacts with the objects which provide the *external* behaviour of the system as seen from that agent's point of view. A major part of the construction of the model is the resolution of inconsistencies and contradictions in the views and expectations of the different agents.

In general, an agent's interaction with the system will involve other agents. One agent may initiate a scenario, e.g. make a phone call, but that also involves a second (dependent) agent, namely the person being called. Initially, we concentrate on *single-agent views.* In the phone call example we would have two single-agent views, one for the caller and one for the person being called. Later, the two single-agent views can be composed to form a *multi-agent view* which describes the set of possible interaction sequences involving both the calling and receiving agents and the telephone system.

The main advantage of dealing with single-agent views is their simplicity. They give us a place to start when handling large complex requirements. They provide a separation of concerns; at any one time we focus on a single agent's view of the services the system is to provide. They are therefore ideal in giving feedback to potential users who can check that their requirements have been interpreted correctly. For similar reasons, in their modification of OOSE, Regnell *et al.* only deal with single-agent use cases.

If agent views are to give feedback to potential users, it is important that they can be executed. The formal language used in their representation must therefore be executable. As we can regard the definition of an agent view to be a class template, it can be instantiated to give an object which can be executed to show different possible sequences of interactions between the environment and the system.

2.6 Creating a user-centred model

2.6.1 Identifying agents and their views

From the requirements, and in consultation with the users, we identify:

- the agents which make up the environment,
- the entities (interface objects) through which the environment and the system interact.

This information is used to create an initial object model. (Typically, more entities and agents will be identified later during the analysis process and the object model will be refined.)

The next step is to identify the ways in which the agents can interact with the system. These are the agent views. For each agent view, we:

- construct an agent tree to give a graphical description of the possible event sequences which make up each agent view by:
  - identifying the events which make up the interaction between the environment and the system,
  - allocating the events to the appropriate interface objects,
  - identifying the alternative event sequences which make up the agent view,
  - adding the ordered events to the tree as nodes with each event being represented by the pair: ⟨class template⟩.⟨event⟩,
- refine the object model by:
  - adding any new interface objects to the object model,
  - determining which events correspond to services offered by interface objects to the environment,
  - adding these offered services to the object model.

Several agent views will often use the services offered by the same object. Creation of the object model helps ensure that the different agent views use identical names for objects and services which are essentially the same.

In the specification of an automated banking system customers may interact with the system through ATMs or by going to the bank and dealing with human tellers. Although a customer may do either, they act as different agents in these two circumstances.

We must construct trees for these two agent views together with trees for all the other agent views of the banking system. A tree for the ATM agent view is given in Figure 2.1. Each path through the tree provides an agent scenario. In this example, the agent view is self standing and so there is no difference between a single and a multi-agent view.

The requirements will show that the ATM is composed of the following entities: card reader, keypad, screen, dispenser and printer. They correspond to the objects in the system interface with which the agent view may interact. The set of objects may change as the development of the specification proceeds.

Rather than duplicating large parts of the tree, a shorthand is used to represent iteration. The subtree to be duplicated is labelled as is the branch where the copy of the subtree is to be inserted. Care must be taken to ensure that the tree only describes terminating agent scenarios. In the ATM example, for instance, invalid passwords may be given several times. However, all agent scenarios terminate as the requirements will specify a limit on the number of incorrect attempts. When the limit is exceeded, the ATM will retain the card.

2.6.2 Formalising agent views

As the semantics of a LOTOS process definition can be represented as a tree, it is straightforward to construct the outline of a LOTOS process definition from the tree representing an agent view:

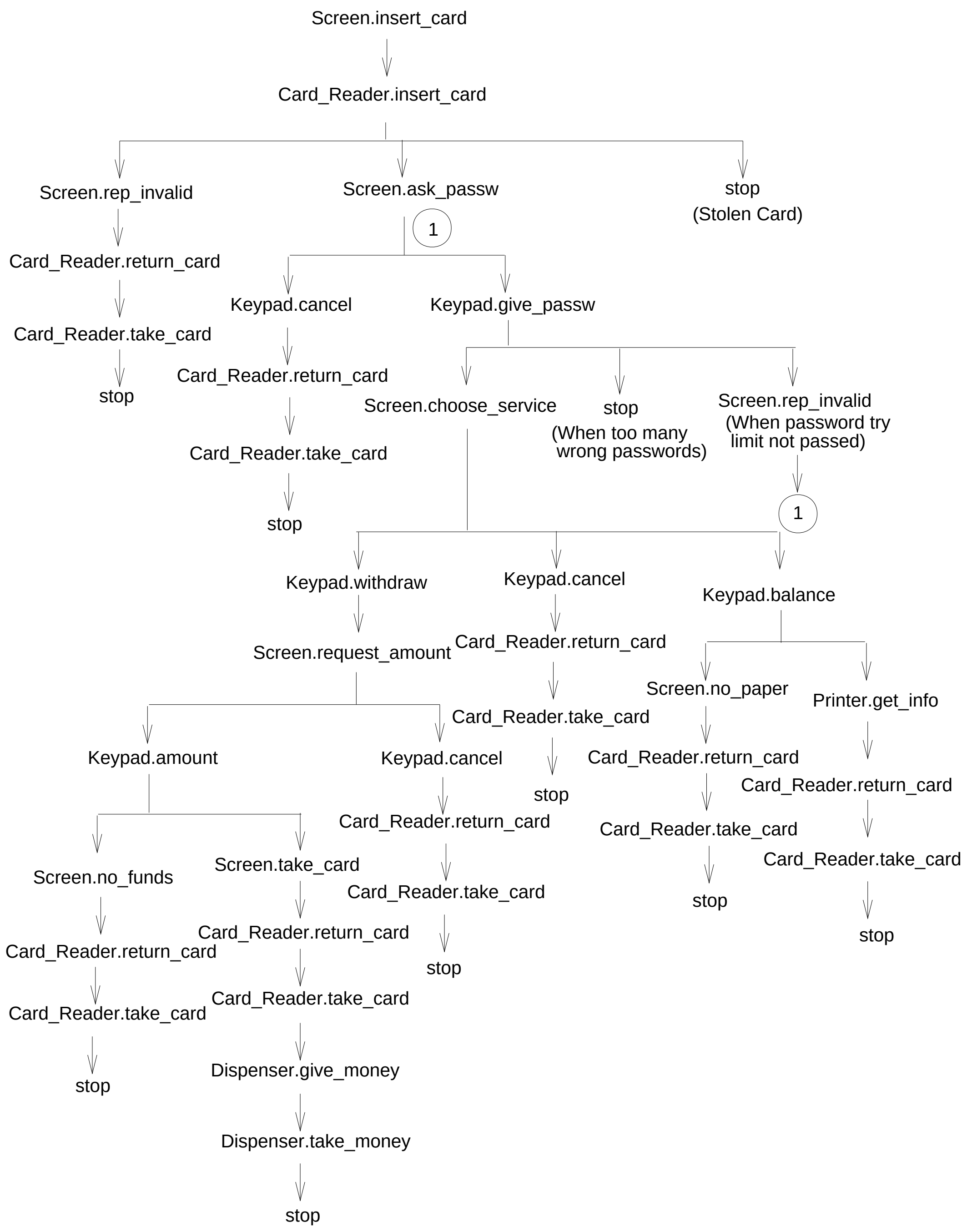

Figure 2.1: ATM Tree for an Agent View

- At each node of the tree, the first event of each branch will be offered as an option in a LOTOS choice expression.
- If the same subtree appears more than once, it is defined as a separate process and an instantiation of that process is inserted instead of the LOTOS code for the subtree.

The LOTOS process definition does however contain a significant amount of additional material, namely the information which is to be passed when the agent and the system interact. We must return to the informal requirements to determine what information is required at each stage and its source. This frequently shows up omissions, ambiguities and general vagueness in the requirements which must be rectified.

In LOTOS, processes interact with one another through gates. We can regard the LOTOS process definition of an agent view to be a class template with an object being represented by the instantiation of the LOTOS process. As class templates offer their services at different gates, the LOTOS definition of an agent view requires a separate gate for each class of interface object with which it interacts. We do not give an introduction to the LOTOS language in this report. A tutorial introduction is available in [3]

The LOTOS process definition for the ATM agent view given in Figure 2.1 is given below. It uses the gates: `cdr`, `kpd`, `scrn`, `disp` and `pr` to interact with the five interface objects given above. Each event has the following structure:

<gate> <message name> <object identifier> <optional parameters>

The complete LOTOS specification is given in Appendix A and includes the definitions of the sorts `Atm` and `Customer`.

```
process Atm_Agent[cdr, kpd, scrn, disp, pr](a: Atm, c: Customer) : exit :=
   cdr !insert_card !Id(a) !Id(c) !Card_No(c);
   (   scrn !rep_invalid !Id(a) !Id(c);
       Return_Card[cdr](a, c)
    []
       (* Stolen card, transaction stopped, card not returned *)
       exit
    []
       scrn !ask_passw !Id(a) !Id(c);
       Check_Passw[cdr, kpd, scrn, disp, pr](a, c, 0)
   )
endproc (* Atm_Agent *)

process Return_Card[cdr](a: Atm, c: Customer) : exit :=
  cdr !return_card !Id(a) !Id(c);
  cdr !take_card !Id(a) !Id(c);
  exit
endproc (* Return_Card *)

process Check_Passw[cdr, kpd, scrn, disp, pr](a: Atm, c: Customer, num: Nat)
                   : exit :=
    kpd !give_passw !Id(a) !Id(c) !Password(c);
```

```
      (   [num lt succ(succ(0))] ->
              scrn !rep_invalid !Id(a) !Id(c);
              Check_Passw[cdr, kpd, scrn, disp, pr](a, c, succ(num))
       []
          [num eq succ(succ(0))] ->
              (* Too many wrong passwords *)
              exit
       []
          Atm_Trans[cdr, kpd, scrn, disp, pr](a, c)
      )
    []
      Cancel[cdr, kpd](a, c)
endproc (* Check_Passw *)

process Atm_Trans[cdr, kpd, scrn, disp, pr](a: Atm, c: Customer) : exit :=
  scrn !choose_service !Id(a) !Id(c);
  (    kpd !withdraw !Id(a) !Id(c);
       scrn !request_amount !Id(a) !Id(c);
       kpd !amount !Id(a) !Id(c) ?am: Money;
       (   scrn !no_funds !Id(a) !Id(c);
           Return_Card[cdr](a, c)
        []
           scrn !Take_Card !Id(a) !Id(c);
           Return_Card[cdr](a, c)
           >> ( disp !give_money !Id(a) !Id(c);
                disp !take_money !Id(a) !Id(c);
                exit
              )
       )
    []
       kpd !balance !Id(a) !Id(c);
       (   scrn !no_paper !Id(a) !Id(c);
           Return_Card[cdr](a, c)
        []
           pr !get_info ?am: Money;
           Return_Card[cdr](a, c)
       )
    []
      Cancel[cdr, kpd](a, c)
  )
endproc (* Atm_Trans *)

process Cancel[cdr, kpd](a: Atm, c: Customer) : exit :=
  kpd !cancel !Id(a) !Id(c);
  Return_Card[cdr](a, c)
endproc (* Cancel *)
```

There is repetition in the agent tree as some events are possible at different times; for example, cancellation of the transaction by the agent. We factor such events to make the specification simpler. The LOTOS specification also gives more information as it shows the information to be passed during each event.

It will be noted that many of the tree nodes have alternative child nodes. In some of these cases, it is the agent who decides what is to be done next; in others it is the system which will decide. We do not need to distinguish between the two situations at this stage.

2.6.3 Combining agent views

Although, in this case, the individual agent view is self standing, in general dealing with individual agent views is not sufficient to specify external system behaviour fully; e.g. a person cannot respond to a phone call before the caller has made the call.

Agent views must therefore be combined. If two agent views `View_1` and `View_2` are completely independent of one another, then they can be composed using the interleaving operator:

```
View_1[s] ||| View_2[t]
```

However, when two views are not independent, this composition does not work as it does not constrain the relative ordering of the events in `View_1` and `View_2`.

In the specification of OSI systems [25], a service specification is produced using the *constraint oriented style* in which behaviour is expressed as sets of constraints operating at external interfaces. There are *local constraints* which define the order of the services provided to (or required by) a single agent and *end-to-end constraints* which define the interaction between agents.

A similar approach can be used to construct multi-agent views from single-agent views. Given that `View_1` and `View_2` define two single-agent views which are dependant on one another, their combined effect can be specified by the LOTOS behaviour expression:

```
(View_1[s] ||| View_2[t])
|[s, t]|
Constraint[s,t]
```

where process `Constraint` defines the interaction between the single-agent views.

An example of such a constraint is given in Section 3.3.3.

2.6.4 Verifying, instantiating and validating agent views

The LOTOS process definition of each single-agent view can be instantiated to give a single-agent view object. Such an object offers a set of alternative execution sequences. Execution of a LOTOS behaviour expression using a simulator such as SMILE [7] creates a tree showing all possible event traces. As this should have the same structure as the original agent tree, we have a simple check to verify that an agent tree has been correctly specified in LOTOS.

The LOTOS SMILE simulator allows the use of uninstantiated variables instead of values. As SMILE is able to determine when a combination of conditions can never be true, a single instantiation of an agent view can deal with a set of similar agent instances rather than be instantiated for a particular agent instance.

The LOTOS process definition for an agent view can be instantiated from a process with the structure:

```
process X_Tree[gates, success](parameters) : noexit :=
   X_Agent[gates](parameters) >> success; stop
endproc
```

The occurrence of the `success` event during simulation indicates that the an agent view has been successfully executed.

Initially, we execute single-agent view objects independently; this demonstrates the possible behaviours that an agent expects from the system. When an agent's interaction with the system involves other agents, single-agent views can be composed as described in Section 2.6.3. Their instantiation results in a multi-agent view object which can be executed to ensure that the complete set of possible events has been properly described.

The process of composing agent views can show up inconsistencies. When this happens, we return to the agent view identification step, make the appropriate changes in one or more single-agent views and repeat the subsequent steps.

It will often be the case that two or more agent views will describe what are best described as variants of one another. When two such agent views require the same initial sequence of events from the system, they must be combined to form a single agent view.

2.6.5 The user-centred model

The behaviour that the environment expects from the system is the composition of all the agent views. Hence, the formal user-centred model is the composition, using the interleaving operator, of the LOTOS specifications of all the agent views. This behaviour expression is then composed in parallel with the constraint processes which define their interactions.

If the resulting user-centred model is prototyped, the number of alternative events offered is too large to be handled conveniently. The user-centred model must therefore be composed with a `Test` process with the following structure [19]:

```
process Test[gates, success](parameters) : noexit :=
  Scenario[gates](parameters) >> success; stop
endproc
```

Each scenario is derived directly from a single or multi-agent view and defines a behaviour which the user-centred model must satisfy.

Validation of the user-centred model is by executing a series of such test compositions to demonstrate that the expected behaviour is supported, i.e. that the `success` event is reached. Initially, we prototype interactively to explore particular execution paths. However, that will not usually allow all possible paths to be explored. Hence, once we have shown that the composition is successful for at least some execution paths, we replace interactive prototyping with the use of LOLA [19] which can automatically perform a complete state exploration of the composition of a specification and a test process. LOLA reports one of three possible results:

MUST where all possible execution paths lead to the `success` event.

MAY where at least one execution path leads to the `success` event.

REJECT where no execution path leads to the `success` event.

This enables us to verify whether or not a particular test has been satisfied.

2.7 Creating and verifying the system-centred model

2.7.1 Model generation

We now construct a formal system-centred model which satisfies the agent views. The formal user-centred model is a major help in the development of the system-centred model. For example, as the interactions between an agent and the system are often not through a single interface object, the agent view specifications help partition behaviour among the different interface objects.

Each event in an agent view can be the starting event for (part of) an event trace diagram. Many of the required objects will already have been identified and be represented in the initial object model. Constructing event trace diagrams identifies the message passing between these objects and often shows the need for new operations and may show the need for new objects.

When several agent views use the services offered by one interface object, we must check that they do not make incompatible demands. This gives us a further means of discovering inconsistencies and contradictions in the requirements. Their resolution will often require us to go back to the users for further or revised information.

It is possible that two agent views will have used different names for objects or services which are essentially the same. The definition of the system-centred model will help identify any such occurrences so that they can be rectified.

2.7.2 Validation

The system-centred model must support the behaviour specified by the user-centred model. We must verify that all the required services are provided.

As the system centred model is expressed in the same formal language as the user-centred model, the two specifications can be composed and the composition executed using the same test cases that were used for the validation of the user-centred model. Interactively prototyping the composed system allows us to explore interesting execution paths to increase our confidence that the system-centred model offers the expected behaviour. This is very important during the creation of the system-centred model. Interactive prototyping will normally be too time consuming to demonstrate that all execution paths of the composed system terminate successfully; for that we use LOLA.

The user-centred and system-centred models are two alternative ways of formally specifying the same system. Our next step is therefore to demonstrate that the two specifications are equivalent.

2.7.3 Verification

There are many different definitions of what it means for two specifications to be equivalent, but the only one which can be applied in practice to large specifications is *testing equivalence*.

In testing equivalence, the specifications under test are regarded as black boxes, i.e. we only consider external behaviour, internal structure is ignored. The two specifications are composed with test processes in the same way as was described in the validation of the user-centred model in Section 2.6.5. However, we are not now concerned with the informal process of showing that the formal user-centred model satisfies the informal requirements, but with the formal process of demonstrating that the two formal models cannot be distinguished by experiment.

Two specifications A and B are testing equivalent if every MUST or MAY test on A gives the same result with B and vice versa. Using LOLA allows a complete state exploration of the composition of each of the user-centred and system-centred models with each test process and reports each of the results as MUST, MAY or REJECT. The drawback is, of course, that the set of test processes is not complete. However, by judicious choice of a sufficiently large number of tests we can have reasonable confidence that the specifications are indeed testing equivalent [19].

2.8 Integration of the formal user-centred model into ROOA

2.8.1 The ROOA Method

The aim of the ROOA method is to construct a formal object-oriented model of the observable behaviour of a system. It involves three tasks. In Tasks 1 and 2, it uses the techniques of informal object-oriented analysis methods such as [6, 11, 21, 23] to build an object model and part of the dynamic model from a set of informal requirements. In Task 3, we build the formal system-centred model (the ROOA model). The construction of a system-centred model is shown in Chapter 3.

The ROOA model is expressed in LOTOS and acts as a formal requirements specification. Prototyping is used to validate the specification against the requirements. Informal requirements normally contain inconsistencies, contradictions, ambiguities and omissions. Some of these errors will be detected and resolved during the process of constructing the formal specification while others will be found when prototyping the specification. The ROOA method enables these errors to be detected sufficiently early in the development so that feedback can be given to the requirements capture process.

When a problem is process-oriented, only a very simple object model showing the most obvious objects and associations can be constructed without considering dynamic behaviour [17]. This simple object model corresponds to what Jacobson calls the *domain object model* [11]. In such cases, scenarios and event trace diagrams already play an important part in the ROOA method and help to identify the objects and their interactions and to identify dynamic behaviour. However, the introduction of a formal user-centred model is a major change of emphasis. Once the user-centred model has been created, it is used directly in the construction of the ROOA specification.

In the validation phase of the ROOA method, scenarios are encoded in LOTOS, and used as test cases to drive the ROOA model to demonstrate that the specification has the expected external behaviour. Here again, the introduction of a formal user-centred model gives a major change of emphasis. By the time we reach the validation phase, we already have two formal models. The two models are composed and their composition executed. Interesting execution paths are explored interactively to increase our confidence in the behaviour of the system-centred model. Prototyping with agent views rather than with agent scenarios gives a much wider coverage of the required behaviour.

We then use a tool such as LOLA which supports the automatic exploration of the complete state space of the composition of a specification and a test process to verify that the user-centred and ROOA models are testing equivalent.

2.8.2 The Integration

The system-centred model is built following the ROOA method, but takes advantage of the user-centred model and of the initial object model which has already been created.

The ROOA method involves three main tasks:

1. build an object model;
2. refine the object model;
3. build and validate the LOTOS ROOA model.

Task 1

Task 1 includes the construction of an initial object model which is then enhanced by applying the ideas of object-oriented analysis methods, such as OOSE [11] or OMT [23]. The user-centred model is a major help in identifying objects.

Task 2

To refine the object model we use the user-centred model to identify dynamic behaviour to help us:

- add more objects to the object model;
- identify services offered by each object;
- identify message connections between objects.

We start with an event in an agent view. This identifies the interface object which is to synchronise with this event. We then determine the internal system objects and the object interactions (message connections) which are required to satisfy the requested behaviour and record the information in an event trace diagram (ETD). Each agent view may identify the need for new class templates or it may suggest additional services to be added to the existing class templates in the object model.

The requested behaviour will often require the involvement of other agents. This process therefore enables multi-agent views to be constructed. The composition of single-agent views to form a multi-agent view will show up any contradictions in the different agent views.

Task 3

The first step is to construct an *Object Communication Diagram* (OCD), i.e. a graphical representation of the ROOA model. The development of the ROOA model starts with the specification of the model's interface objects which are identified by examining the user-centred model. They must offer the services demanded by the agent views.

Although the ETDs show the possible sequences of object interactions (event synchronisations) required to respond to events in the user-centred model, they do not give any information about the data to be passed. It is the formal user-centred model which details the data passed to the system, and expected from the system, when an agent view and an interface object synchronise on an event. This information is used directly in the construction of the definitions of the interface objects in the system-centred model: each external event in an interface object synchronises with an event in one or more agent views and so the number and sorts of its parameters is fully determined by the number and sorts of the parameters in the corresponding event in the agent view.

The synchronisation between an interface object and an agent view is typically followed by one or more object interactions between the interface object and other objects in the system-centred model. These object interactions may then be followed by an event where the interface object reports back to the agent view. The purpose of each object interaction is to carry out (part of) the wishes of the agent view. The decision about the number and sorts of the parameters required in the object interactions is therefore driven by the data exchanged in the interactions between the agent view and the interface object.

Once the event structure of the object interactions in an interface object has been determined, the same approach can be used to determine the structure of the events in the objects with which it synchronises with the interface object playing the role of an agent view. In this way the number and sorts of the parameters in all the object interactions can be determined. As we are using a formal language, we can ensure that synchronising events properly match with one another with respect to the number and sorts of their parameters.

The single ROOA model must cater for all the different agent views. Each agent view describes the expected behaviour of the system from the point of view of a particular agent. When we come to specify the system-centred model which is to satisfy this behaviour, it is not unusual for incompatibilities to appear in the expectations of different agents. The resolution of these contradictions will often require us to go back to the users for further or revised information.

The user-centred model plays an important role when we validate the ROOA specification. As the user-centred model has already been validated with respect to the informal requirements, validation of the ROOA specification is now concerned with the formal task of verifying that it is equivalent to the user-centred model.

The ROOA method is iterative. In later iterations, the formal user-centred model, the informal object model and the formal ROOA model are developed in conjunction. This is especially important in tracing the consequences of a change in the requirements.

2.8.3 Summary

Having the ROOA and user-centred models represented in the same formal language has significant advantages:

- We can integrate the information (services and objects) found during the construction of the user-centred model into the system-centred model.

- We can compose both models to verify that the system-centred model offers all the services required by the user-centred model and execute the resulting specification to validate the system-centred model. This may require changes in both models and in the requirements.

- We can trace how a change in one model affects the other.

- As the ROOA model must be able to provide the behaviour expected by all the agent views, we can identify inconsistencies and contradictions in the views making up the user-centred model. Resolving such inconsistencies plays a major part in clarifying the requirements.

- We can validate the user-centred model against the informal requirements and then verify that the user-centred and ROOA models are equivalent.

2.9 Software development

Object-oriented development is best suited to software life-cycle models in which incremental and iterative development is an explicit and important component. Two examples are the Spiral Model [2] and the Fountain Model [9].

We are concerned with the activity of object-oriented analysis and believe that our approach can fit within these development methods; it can be continually revisited as a system is incrementally developed. Also, as we produce a formal requirements specification, our method can form the first stage of a formal development trajectory where the design and eventual implementation are automatically developed from, and can be verified against, the specification.

We propose the use of an executable specification language. Our approach therefore not only consists of constructing formal user-centred and system-centred models, but also prototyping these models both to give feedback to requirements capture and to validate the models. It therefore contains, within the analysis phase, activities which normally take place in later phases.

Requirements are often vague, inconsistent or incomplete and be open to misinterpretation. Capturing and then specifying the requirements as agent views forces their unambiguous description. This enables us to identify problems in the requirements very early in the development and to formulate questions to users so that we can have clarification. Execution of the agent views shows up the consequences of the clients' demands and these can be passed back to the clients to ensure that they are indeed intended.

This is especially useful when the requirements have not been fully developed and exploration is needed to help clarify what is required. As prototyping can be done quickly, it allows us to try out alternative ideas and report results back to the clients for comment. This is a great aid in helping them clarify their ideas; seeing the consequences of their initial requirements can help a client refine the existing, or formulate further, requirements.

The creation and execution of the formal system-centred model enables further checks to be made on the suitability of the clients' requirements. It is at this stage that we might find that two agent views lead to incompatible demands on the system which cannot be satisfied. This is especially useful during exploration as it indicates that the requirements must be rethought.

We do not propose that a complete user-centred model is created before we proceed to the creation of the system-centred model. Instead, it is best if an incremental approach is adopted. With incomplete requirements, some parts will be well understood while others will not. Those parts of the requirements which are best understood can be modelled first. This subset of the user-centred model can be executed to ensure that it behaves according to the clients' expectations. This will give both us and the clients a better understanding of the system. The corresponding part of the system-centred model can then be created and, when the two models are written in the same executable specification language, the models can be composed and the composition executed. Prototyping of both the user-centred and the composed models gives feedback to the requirements capture process enabling further agent views to be formulated and existing ones refined. In this way both the user and system-centred models can be developed incrementally.

2.10 Conclusion

This chapter presents a process we can follow in building a formal user-centred model. A user-centred model consists of a set of agent views which represent the external behaviour that agents expect from the proposed system. The advantage in defining a user-centred model is that we can can easily identify the behaviour that the system is expected to provide. Also, a user-centred model can help in the construction of the formal system-centred model and in the clarification of the informal requirements.

When the system-centred model is built using the ROOA method, we have the advantage that processes in both are specified using LOTOS. Therefore, we can use LOTOS tools to prototype the user-centred model to validate it with respect to the requirements and then verify that the system-centred ROOA model is equivalent to the user-centred model.

Chapter 3

The road pricing system: a case study

3.1 Introduction

In this chapter we use a case study to show how a user-centred model can be built by following the process presented in Chapter 2. We then show how this model can be used to help create and validate a system-centred model.

The user-centred model is specified in LOTOS. We use LOTOS tools to prototype the external behaviour of the system and to identify ambiguities, omissions and contradictions in the original set of requirements.

As the system-centred model is built using the ROOA method, it is also specified using LOTOS. The obvious advantage is that we can integrate the two models and use the user-centred model to verify and validate the system-centred model. Another important advantage is that we can easily integrate the information (services and objects) found during the construction of the user-centred model into the system-centred model.

3.2 The road pricing system requirements

As part of a road traffic pricing system, we require an automated system which will enable drivers of authorized vehicles to be charged at toll gates without having to stop. Drivers must install a device (a gizmo) in their vehicle. (Each gizmo can only be used within a single vehicle. Therefore, if a person has two cars, he or she has to have two gizmos.) Authorized vehicles must have a registration which includes vehicle details, the owner's personal data and an account number from where automatic debits are done monthly.

Each gizmo has an identification number which is read by the sensors installed in special lanes at the toll gates. Let us call those lanes *green lanes.* With the information given by a sensor, the system can then store the necessary information so that the specified bank account can be debited.

When an authorized vehicle passes through a green lane, a green light is turned on and the amount being debited is shown in a big display. If an unauthorized vehicle passes through a green lane, a yellow light turns on and a camera photographs the vehicle registration number. (Later, the owner of the vehicle will be fined.)

Different types of vehicles pay different rates.

There are two kinds of green lane: one where all vehicles of the same type pay the same fixed amount (e.g. at a toll bridge) and one where the amount to be paid depends on the distance travelled. For example, a car which uses 50 miles of a motorway pays less then a car which uses 100 miles of the same motorway. For this, the system has to store the entrance point and the exit point, for each vehicle.

We want to model the part of the system which receives the information from the gizmos, determines whether the vehicle is authorized, displays the amount to be debited, turns on the appropriate light, triggers a photograph when necessary, and deals with the automatic account debits.

3.3 Creating a user-centred model

To create a user-centred model we must accomplish four main tasks:

1. identify agents and their views;
2. formalise agent views;
3. combine agent views;
4. instantiate and validate agent views.

3.3.1 Identifying agents and their views

By the end of this task we will have identified the agents which interact with the system, the views each one has of the system, and, as a result, the interface objects which allow those interactions.

Identifying agents, interface objects and agent views

The first question to ask ourselves is: what interacts with the system? The first agent we may think of is the gizmo installed in a vehicle. A vehicle uses a toll gate and has a driver. We regard the composite vehicle-driver-gizmo to be a single agent which we refer to as *vehicle*. Another agent is the owner of the vehicle. He or she has to buy a gizmo and be registered before being able to use the green lanes. A third agent is the bank. This agent behaves differently from the other two. While the first two require services from the system, the bank offers services to the system.

The second question to answer is: how does each agent view the system? In order to answer this, we imagine ourselves to be the agent and describe what we see when we use the system. As part of this process, we identify the interface objects with which we, the agents, interact. This has the advantage of helping in the creation of the object model of the system-centred model.

The first obvious interface object is a toll gate. The vehicle agent (i.e. the driver) sees and uses a toll gate, but it also knows about its components. For example, it sees a light turning green or yellow, sees the amount shown in the display, knows that there must be a sensor which identifies its gizmo, and also knows that if the light turns yellow a photograph of its plate number will be taken. Therefore, a toll gate is an object which has four components. The vehicle agent also knows that there are different types of toll gate (e.g. entry and exit gates for a motorway and a single gate for a toll bridge) and that the display is only used when it is either leaving the motorway or when it passes through a single toll gate.

All this information should be added to the object model and represented by using object-oriented concepts. An object model using OMT notation is shown in Figure 3.1.

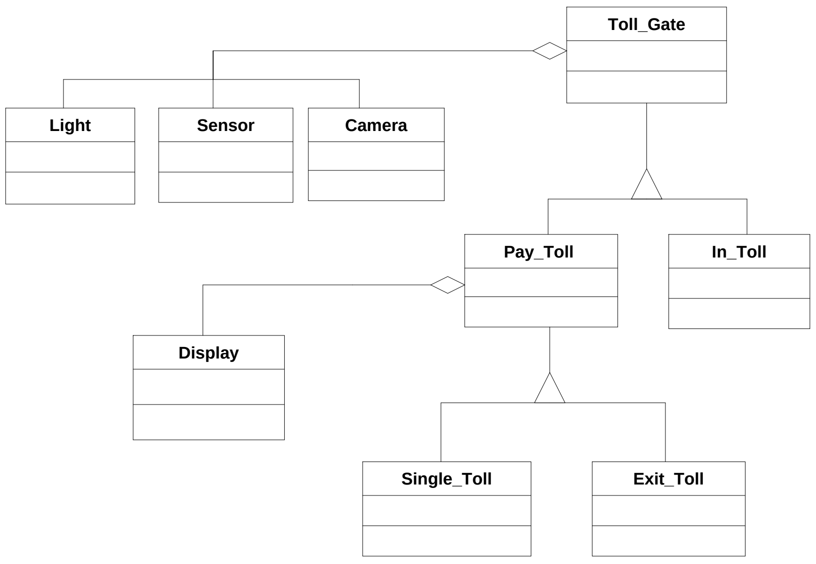

Figure 3.1: Initial object model

Now, we start identifying the views of the vehicle agent. This agent knows that it can use the system in two different ways: one where it always pays the same amount, let us call it the one-point vehicle view, and another where the amount paid depends on the distance travelled, let us call it the two-point vehicle view.

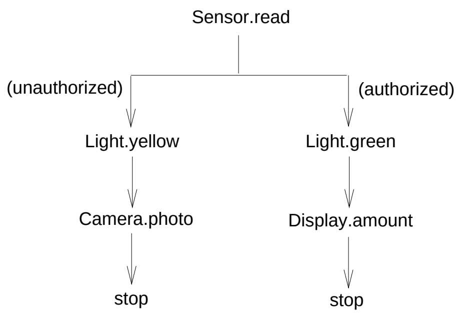

Figure 3.2: One-point vehicle view

Constructing a tree for each agent view

An agent view is graphically represented as a tree. In order to build this tree we:

- identify the events which make up the interaction between the environment and the system,
- allocate the events to the appropriate interface objects,

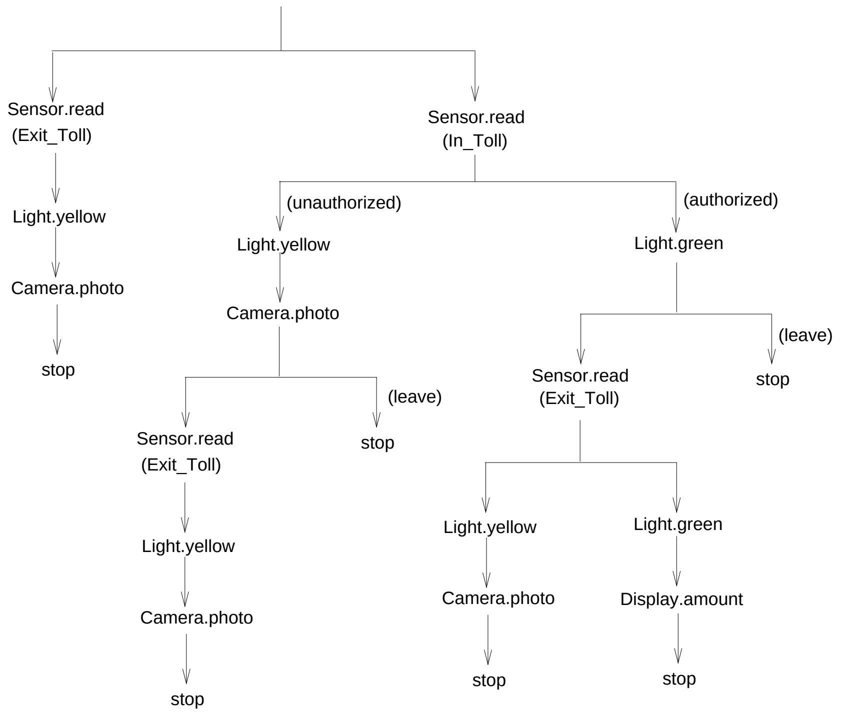

Figure 3.3: Two-point vehicle view

- identify the possible alternative event sequences which make up the agent view,
- add the ordered events to the tree as nodes with each event being represented by the pair: ⟨class template⟩.⟨event⟩.

The one-point vehicle view is depicted in Figure 3.2, When the vehicle passes the toll gate, the sensor reads the vehicle's gizmo identifier. The system will then either show a green light and display the amount to be debited or show a yellow light and a camera will take a photo of the vehicle's plate number. The yellow light is shown to vehicles whose gizmo is not accepted by the system because, for example, it has been reported stolen or its registration has been cancelled.

The two-point vehicle view is shown in Figure 3.3. When the vehicle enters a motorway, first it passes an In_Toll gate. Here, a sensor reads its gizmo identifier and the system either responds by showing a green light or by showing a yellow light followed by triggering a camera. After this, the vehicle proceeds onto the motorway and it may just leave it by some unknown method (it could have an accident, for example, and so never reach the exit toll, or it can leave by using a non-green lane) or it passes through an Exit_Toll gate. If a yellow light in the In_Toll was shown, now the vehicle can only get another yellow. On the other hand, if a green light was shown, the vehicle may get a yellow light if the state of the system has changed (for example, the system now knows that that vehicle was stolen) or a green light again, meaning that everything is normal.

The agent view also describes the situation where a vehicle passes through an Exit_Toll gate, but did not enter the motorway through an In_Toll gate. This leads to a yellow light being shown and a photograph taken. It should be noted that this situation was omitted from the original informal statement of requirements.

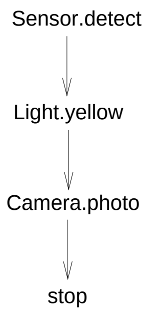

Figure 3.4: One-point vehicle view with no gizmo

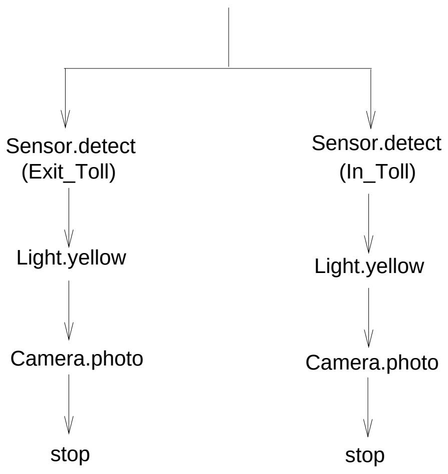

Figure 3.5: Two-point vehicle view with no gizmo

In our vehicle agent views, we have assumed that the vehicle has a gizmo which is read by a sensor. We must also consider the agent view where a vehicle goes through a green lane either without a gizmo or with a gizmo which is broken and cannot be detected. In such a situation, a sensor must be able to detect that a vehicle has passed, so that it can show a yellow light and cause a photograph to be taken. This situation leads to two more agent views. They are shown in Figures 3.4 and 3.5. When a vehicle has no detectable gizmo, its action at an In_Toll and at an Exit_Toll gate are not linked and so each are considered separately in Figure 3.5.

The benefit of constructing a formal user centred model and structuring the requirements as a set of agent views is that it can identify omissions in the requirements. The requirements state that a light can turn green or yellow, but say nothing about whether the light stays on or whether it is cleared before the next vehicle is processed. The requirements are amended so that the light (and the display) are to be cleared before the next vehicle is processed.

We now consider the views of the owner and bank agents. They help us to identify two more interface objects: Reg_Desk, to deal with the registration of a vehicle by its owner, and Bank_Interface, to deal with the bank agent.

The owner agent only has one view of the system which is shown in Figure 3.6. It gives information which includes vehicle details, the vehicle owner's personal data and an account number from where the monthly debits will be done. The registration may or may not be

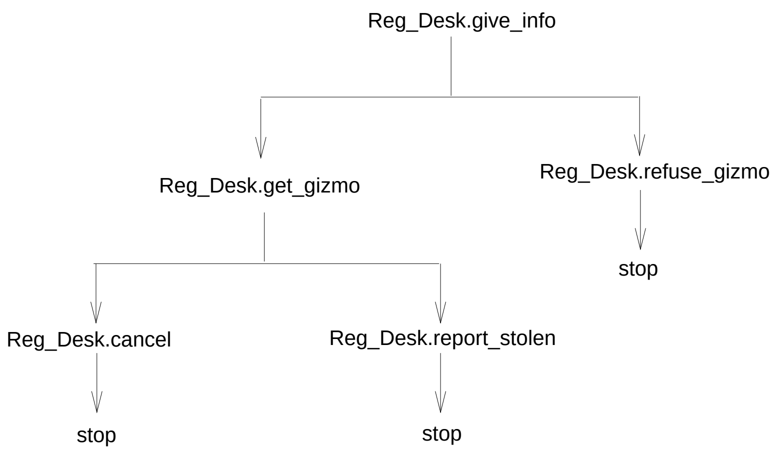

Figure 3.6: Owner agent view

accepted.

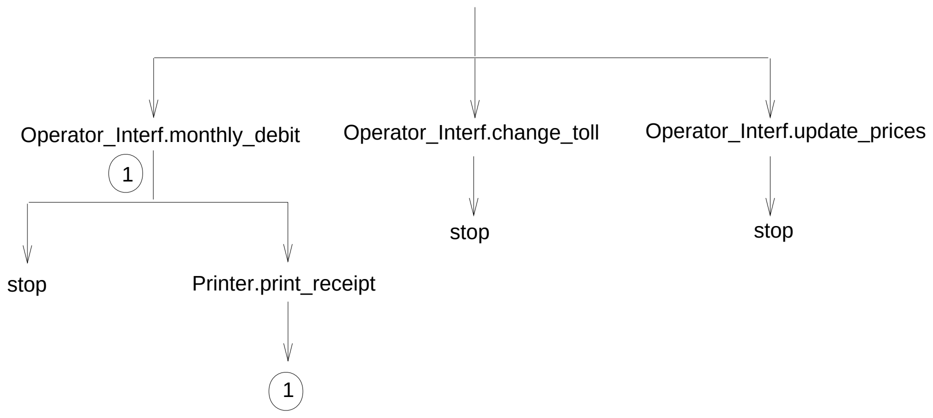
Operator_Interf.monthly_debit
Operator_Interf.change_toll
Operator_Interf.update_prices
1
stop
Printer.print_receipt
stop
stop
1

Figure 3.7: Operator agent view

An agent which almost always appears in a system is the operator. This is shown in Figure 3.7. The system offers three services to the operator: `monthly debit` so that the operator can initiate the monthly debits of gizmo owner's accounts, `change toll` so that toll gates can be added to, or removed from, the system and `update prices` so that the toll prices can be modified. Although these operations are not mentioned explicitly in the requirements, they are identified as being necessary for the initiation and correct working of the system. A consequence of the operation initiating monthly debits is that a series of receipts will be produced. As these receipts must be handled by an operator, we can regard this as part of the operator agent view.

The bank agent has two different views of the system: the view when it is asked for information about an account number and the view when it is asked for automatic debits. These two views are shown in Figures 3.8 and 3.9.

When an owner agent attempts to register with the system, the agent view in Figure 3.8 is activated by the system to check with the bank whether or not that account number exists. Therefore, the two single agent views, owner and bank debit can be combined later to give a

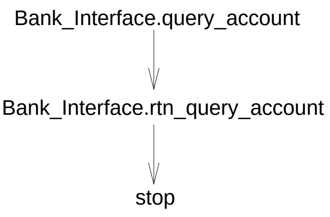
Bank_Interface.query_account
Bank_Interface.rtn_query_account
stop

Figure 3.8: Bank view for account information

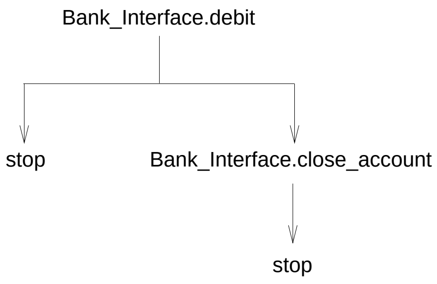

Figure 3.9: Bank view for automatic debits

multi-agent view.

The bank agent view dealing with the debits is very simple. It either just receives the event and finishes or it responds by informing the system that the account has been closed and the debit cannot therefore take place.

Refining the object model

Based on the information in the agent views, we refine the initial object model by:

- adding any new interface objects to the object model,
- determining which events correspond to services offered by interface objects to the environment,
- adding these offered services to the object model.

At this point we have no more interface objects to be added to the object model. We have to analyse each agent view and decide whether or not a given event corresponds to a call of a service offered by an interface object or if it is merely a response. Responses do not appear in an object model. The best way to do this is, once again, to stand outside the system, and follow the tree identifying which messages are reaching the system and which are leaving it.

For example, `Reg_Desk.give_info` in Figure 3.6 is a service offered to the environment. When an owner agent calls this service, it is sent the message `Reg_Desk.refuse_gizmo` or `Reg_Desk.get_gizmo` as a response.

In some situations, it may appear that an interface object sends a message of its own volition, e.g. `Light` can send the message `Light.yellow`. In fact, `Light` will offer a service such as `turn_yellow` to internal objects in the system and it is a call of that service which will trigger `Light.yellow` being sent to the environment. However, we do not, at this stage, show services such as `turn_yellow` in the object model as we are only concerned with services offered to the environment, not with services offered to internal system objects; they will be added later during the construction of the system-centred model.

The result of analysing the events in each agent view to determine which are services offered by interface objects is shown in Figure 3.10.

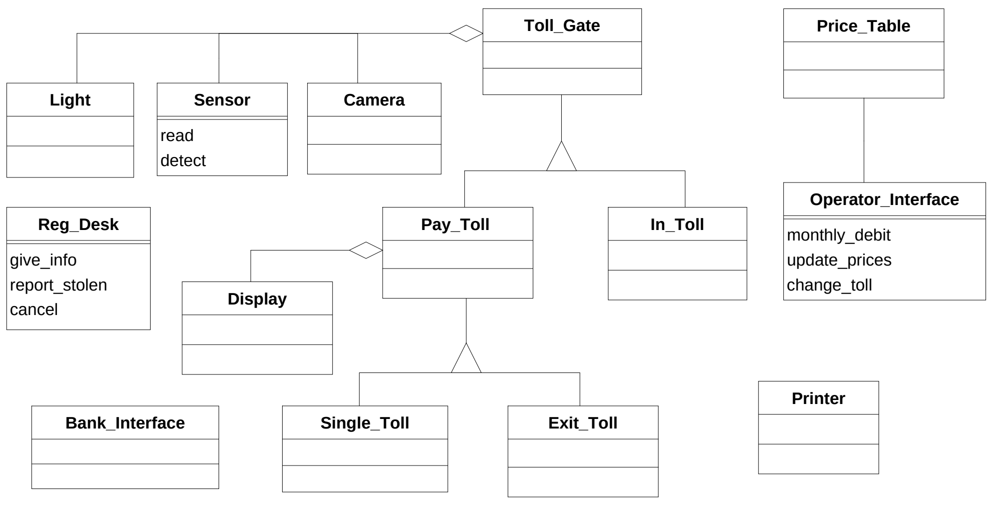

Figure 3.10: Initial object model with externally offered services

3.3.2 Formalising agent views

Each agent view is formalised using LOTOS. The definition of an agent view is a LOTOS process which requires possible alternative series of services from the system. It is straightforward to construct the outline of a LOTOS process definition from a tree by using the choice operator to offer each branch of the tree as a LOTOS choice expression. We can also factorize out those parts of the tree which are similar and define them in a separate process.

The LOTOS formal model contains much more information than was held in the tree. We have to return to the requirements or, when necessary, the clients to determine the information which is to be passed between the agent and the system. The need for this extra detail is very useful in clarifying the requirements.

A LOTOS process definition for the `Two Point Vehicle` agent view corresponding to the tree given in Figure 3.3 is given below. It includes a number of LOTOS gates for communication (one for each toll gate component). The agent first interacts with the components of an `In_Toll` gate and then later with the components of an `Exit Toll` gate. The components are identified by the object identifier of their toll gate. Sort `Toll Id` is defined in an ADT. Toll gate identifiers are introduced as variables into the process, as in:

```
?int: Toll_Id
```

The variable `int` is then constrained to be the identifier of an `In Toll` gate by the predicate

`Is_Enter(int)`. An `Exit Toll` gate identifier `ext` is later introduced and constrained in a similar way.

The rest of the body of the process follows the directions given in Chapter 2. Mainly:

- for each node of the tree, we define a choice operator in which each option defines a branch;
- if the same subtree appears more than once, it is defined as a separate process and an instantiation of that process is inserted instead of the LOTOS code for the subtree.

Much of the behaviour at an exit gate is independent of whether a vehicle agent was shown a green or a yellow light at the entry gate. This behaviour can therefore be factored out. It is defined in process `Leaving`. Process `Missing` deals with the case where a gizmo identifier was not detected at the `In Toll`, but was detected at the `Exit Toll` gate.

```
process Two_Point_Vehicle[sen, lgt, dpl, cam] : exit :=
     (  sen !read ?int: Toll_Id ?id_g: Gizmo_Id [Is_Enter(int)];
        (  ( Yellow[lgt, cam](id_g, int)
              >> Leaving[sen, lgt, dpl, cam](id_g, was_yellow)
           )
         []
           lgt !green !int !id_g;
           lgt !clear !int;
           Leaving[sen, lgt, dpl, cam](id_g, was_green)
        )
       []
        Yellow_No_G[lgt, cam](int)
     )
   []
     Missing[sen, lgt, dpl, cam]
where
   process Leaving[sen, lgt, dpl, cam](id_g: Gizmo_Id, s: Entry_Val)
              : exit :=
         (  sen !read ?ext: Toll_Id !id_g [Is_Exit(ext)];
            ( [s eq was_green] -> Green[lgt,dpl](id_g, ext)
              []
                Yellow[lgt, cam](id_g, ext)
            )
          []
            Yellow_No_G[lgt, cam](ext)
         )
        []
         (* leave in a non-green lane *)
         exit
      )
   endproc (* Leaving *)

   process Missing[sen, lgt, dpl, cam]
```

```
                : exit :=
         sen !read ?ext: Toll_Id ?id_g: Gizmo_Id [Is_Exit(ext)];
         Yellow[lgt, cam](id_g, ext)
      endproc (* Leaving *)
   endproc (* Two_Point_Vehicle *)
```

The two processes given below are part of the Two Point Vehicle process. Process Yellow defines part of the external behaviour of the toll gate when the light turns yellow and a photograph is taken while Green defines the behaviour when the light turns green at the exit gate.

```
process Yellow[lgt, cam](id_g: Gizmo_Id, iden: Toll_Id) : exit :=
   lgt !yellow !iden !id_g;
   cam !photo !iden !id_g;
   lgt !clear !iden;
   exit
endproc (* Yellow *)

process Green[lgt, dpl](id_g: Gizmo_Id, iden: Toll_Id) : exit :=
   lgt !green !iden !id_g;
   dpl !amount !iden !some_am;
   dpl !clear !iden;
   lgt !clear !iden;
   exit
endproc (* Green *)
```

The other agent views are defined in a similar way. A full LOTOS specification can be found in Appendix B although it differs slightly from the above for reasons which are given later.

3.3.3 Combining agent views

Some agent views are self standing while others interact with one another and so can be combined to give a multi-agent view. In Chapter 2 we discussed how this can be accomplished.

For example, when an owner agent asks for a registration, the desk contacts a bank agent to check on the specified bank account before it accepts the registration. Therefore the owner agent view, depicted in Figure 3.6, and the bank view for account information, depicted in Figure 3.8, can be combined. The LOTOS definitions of the two single-agent views are left unchanged. They are instantiated and composed using the interleaving operator. The two processes then fully synchronise with a new process O B Constraint where we define the sequencing of the events between the two single-agent views.

```
      process O_B_Constraint[rd, bk] : exit :=
         rd !give_info ?dsk: Desk_Id ?g_id: Gizmo_Id ?pl_id: Plate_Id
            ?ac_no: Ac_Nr ?inf: Info;
         bk !query_account ?bnk: Bank_Interf_Id !ac_no;
         bk !rtn_query_account !bnk !ac_no ?ok: Bool;
         (  [ok]-> rd !get_gizmo !g_id !dsk;
                   O_B_Constraint[rd, bk]
          []
            [not(ok)] -> rd !refuse_gizmo !g_id !dsk;
                         O_B_Constraint[rd, bk]
         )
       []
         rd ?ev: Event ?dsk: Desk_Id ?g_id: Gizmo_Id;
         O_B_Constraint[rd, bk]
       []
         exit
      endproc (* O_B_Constraint *)
```

Process O_B_Constraint ensures that event give_info occurs before event query_account and that event rtn_query_account occurs before either get_gizmo or refuse_gizmo. It also relates the response by the bank to the choice of whether the owner gets, or is refused, a gizmo.

The other events in O_B_Constraint add no new information and are present as a LOTOS technical feature to ensure that, for every event in the single-agent view process definitions, there is a corresponding event in O_B_Constraint.

A system must be initialised before it can be used. Hence toll gates cannot detect vehicles before the toll gates have been added to the system or a gizmo used before it has been registered. Let us consider the situation where we ensure that a gizmo has been registered before it can be used and that a gizmo must have passed successsfully through an In_Toll before it can pass successfully through an Exit_Toll. Registration occurs in the Operator_Agent view while gizmos are used at toll gates.

Initially, we regarded a vehicle's behaviour at an In_Toll and its subsequent behaviour at an Exit_Toll as a single agent view which we referred to as the Two_Point_Vehicle view. This led to problems when we dealt with multiple instances of the vehicle views. The specification in Appendix B splits the Two_Point_Vehicle view into two separate single-agent views: In_Vehicle describing behaviour at an In_Toll and Exit_Vehicle describing behaviour at an Exit_Toll. In_Vehicle and Exit_Vehicle are composed into a multi-agent view using the Gizmos constraint process given below.

The Gizmos constraint process is defined to maintain the set of currently registered gizmos and the set of gizmos which have entered, but have not yet left a motorway. Other restrictions are concerned with when a vehicle can pass through a toll gate and get a green light, e.g. a vehicle cannot pass through a second In_Toll before it has passed through an Exit_Toll. Process Gizmos interacts with Operator_Agent on gate rd and with the each of the processes specifying behaviour at a toll gate on gate lgt.

```
process Gizmos[rd, lgt](idgs, curr: Gizmo_Id_Set) : noexit :=
   rd !give_info ?dsk : Desk_Id ?idg: Gizmo_Id ?pl: Plate_Id ?ac: Ac_Nr !inf;
   Gizmos[rd, lgt](idgs, curr)
 []
   rd !get_gizmo ?idg: Gizmo_Id ?dsk : Desk_Id;
   Gizmos[rd, lgt](Insert(idg, idgs), curr)
 []
   rd !refuse_gizmo ?idg: Gizmo_Id ?dsk : Desk_Id;
   Gizmos[rd, lgt](idgs, curr)
 []
   (* cancel or report stolen *)
   rd ?op : Event ?dsk : Desk_Id ?idg: Gizmo_Id;
   Gizmos[rd, lgt](Remove(idg, idgs), curr)
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
       [(idg isin idgs) and Is_Enter(t_id) and not(idg isin curr)];
   Gizmos[rd, lgt](idgs, Insert(idg, curr))
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
       [Is_Exit(t_id) and (idg isin curr) and (idg isin idgs)];
   Gizmos[rd, lgt](idgs, Remove(idg, curr))
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
       [Is_Single(t_id) and(idg isin idgs) and not(idg isin curr)];
   Gizmos[rd, lgt](idgs, curr)
 []
   lgt !yellow ?t_id:Toll_Id ?idg: Gizmo_Id
       [(idg notin idgs) or (Is_Exit(t_id) and (idg notin curr))
        or ((idg isin curr) and (Is_Enter(t_id) or Is_Single(t_id)))];
   Gizmos[rd, lgt](idgs, Remove(idg, curr))
 []
   lgt !yellow ?t_id:Toll_Id;
   Gizmos[rd, lgt](idgs, curr)
 []
   lgt !clear ?t_id:Toll_Id;
   Gizmos[rd, lgt](idgs, curr)
endproc (* Gizmos *)
```

Another constraint to be considered is that two vehicles cannot be processed simultaneously at the same toll gate.

3.3.4 Instantiating and validating the user-centred model

Once we have specified an agent view, it can be instantiated from an enclosing process, as described in Section 2.6.4, so that we can check, using the SMILE simulator, that it executes as expected. Multiple agent views, together with their constraints can be checked in a similar way.

As described in Section 2.6.5, the user-centred model can then be created by composing all the agent views and their constraints in a behaviour expression. The full user-centred model must deal with multiple instances of the vehicle views. As number of alternative events offered is now too large to be handled conveniently, the user model is composed with test processes to check that it behaves as expected. The test processes are derived directly from single or multi-agent views. Using LOLA, all possible execution paths of the composition of the user-centred model and a test process can be explored and the result of MUST, MAY or REJECT reported. In this way, the user-centred model can be validated.

The modified LOTOS specification of user-centred model together with a sample test process is shown in Appendix B.

3.4 Creating the system-centred model

The system centred-model is built by following the ROOA method, but it takes advantage of the user-centred model and of the initial object model already created.

The ROOA method involves three main tasks as described in Section 2.8.2.

Task 1: Build an object model

An object model shows the class templates that compose the system and the relationships between their objects. An initial object model, containing interface objects, already exists from the construction of the user-centred model (see Figure 3.10).

Producing an analysis model from a set of requirements is not easy. Object-oriented methods propose strategies for the identification of objects and their attributes, services and relationships. With problems which are primarily data-oriented, an object model can be constructed without paying too much attention to dynamic behaviour. With process-oriented problems, on the other hand, the dynamic aspects of a problem must be explored to help identify the static object structure. Indeed, Rubin and Goldberg [21] suggest that the best way of identifying objects is to focus on their behaviour. When a method such as OMT [23] is used to deal with process-oriented problems, we have found that agent views are a major help in identifying the required objects.

From the requirements we can identify some objects and their attributes which we add to the initial object model. However, as the problem we are analysing has a significant dynamic component, we cannot identify all the objects at this stage. The resulting incomplete object model is shown in Figure 3.11.

Task 2: Refine the object model

During this task we use the user-centred model and the event traces initiated by the agent views to help us identify:

- more objects;
- static relationships;
- attributes and services in each class template;
- message connections.

Task 2.1: Standardise the object model

Some analysis methods do not distinguish between static relationships and message connections, or have services or attributes in their object model. In this task we add those services, attributes, static relationships and message connections which can be easily derived from the requirements and from the user-centred model. We do not worry if the result is incomplete, as more detail will be added in Task 2.2.

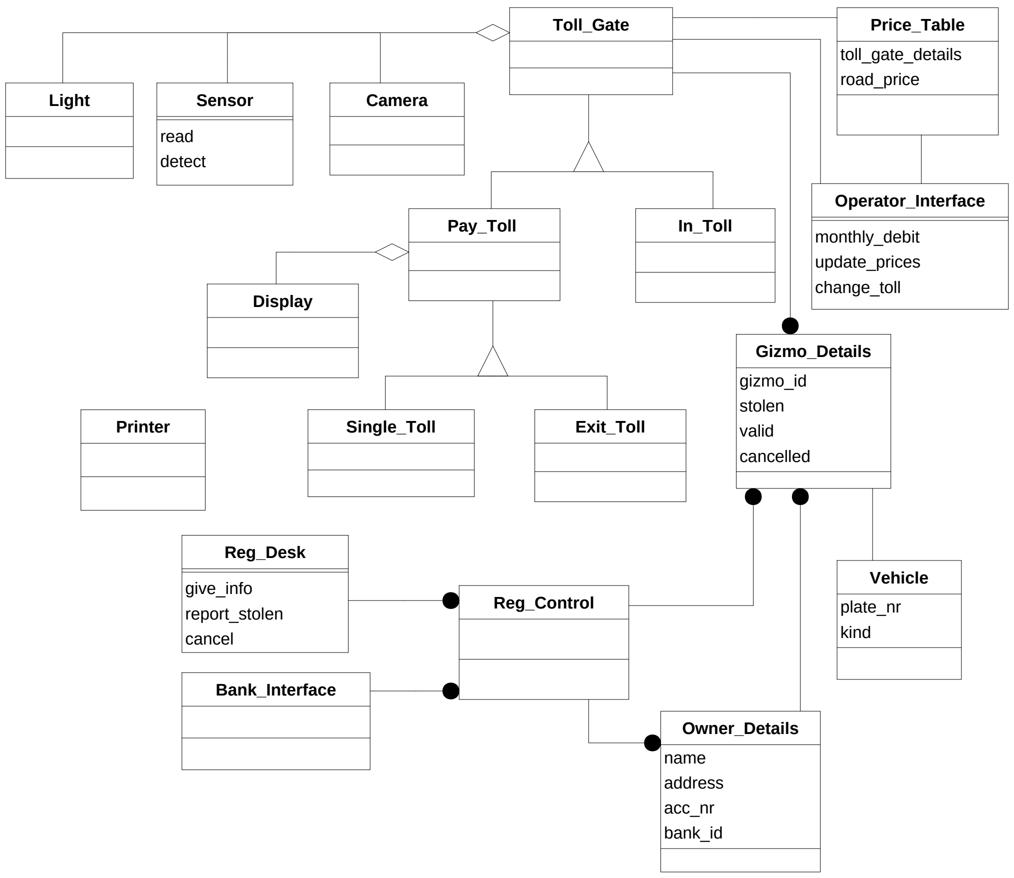

Figure 3.11: Initial OMT object model

We examine the agent views and identify events which are sent to the environment and determine if the relevant interface object must offer a service to some internal object to trigger the external event. For example, in the one-point vehicle view, for `Light` to generate the events `Light.yellow` and `Light.green`, it must offer the services **`turn yellow`** and **`turn green`**.

Also, by analysing the object model in Figure 3.11, we can see that some of the static relationships are in fact message connections. These additions are shown in Figure 3.12 [1].

Task 2.2: Identify dynamic behaviour

In order to identify dynamic behaviour we:

- build Event Trace Diagrams (ETDs) to show sequences of object interactions;
- collect the information in the ETDs in an Object Communication Table (OCT).

Agent views show the interactions between agents and interface objects. The starting event in an agent scenario will trigger a series of interactions between objects internal to the

[1] As LOV/OMT does not give us a notation for messages connections, we use the relationship notation and name it "calls".

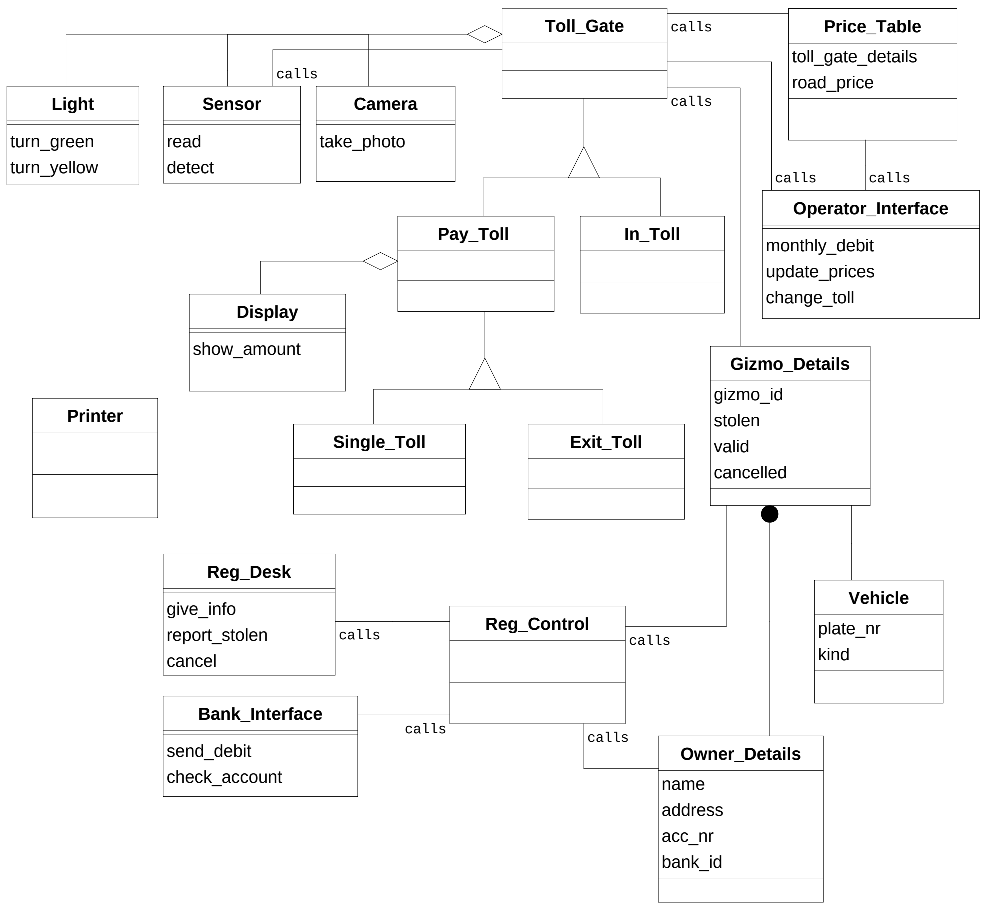

Figure 3.12: OMT object model with message connections

system. We describe these sequences by means of event trace diagrams. Some of the required objects will already have been identified and be represented in the initial object model. An event trace diagram will show the message passing between these objects and its construction may show new objects and services. Each new object and service identified is added to the object model.

Let us analyse the one-point vehicle view. The scenario where the vehicle gets a green light results in the ETD shown in Figure 3.13. This ETD allowed us to identify new services and two new objects: `Gate Processor` and `Usage Details`. We must ensure that a toll gate fully processes a vehicle before the next vehicle arrives. That is the reason for the objects `Light` and `Display` sending a `clear` event to the toll gate to indicate that they are ready. The Toll then sends an `enable` event to allow the sensor to read the next gizmo. This `enable` event must be sent sufficiently quickly so that no vehicle is missed, but it is not sent until `Light` and `Display` have indicated that they are ready for the next vehicle. The presence of `enable` makes this requirement explicit. What we are therefore doing is making explicit requirements which were left vague in the original informal requirements.

We are not saying that explicit `clear` and `enable` signals must be sent from `Light` and `Display` to the toll gate in an implementation. A specification language which modelled time explicitly could use time constraints to specify that `Light` and `Display` must process a vehicle

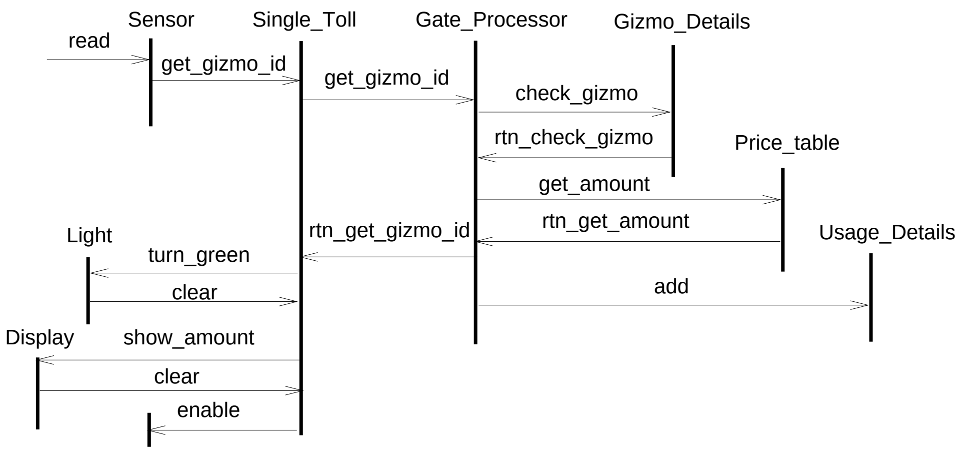

Figure 3.13: ETD for a vehicle passing a single toll gate and getting a green light

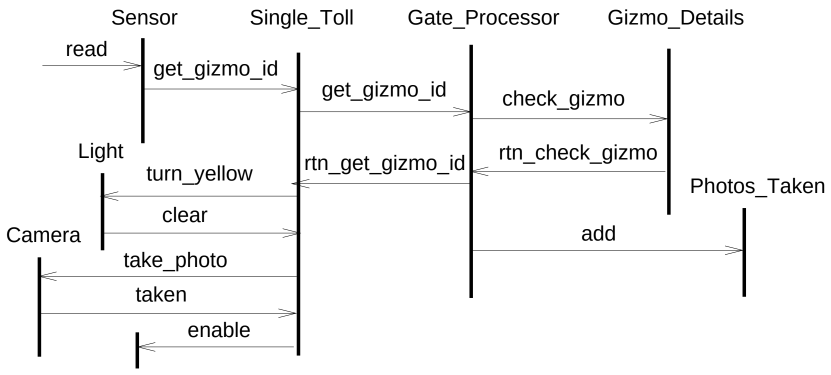

Figure 3.14: ETD for a vehicle passing a single toll gate and getting a yellow light

sufficiently quickly so that they are always ready for the next vehicle. An implementation would then just have to demonstrate that it performed the operations sufficiently quickly. We do not therefore show these events in our object model.

Figure 3.14 shows the ETD which results from the agent scenario where a vehicle gets a yellow light. This ETD allowed us to identify the object `Photos Taken`. The `taken` event ensures that the photograph taken is of the offending vehicle, not some subsequent vehicle. The ETD for the case when a vehicle passes a single toll gate and has no readable gizmo is shown in Figure 3.15.

We now analyse the two-point vehicle view. The agent scenario where the vehicle gets a green light at both the entrance and at the exit gates leads to the ETD shown in Figure 3.16. The ETD corresponding to the two-point vehicle view where a vehicle has no gizmo is shown in Figure 3.17.

The ETD dealing with the agent view where a client makes a registration is depicted in Figure 3.18.

Usually we only draw ETDs for complex agent views. For simple agent views we can identify all the services and objects involved without having to draw the corresponding ETDs.

As we construct the ETDs and identify new objects and services we add them to the existing object model. A version of the object model, including the objects found when

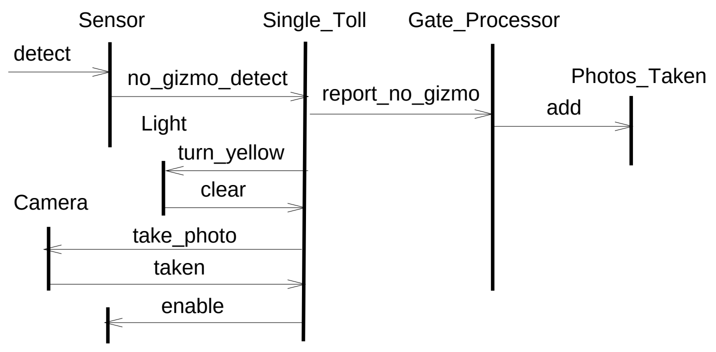

Figure 3.15: ETD for a vehicle passing a single toll gate with no gizmo

identifying dynamic behaviour, is presented in Figure 3.19.

The next step is to collect all to information given in the ETDs and build an OCT. Eventually, this table will be composed of five columns, but now we are only building the first four columns. In the first column we list the class templates that form the object model; in the second column we list the services offered by each class template; in the third column we list, for each service offered by a class template in column one, the services that that class template requires from other class templates to accomplish that particular service; in the fourth column we list, for each service offered in column one, the class templates (clients) which require that service (for each offered service we may have a list of clients). The initial OCT is shown in Tables C.1 and C.2 in Appendix C.

Task 2.3: Structure the object model

Grouping class templates into subsystems or into aggregates is necessary when we are dealing with large complex systems.

This task is difficult to accomplish and so we cannot expect to do it completely and correctly in the first iteration. The low level class templates in the object model often remain almost unchanged during the development, but the high level structure is less stable. Our suggestion is to do only what is obvious to begin with, and then come back to it as our knowledge about each individual class template increases. We use aggregates and subsystems to structure a large system. The fundamental difference between an aggregate and a subsystem is that, while the components are an intrinsic part of the aggregate, and the aggregate is itself an object from the problem domain, a subsystem is merely a grouping of related class templates.

In our road pricing system, we propose, at this stage, two subsystems: `Client Details` and `Toll`. `Toll` encloses the hierarchy defined for the toll gates, with its inheritance and aggregation relationships. `Client Details` incorporates all the data which defines a client: `Gizmo_Details`, `Owner Details` and `Vehicles`. The resulting OCT is shown in Table C.3 shown in Appendix C.

The standard OCT shows, for each class template, the services it offers to other class templates. The services offered by the object components of Toll are given in the OCT shown in Table C.4.

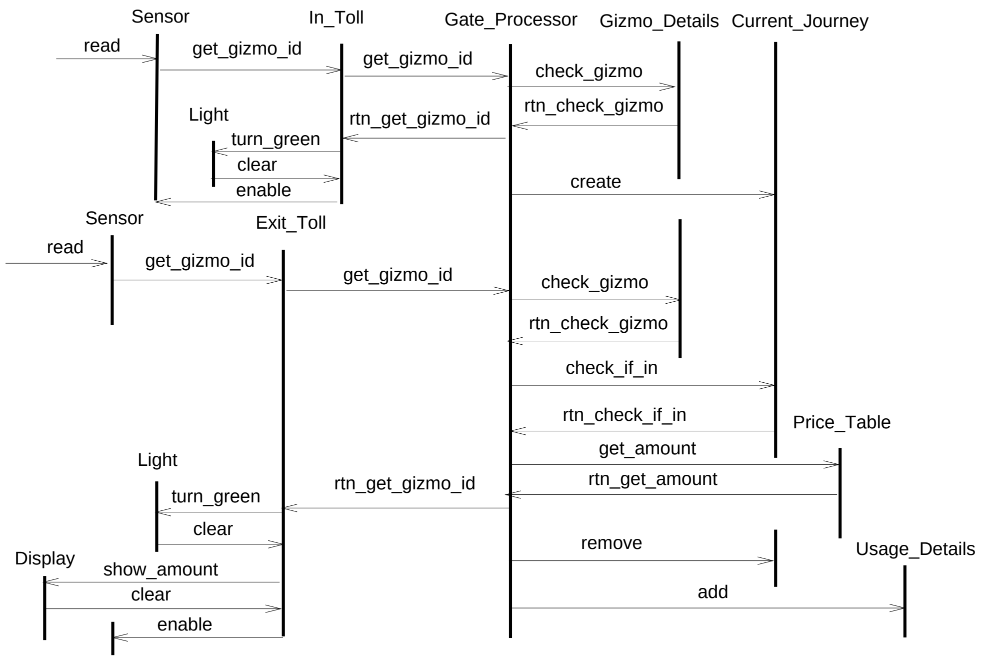

Figure 3.16: ETD for a vehicle successfully passing In and Exit toll gates

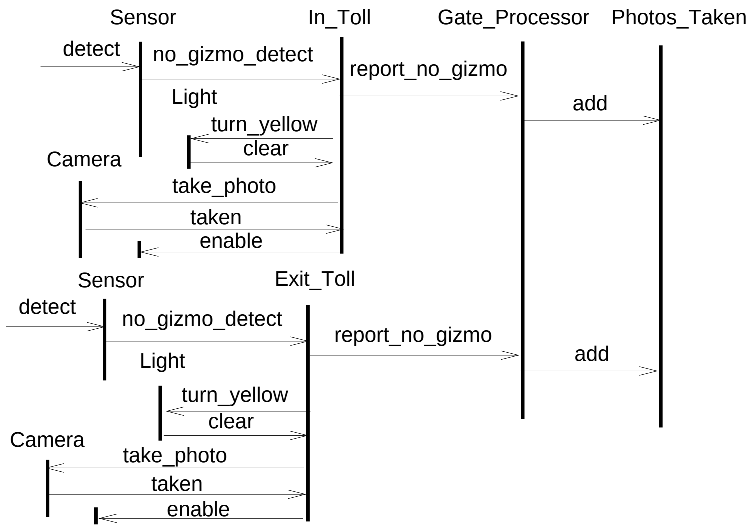

Figure 3.17: ETD for a vehicle passing In and Exit toll gates with no gizmo

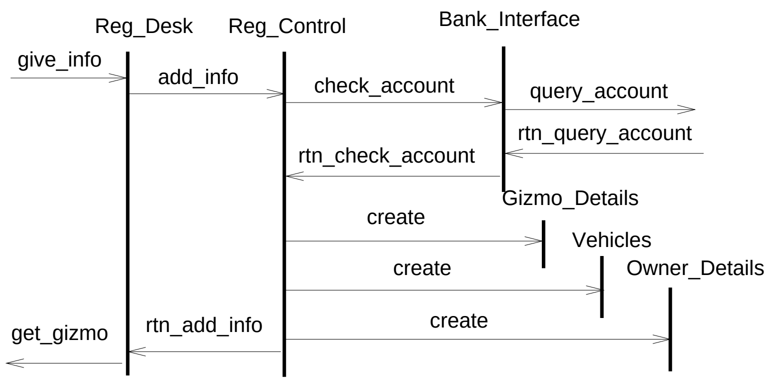

Figure 3.18: Registration of a gizmo

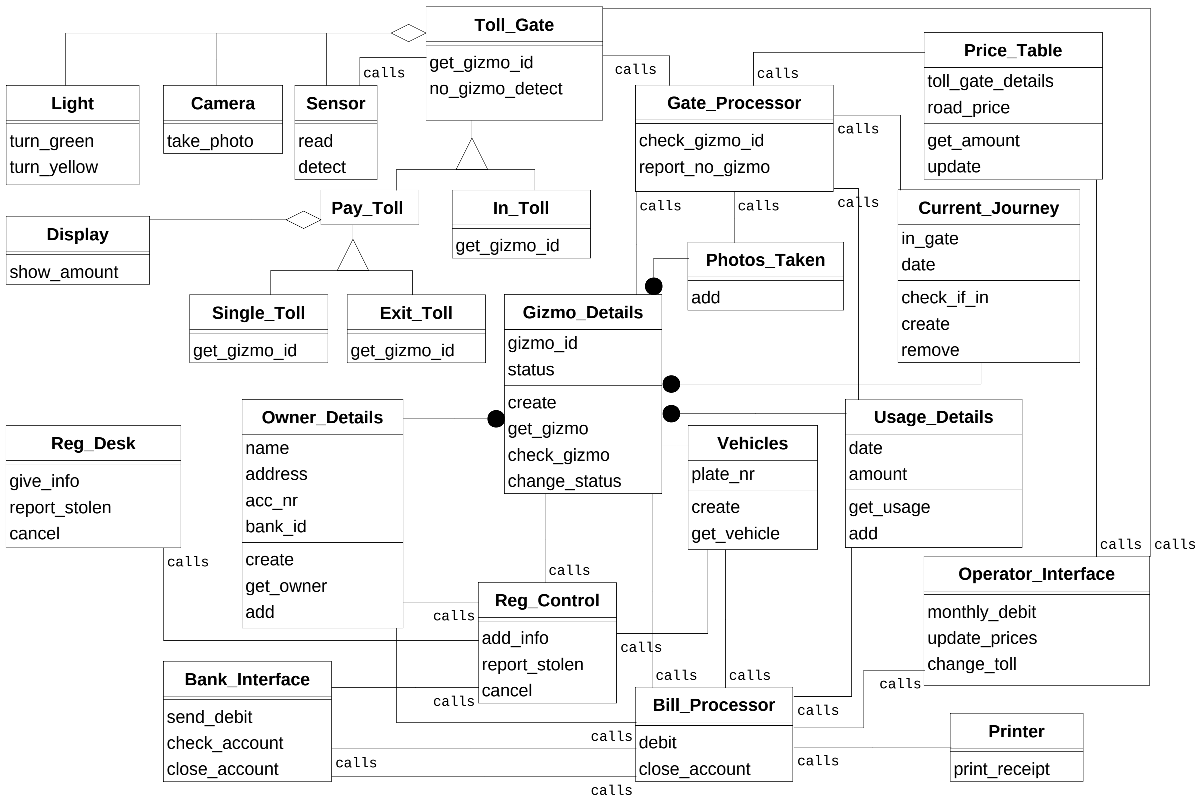

Figure 3.19: Refined OMT object model

Task 3: Build the LOTOS formal model

Task 3.1: Create an Object Communication Diagram (OCD)

The OCD is a graph where, in the first iteration, each node represents an object and each arc connecting two objects represents a gate of communication between them. In later iterations, the diagram is generalised to deal with multiple instances of the same class template. In the beginning, some of the objects may not be connected by arcs to the rest of the diagram. As the method is applied, these objects will either disappear or be connected to the others, and new groupings may appear, refining the diagram.

In order to build the OCD, we first have to complete the OCT, by adding to it the column *Gates*. This column gives the name of the gates that the objects in column one and column four use to communicate between each other.

To identify the gates, we follow three basic rules [13]:

1. Give the same gate name for the object communications which require the same set, or subset, of services; i.e. where there is an overlap between the set of services required by different clients.

2. Give different gate names for object communications which require a different set of services, i.e. where there is no overlap between the set of services required by each client.

   These two rules are the result of requiring that an object cannot use the same gate to communicate with both an object at the same level of abstraction and another object at a different level of abstraction.

3. For each pure server, define a single gate from where it offers all its services.

The completed OCTs are presented in Tables C.5 and C.6 in Appendix C. Based on these OCTs we build the first version of the OCD, presented in Figure 3.20.

Task 3.2: Specify class templates

To specify the behaviour of an object we place ourselves inside that object and act as if we were the centre of the system. By following this strategy, and using the information in the OCT and event trace diagrams, we identify the events the object takes part in and their order. These events correspond to the services the object offers to, or requires of, its environment (i.e. the rest of the system) and are often shown as the options of a choice expression.

We start by specifying the interface objects. For each event in the LOTOS process definition of an agent view, there is an event in the LOTOS process definition of an interface object which has an identical signature. The formal user-centred model is therefore used directly in the construction of the LOTOS definition of the interface objects.

Some events in the user-centred model show the initiation of some action which is to be carried out by the system together with the information which is required from the environment. The others show responses from the system and the information which is to be passed to the environment. The system-centred model must demonstrate how the input information is distributed among the internal objects and how the objects combine to produce the desired output. Details of the events on which the internal objects are to synchronise are already held in the ETDs, but it is the user-centred model that determines the information which is to be passed during event synchronisation.

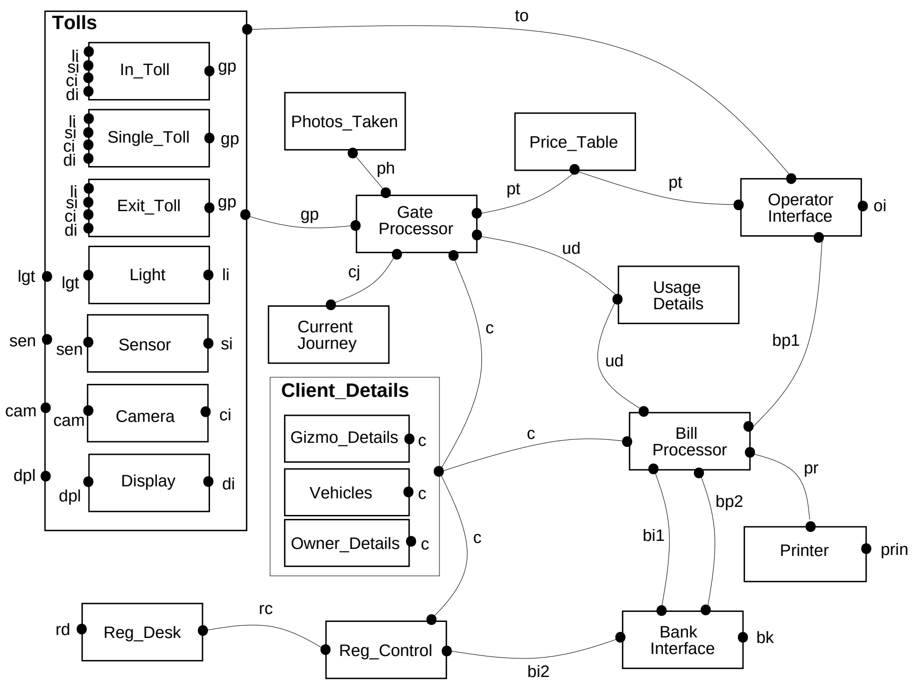

Figure 3.20: First version of the OCD

Inconsistencies in Agent Views

Each agent view describes the expected behaviour of the system from the point of view of a particular agent. When we come to specify the objects in the system-centred model which are to satisfy this behaviour, it will not be unusual for incompatibilities to appear in the expectations of different agents.

When such incompatibilities appear, we must return to the agent views to resolve the differences. Consider, for example, the view of the driver of a vehicle with a gizmo. They expect their gizmo to be read and checked to determine if it is valid. A driver of a vehicle without a gizmo will, on the other hand, expect their *vehicle* to be detected. When these two views are considered together, we see that the sensor must take part in two events; it first detects the presence of a vehicle and then attempts to read its gizmo. The agent views in Figures 3.2 and 3.3, and the LOTOS specification of the user-centred model, should therefore be modified to include the detection of a vehicle. We can, in fact, amalgamate the views of a vehicle with and without a gizmo and that is shown in Figures 3.21 and 3.22. A vehicle is detected and if the sensor cannot read a gizmo, a yellow light is shown.

The user-centred LOTOS specification given in Appendix B shows the amalgamated agent views.

Task 3.3: Compose the objects into a behaviour expression

Following the structure of the OCD, we compose the objects defined in Task 3.2 into a LOTOS behaviour expression by using the parallel operators. A possible top-level LOTOS behaviour expression corresponding to the OCD of Figure 3.20 is:

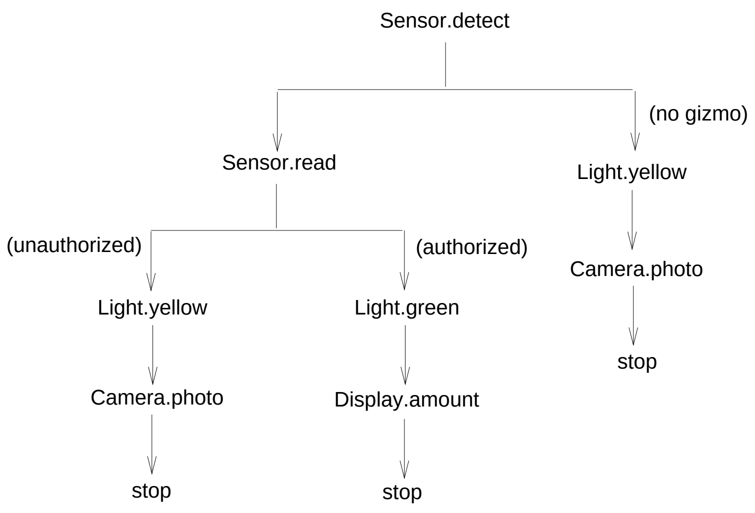

Figure 3.21: One-point vehicle view

```
hide gp, to in
Toll_Gates[sen, lgt, dpl, cam, gp, to]
|[gp, to]|
(hide c, ud, bi1, bi2, bp2 in
( hide bp1 in
  ( hide pt, ph, cj in
    ( Gate_Processor[gp, ph, c, cj, ud, pt](gp_id)
      |||
      Operator_Interface[pt, bp1, oi, to](oper_id)
    )
    |[pt, ph, cj]|
    (Photos_Taken[ph](ph_id)
     |||
     Current_Journey[cj](cj_id, {} of Gizmo_Id_Set)
     |||
     Price_Table[pt](pt_id)
    )
  )
  |[bp1]|
  ( hide pr in
    Bill_Processor[c, ud, bp1, bp2, bi1, pr](bp_id)
    |[pr]|
    Printer[pr, prin](prin_id)
  )
  |||
  ( hide rc in
    Reg_Desk[rd, rc](dsk_id)
    |[rc]|
    Reg_Control[rc, bi2, c](rc_id)
```

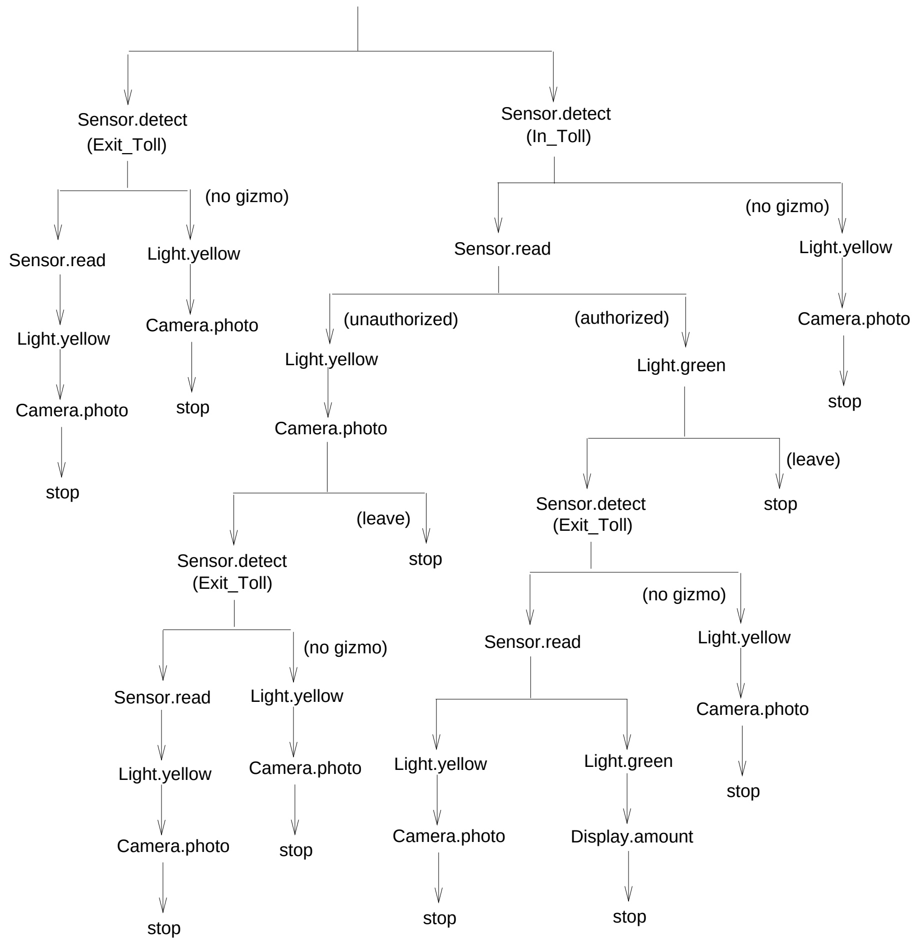

Figure 3.22: Two-point vehicle view

```
    )
  )
  |[c, ud, bi1, bi2, bp2]|
  ( Client_Details[c]
    |||
    Usage_Details[ud](us_id, {} of Gizmo_Id_Set)
    |||
    Bank_Interface[bk, bi1, bi2, bp2](bank_id)
  ))
```

Gates are introduced as locally as possible using the **hide** operator. The full specification is given in Appendix D.

Task 3.4: Prototype the specification

The top-level behaviour expression of the system-centred LOTOS specification is composed in turn with each of the multi-agent views. The resulting compositions are then executed using SMILE to ensure that the system-centred model can satisfy each agent view, i.e. that the `success` event is reached as described in Section 2.6.4.

Each event in the user-centred model should have a corresponding event in the system-centred model which has exactly the same structure and with which it can synchronise. The occurrence of the `success` event in a simulation indicates when a path through the agent view has been successfully completed.

Many agent views assume that the system has already been initialised, e.g. that toll gates have already been added to the system and that there is at least one registered gizmo. That is why the composition is with multi-agent views. Their constraints can then ensure that proper initialisation has taken place.

Execution of the SMILE simulator creates an event tree. This may be created either depth first or breadth first. The branching is kept within bounds because:

- we simulate with uninstantiated variables rather than with values,
- SMILE does not expand subtrees which it recognises as providing behaviour identical to that already analysed,
- composing the system model with an agent view enables us to focus on a particular feature of the overall behaviour.

All paths in the tree should lead to the success event defined in the user-centred specification as the system-centred model should not offer any behaviour which is not expected by the user-centred model.

Testing Equivalence

Interactive prototyping with SMILE is used during the development of the ROOA model to increase our confidence that it satisfies the behaviour expected by the user-centred model. The next step is to demonstrate that the user-centred and ROOA models both describe the same system. As described in Section 2.7.3, we use LOLA to verify that the user-centred and ROOA models cannot be distinguished by experiment, i.e. that they are testing equivalent. Although the system-centred model was developed from the user-centred model it contains a lot of extra internal detail. It is therefore very easy for inconsistencies to be introduced.

The set of test cases used to validate the user-centred model with respect to the informal requirements are now used to check that the two models offer the same behaviour. As the ROOA model has significant internal structure, the number of states generated, and which have to be explored, is very much greater than in the case of the user-centred model. However, as testing equivalence is only concerned with external behaviour, the two specifications can still be equivalent. The much larger number of states generated with the ROOA model means that LOLA takes much longer to produce each result.

The main problem with LOLA is combinatorial explosion where the number of states becomes too large. A major cause of this is an internal event which, even though it only occurs once, can occur at any time once it has been allowed. As it can then be interleaved with all subsequent events, the number of possible traces quickly becomes unmanageable. Such an event is:

```
p !update ?ptid: P_T_Id !inf
```

in `Operator_Interface`. When this event was commented out, the number of transitions generated when the system-centred model was composed with the test where two vehicles both successfully passed through a pair of `In Toll` and `Exit Toll` gates was 4198; when it was present the simulation was aborted after more than 100000 transitions! Specifications should therefore be constructed so that such free floating events do not occur.

When the number of transitions becomes too large, LOLA supports exploration of randomly selected executions. Although that can increase ones confidence that a specification has the expected behaviour, it cannot be used to show testing equivalence.

Task 3.5: Refine the specification

We refine the specification by re-applying Tasks 2.3, 3.1, 3.2, 3.3 and 3.4. During successive refinements we may:

1. Model static relationships.
2. Introduce more object generators.
3. Identify new higher level objects.
4. Demote an object to be specified only as an ADT.
5. Promote an object from an ADT to a process and an ADT.
6. Refine processes and ADTs by introducing more detail.

3.5 Conclusions

In this chapter we showed, by means of an example, how a user centred-model can be built to help both in understanding the requirements and in the construction of a system-centred model. We followed the process described in Chapter 2 to build the user-centred model, and used the ROOA method to build the system centred model. The initial part of the ROOA method had to be changed to intregrate the results produced by the user-centred process. As we used LOTOS to specify both models, we were able to use LOTOS tools to validate that the user-centred model offers the expected external behaviour and to attempt to verify that the user and system-centred models are equivalent.

Chapter 4

Conclusions

Our approach has been to take ideas from several disparate areas, to combine them and to apply them in a novel setting.

From formal methods, we have taken the idea of a formal user-centred model [10]; from requirements engineering, the notion of expressing requirements as a set of multiple viewpoints [12, 18] and from object-oriented analysis, the notion of expressing requirements as a set of use cases [11].

We have adapted the LOTOS constraint-oriented style used for the specification of a telecommunication service to the specification of the user-centred model, have demonstrated how LOTOS can be used in the specification of an object-oriented model and have then used standard LOTOS techniques to validate the user-centred model and to show that the user and system-centred models are testing equivalent [19].

Finally, although modelling the environment as part of the specification of a system is standard practice, especially with embedded systems [4, 24], we believe that our modelling of the environment as a user-centred model, and its subsequent use in the creation and validation of a formal object-oriented specification, is new.

The problem that we have addressed is the large gap that has to be bridged during the transition from informal requirements to an inititial formal requirements specification (i.e. the system-centred model). We have proposed that a formal and executable user-centred model should be constructed to bridge this gap. The construction, and subsequent execution, of the user-centred model helps us to understand, structure and clarify the requirements. The user-centred model is then used to aid the construction of the system-centred model. Validation of the system-centred model is then concerned with verifying that it is equivalent to the user-centred model.

We have shown how the development of a formal user-centred model can be integrated into the ROOA method and have described how the formal user-centred model can complement the OCD and ETDs in the construction of the ROOA model. We have then shown how LOLA can demonstrate that the user-centred and ROOA models cannot be distinguished by testing.

Although we have been concerned with LOTOS and the ROOA method, we believe that the approach is applicable to other formal languages and development methods.

Bibliography

[1] Amyot, D., Bordeleau, F., Buhr, R.J.A., Logrippo, L.: Formal support for design techniques: a Timethreads-LOTOS approach. In: Proceedings FORTE'95, Chapman and Hall 1996, pp. 57-72.

[2] Boehm, B.W.: A Spiral Model of Software Development and Enhancement. *IEEE Computer*, **21**(5), 61-72, 1988.

[3] Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, **14**, 25-59, 1987.

[4] Clark, R.G.: 'Using LOTOS in the Object-Based Development of Embedded Systems'. Unified Computation Laboratory, Oxford University Press, 1992, pp.307-319

[5] Clark, R.G., Jones, V.: The Use of LOTOS in the Formal Development of an OSI Protocol. *Computer Communications*, **15**(2), 86-92, 1992.

[6] Coad, P., Yourdon, E.: *Object Oriented Analysis (Second Edition)*, Yourdon Press, Prentice-Hall 1991.

[7] Eertink H., Wolz D.: Symbolic Execution of LOTOS Specifications. In: Diaz M., Groz R. (eds): *Formal Description Techniques V*, North-Holland 1993, pp. 295-310.

[8] Glinz, P.: An Integrated Formal Model of Scenarios Based on Statecharts. *ESEC'95*, LNCS 989, Springer-Verlag, pp.254-271, 1995.

[9] Henderson-Sellers B., Edwards, J.M.: The Object-Oriented Systems Life Cycle, *Comm ACM*, **33**(9), 142-159, 1990.

[10] Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., and Chen, C.: Formal Approach to Scenario Analysis. *IEEE Software*, 33-41, March 1994.

[11] Jacobson, I.: *Object-Oriented Software Engineering.* Addison-Wesley 1992.

[12] Kotonya, G., Sommerville, I.: Requirements Engineering with Viewpoints. *Software Engineering Journal*, **11**(1), 5-18, 1996.

[13] Moreira, A.M.D.: Rigorous Object-Oriented Analysis. PhD Thesis. Department of Computing Science and Mathematics, University of Stirling, Scotland, 1994.

[14] Moreira, A.M.D., Clark, R.G.: Combining Object-Oriented Analysis and Formal Description Techniques. In: Tokoro, M. and Pareschi, R. (ed): *8th European Conference on Object-Oriented Programming: ECOOP '94*, LNCS 821, Springer-Verlag, 1994 pp. 344-364.

[15] Moreira, A.M.D., Clark, R.G.: Rigorous Object-Oriented Analysis. In: Bertino, E and Urban, S (ed): *ISOOMS: International Symposium on Object Oriented Methodologies and Systems*, LNCS 858, Springer-Verlag, 1994 pp. 65-78.

[16] Moreira, A.M.D., Clark, R.G.: LOTOS in the Object-Oriented Analysis Process. In: Goldsack, S and Kent, S (eds): *Formal Methods in Object Technology*, Springer-Verlag, 1996 pp. 33-46.

[17] Moreira, A.M.D., Clark, R.G.: Adding Rigour to Object-Oriented Analysis. *Software Engineering Journal* **11**(5), 270-280, 1996.

[18] Nuseibeh, B., Kramer, J., Finkelstein, A.: A Framework for Expressing the Relationships between Multiple Views in Requirements Specification. *IEEE Transactions on Software Engineering*, **20**(10), 760-773, 1994.

[19] Quemada, J., Azcorra, A. and Pavon, S.: The Lotosphere Design Methodology. In: Bolognesi, T., van de Lagemaat, J. and Vissers, C. (eds): *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, 1995, pp 29-58.

[20] Regnell, B, Kimbler, K and Wesslen, A: Improving the Use Case Driven Approach to Requirements Engineering, *Second IEEE Int Symposium on Requirements Engineering*, IEEE Press, 1995, pp 40-47.

[21] Rubin, K.S., Goldberg, A.: Object Behaviour Analysis. *Comm ACM*, **35**(9), 48-62, 1992.

[22] Rumbaugh, J.: Getting started. Using use cases to capture requirements, *J Object Oriented Programming*, **7**(5), 8-23, September 1994.

[23] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modelling and Design*, Prentice-Hall 1991.

[24] Sipma, H.B. and Manna, Z.: Specification and Verification of Controlled Systems, *Formal Techniques in Real-Time and Fault Tolerant Systems*, LNCS 863, Springer-Verlag, 1994, pp 641-659.

[25] Turner, K.J., van Sinderen, M.: LOTOS Specification Style for OSI. *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, pp.137-159, 1995.

Appendix A

LOTOS specification of the ATM user-centred model

```
specification Atm [rdr, kpd, scrn, disp, pr, success]: noexit

library
    NaturalNumber
endlib

type Id_Type is
   sorts Id
   opns id1, id2 : -> Id
endtype

type Money_Type is
   sorts Money
   opns m : -> Money
endtype

type Customer_Type is Id_Type
   sorts Customer
   opns Make_Customer : Id, Id, Id -> Customer
        Id : Customer -> Id
        Password : Customer -> Id
        Card_No : Customer -> Id
   eqns forall iden, pass, card : Id
     ofsort Id
        Id(Make_Customer(iden, pass, card)) = iden;
        Password(Make_Customer(iden, pass, card)) = pass;
        Card_No(Make_Customer(iden, pass, card)) = card;
endtype

type ATM_Type is Id_Type
```

```
   sorts Atm
   opns Make_ATM : Id -> Atm
        Id : Atm -> Id
   eqns forall iden : Id
     ofsort Id
        Id(Make_ATM(iden)) = iden;
endtype

type Events is
   sorts Event
   opns insert_card, rep_invalid, return_card, take_card, ask_passw,
        give_passw, withdraw, amount, request_amount, choose_service,
        no_funds, give_money, take_money, balance,
        no_paper, get_info, cancel : -> Event
endtype

behaviour
   Atm_Tree[rdr, kpd, scrn, disp, pr, success]
where

process Atm_Tree[rdr, kpd, scrn, disp, pr, success] : noexit :=
   Atm_Agent[rdr, kpd, scrn, disp, pr]
   >> success; stop
where
   process Atm_Agent[cdr, kpd, scrn, disp, pr] : exit :=
      choice a: Atm, c: Customer[]
      cdr !insert_card !Id(a) !Id(c) !Card_No(c);
      (   scrn !rep_invalid !Id(a) !Id(c);
          Return_Card[cdr](a, c)
       []
          (* Stolen card, transaction stopped, card not returned *)
          exit
       []
          scrn !ask_passw !Id(a) !Id(c);
          Check_Passw[cdr, kpd, scrn, disp, pr](a, c, 0)
      )
   endproc (* Atm_Agent *)

   process Return_Card[cdr](a: Atm, c: Customer) : exit :=
     cdr !return_card !Id(a) !Id(c);
     cdr !take_card !Id(a) !Id(c);
     exit
   endproc (* Return_Card *)

   process Check_Passw[cdr, kpd, scrn, disp, pr](a: Atm, c: Customer, num: Nat)
                      : exit :=
       kpd !give_passw !Id(a) !Id(c) !Password(c);
```

```
        (   [num lt succ(succ(0))] ->
                 scrn !rep_invalid !Id(a) !Id(c);
                 Check_Passw[cdr, kpd, scrn, disp, pr](a, c, succ(num))
         []
            [num eq succ(succ(0))] ->
                 (* Too many wrong passwords *)
                 exit
         []
            Atm_Trans[cdr, kpd, scrn, disp, pr](a, c)
        )
      []
        Cancel[cdr, kpd](a, c)
   endproc (* Check_Passw *)

   process Atm_Trans[cdr, kpd, scrn, disp, pr](a: Atm, c: Customer) : exit :=
     scrn !choose_service !Id(a) !Id(c);
     (    kpd !withdraw !Id(a) !Id(c);
          scrn !request_amount !Id(a) !Id(c);
          kpd !amount !Id(a) !Id(c) ?am: Money;
          (   scrn !no_funds !Id(a) !Id(c);
              Return_Card[cdr](a, c)
           []
              scrn !take_card !Id(a) !Id(c);
              Return_Card[cdr](a, c)
              >> ( disp !give_money !Id(a) !Id(c);
                   disp !take_money !Id(a) !Id(c);
                   exit
                 )
          )
       []
          kpd !balance !Id(a) !Id(c);
          (   scrn !no_paper !Id(a) !Id(c);
              Return_Card[cdr](a, c)
           []
              pr !get_info ?am: Money;
              Return_Card[cdr](a, c)
          )
       []
         Cancel[cdr, kpd](a, c)
     )
   endproc (* Atm_Trans *)

   process Cancel[cdr, kpd](a: Atm, c: Customer) : exit :=
     kpd !cancel !Id(a) !Id(c);
     Return_Card[cdr](a, c)
   endproc (* Cancel *)
endproc (* Atm_Tree *)
```

```
endspec
```

Appendix B

LOTOS specification of the user-centred model of the road pricing system

```
specification rps [sen, lgt, dpl, cam, bk, rd, oi, prin, success]: noexit

library
    NaturalNumber, Boolean
endlib

type Id_Type is Boolean, NaturalNumber
   sorts Id
   opns id1, id2, id3, id4, id5, id6,
        id7, id8, id9, id10, id11,
        id12, id13, id14, id15, id16,
        id17, id18, id19, id20 : -> Id
        h   : Id -> Nat
        _eq_, _ne_, _lt_, _gt_ : Id, Id -> Bool
        First_Set  : Id -> Bool
        Second_Set : Id -> Bool
        Third_Set  : Id -> Bool
        Fourth_Set : Id -> Bool
   eqns forall n1, n2: Id
     ofsort Nat
       h(id1) = 0;
       h(id2) = succ(h(id1));
       h(id3) = succ(h(id2));
       h(id4) = succ(h(id3));
       h(id5) = succ(h(id4));
       h(id6) = succ(h(id5));
       h(id7) = succ(h(id6));
       h(id8) = succ(h(id7));
       h(id9) = succ(h(id8));
       h(id10) = succ(h(id9));
       h(id11) = succ(h(id10));
       h(id12) = succ(h(id11));
       h(id13) = succ(h(id12));
       h(id14) = succ(h(id13));
       h(id15) = succ(h(id14));
       h(id16) = succ(h(id15));
       h(id17) = succ(h(id16));
       h(id18) = succ(h(id17));
       h(id19) = succ(h(id18));
       h(id20) = succ(h(id19));
     ofsort Bool
       n1 eq n2 = h(n1) eq h(n2);
       n1 ne n2 = h(n1) ne h(n2);
       n1 lt n2 = h(n1) lt h(n2);
       n1 gt n2 = h(n1) gt h(n2);
       First_Set(n1)  = h(n1) lt h(id5);
       Second_Set(n1) = not(h(n1) lt h(id5))  and (h(n1) lt h(id10));
       Third_Set(n1) = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
       Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
endtype

type Set_Id_Type is Set actualizedby Id_Type using
    sortnames Id   for Element
              Bool for FBool
endtype

type Toll_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Toll_Id for Id
             Toll_Id_Set for Set
   opnnames Is_Enter for First_Set
            Is_Exit for Second_Set
            Is_Single for Third_Set
endtype

type Desk_Id_Type is
   sorts Desk_Id
   opns dsk_id : -> Desk_Id
endtype

type Info_Type is
   sorts Info
   opns inf : -> Info
endtype

type Gizmo_Id_Set_Type is Set_Id_Type
```

```
   renamedby
   sortnames Gizmo_Id for Id
             Gizmo_Id_Set for Set
endtype

type Plate_Id_Type is Id_Type
   renamedby
   sortnames Plate_Id for Id
endtype

type Ac_Nr_Type is Id_Type
   renamedby
   sortnames Ac_Nr for Id
endtype

type Vehicle_Type is Gizmo_Id_Set_Type, Plate_Id_Type, Ac_Nr_Type, Info_Type
   sorts Vehicle
   opns Make_Vehicle : Gizmo_Id, Plate_Id, Ac_Nr, Info -> Vehicle
        Id : Vehicle -> Gizmo_Id
        Plate_Nr : Vehicle -> Plate_Id
        Account_Nr : Vehicle -> Ac_Nr
        Name_Info : Vehicle -> Info
   eqns forall iden: Gizmo_Id, plate: Plate_Id, acc: Ac_Nr, fo: Info
     ofsort Gizmo_Id
        Id(Make_Vehicle(iden, plate, acc, fo)) = iden;
     ofsort Plate_Id
        Plate_Nr(Make_Vehicle(iden, plate, acc, fo)) = plate;
     ofsort Ac_Nr
        Account_Nr(Make_Vehicle(iden, plate, acc, fo)) = acc;
     ofsort Info
        Name_Info(Make_Vehicle(iden, plate, acc, fo)) = fo;
endtype

type Cost_Type is
   sorts Cost
   opns some_am : -> Cost
endtype

type Op_Id_Type is
   sorts Op_Id
   opns oper_id : -> Op_Id
endtype

type Pr_Id_Type is
   sorts Pr_Id
   opns prin_id : -> Pr_Id
endtype
```

```
type Entry_Type is Id_Type
   renamedby
   sortnames Entry_Val for Id
   opnnames was_green for id1
            was_yellow for id2
endtype

type Bank_Interf_Id_Type is
   sorts Bank_Interf_Id
   opns bank_id : -> Bank_Interf_Id
endtype

type Events is
   sorts Event
   opns read, yellow, green, photo, amount, refuse_gizmo, get_gizmo,
        monthly_receipt, cancel, report_stolen, give_info, detect,
        rtn_give_info, query_account, rtn_query_account, update_prices,
        close_account, debit, monthly_debit, change_toll, new_toll,
        print_receipt, clear : -> Event
endtype

behaviour
 ((In_Vehicles[sen, lgt, dpl, cam]
   |||
   Exit_Vehicles[sen, lgt, dpl, cam]
   |||
   One_Point_Vehicles[sen, lgt, dpl, cam]
   |||
   Owner_Agent[rd]
   |||
   Bank_Acc_Agent[bk]
   |||
   Bank_Debit_Agent[bk]
   |||
   Operator_Agent[oi, prin]
  )
  |[rd, sen, lgt, bk, oi]|
  ((Toll_Ids[sen, oi]({} of Toll_Id_Set)
    |||
    Gizmos[rd, lgt]({} of Gizmo_Id_Set, {} of Gizmo_Id_Set)
    |[rd]|
    O_B_Constraint[rd, bk]
   )
   |[sen, lgt]|
   Busy_Toll[sen, lgt]({} of Toll_Id_Set)
  )
 )

where
process In_Vehicles[sen, lgt, dpl, cam] : noexit :=
   sen !detect ?int: Toll_Id [Is_Enter(int)];
   (In_Vehicle[sen, lgt, dpl, cam](int)
    |||
    In_Vehicles[sen, lgt, dpl, cam]
   )
where
   process In_Vehicle[sen, lgt, dpl, cam](int : Toll_Id) : noexit :=
       sen !read !int ?id_g: Gizmo_Id;
       (  Yellow[lgt, cam](id_g, int)
        []
          lgt !green !int !id_g;
          lgt !clear !int;
          stop
       )
     []
       sen !read !int;
       Yellow_No_G[lgt, cam](int)
   endproc (* In_Vehicle *)
endproc (* In_Vehicles *)

process Exit_Vehicles[sen, lgt, dpl, cam] : noexit :=
   sen !detect ?ext: Toll_Id [Is_Exit(ext)];
   (Exit_Vehicle[sen, lgt, dpl, cam](ext)
    |||
    Exit_Vehicles[sen, lgt, dpl, cam]
   )
where
   process Exit_Vehicle[sen, lgt, dpl, cam](ext : Toll_Id) : noexit :=
        sen !read !ext ?id_g: Gizmo_Id;
        (   Green[lgt,dpl](id_g, ext)
          []
            Yellow[lgt, cam](id_g, ext)
        )
      []
        sen !read !ext;
        Yellow_No_G[lgt, cam](ext)
   endproc (* Exit_Vehicle *)
endproc (* Exit_Vehicles *)

process Yellow[lgt, cam](id_g: Gizmo_Id, iden: Toll_Id) : noexit :=
   lgt !yellow !iden !id_g;
   cam !photo !iden !id_g;
   lgt !clear !iden;
   stop
endproc (* Yellow *)

process Green[lgt, dpl](id_g: Gizmo_Id, iden: Toll_Id) : noexit :=
   lgt !green !iden !id_g;
   dpl !amount !iden !some_am;
   dpl !clear !iden;
   lgt !clear !iden;
   stop
endproc (* Green *)

process Yellow_No_G[lgt, cam](iden: Toll_Id) : noexit :=
   lgt !yellow !iden;
   cam !photo !iden;
   lgt !clear !iden;
   stop
endproc (* Yellow_No_G *)

process One_Point_Vehicles[sen, lgt, dpl, cam] : noexit :=
   sen !detect ?sin: Toll_Id [Is_Single(sin)];
   (One_Point_Vehicle[sen, lgt, dpl, cam](sin)
    |||
    One_Point_Vehicles[sen, lgt, dpl, cam]
   )
where
   process One_Point_Vehicle[sen, lgt, dpl, cam](sin : Toll_Id) : noexit :=
        sen !read !sin ?id_g: Gizmo_Id;
        (  Green[lgt, dpl](id_g, sin)
         []
           Yellow[lgt, cam](id_g, sin)
        )
      []
        sen !read !sin;
        Yellow_No_G[lgt, cam](sin)
   endproc (* One_Point_Vehicle *)
endproc (* One_Point_Vehicles *)

process Owner_Agent[rd] : noexit :=
   rd !give_info ?dsk : Desk_Id ?idg: Gizmo_Id ?pl: Plate_Id ?ac: Ac_Nr !inf;
   (  rd !get_gizmo !idg !dsk;
      Owner_Agent[rd]
    []
      rd !refuse_gizmo !idg !dsk;
      Owner_Agent[rd]
   )
  []
```

```
    Desk_Services[rd]
where
   process Desk_Services[rd] : noexit :=
        rd !cancel ?dsk : Desk_Id ?idg: Gizmo_Id;
        Owner_Agent[rd]
      []
        rd !report_stolen ?dsk : Desk_Id ?idg: Gizmo_Id;
        Owner_Agent[rd]
   endproc (* Desk_Services *)
endproc (* Owner_Agent *)

process Bank_Acc_Agent[bk] : noexit :=
   bk !query_account ?bnk: Bank_Interf_Id ?ac_no: Ac_Nr;
   bk !rtn_query_account !bnk !ac_no ?account_ok: Bool;
   Bank_Acc_Agent[bk]
endproc (* Bank_Acc_Agent *)

   process Bank_Debit_Agent[bk] : noexit :=
        bk !debit ?bnk: Bank_Interf_Id ?ac_no: Ac_Nr;
        (  Bank_Debit_Agent[bk]
         []
           bk !close_account !bnk !ac_no;
           Bank_Debit_Agent[bk]
        )
   endproc (* Bank_Debit_Agent *)

   process Operator_Agent[oi, prin]  : noexit :=
        oi !update_prices ?opr_id: Op_Id !inf;
        Operator_Agent[oi, prin]
      []
        oi !monthly_debit ?opr_id: Op_Id;
        (Print_Out[prin] >> Operator_Agent[oi, prin])
      []
        oi !change_toll ?opr_id: Op_Id ?idt: Toll_Id !inf;
        Operator_Agent[oi, prin]
      []
        oi !new_toll ?opr_id: Op_Id ?idt: Toll_Id !inf;
        Operator_Agent[oi, prin]
   where
      process Print_Out[prin] : exit :=
           prin !print_receipt ?p_id: Pr_Id !inf ?pl: Plate_Id
                !some_am ?ac: Ac_Nr ?idt: Toll_Id;
           Print_Out[prin]
         []
           exit
      endproc (* Print_Out *)
   endproc (* Operator_Agent *)
```

```
process Gizmos[rd, lgt](idgs, curr: Gizmo_Id_Set) : noexit :=
   rd !give_info ?dsk : Desk_Id ?idg: Gizmo_Id ?pl: Plate_Id ?ac: Ac_Nr !inf;
   Gizmos[rd, lgt](idgs, curr)
 []
   rd !get_gizmo ?idg: Gizmo_Id ?dsk : Desk_Id;
   Gizmos[rd, lgt](Insert(idg, idgs), curr)
 []
   rd !refuse_gizmo ?idg: Gizmo_Id ?dsk : Desk_Id;
   Gizmos[rd, lgt](idgs, curr)
 []
   (* cancel or report stolen *)
   rd ?op : Event ?dsk : Desk_Id ?idg: Gizmo_Id[idg isin idgs];
   Gizmos[rd, lgt](Remove(idg, idgs), curr)
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
      [(idg isin idgs) and Is_Enter(t_id) and not(idg isin curr)];
   Gizmos[rd, lgt](idgs, Insert(idg, curr))
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
      [Is_Exit(t_id) and (idg isin curr) and (idg isin idgs)];
   Gizmos[rd, lgt](idgs, Remove(idg, curr))
 []
   lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id
      [Is_Single(t_id) and(idg isin idgs) and not(idg isin curr)];
   Gizmos[rd, lgt](idgs, curr)
 []
   lgt !yellow ?t_id:Toll_Id ?idg: Gizmo_Id
      [(idg notin idgs) or (Is_Exit(t_id) and (idg notin curr))
       or ((idg isin curr) and (Is_Enter(t_id) or Is_Single(t_id)))];
   Gizmos[rd, lgt](idgs, Remove(idg, curr))
 []
   lgt !yellow ?t_id:Toll_Id;
   Gizmos[rd, lgt](idgs, curr)
 []
   lgt !clear ?t_id:Toll_Id;
   Gizmos[rd, lgt](idgs, curr)
endproc (* Gizmos *)

process Toll_Ids[sen, oi](tids: Toll_Id_Set) : noexit :=
     sen !detect ?t_id:Toll_Id [t_id isin tids];
     Toll_Ids[sen, oi](tids)
   []
     sen !read ?t_id:Toll_Id ?idg: Gizmo_Id[t_id isin tids];
     Toll_Ids[sen, oi](tids)
   []
     sen !read ?t_id:Toll_Id[t_id isin tids];
     Toll_Ids[sen, oi](tids)
   []
     oi !update_prices ?o_id: Op_Id !inf;
     Toll_Ids[sen, oi](tids)
   []
     oi !monthly_debit ?o_id: Op_Id;
     Toll_Ids[sen, oi](tids)
   []
     oi !new_toll ?o_id: Op_Id ?idt: Toll_Id !inf;
     Toll_Ids[sen, oi](Insert(idt, tids))
   []
     oi !change_toll ?o_id: Op_Id ?idt: Toll_Id !inf[idt isin tids];
     Toll_Ids[sen, oi](tids)
endproc (* Toll_Ids *)

process Busy_Toll[sen, lgt](busy: Toll_Id_Set) : noexit :=
     sen !detect ?t_id:Toll_Id [not(t_id isin busy)];
     Busy_Toll[sen, lgt](Insert(t_id,busy))
   []
     sen !read ?t_id:Toll_Id ?idg: Gizmo_Id;
     Busy_Toll[sen, lgt](busy)
   []
     sen !read ?t_id:Toll_Id;
     Busy_Toll[sen, lgt](busy)
   []
     lgt !clear ?t_id:Toll_Id[t_id isin busy];
     Busy_Toll[sen, lgt](Remove(t_id,busy))
   []
     lgt !yellow ?t_id:Toll_Id;
     Busy_Toll[sen, lgt](busy)
   []
     lgt !green ?t_id:Toll_Id ?idg: Gizmo_Id;
     Busy_Toll[sen, lgt](busy)
   []
     lgt !yellow ?t_id:Toll_Id ?idg: Gizmo_Id;
     Busy_Toll[sen, lgt](busy)
endproc (* Busy_Toll *)

process O_B_Constraint[rd, bk] : exit :=
   rd !give_info ?dsk: Desk_Id ?g_id: Gizmo_Id ?pl_id: Plate_Id
      ?ac_no: Ac_Nr ?inf: Info;
   bk !query_account ?bnk: Bank_Interf_Id !ac_no;
   bk !rtn_query_account !bnk !ac_no ?ok: Bool;
   (  [ok]-> rd !get_gizmo !g_id !dsk;
             O_B_Constraint[rd, bk]
    []
      [not(ok)] -> rd !refuse_gizmo !g_id !dsk;
```

```
                   O_B_Constraint[rd, bk]
   )
 []
   rd ?ev: Event ?dsk: Desk_Id ?g_id: Gizmo_Id;
   O_B_Constraint[rd, bk]
 []
   exit
endproc (* O_B_Constraint *)

(* Process Testing is a scenario involving two vehicles both
   going through the enter and exit gates of a motorway *)
process Testing[oi, rd, bk, sen, lgt, dpl, cam, success] : noexit :=
    Init[rd, oi, bk]>> Two_Point_Vehicle_A[sen, lgt, dpl, cam] >> success; stop
where

process Init[rd, oi, bk] : exit :=
   oi !new_toll !oper_id !id10 of Toll_Id !inf
        [Is_Single(id10)];
   oi !new_toll !oper_id !id1 of Toll_Id !inf
        [Is_Enter(id1)];
   oi !new_toll !oper_id !id5 of Toll_Id !inf
        [Is_Exit(id5)];
   (let v1: Vehicle = Make_Vehicle(id1 of Gizmo_Id, id1 of  Plate_Id,
                    id1 of  Ac_Nr, inf),
        v2: Vehicle = Make_Vehicle(id2 of Gizmo_Id, id2 of  Plate_Id,
                    id2 of  Ac_Nr, inf) in
    rd !give_info !dsk_id !Id(v1) !Plate_Nr(v1)
         !Account_Nr(v1) !Name_Info(v1);
    bk !query_account !bank_id !Account_Nr(v1);
    bk !rtn_query_account !bank_id !Account_Nr(v1) !true;
    rd !get_gizmo !Id(v1) !dsk_id;
    rd !give_info !dsk_id !Id(v2) !Plate_Nr(v2)
         !Account_Nr(v2) !Name_Info(v2);
    bk !query_account !bank_id !Account_Nr(v2);
    bk !rtn_query_account !bank_id !Account_Nr(v2) !true;
    rd !get_gizmo !Id(v2) !dsk_id;
    exit)
endproc

   process Two_Point_Vehicle_A[sen, lgt, dpl, cam] : exit :=
     sen !detect !id1 of Toll_Id (*Is_Enter(id1)*);
     sen !read !id1 of Toll_Id !id1 of Gizmo_Id;
     lgt !green !id1 of Toll_Id !id1 of Gizmo_Id;
     lgt !clear !id1 of Toll_Id;

     sen !detect !id1 of Toll_Id (*Is_Enter(id1)*);
     sen !read !id1 of Toll_Id !id2 of Gizmo_Id;
     lgt !green !id1 of Toll_Id !id2 of Gizmo_Id;
     lgt !clear !id1 of Toll_Id;

     sen !detect !id5 of Toll_Id (*Is_Exit(id5)*);
     sen !read !id5 of Toll_Id !id2 of Gizmo_Id;
     lgt !green !id5 of Toll_Id !id2 of Gizmo_Id;
     dpl !amount !id5 of Toll_Id ?the_cost : Cost;
     dpl !clear !id5 of Toll_Id;
     lgt !clear !id5 of Toll_Id;

     sen !detect !id5 of Toll_Id (*Is_Exit(id5)*);
     sen !read !id5 of Toll_Id !id1 of Gizmo_Id;
     lgt !green !id5 of Toll_Id !id1 of Gizmo_Id;
     dpl !amount !id5 of Toll_Id ?the_cost : Cost;
     dpl !clear !id5 of Toll_Id;
     lgt !clear !id5 of Toll_Id;
     exit
   endproc (* Two_Point_Vehicle_A *)
endproc (* Testing *)

endspec
```

Appendix C

Object Communication Tables

Class Templates	**Offered Services**	**Required Services**	**Clients**
Sensor (S)	read	In Toll.get gizmo id	External
		Exit_Toll.get gizmo id	External
		Single_Toll.get gizmo id	External
	detect	In_Toll.no gizmo detect	External
		Exit_Toll.no gizmo detect	External
		Single_Toll.no gizmo detect	External
Light (L)	turn green		In Toll, Exit Toll, Single_Toll
	turn_yellow		In Toll, Exit Toll, Single_Toll
Display (D)	show amount		Exit Toll, Single Toll
Camera (C)	take photo		In Toll, Exit Toll, Single_Toll
Single_Toll (S_T)	get gizmo id	Gate Processor.check gizmo id_	Sensor
		Light.turn green	
		Display.show amount	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo detect	Gate Processor.report no gizmo	Sensor
		Light.turn yellow	
		Camera.take photo	
In_Toll (I_N)	get gizmo id	Gate Processor.check gizmo id_	Sensor
		Light.turn green	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo detect	Gate Processor.report no gizmo	Sensor
		Light.turn yellow	
		Camera.take photo	
Exit_Toll (E_T)	get gizmo id	Gate Processor.check gizmo id_	Sensor
		Light.turn green	
		Display.show amount	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo detect	Gate Processor.report no gizmo	Sensor
		Light.turn yellow	
		Camera.take photo	
Pay_Toll (P_T)	get gizmo id	Gate Processor.check gizmo id_	Sensor
		Light.turn green	
		Display.show amount	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo detect	Gate Processor.report no gizmo	Sensor
		Light.turn yellow	
		Camera.take photo	
Toll_Gate (T_G)	get gizmo id	Gate Processor.check gizmo id_	Sensor
		Light.turn green	
		Display.show amount	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo detect	Gate Processor.report no gizmo	Sensor
		Light.turn yellow	
		Camera.take photo	
	create		Operator Interface

Table C.1: OCT with class templates, services offered, services required and clients

Class Templates	**Offered Services**	**Required Services**	**Clients**
Gate_Processor (G_P)	check gizmo id	Gizmo Detail.check gizmo	In Toll, Exit Toll, Single_Toll
		Current Journey.check if in	
		Current Journey.create	
		Price_Table.get amount	
		Usage_Details.add	
		Current Journey.remove	
		Photos_Taken.add	
	report no_gizmo	Photos Taken.add	In Toll, Exit Toll, Single_Toll
Price_Table (P)	get amount		Gate Processor
	update		Operator Interface
Gizmo_Detail (G D)	create		Reg Control
	get_gizmo		Bill Processor
	check_gizmo		Gate Processor
	change status		Reg Control
Usage_Details (U D)	get usage		Bill Processor
	add		Gate Processor
Current Journey (C_J)	create		Gate Processor
	check if in		Gate Processor
	remove		Gate Processor
Photos_Taken (Ph T)	add		Gate Processor
Owner_Detail (O_D)	create		Reg Control
	get_owner		Bill Processsor
	add		Reg_Control
Vehicle (V)	create		Reg Control
	get_vehicle		Bill Processsor
Reg_Control (R_C)	add info	Bank Interface.check account	Reg Desk
		Vehicle.create	
		Gizmo_Detail.create	
		Owner_Detail.create	
	report stolen	Gizmo Detail.change status	Reg Desk
	cancel	Gizmo Detail.change status	Reg Desk
Reg_Desk (R D)	give info	Reg Control.add info	External
	report stolen	Reg Control.report stolen	External
	cancel	Reg_Control.cancel	External
Bank_Interface (B_I)	send debit		Bill Processor
	check account		Reg Control
	close_account	Bill Processor.close account	External
Bill_Processor (B_P)	debit	Usage Details.get usage	Operator Interface
		Gizmo Detail.get gizmo	
		Owner_Detail.get owner	
		Vehicle.get vehicle	
		Bank_Interface.send debit	
		Printer.print receipt	
	close_account	Owner Detail.get owner	Bank Interface
		Gizmo_Detail.change status	
Operator Interface (O_I)	monthly debit	Bill Processor.debit	External
	update prices	Price Table.update	External
	change toll	Toll Gate.create	External
		Price_Table.update	
Printer	print receipt		Bill Processor

Table C.2: OCT with class templates, services offered, services required and clients (continued)

Class Templates	Offered Services	Required Services	Clients
Toll [T_G + I_T + E_T + S_T + P_T + L + D + S + C]	read (S) detect (S) create	T G.get gizmo id T G.no gizmo detect	External External O I
Gate_Processor (G_P)	check gizmo id report no gizmo	Gizmo Detail.check gizmo Current Journey.check if in Current Journey.create Price_Table.get amount Usage_Details.add Current Journey.remove Photos Taken.add Photos Taken.add	T G T G
Price_Table (P)	get amount update		G P O I
Usage_Details (U_D)	get usage add		B P G P
Current Journey (C_J)	create check if in remove		G P G P G P
Photos Taken (Ph T)	add		G P
Client_Details [G_D + O_D + V]	create (G D) get gizmo (G D) check gizmo (G D) change status (G D) create (O D) get owner (O D) add (O D) create (V) get vehicle (V)		R C B P G P R C R C B P R C R C B P
Reg_Control (R C)	add info report stolen cancel	Bank Interface.check account Vehicle.create Owner_Detail.create Gizmo_Detail.create Gizmo Detail.change status Gizmo Detail.change status	R D R D R D
Reg_Desk (R D)	give info report stolen cancel	Reg Control.add info Reg Control.report stolen Reg Control.cancel	External External External
Bank Interface (B_I)	send debit check account close account	 Bill Processor.close account	B P R C External
Bill_Processor (B P)	debit close account	Usage Details.get usage Gizmo_Detail.get gizmo Owner_Detail.get owner Vehicle.get vehicle Bank Interface.send debit Printer.print receipt Owner Detail.get owner Gizmo_Detail.change status	O I B I
Operator Interface (O_I)	monthly debit update prices change toll	Bill Processor.debit Price Table.update Toll Gate.create Price_Table.update	External External External
Printer	print receipt		B P

Table C.3: OCT with subsystems

Class Templates	**Offered Services**	**Required Services**	**Clients**
Sensor	read	T_G.get gizmo id	External
	detect	T_G.no gizmo detect	External
Light	turn_green		T_G
	turn_yellow		T_G
Display	show_amount		P_T
Camera	take_photo		T_G
Toll_Gate	get gizmo id	Gate Processor.check gizmo id_ S	
		Light.turn green	
		Display.show amount	
		Light.turn yellow	
		Camera.take photo	
	no_gizmo_detect	Gate Processor.report no gizmo S	
		Light.turn yellow	
		Camera.take photo	

Table C.4: OCT showing the objects within subsystem Toll

Class Templates	**Offered Services**	**Required Services**	**Clients**	**Gates**
Toll	read (S)	T_G.get gizmo id	External	t
[T_G + I_T + E_T +	detect (S)	T_G.no gizmo detect	External	t
S_T + P_T + L +	create		O_I	to
D + S + C]				
Gate_Processor	check gizmo id	Gizmo_Detail.check gizmo	T_G	gp
(G_P)		Current_Journey.check if in		
		Current_Journey.create		
		Price_Table.get amount		
		Usage_Details.add		
		Current_Journey.remove		
		Photos_Taken.add		
	report_no_gizmo	Photos_Taken.add	T_G	gp
Price_Table (P)	get amount		G_P	pt
	update		O_I	pt
Usage_Details	get_usage		B_P	ud
(U_D)	add		G_P	ud
Current_Journey	create		G_P	cj
(C_J)	check if in		G_P	cj
	remove		G_P	cj
Photos_Taken (Ph_T)	add		G_P	ph
Client_Details	create (G_D)		R_C	c
[G_D + O_D + V]	get gizmo (G_D)		B_P	c
	check_gizmo (G_D)		G_P	c
	change_status (G_D)		R_C	c
	create (O_D)		R_C	c
	get_owner (O_D)		B_P	c
	add (O_D)		R_C	c
	create (V)		R_C	c
	get_vehicle (V)		B_P	c
Reg_Control	add info	Bank_Interface.check account	R_D	rc
(R_C)		Vehicle.create		rc
		Owner_Detail.create		
		Gizmo_Detail.create		
	report_stolen	Gizmo_Detail.change status	R_D	rc
	cancel	Gizmo_Detail.change status	R_D	rc
Reg_Desk (R_D)	give info	Reg_Control.add info	External	rd
	report_stolen	Reg_Control.report stolen	External	rd
	cancel	Reg_Control.cancel	External	rd
Bank_Interface	send debit		B_P	bi1
(B_I)	check account		R_C	bi2
	close_account	Bill_Processor.close account	External	bk
Bill_Processor (B_P)	debit	Usage_Details.get usage	O_I	bp1
		Gizmo_Detail.get gizmo		
		Owner_Detail.get owner		
		Vehicle.get vehicle		
		Bank_Interface.send debit		
		Printer.print receipt		
	close_account	Owner_Detail.get owner	B_I	bp2
		Gizmo_Detail.change status		
Operator_Interface	monthly debit	Bill_Processor.debit	External	oi
(O_I)	update prices	Price_Table.update	External	oi
	change_toll	Toll_Gate.create	External	oi
		Price_Table.update		
Printer	print receipt		B_P	p

Table C.5: OCT with gates

Class Templates	**Offered Services**	**Required Services**	**Clients**	**Gates**
Sensor	read	T_G.get_gizmo_id	External	sen
	detect	T_G.no_gizmo_detect	External	sen
Light	turn_green		T_G	li
	turn_yellow		T_G	li
Display	show_amount		P_T	di
Camera	take_photo		T_G	ci
Toll_Gate	get_gizmo_id	Gate_Processor.check_gizmo_id_	S	si
		Light.turn_green		
		Display.show_amount		
		Light.turn_yellow		
		Camera.take_photo		
	no_gizmo_detect	Gate_Processor.report_no_gizmo	S	si
		Light.turn_yellow		
		Camera.take_photo		

Table C.6: OCT with gates for the objects within subsystem Toll

Appendix D

LOTOS specification of the system-centred model of the road pricing system

```
specification rps [sen, lgt, dpl, cam, bk,
                   rd, prin, oi, success]: noexit

library
    NaturalNumber, Boolean, Set
endlib

type Id_Type is Boolean, NaturalNumber
   sorts Id
   opns id1, id2, id3, id4, id5, id6,
        id7, id8, id9, id10, id11,
        id12, id13, id14, id15, id16,
        id17, id18, id19, id20 : -> Id
        h    : Id -> Nat
        _eq_, _ne_, _lt_, _gt_ : Id, Id -> Bool
        First_Set  : Id -> Bool
        Second_Set : Id -> Bool
        Third_Set  : Id -> Bool
        Fourth_Set : Id -> Bool
   eqns forall n1, n2: Id
     ofsort Nat
       h(id1) = 0;
       h(id2) = succ(h(id1));
       h(id3) = succ(h(id2));
       h(id4) = succ(h(id3));
       h(id5) = succ(h(id4));
       h(id6) = succ(h(id5));
       h(id7) = succ(h(id6));
       h(id8) = succ(h(id7));
       h(id9) = succ(h(id8));
       h(id10) = succ(h(id9));
       h(id11) = succ(h(id10));
       h(id12) = succ(h(id11));
       h(id13) = succ(h(id12));
       h(id14) = succ(h(id13));
       h(id15) = succ(h(id14));
       h(id16) = succ(h(id15));
       h(id17) = succ(h(id16));
       h(id18) = succ(h(id17));
       h(id19) = succ(h(id18));
       h(id20) = succ(h(id19));
     ofsort Bool
       n1 eq n2 = h(n1) eq h(n2);
       n1 ne n2 = h(n1) ne h(n2);
       n1 lt n2 = h(n1) lt h(n2);
       n1 gt n2 = h(n1) gt h(n2);
       First_Set(n1)  = h(n1) lt h(id5);
       Second_Set(n1) = not(h(n1) lt h(id5))  and (h(n1) lt h(id10));
       Third_Set(n1) = not(h(n1) lt h(id10)) and (h(n1) lt h(id15));
       Fourth_Set(n1) = not(h(n1) lt h(id15)) and (h(n1) lt h(id20));
endtype

type Set_Id_Type is Set actualizedby Id_Type using
    sortnames Id   for Element
              Bool for FBool
endtype

type Toll_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Toll_Id for Id
             Toll_Id_Set for Set
   opnnames Is_Enter for First_Set
            Is_Exit for Second_Set
            Is_Single for Third_Set
endtype

type Desk_Id_Type is
   sorts Desk_Id
   opns dsk_id : -> Desk_Id
endtype

type Gizmo_Stat_Type is Id_Type
   renamedby
   sortnames Gizmo_Status for Id
   opnnames g_ok for id1
```

```
            g_stolen for id2
            g_cancel for id3
            g_closed for id4
endtype

type Cost_Type is
   sorts Cost
   opns some_am : -> Cost
endtype

type Date_Type is
   sorts Date
   opns the_date : -> Date
endtype

type Info_Type is
   sorts Info
   opns inf : -> Info
endtype

type Ac_Nr_Set_Type is Set_Id_Type
   renamedby
   sortnames Ac_Nr for Id
             Ac_Nr_Set for Set
endtype

type Plate_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Plate_Id for Id
             Plate_Id_Set for Set
endtype

type Gizmo_Id_Set_Type is Set_Id_Type
   renamedby
   sortnames Gizmo_Id for Id
             Gizmo_Id_Set for Set
endtype

type G_P_Id_Type is
   sorts G_P_Id
   opns gp_id : -> G_P_Id
endtype

type Ph_T_Id_Type is
   sorts Ph_T_Id
   opns ph_id : -> Ph_T_Id
endtype
```

```
type P_T_Id_Type is
   sorts P_T_Id
   opns pt_id : -> P_T_Id
endtype

type C_J_Id_Type is
   sorts C_J_Id
   opns cj_id : -> C_J_Id
endtype

type R_C_Id_Type is
   sorts R_C_Id
   opns rc_id : -> R_C_Id
endtype

type Op_Id_Type is
   sorts Op_Id
   opns oper_id : -> Op_Id
endtype

type Bill_P_Id_Types is
   sorts Bill_P_Id
   opns bp_id : -> Bill_P_Id
endtype

type Usage_List_Id_Type is
   sorts Usage_List_Id
   opns us_id : -> Usage_List_Id
endtype

type Bank_Interf_Id_Type is
   sorts Bank_Interf_Id
   opns bank_id : -> Bank_Interf_Id
endtype

type Pr_Id_Type is
   sorts Pr_Id
   opns prin_id : -> Pr_Id
endtype

type Vehicle_Type is Gizmo_Id_Set_Type, Plate_Id_Set_Type,
     Ac_Nr_Set_Type, Info_Type
   sorts Vehicle
   opns Make_Vehicle : Gizmo_Id, Plate_Id, Ac_Nr, Info -> Vehicle
        Id : Vehicle -> Gizmo_Id
        Plate_Nr : Vehicle -> Plate_Id
        Account_Nr : Vehicle -> Ac_Nr
        Name_Info : Vehicle -> Info
   eqns forall iden: Gizmo_Id, plate: Plate_Id, acc: Ac_Nr, fo: Info
     ofsort Gizmo_Id
        Id(Make_Vehicle(iden, plate, acc, fo)) = iden;
     ofsort Plate_Id
        Plate_Nr(Make_Vehicle(iden, plate, acc, fo)) = plate;
     ofsort Ac_Nr
        Account_Nr(Make_Vehicle(iden, plate, acc, fo)) = acc;
     ofsort Info
        Name_Info(Make_Vehicle(iden, plate, acc, fo)) = fo;
endtype

type Events is
   sorts Event
   opns read, yellow, green, photo, amount, refuse_gizmo,
        monthly_receipt, cancel, report_stolen, give_info, detect,
        query_account, rtn_query_account, clear, enable, taken,
        debit, check_gizmo_id, no_gizmo_detect,
        turn_green, turn_yellow, take_photo, show_amount,
        rtn_check_gizmo_id, check_gizmo, change_status,
        get_amount, rtn_check_gizmo, add, update_prices,
        rtn_check_if_in, check_if_in, get_usage, update,
        check_account, rtn_check_account, send_debit, change_toll,
        get_vehicle, get_owner, get_gizmo_id, monthly_debit,
        report_no_gizmo, create, remove, close_account, new_toll,
        add_info, print_receipt, rtn_add_info, get_gizmo : -> Event
endtype

behaviour
  (hide gp, to in
   Toll_Gates[sen, lgt, dpl, cam, gp, to]
   |[gp, to]|
   (hide c, ud, bi1, bi2, bp2 in
   ( hide bp1 in
     ( hide pt, ph, cj in
       ( Gate_Processor[gp, ph, c, cj, ud, pt](gp_id)
         |||
         Operator_Interface[pt, bp1, oi, to](oper_id)
       )
       |[pt, ph, cj]|
       (Photos_Taken[ph](ph_id)
        |||
        Current_Journey[cj](cj_id, {} of Gizmo_Id_Set)
        |||
        Price_Table[pt](pt_id)
       )
```

```
      )
      |[bp1]|
      ( hide pr in
        Bill_Processor[c, ud, bp1, bp2, bi1, pr](bp_id)
        |[pr]|
        Printer[pr, prin](prin_id)
      )
      |||
      ( hide rc in
        Reg_Desk[rd, rc](dsk_id)
        |[rc]|
        Reg_Control[rc, bi2, c](rc_id)
      )
    )
    |[c, ud, bi1, bi2, bp2]|
    ( Client_Details[c]
      |||
      Usage_Details[ud](us_id, {} of Gizmo_Id_Set)
      |||
      Bank_Interface[bk, bi1, bi2, bp2](bank_id)
    ))
  )
  where

(* Process Testing is a scenario involving two vehicles both
   going through the enter and exit gates of a motorway *)
process Testing[oi, rd, bk, sen, lgt, dpl, cam, success] : noexit :=
    Init[rd, oi, bk]>> Two_Point_Vehicle_A[sen, lgt, dpl, cam] >> success; stop
where

process Init[rd, oi, bk] : exit :=
   oi !new_toll !oper_id !id10 of Toll_Id !inf
        [Is_Single(id10)];
   oi !new_toll !oper_id !id1 of Toll_Id !inf
        [Is_Enter(id1)];
   oi !new_toll !oper_id !id5 of Toll_Id !inf
        [Is_Exit(id5)];
   (let v1: Vehicle = Make_Vehicle(id1 of Gizmo_Id, id1 of  Plate_Id,
                   id1 of  Ac_Nr, inf),
        v2: Vehicle = Make_Vehicle(id2 of Gizmo_Id, id2 of  Plate_Id,
                   id2 of  Ac_Nr, inf) in
    rd !give_info !dsk_id !Id(v1) !Plate_Nr(v1)
         !Account_Nr(v1) !Name_Info(v1);
    bk !query_account !bank_id !Account_Nr(v1);
    bk !rtn_query_account !bank_id !Account_Nr(v1) !true;
    rd !get_gizmo !Id(v1) !dsk_id;
    rd !give_info !dsk_id !Id(v2) !Plate_Nr(v2)
```

```
          !Account_Nr(v2) !Name_Info(v2);
     bk !query_account !bank_id !Account_Nr(v2);
     bk !rtn_query_account !bank_id !Account_Nr(v2) !true;
     rd !get_gizmo !Id(v2) !dsk_id;
     exit)
endproc

   process Two_Point_Vehicle_A[sen, lgt, dpl, cam] : exit :=
     sen !detect !id1 of Toll_Id (*Is_Enter(id1)*);
     sen !read !id1 of Toll_Id !id1 of Gizmo_Id;
     lgt !green !id1 of Toll_Id !id1 of Gizmo_Id;
     lgt !clear !id1 of Toll_Id;

     sen !detect !id1 of Toll_Id (*Is_Enter(id1)*);
     sen !read !id1 of Toll_Id !id2 of Gizmo_Id;
     lgt !green !id1 of Toll_Id !id2 of Gizmo_Id;
     lgt !clear !id1 of Toll_Id;

     sen !detect !id5 of Toll_Id (*Is_Exit(id5)*);
     sen !read !id5 of Toll_Id !id2 of Gizmo_Id;
     lgt !green !id5 of Toll_Id !id2 of Gizmo_Id;
     dpl !amount !id5 of Toll_Id ?the_cost : Cost;
     dpl !clear !id5 of Toll_Id;
     lgt !clear !id5 of Toll_Id;

     sen !detect !id5 of Toll_Id (*Is_Exit(id5)*);
     sen !read !id5 of Toll_Id !id1 of Gizmo_Id;
     lgt !green !id5 of Toll_Id !id1 of Gizmo_Id;
     dpl !amount !id5 of Toll_Id ?the_cost : Cost;
     dpl !clear !id5 of Toll_Id;
     lgt !clear !id5 of Toll_Id;
     exit
   endproc (* Two_Point_Vehicle_A *)
endproc (* Testing *)

process Sensor[sen, si](id: Toll_Id) : exit :=
   sen !detect !id;
   (  sen !read !id ?gi: Gizmo_Id;
      si !get_gizmo_id !id !gi;
      si !enable !id;
      Sensor[sen, si](id)
    []
      sen !read !id; (* fail to read gizmo *)
      si !no_gizmo_detect !id;
      si !enable !id;
      Sensor[sen, si](id)
   )
```

```
endproc (* Sensor *)

process Light[li, lgt](id: Toll_Id) : exit :=
     li !turn_green !id ?gi: Gizmo_Id;
     lgt !green !id !gi;
     lgt !clear !id;
     li !clear !id;
     Light[li, lgt](id)
   []
     li !turn_yellow !id ?gi: Gizmo_Id;
     lgt !yellow !id !gi;
     lgt !clear !id;
     li !clear !id;
     Light[li, lgt](id)
   []
     li !turn_yellow !id;
     lgt !yellow !id;
     lgt !clear !id;
     li !clear !id;
     Light[li, lgt](id)
endproc (* Light *)

process Camera[ci, cam](id: Toll_Id) : exit :=
     ci !take_photo !id ?gi: Gizmo_Id;
     cam !photo !id !gi;
     ci !taken !id;
     Camera[ci, cam](id)
     (* gizmo id imprinted on photograph *)
   []
     ci !take_photo !id;
     cam !photo !id;
     ci !taken !id;
     Camera[ci, cam](id)
endproc (* Camera *)

process Display[di, dpl](id: Toll_Id) : exit :=
   di !show_amount !id ?am: Cost;
   dpl !amount !id !am;
   dpl !clear !id;
   di !clear !id;
   Display[di, dpl](id)
endproc (* Display *)

process Toll_Gates[sen, lgt, dpl, cam, t, to] : noexit :=
   In_Tolls[sen, lgt, cam, t, to]({} of Toll_Id_Set)
   |||
   Single_Tolls[sen, lgt, dpl, cam, t, to]({} of Toll_Id_Set)
   |||
   Exit_Tolls[sen, lgt, dpl, cam, t, to]({} of Toll_Id_Set)
where
   process Toll[sen, si, li, lgt, ci, cam](id: Toll_Id) : exit :=
      Light[li, lgt](id)
      |||
      Sensor[sen, si](id)
      |||
      Camera[ci, cam](id)
   endproc (* Toll *)

   process Toll_Gate[si, li, ci, t](id: Toll_Id) : exit :=
      si !no_gizmo_detect !id;
      t !report_no_gizmo ?gpid: G_P_Id !id;
      li !turn_yellow !id;
      ci !take_photo !id;
      ci !taken !id;
      li !clear !id;
      si !enable !id;
      exit
   endproc (* Toll_Gate *)

   process In_Tolls[sen, lgt, cam, t, to](ids: Toll_Id_Set) : noexit :=
      to !create ?id: Toll_Id [(id notin ids) and Is_Enter(id)];
      (In_Toll[sen, lgt, cam, t](id)
       |||
       In_Tolls[sen, lgt, cam, t, to](Insert(id, ids))
      )
   where
      process In_Toll[sen, lgt, cam, t](id: Toll_Id) : noexit :=
         hide si, li, ci in
            The_Toll[si, li, ci, t](id)
            |[si, li, ci]|
            Toll[sen, si, li, lgt, ci, cam](id)
      where
         process The_Toll[si, li, ci, t](id: Toll_Id) : noexit :=
            si !get_gizmo_id !id ?gi: Gizmo_Id;
            t !check_gizmo_id ?gpid: G_P_Id !gi !id;
            t !rtn_check_gizmo_id !gpid !gi
              !id ?ok:Bool ?an_amount: Cost;
            ([ok] -> li !turn_green !id !gi;
                     li !clear !id;
                     si !enable !id;
                     The_Toll[si, li, ci, t](id)
             []
             [not(ok)] -> li !turn_yellow !id !gi;
                          ci !take_photo !id !gi;
```

```
                         ci !taken !id;
                         li !clear !id;
                         si !enable !id;
                         The_Toll[si, li, ci, t](id)
             )
           []
             (Toll_Gate[si, li, ci, t](id)
              >> The_Toll[si, li, ci, t](id))
          endproc (* The_Toll *)
       endproc (* In_Toll *)
    endproc (* In_Tolls *)

    process Pay_Toll[sen, si, li, lgt, ci, cam, di, dpl](id: Toll_Id) : exit :=
          Toll[sen, si, li, lgt, ci, cam](id)
          |||
          Display[di, dpl](id)
    endproc (* Pay_Toll *)

    process Single_Tolls[sen, lgt, dpl, cam, t, to](ids: Toll_Id_Set) : noexit :=
       to !create ?id: Toll_Id [(id notin ids) and Is_Single(id)];
       (Single_Toll[sen, lgt, dpl, cam, t](id)
        |||
        Single_Tolls[sen, lgt, dpl, cam, t, to](Insert(id, ids))
       )
    where
       process Single_Toll[sen, lgt, dpl, cam, t](id: Toll_Id) : noexit :=
          hide si, li, ci, di in
             The_Toll[si, li, ci, di, t](id)
             |[si, li, ci, di]|
             Pay_Toll[sen, si, li, lgt, ci, cam, di, dpl](id)
       where
          process The_Toll[si, li, ci, di, t](id: Toll_Id) : noexit :=
             si !get_gizmo_id !id ?gi: Gizmo_Id;
             t !check_gizmo_id ?gpid: G_P_Id !gi !id;
             t !rtn_check_gizmo_id !gpid !gi
               !id ?ok:Bool ?an_amount: Cost;
             ([ok] -> li !turn_green !id !gi;
                      di !show_amount !id !an_amount;
                      di !clear !id;
                      li !clear !id;
                      si !enable !id;
                      The_Toll[si, li, ci, di, t](id)
              []
              [not(ok)] -> li !turn_yellow !id !gi;
                           ci !take_photo !id !gi;
                           ci !taken !id;
                           li !clear !id;
```

```
                          si !enable !id;
                          The_Toll[si, li, ci, di, t](id)
            )
          []
            (Toll_Gate[si, li, ci, t](id)
             >> The_Toll[si, li, ci, di, t](id))
         endproc (* The_Toll *)
      endproc (* Single_Toll *)
   endproc (* Single_Tolls *)

   process Exit_Tolls[sen, lgt, dpl, cam, t, to](ids: Toll_Id_Set) : noexit :=
      to !create ?id: Toll_Id [(id notin ids) and Is_Exit(id)];
      (Exit_Toll[sen, lgt, dpl, cam, t](id)
       |||
       Exit_Tolls[sen, lgt, dpl, cam, t, to](Insert(id, ids))
      )
   where
      process Exit_Toll[sen, lgt, dpl, cam, t](id: Toll_Id) : noexit :=
         hide si, li, ci, di in
            The_Toll[si, li, ci, di, t](id)
            |[si, li, ci, di]|
            Pay_Toll[sen, si, li, lgt, ci, cam, di, dpl](id)
      where
         process The_Toll[si, li, ci, di, t](id: Toll_Id) : noexit :=
               si !get_gizmo_id !id ?gi: Gizmo_Id;
               t !check_gizmo_id ?gpid: G_P_Id !gi !id;
               t !rtn_check_gizmo_id !gpid !gi
                 !id ?ok:Bool ?an_amount: Cost;
               ([ok] -> li !turn_green !id !gi;
                        di !show_amount !id !an_amount;
                        di !clear !id;
                        li !clear !id;
                        si !enable !id;
                        The_Toll[si, li, ci, di, t](id)
                []
                [not(ok)] -> li !turn_yellow !id !gi;
                             ci !take_photo !id !gi;
                             ci !taken !id;
                             li !clear !id;
                             si !enable !id;
                             The_Toll[si, li, ci, di, t](id)
               )
             []
               (Toll_Gate[si, li, ci, t](id)
                >> The_Toll[si, li, ci, di, t](id))
         endproc (* The_Toll *)
      endproc (* Exit_Toll *)
   endproc (* Exit_Tolls *)
endproc (* Toll_Gates *)

process Gate_Processor[t, pt, g, cj, u, p](id: G_P_Id) : noexit :=
      t !check_gizmo_id !id ?gi: Gizmo_Id ?idt: Toll_Id;
      g !check_gizmo !gi !idt;
      g !rtn_check_gizmo !gi !idt ?ok:Bool;
      (  [ok and Is_Single(idt)] ->
            p !get_amount ?ptid: P_T_Id !idt ?an_amount: Cost;
            u !add ?us_id: Usage_List_Id !idt !gi !an_amount !the_date;
            t !rtn_check_gizmo_id !id !gi !idt !ok !an_amount;
            Gate_Processor[t, pt, g, cj, u, p](id)
       []
         [not(ok) and Is_Single(idt)] ->
            pt !add ?phid: Ph_T_Id !gi !idt;
            t !rtn_check_gizmo_id !id !gi !idt !ok ?an_amount: Cost;
            Gate_Processor[t, pt, g, cj, u, p](id)
       []
         [ok and Is_Enter(idt)] ->
            cj !create ?cj_i: C_J_Id !idt !gi;
            t !rtn_check_gizmo_id !id !gi !idt !ok ?an_amount: Cost;
            Gate_Processor[t, pt, g, cj, u, p](id)
       []
         [not(ok) and Is_Enter(idt)] ->
            pt !add ?phid: Ph_T_Id !gi !idt;
            t !rtn_check_gizmo_id !id !gi !idt !ok ?an_amount: Cost;
            Gate_Processor[t, pt, g, cj, u, p](id)
       []
         [ok and Is_Exit(idt)] ->
            cj !check_if_in ?cj_i: C_J_Id !idt !gi;
            cj !rtn_check_if_in !cj_i !idt !gi ?is_in: Bool;
            (  [is_in] ->
                  p !get_amount ?ptid: P_T_Id !idt ?an_amount: Cost;
                  cj !remove ?cj_i: C_J_Id !idt !gi;
                  u !add ?us_id: Usage_List_Id !idt !gi !an_amount !the_date;
                  t !rtn_check_gizmo_id !id !gi
                    !idt !is_in !an_amount;
                  Gate_Processor[t, pt, g, cj, u, p](id)
             []
               [not(is_in)] ->
                  pt !add ?phid: Ph_T_Id !gi !idt;
                  t !rtn_check_gizmo_id !id !gi
                    !idt !is_in ?an_amount: Cost;
                  Gate_Processor[t, pt, g, cj, u, p](id)
            )
       []
         [not(ok) and Is_Exit(idt)] ->
```

```
            pt !add ?phid: Ph_T_Id !gi !idt;
            t !rtn_check_gizmo_id !id !gi !idt !ok ?an_amount: Cost;
            Gate_Processor[t, pt, g, cj, u, p](id)
      )
    []
      t !report_no_gizmo !id ?idt: Toll_Id;
      pt !add ?phid: Ph_T_Id !idt;
      Gate_Processor[t, pt, g, cj, u, p](id)
endproc (* Gate_Processor *)

process Price_Table[p](id: P_T_Id) : noexit :=
      p !get_amount !id ?idt: Toll_Id !some_am;
      Price_Table[p](id)
    []
      p !update !id !inf;
      Price_Table[p](id)
endproc (* Price_Table *)

process Usage_Details[u](id: Usage_List_Id, idgs: Gizmo_Id_Set) : noexit :=
      u !add !id ?idt: Toll_Id ?gi: Gizmo_Id !some_am !the_date;
      Usage_Details[u](id, Insert(gi, idgs))
    []
      u !get_usage !id ?idg: Gizmo_Id !some_am !the_date
         ?idt: Toll_Id [idg isin idgs];
      Usage_Details[u](id, Remove(idg, idgs))
endproc (* Usage_Details *)

process Current_Journey[cj](id: C_J_Id, idgs: Gizmo_Id_Set) : noexit :=
      cj !create !id ?idt: Toll_Id ?gi: Gizmo_Id;
      Current_Journey[cj](id, Insert(gi, idgs))
    []
      cj !check_if_in !id ?idt: Toll_Id ?gi: Gizmo_Id;
      cj !rtn_check_if_in !id !idt !gi !(gi isin idgs);
      Current_Journey[cj](id, idgs)
    []
      cj !remove !id ?idt: Toll_Id ?gi: Gizmo_Id;
      Current_Journey[cj](id, Remove(gi, idgs))
endproc (* Current_Journey *)

process Photos_Taken[pt](id: Ph_T_Id) : noexit :=
      pt !add !id ?gi: Gizmo_Id ?idt: Toll_Id;
      Photos_Taken[pt](id)
    []
      pt !add !id ?idt: Toll_Id;
      Photos_Taken[pt](id)
endproc (* Photos_Taken *)
```

```
process Client_Details[c] : noexit :=
   Gizmo_Details[c]({} of Gizmo_Id_Set)
   |||
   Owner_Details[c]({} of Ac_Nr_Set)
   |||
   Vehicles[c]({} of Plate_Id_Set)
where
   process Gizmo_Details[c](ids: Gizmo_Id_Set) : noexit :=
        c !create ?idg: Gizmo_Id ?pl: Plate_Id ?ac: Ac_Nr;
        (  Gizmo_Detail[c](idg, pl, ac, g_ok)
           |||
           Gizmo_Details[c](Insert(idg, ids))
        )
   where

      process Gizmo_Detail[c](id: Gizmo_Id, pl: Plate_Id,
                   ac: Ac_Nr, stat: Gizmo_Status) : noexit :=
           c !check_gizmo !id ?idt: Toll_Id;
           c !rtn_check_gizmo !id !idt !(stat eq g_ok) ;
           Gizmo_Detail[c](id, pl, ac, stat)
         []
           c !get_gizmo !id !ac;
           Gizmo_Detail[c](id, pl, ac, stat)
         []
           c !change_status !id ?st: Gizmo_Status;
           Gizmo_Detail[c](id, pl, ac, st)
      endproc (* Gizmo_Detail *)
   endproc (* Gizmo_Details[ *)

   process Owner_Details[c](ids: Ac_Nr_Set) : noexit :=
      c !create ?ac: Ac_Nr ?inf: Info;
      (  [ac isin ids] -> Owner_Details[c](ids)
         []
         [ac notin ids] ->
         (  Owner_Detail[c](ac, {} of Gizmo_Id_Set, inf)
            |||
            Owner_Details[c](Insert(ac, ids))
         )
      )
   where
      process Owner_Detail[c](ac: Ac_Nr, idgs: Gizmo_Id_Set, inf: Info)
                : noexit :=
           c !add !ac ?idg: Gizmo_Id;
           Owner_Detail[c](ac, Insert(idg, idgs), inf)
         []
           c !get_owner !ac !idgs !inf;
           Owner_Detail[c](ac, idgs, inf)
      endproc (* Owner_Detail *)
   endproc (* Owner_Details *)

   process Vehicles[c](ids: Plate_Id_Set) : noexit :=
        c !create ?pl: Plate_Id ?idg: Gizmo_Id [pl notin ids];
        (  Vehicle[c](pl, idg)
           |||
           Vehicles[c](Insert(pl, ids))
        )
   where
      process Vehicle[c](plate_nr: Plate_Id, idg: Gizmo_Id) : noexit :=
         c !get_vehicle !plate_nr !idg;
         Vehicle[c](plate_nr, idg)
      endproc (* Vehicle *)
   endproc (* Vehicles *)
endproc (* Client_Details *)

process Reg_Desk[dk, rc](id: Desk_Id) : noexit :=
     dk !give_info !id ?idg: Gizmo_Id ?pl: Plate_Id ?ac: Ac_Nr ?inf: Info;
     rc !add_info ?rcid: R_C_Id !idg !pl !ac !inf;
     rc !rtn_add_info !rcid !idg !ac ?acc_ok: Bool;
     (  [acc_ok] -> dk !get_gizmo !idg !id;
                  Reg_Desk[dk, rc](id)
      []
        [not(acc_ok)] -> dk !refuse_gizmo !idg !id;
                       Reg_Desk[dk, rc](id)
     )
   []
     dk !report_stolen !id ?idg: Gizmo_Id;
     rc !report_stolen ?rcid: R_C_Id !idg;
     Reg_Desk[dk, rc](id)
   []
     dk !cancel !id ?idg: Gizmo_Id;
     rc !cancel ?rcid: R_C_Id !idg;
     Reg_Desk[dk, rc](id)
endproc (* Reg_Desk *)

process Reg_Control[rc, bi2, c](id: R_C_Id) : noexit :=
     rc !add_info !id ?idg: Gizmo_Id ?pl: Plate_Id
        ?ac: Ac_Nr !inf;
     bi2 !check_account ?bnk: Bank_Interf_Id !ac;
     (  bi2 !rtn_check_account !bnk !ac !true;
        c !create !idg !pl !ac;
        c !create !pl !idg;
        c !create !ac !inf;
        c !add !ac !idg;
        rc !rtn_add_info !id !idg !ac !true;
```

```
          Reg_Control[rc, bi2, c](id)
        []
          bi2 !rtn_check_account !bnk !ac !false;
          rc !rtn_add_info !id !idg !ac !false;
          Reg_Control[rc, bi2, c](id)
      )
    []
      rc !cancel !id ?idg: Gizmo_Id;
      c !change_status !idg !g_cancel;
      Reg_Control[rc, bi2, c](id)
    []
      rc !report_stolen !id ?idg: Gizmo_Id;
      c !change_status !idg !g_stolen;
      Reg_Control[rc, bi2, c](id)
endproc (* Reg_Control *)

process Bank_Interface[bk, bi1, bi2, bp2](id: Bank_Interf_Id) : noexit :=
      bi2 !check_account !id ?ac: Ac_Nr;
      bk !query_account !id !ac;
      bk !rtn_query_account !id !ac ?account_ok: Bool;
      bi2 !rtn_check_account !id !ac !account_ok;
      Bank_Interface[bk, bi1, bi2, bp2](id)
    []
      bi1 !send_debit !id ?ac: Ac_Nr !some_am;
      bk !debit !id !ac;
      Bank_Interface[bk, bi1, bi2, bp2](id)
    []
      bk !close_account !id  ?ac: Ac_Nr;
      bp2 !close_account ?bp_id: Bill_P_Id !ac;
      Bank_Interface[bk, bi1, bi2, bp2](id)
endproc (* Bank_Interface *)

process Operator_Interface[p, bp1, oi, to](id: Op_Id) : noexit :=
      oi !update_prices !id !inf;
      p !update ?ptid: P_T_Id !inf;
      Operator_Interface[p, bp1, oi, to](id)
    []
      oi !monthly_debit !id;
      bp1 !debit ?bpid: Bill_P_Id !the_date;
      Operator_Interface[p, bp1, oi, to](id)
    []
      oi !new_toll !id ?idt: Toll_Id !inf;
      to !create !idt;
      (*p !update ?ptid: P_T_Id !inf;*)
      Operator_Interface[p, bp1, oi, to](id)
    []
      oi !change_toll !id ?idt: Toll_Id !inf;
```

```
     p !update ?ptid: P_T_Id !inf;
     Operator_Interface[p, bp1, oi, to](id)
endproc (* Operator_Interface *)

process Bill_Processor[c, u, bp1, bp2, bi1, pr](id: Bill_P_Id) : noexit :=
     bp1 !debit !id !the_date;
     (  Send_Bills[c, u, bi1, pr](the_date)
        >> Bill_Processor[c, u, bp1, bp2, bi1, pr](id)
     )
   []
     bp2 !close_account !id ?ac:Ac_Nr;
     c !get_owner !ac ?idgs: Gizmo_Id_Set ?inf: Info;
     (  Close_Gs[c](idgs)
        >> Bill_Processor[c, u, bp1, bp2, bi1, pr](id)
     )
   where
      process Send_Bills[c, u, bi, pr](d: Date) : exit :=
           u !get_usage ?us_id: Usage_List_Id
             ?idg: Gizmo_Id !some_am !d ?idt: Toll_Id;
           c !get_gizmo !idg ?ac: Ac_Nr;
           c !get_owner !ac ?idgs: Gizmo_Id_Set !inf;
           c !get_vehicle ?pl: Plate_Id !idg;
           bi !send_debit ?b_id: Bank_Interf_Id !ac !some_am;
           pr !print_receipt ?p_id: Pr_Id
              !inf !pl !some_am !ac !idt;
           Send_Bills[c, u, bi, pr](d)
        []
           (*  When all bills sent *)
           exit
      endproc (* Send_Bills *)

      process Close_Gs[c](idgs: Gizmo_Id_Set) : exit :=
           c !change_status ?idg: Gizmo_Id !g_closed [idg isin idgs];
           Close_Gs[c](Remove(idg, idgs))
         []
           [idgs eq {}] -> exit
      endproc (* Close_Gs *)
endproc (* Bill_Processor *)

process Printer[pr, prin](id: Pr_Id) : noexit :=
     pr !print_receipt !id ?inf: Info ?pl: Plate_Id
         ?am: Cost ?ac: Ac_Nr ?idt: Toll_Id;
     prin !print_receipt !id !inf !pl !am !ac !idt;
     Printer[pr, prin](id)
endproc (* Printer *)
endspec
```