Department of Computing Science and Mathematics
University of Stirling

ROOA with SDL

Robert G. Clark and Ana M. D. Moreira

Technical Report CSM-147

November 1998

Department of Computing Science and Mathematics
University of Stirling

ROOA with SDL

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-147

November 1998

Contents

Abstract

Our interest is in making object-oriented analysis a more rigorous process. As we wish to create a practical and usable method, we do not propose a new specification language. Instead, we base our work on standard formal description techniques which provide executable specifications and which are supported by validation and simulation tools so that prototyping can be used validate a specification against the requirements. Also, the first steps in our rigorous object-oriented analysis method are based on widely used informal methods such as OMT and OOSE. In this report, we show how SDL can be applied during object-oriented analysis to produce a formal object-oriented requirements specification. SDL is a standard formal description technique that is normally used in the design phase of systems development.

Building a formal specification from informal requirements is difficult. To simplify this, our method builds two formal models: a user-centred model and a system-centred model. The user-centred model is based on scenarios and specifies the external behaviour of a system from the viewpoint of the environment. It is used to support the construction of the system-centred model which is the formal object-oriented requirements specification.

We represent both models in the same formal language (in this case SDL, but it could be another formal description technique such as LOTOS). From the external point of view, the two models should exhibit the same behaviour. We validate the user-centred model against the requirements. Validation of the system-centred model can then be achieved by verifying that it provides the behaviour expected by the user-centred model.

Chapter 1

Introduction

An essential first step in the creation of any system is to ensure that the requirements are properly understood, are complete and do not contain inherent contradictions. Unfortunately, informal requirements are usually inconsistent, ambiguous and incomplete. However, by forcing the analyst to be precise, the process of creating a formal specification can show up inconsistencies, omissions and ambiguities sufficiently early in the development process so that their correction is relatively inexpensive. Creating a formal requirements specification is therefore of major value even when the rest of the development is not a formal process.

To support this approach, we have been investigating how formal description techniques (FDTs) can be integrated into object-oriented analysis. The goal is to obtain a formal object-oriented requirements specification while we are still dealing with the problem domain. In this report, we demonstrate the suitability of using SDL as the FDT [1]. This differs from other work on SDL which has concentrated on the design phase of the software life cycle.

Our previous work in this area used LOTOS [2, 3] and the result is the Rigorous Object Oriented Analysis (ROOA) method [4, 5]. Here we use ROOA as the vehicle to demonstrate the suitability of using SDL in the analysis phase. Our interest has been to produce a rigorous process that can be used by systems engineers as part of a development process. Our goal is to make FDTs more acceptable to those systems engineers who currently have little interest in formal methods. That is why we only use standard FDTs, such as SDL or LOTOS, which produce executable specifications and which are widely supported by prototyping tools.

A major problem is the wide gap between the requirements and a formal object-oriented specification. As informal requirements are normally expressed in terms of user needs, ROOA bridges the gap by first creating a formal user-centred model which specifies the observable behaviour as a set of scenarios describing how external users expect to use the system [6, 7]. The user-centred model is validated with respect to the requirements and is then used in the construction and validation of an object-oriented requirements specification, i.e. a system-centred model. We use the same formal language in both models.

This report concentrates on how the user-centred and the system-centred models can be represented in SDL and how SDL tools can be used to guarantee their internal consistency, to support their validation against the requirements and to demonstrate that the two models are equivalent. We also discuss our previous experiences with LOTOS and compare the representation of object-oriented concepts in LOTOS and SDL.

This report consists of six chapters and two appendices. Chapter 2 discusses the need for a rigorous method that builds a formal specification from informal requirements. It then gives a brief overview of the ROOA method, analysing its strengths and weaknesses. Chapter 3 presents a mapping of the most important object-oriented concepts into LOTOS and SDL, describing how each of these concepts can be modelled in the two specification languages. Chapter 4 gives a detailed description of the ROOA method by means of a case study of a road pricing system and describes how SDL can be used to represent both the user-centred and the system-centred models. Chapter 5 compares the advantages and disadvantages of using LOTOS and SDL within ROOA. Chapter 6 presents our main conclusions. Finally, Appendix A presents the complete SDL specification of the user-centred model and Appendix B does the same for the system-centred model.

Chapter 2

A rigorous process

2.1 Introduction

We have been investigating the advantages of adding formality to the object-oriented analysis process. Existing object-oriented analysis methods suffer from two fundamental problems: weak integration between the static and dynamic models and lack of formality. We have created the rigorous object-oriented analysis (ROOA) method which addresses both these problems.

It is impossible for an analysis method to be completely formal as the starting point is a set of informal requirements with probable inconsistencies, ambiguities and omissions. That is why we have produced a rigorous method.

In this chapter we discuss the needs for a rigorous method and the benefits we can gain from using formality.

2.2 Reasons for a rigorous analysis process

Given that, in order to be useful, software must be correct and reliable, it is clear that we only gain by using formal techniques. Our work is concerned with enhancing the object-oriented analysis process. We know that the object-oriented analysis methods that are currently used suffer from several deficiencies, namely:

- they are weak at representing dynamic views;
- they do not integrate the static and dynamic properties;
- they lack formality in the method and in the models used;
- they lack supporting tools to check the semantics of the models.

The use of a formal description techniques, such as SDL or LOTOS, during the analysis phase can overcome these deficiencies. Both those languages are good at describing the behaviour of a system (after all that is what they were designed for). The ROOA method guides us on how to write formal specifications, in SDL or LOTOS, which integrate the static, dynamic and functional properties of a system. Formal description techniques have a precise syntax and semantics, and therefore the resulting ROOA model is formal; also, they have supporting tools, such as syntax and static semantic checkers, validators and simulators, which can be used to check that the final model is internally self-sufficient and to validate it against the requirements.

The validation of a specification is a difficult task. In ROOA, we have devised a process to simplify the validation task by dividing that task into two. We first create a user-centred specification which is then used to support the creation and validation of the object-oriented specification. Both specifications are executable and so prototyping can be used in the validation process.

ROOA is especially useful for specifying distributed, embedded and concurrent system. As their implementations are very difficult (if not impossible) to test on real hardware, being able to validate their specifications is very important.

2.3 What about formal methods?

It is possible to provide a formal design process if a formal requirements specification already exists. However, the analysis process starts from a set of informal requirements and may include the capture of the requirements, involving discussions with clients or the users of the system. Thus, an analysis method cannot provide a formal process.

Nevertheless, we can reduce the distance between informal requirements and formal specifications by providing a process which creates formal specifications very early in the development cycle. Shortening this distance is the main goal of our work, and the primary result of the ROOA method. The final specification provided by ROOA is a requirements specification which can then be used as the starting point of a formal design process.

The following are the characteristics that make ROOA a rigorous method:

- ROOA produces a formal requirements specification, expressed in a specification language which is formal and has a mathematical semantics.
- ROOA uses prototyping to validate the formal specification against the requirements.
- ROOA proposes rules to be followed when constructing the specifications.
- ROOA provides a systematic development process, by offering a set of well-defined steps, heuristics and mappings from object-oriented concepts into SDL or LOTOS.
- ROOA builds on well-established methods and tools.

Therefore, ROOA is a *rigorous method*, as it is less formal than a formal method should be. But formality is not a goal in itself; it is only useful as a means towards more effective software development [8].

2.4 Strengths of the ROOA method

2.4.1 The ROOA Process

The first major strength of ROOA is that it combines two important software development techniques: object-oriented analysis and formal methods. The specification resulting from an application of ROOA acts as an initial formal requirements specification. Creating a formal requirements specification is useful, even when we do not follow a formal development trajectory. It gives us the opportunity to reason about the requirements early, helping us to get them right and therefore to understand the user needs.

ROOA builds on the work already available for object-oriented analysis methods. By using an executable specification language such as SDL or LOTOS, ROOA produces a prototype where the validators and simulators may be used to validate the requirements.

Our work with ROOA started using LOTOS. We decided to show how difficult (or easy) it would be to use a different formal description technique. The result of our work is that ROOA was easily adapted to use SDL and we are convinced that it can easily embrace other specification languages.

ROOA proposes a set of tasks to produce an initial formal requirements specification systematically. During the method we give rules to be followed, so that tools can automate part of the ROOA process.

ROOA offers a mapping on how to model object-oriented analysis concepts in either LOTOS or SDL. The main concepts we deal with are: class (concrete and abstract), services, attributes, objects, object identity, message passing, aggregates, subsystems and inheritance.

2.4.2 Assessing the resulting specifications

While assessing the formal specifications produced by ROOA, two main questions come to mind:

- Do the specifications reflect the analysis model being created?
- Are these specifications readable?

Both SDL and LOTOS have a very rich set of constructs which allow us to express many different ideas.

The difficulties of understanding formal specifications are a problem shared by most, if not all, formal specification languages. We believe that the problem in writing specifications is in producing a good model and in getting the right level of abstraction. When writing programs we use a low level of abstraction, as compared with what is usual in analysis, and we spend much time in so-called "implementation details". When writing specifications we use a higher level of abstraction where the details we have to consider are of a different kind. Hall observes that many people find it difficult to write specifications because it is difficult for them to get away from the detailed descriptions they are used to when writing programs [9].

In order to make our specifications less difficult to read and to understand, we propose an object-oriented style together with meaningful names for processes, events, abstract data types, operations, parameters and gate names. Meaningful names do not always come to mind immediately; however, we cannot overlook this point, as requirements specifications are used for communication between users, analysts and designers. Moreover, the ROOA-style maintains a correspondence between the concepts and names used from a class diagram through to the formal specification.

2.5 Overview of ROOA

ROOA is an iterative and incremental process. It creates a formal and executable specification from informal requirements. As this is difficult, ROOA proposes that it should be done in two steps. We first build a user-centred model, which is closer to the informal requirements, and then, based on this, we build a system-centred model which is the formal object-oriented requirements specification.

Figure 2.1 gives a simple schematic view of how the user-centred and the system-centred models relate.

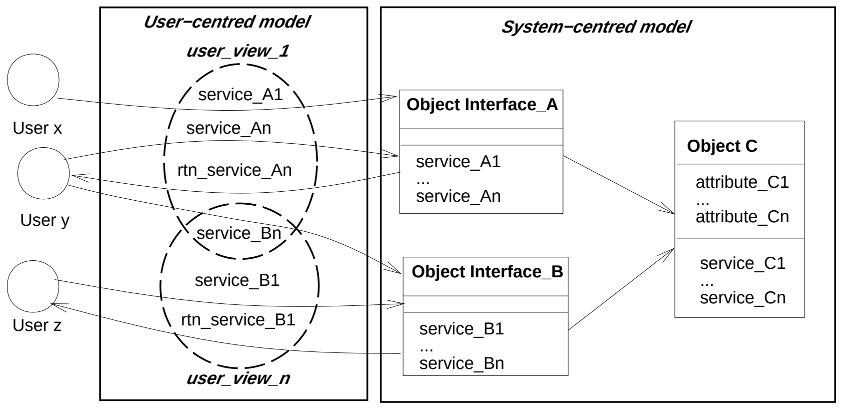


Figure 2.1: Relationship between the user-centred and the system-centred models

The user-centred model is a set of *user views* where each user view provides scenarios of how the system is to be used. The users may be humans, hardware devices or software systems. They belong to the external environment, and are not therefore part of the model we want to build. A user view shows interactions between a user and the system; it 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 user view, while several different people can play a similar role and therefore take part in different instances of the same user view.

We deal initially with views involving a single user. We represent these single-user views informally as a directed graph and then formalise them in SDL or LOTOS. Later we compose these single-user views to form multi-user views. The user-centred model is the set of multi-user views and it is validated with respect to the requirements using simulation and validation tools.

The user-centred model is then used as a stepping stone in the construction of the system-centred model. 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. Both models are represented in the same formal language and are simulated with an equivalent set of scenarios. Instead of the difficult and informal task of validating the system-centred model with respect to the requirements, we can verify that it provides the behaviour expected by the user-centred model. Figure 2.2 shows the main tasks of ROOA.

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 the specifiers/analysts and the clients a better understanding of the system. The corresponding part of the system-centred model can then be created. Prototyping of both the user-centred and system-centred models gives feedback to

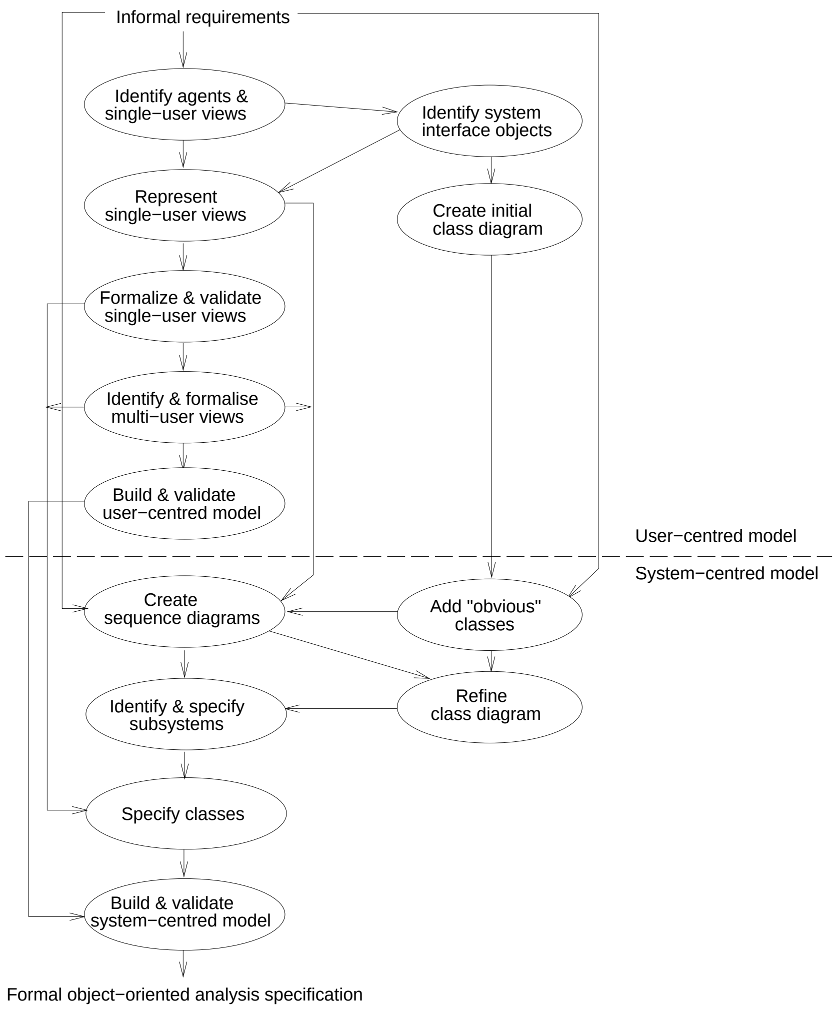

Informal requirements
Identify agents & single−user views
Identify system interface objects
Represent single−user views
Create initial class diagram
Formalize & validate single−user views
Identify & formalise multi−user views
Build & validate user−centred model
User−centred model
System−centred model
Create sequence diagrams
Add "obvious" classes
Identify & specify subsystems
Refine class diagram
Specify classes
Build & validate system−centred model
Formal object−oriented analysis specification

Figure 2.2: The ROOA process

the requirements capture process, enabling further user views to be formulated and existing ones refined. In this way both the user-centred and system-centred models can be developed incrementally.

2.6 Related work

The creation of a formal user-centred model based on scenarios has also been proposed by Hsia *et al.*, Glinz and Somé *et al.* [10, 11, 12]. However, they do not integrate their user-centred model with a formal system-centred model.

In the requirements analysis phase of the OOSE method, Jacobson proposes the construction of a requirements model which corresponds, in some respects, to our user-centred model although his model is not formal [13]. The OOSE requirements model is composed of a use case model and a simplified object model called a domain object model.

Several methods combine SDL with an object-oriented method [14, 15, 16, 17, 18]. However, while we integrate SDL within an analysis process, first to represent the user-centred model and then to represent the object-oriented specification, these other approaches do not introduce SDL until the design phase. In the analysis phase, they use OMT [19] and represent the use case model by message sequence charts (MSCs) [20].

Examples are the SOMT [14] and SISU [16, 21] methods. In the system design phase of SOMT, SDL is used to define the system, block and process structure. Then, in the object design phase, individual objects are modelled as SDL processes or blocks and the testing and validation process begins. SISU deals with reactive systems where modelling the environment as part of the specification is standard practice. SISU introduces an enhanced object notation to provide a domain object model and its interaction with the environment, but SDL process definitions and prototyping are not used until the design phase.

There is a similarity between our approach and work on specification via viewpoints [22, 23]. As with user 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. User 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.

Chapter 3

Object-oriented concepts in SDL and LOTOS

3.1 Introduction

SDL and LOTOS are standard formal description techniques [1, 3]. Although they are primarily used in the specification of communication protocols and standards, they are general purpose specification languages. There have been several versions of SDL since its first release in 1976. An important change was in 1992 when object-oriented features were included in the language.

SDL and LOTOS have a data typing part, based on abstract data types (ADTs), and a process part. Their focus is on the specification of behaviour, especially concurrent behaviour. The process part of LOTOS is based on process algebra while in SDL it is based on communicating extended finite state machines. Both languages specify a system in terms of its interaction with the environment. In LOTOS, behaviour is defined by processes synchronizing with each other and with the environment on events. A major difference in SDL is that communication between processes and with the environment is in terms of asynchronous signals.

Entities in the real world exist in parallel and this should be mirrored when requirements are modelled in a specification language even when the eventual implementation is to be sequential. ROOA achieves this by specifying a class as a process definition, an object as a process instance and by modelling a problem as a set of communicating concurrent objects. LOTOS and SDL support this approach.

3.2 Classes and objects

We use control objects, entity objects and interface objects in a specification. We treat control objects as transient entities which are created dynamically to carry out some function and then terminate.

An object's attributes are specified by one or more ADTs and are given, in LOTOS, as the parameters of a process [1] and, in SDL, either as the parameters of a process or as local variables. Each object has a distinct identity, represented by an object identifier. In LOTOS,

[1] We use the term *process* as shorthand for *process instance.*

each process has a special attribute, defined as an ADT, to represent its object identifier. In SDL, each process is allocated a process identifier (`PId`) when it is created. The `PId` acts as the object identifier.

An SDL process is specified as an extended finite state machine and so the behaviour of a class is defined by means of transitions between states. In a given state, a process is willing to accept one of a set of signals from its environment, i.e. accept one of a set of service calls. When an acceptable signal is received it initiates a transition, which leads to a new state. During a transition, attributes can be modified and signals can be sent to other processes (i.e. we can call a service offered by another object). Multiple instances of a class can exist in a given context and this is represented by having multiple instances of a process type.

3.3 Message passing

Message passing between two objects is represented in LOTOS by two processes synchronizing on an event, while in SDL it is modelled by a client process sending a signal to a server process. As signal passing in SDL is asynchronous, the client is not suspended if the server is not ready to receive the signal. Instead the signal is queued. Each SDL process instance maintains its own input queue.

A second signal is required when information is to be returned from a server. If we make the restriction that the client must wait to receive the return signal, and may not take part in any intermediate activity, we have the equivalent of synchronous communication. We model this by having two signals, `signal name` and `rtn signal name`. When no answer is required, we can take the view that it does not matter if the server carries out the service immediately or at some future time. As received signals are queued by the server, the order is preserved.

If the server process is in a state whose set of acceptable input signals does not include the signal at the front of its input queue, the signal leads to a null transition and so is lost, unless it is explicitly saved. A server may have several clients and the save mechanism can be used to save service requests until the server has completed the series of communications required to satisfy the current request.

When multiple process instances are possible destinations of a signal, the destination process may be explicitly named by means of its `PId`. If no `PId` is specified, we have implicit addressing. We can specify a destination process implicitly by referring to the appropriate channel or signal route.

3.4 Inheritance

A subclass inherits the behaviour of its superclass and can add new transitions and gates or redefine existing transitions. It is only the new or modified transitions and gates that are given in the definition of the subclass. To redefine an SDL transition (i.e. a service in a class diagram), we prefix the input of that transition, in the superclass, with the keyword `virtual` and in the subclass we redefine the transition and prefix the input with the keyword `redefined`.

3.5 Relationships and aggregates

Relationships are modelled as attributes holding the object identifier of the other object involved in the relationship. We can model the relationship either in one direction or in both. Multiple cardinality is modelled using sets of object identifiers. We model an aggregate as a coordinating object together with one or more component objects. In SDL, the coordinating and component objects are represented by processes in a block specification. In LOTOS, each component is represented by a process within the process specifying the aggregate.

The interesting situation in SDL is when there are multiple instances of the aggregate. We define this as a block which contains multiple instances of the processes representing the coordinating and component objects. A process may dynamically create another process, but only when it is in the same block. When the number of aggregate instances is fixed, the required number of coordinating objects is created statically. Each coordinating object then dynamically creates instances of its components and notes their `PIds` so that it can communicate with them. When aggregate instances may be created dynamically, the block must also contain an administration process to create the coordinating objects.

An alternative approach is to model each aggregate, including its components, as a block and to have multiple block instances [14, 16]. However, we regard an aggregate as an object and, as a block does not have identity or local variables, it cannot represent an object. Also, block instances cannot be created dynamically and so this approach cannot deal with the dynamic creation of aggregates. We therefore use SDL blocks purely as a static structuring mechanism and always represent objects by processes and classes by process types.

3.6 Direct language support

Although LOTOS can be used in an object-oriented style [5], a major advantage of using SDL is that it has object-oriented features such as inheritance built in. Another advantage of SDL is that it has specific language constructs, packages and blocks, to deal with the structuring of large systems into manageable and reusable units.

There is a graphical form of LOTOS [24], but the textual form is usually used. The opposite is the case with SDL where specifications are usually represented graphically. This means that tool support is essential for the creation and representation, as well as for the validation, of an SDL specification. The tool used in our experiments is Telelogic SDT [25]. Both SDL and LOTOS lead to executable specifications and are supported by simulation and validation tools which can be used in the process of validating a specification against the requirements.

As SDL has object-oriented features built in, toolsets such as SDT support an object-oriented design method, and this results in better support for ROOA than with LOTOS tools. SDT allows links to be set up between entities in the data dictionary, class diagram and SDL diagrams. This supports both forward and backward traceability. Forward traceability is supported as information in one diagram can be used to help create subsequent diagrams and conformance of the diagrams can be checked. In an iterative method it is important that the diagrams remain consistent as changes are made and that we can trace back to the origin of any inconsistency or error.

Chapter 4

SDL in Object-Oriented Analysis

4.1 Case study

Our examples are taken from the following case study.

> In a road traffic pricing system, drivers of authorized vehicles can be charged at toll gates without having to stop. Vehicle owners register and then install a device (a gizmo) in their vehicle. The system holds information about authorized vehicles including a bank account from where automatic debits are done monthly. Owners may cancel a gizmo or inform the system if the gizmo is stolen.
>
> The gizmos are read by sensors installed in special lanes at toll gates. Different types of vehicle pay different rates. When an authorized vehicle passes through a special lane, it gets a green light and the amount being debited is displayed. If an unauthorized vehicle passes through the lane, it gets a yellow light and a camera photographs its plate number. (Later, the owner will be fined.) There are two kinds of special lane: all vehicles of the same type pay the same amount at a toll bridge while, on a motorway, the amount to be paid depends on the distance travelled.

Complete specifications in SDL of both the user-centred and system-centred models are given in the appendices. We previously applied ROOA to this problem and the resulting specification is available in LOTOS [26].

4.2 Build the user-centred model

4.2.1 Identify and represent user views

The first question to ask ourselves is: what interacts with the system? The first user we may think of is the gizmo installed in a vehicle. A vehicle uses a toll gate and has a driver. Another user is the owner of the vehicle. He or she has to buy a gizmo and be registered before their vehicle can use the special lanes. Other users are the system manager and the bank. The bank behaves differently from the other users. While the bank offers services to the system, the others require services from the system.

The second question to answer is: how does each user view the system? In order to answer this, we imagine ourselves to be the user and describe what we see when we use the system.

Our focus is on modelling behaviour as seen by users, not on modelling the users themselves. Each behaviour must be sufficiently simple so that it is easily understood, but not so simple that we are overwhelmed by a mass of trivial views. Consider a vehicle, its driver and its gizmo pasing through a toll gate. We regard this as a single behaviour. Hence, instead of dealing with vehicle, gizmo and driver as separate users with separate views, we regard the composite vehicle-driver-gizmo to be a single user which we refer to as *vehicle*.

The vehicle user uses the system in different ways: one where it always pays the same amount, the *Single-Point-Vehicle view*, one where it enters a motorway, the *Enter-Motorway-Vehicle view*, and one where it leaves a motorway and the amount paid depends on the distance travelled, the *Exit-Motorway-Vehicle view*.

As part of the identification of user views, we identify the entities (interface objects) through which they interact with the system. For example, the vehicle views use a toll gate. They see a light turning green or yellow, see the amount shown in the display, know that there must be a sensor which identifies their gizmo, and also know that if the light turns yellow a photograph of their plate number will be taken. Therefore, a toll gate is an object which has four components: a sensor, a light, a camera and a display.

As there are three different vehicle user views, we decide that there are three different types of toll gate and that the display is only used when we are either leaving the motorway or when we pass through a single toll gate.

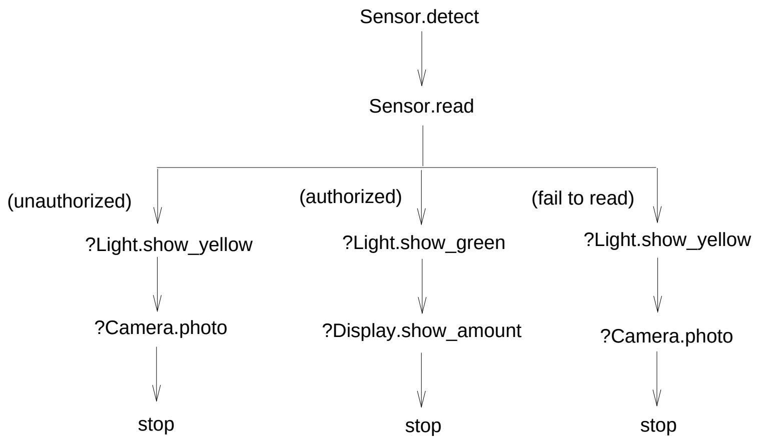


Figure 4.1: Graph representing the Single-Point-Vehicle view

Alternative behaviours within a user view are represented in a tree-like directed graph. Figure 4.1 shows the graph for the Single-Point-Vehicle view. Each path through the graph represents a sequence of events between a user and the system-centred model and is referred to as a user scenario. Events in a user view either represent signals sent by the user (e.g. `detect`) or they correspond to signals being sent by the syatem and being received by a user (e.g. `show green` and `show amount`). In the graph, signals sent by the system are preceded by a '?'. During the creation of a user view, we determine the interface objects involved in each event. These objects belong to the interface classes of the system-centred model and each node in the graph has the structure:

$\langle interface\ class\rangle.\langle event\rangle.$

A class diagram in the system-centred model is created in conjunction with the user views.

Figure 4.2 depicts an initial class diagram showing the interface classes used in the vehicle user views. As the user views are developed, new interface classes are identified and added to the class diagram. The creation of the class diagram helps ensure that different user views use identical names for classes and services which are essentially the same.

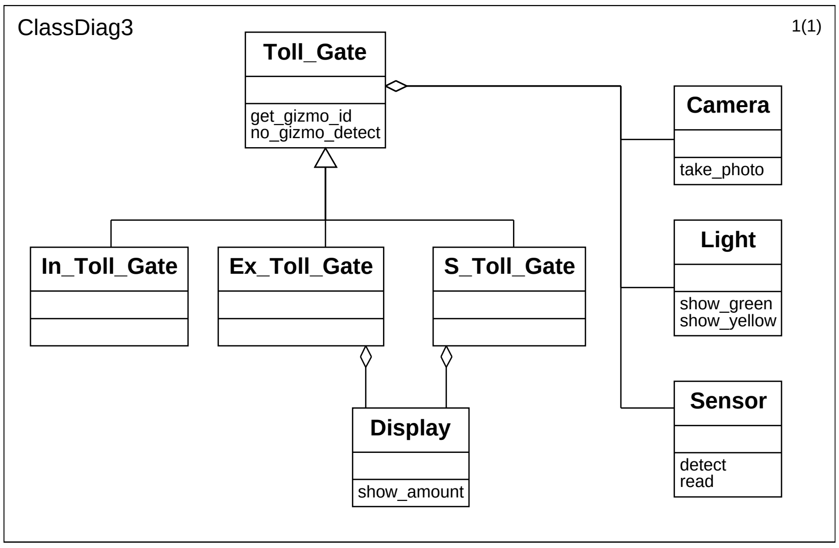


Figure 4.2: Initial class diagram

4.2.2 Formalise single-user views

It is straightforward to specify the graph representing a single-user view as an SDL process type. This process includes signal signatures and is parameterised to deal with the cases of a valid gizmo, an invalid gizmo and failure to read a gizmo. We use a separate SDL gate for each interface class with which a user view interacts. Figure 4.3 shows the SDL definition of the Single-Point-Vehicle view.

Detection of a vehicle is represented by a synchronous communication between the view process and a sensor. This is accomplished by defining two signals: a `detect` signal (to detect that there is a vehicle using the toll) and a `rtn detect` signal (to wait for the answer). Then the view process sends a `read` signal to the sensor with the gizmo identifier as a parameter or indicates failure with a zero value. If the gizmo is valid, the view waits for `show green` and `show_amount` signals, otherwise (if it fails to read or if it is an unauthorized gizmo) it waits for `show_yellow` and `photo` signals.

The graph in Figure 4.1 actually overconstrains the system by expecting a response from `Light` before the response from either `Display` or `Camera`. As the system may send a `show_amount` signal before `show green` or `photo` before `show yellow`, the SDL specification allows these signals to be saved and dealt with after it has dealt with the signal from `Light`. The purpose of the `s_term` signal is explained in 4.2.3.

Although an SDL specification only interacts with its environment (which is represented by the special PId `env`), we construct the SDL definition of a view as if signals were being sent to particular system interface objects, in this case to a particular sensor. The system-centred model and the users play the role of the environment of our SDL user-centred model.

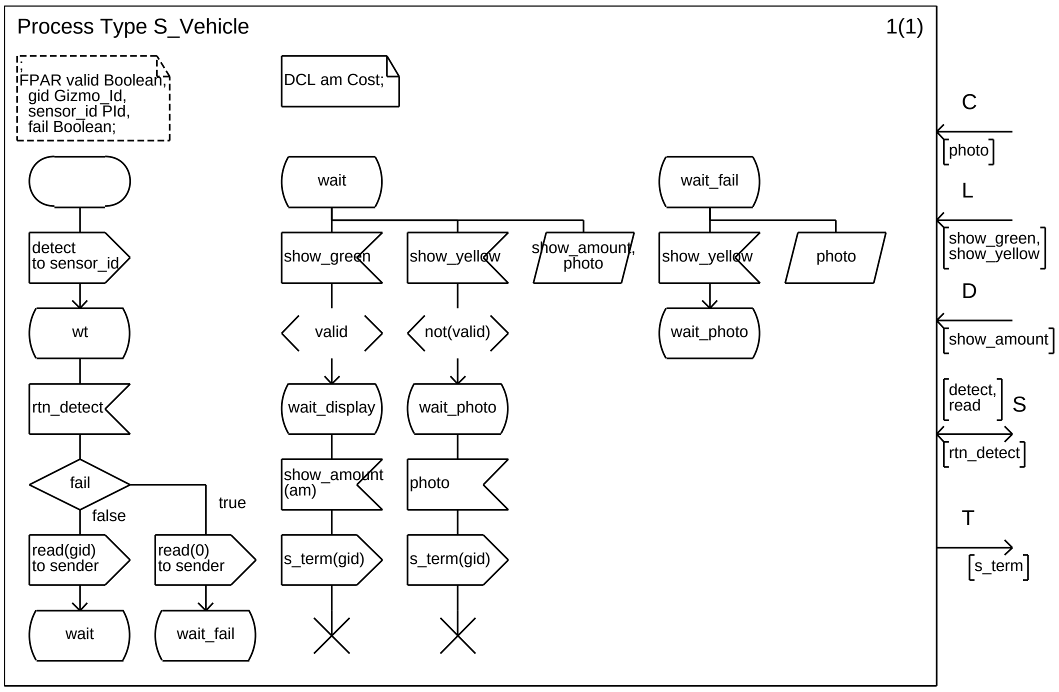


Figure 4.3: SDL Single-Point-Vehicle view

4.2.3 Validate single-user views

Although single-user views are independent of one another, we have found it useful to introduce a `Coordinator` process to the SDL specification to coordinate the execution of the user views. The specification is then represented as an SDL block containing the `Coordinator` process together with the processes representing each single-user view. The specification is shown in Figure 4.4 and the `Coordinator` process in Figure 4.5.

After checking for syntax and static semantics errors, the SDT toolset enables an SDL specification to be translated into an executable form so that it can be manipulated by simulation and validation tools. During interactive simulation in SDL, we initiate scenarios by sending a `drive` signal to `Coordinator` from the environment with the signal parameters determining which particular user-view process is to be created and executed to demonstrate that it displays the expected behaviour. The `drive` signal models a user deciding to initiate some interaction with the system. On completion of its execution, a user-view process sends a terminate signal (represented as ⟨*prefix*⟩ `term`) to `Coordinator`. An example is signal `s term` in Figure 4.3.

Interactive simulation is used to explore the behaviour offered by each single-user view. In addition, we use the validator to show that the specification is internally consistent and does not contain deadlocks. The validator explores the state space of a system. Although this cannot be complete for all but the smallest systems, by using *bit state hashing* the SDT validator allows a large number of states to be examined. A hash table is used to hold the system states which have already been reached so that the same state is not used repeatedly. SDT also offers random walks.

Inconsistencies in the requirements and logical errors in the specification will often result

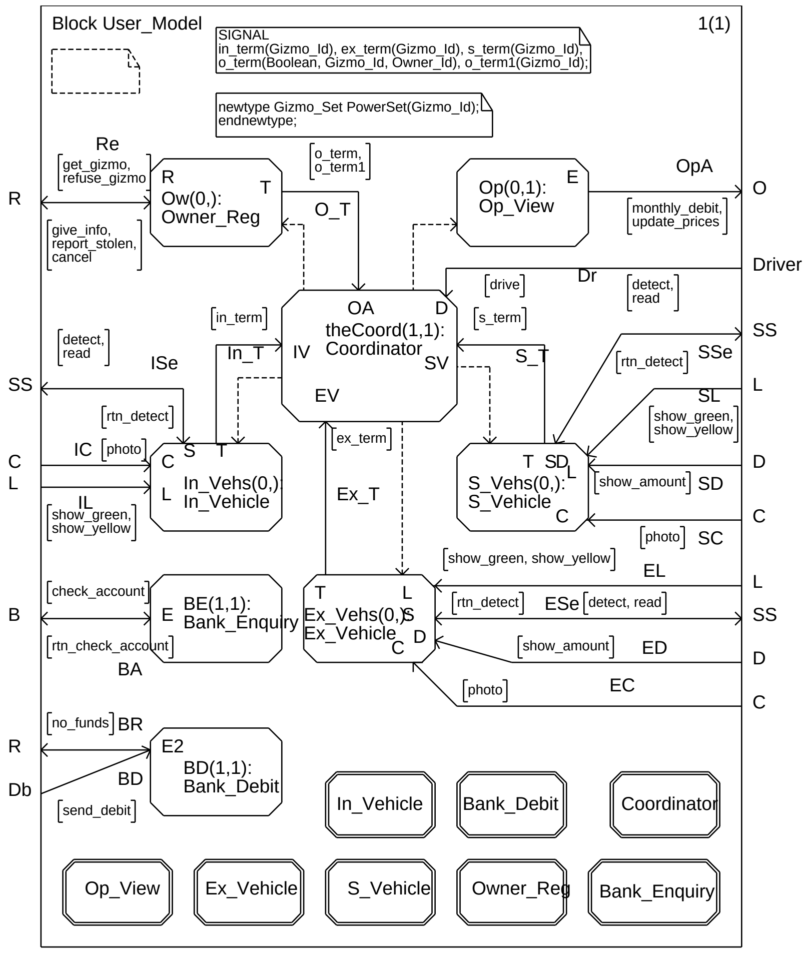


Figure 4.4: User model with single-user views

in reports produced by the tools that, for example, a signal has no acceptable target or that it has been implicitly consumed. When such conditions arise, the validator generates an MSC to help detect the cause. Transitions in the specification which have not been executed are also reported. Except where they correspond to virtual transitions in an abstract superclass, a transition which cannot be reached indicates a logical error in the specification.

The validator automatically generates signals from the environment. We can provide the validator with a list of parameter values which may be associated with these signals. We can also associate priorities with the different kinds of signal. As we are concerned here with executing individual user views, we set the priority of internal signals to be higher than signals generated by the environment so that the emphasis is on completing the execution of a user view rather than creating new views by the generation of `drive` signals.

4.2.4 Identify and formalise multi-user views

In general, the interaction of a user view with the system involves other user views. For example, the user view where an owner attempts to register a vehicle involves another user view where a bank checks the owner's account. The graphs representing these two single-user

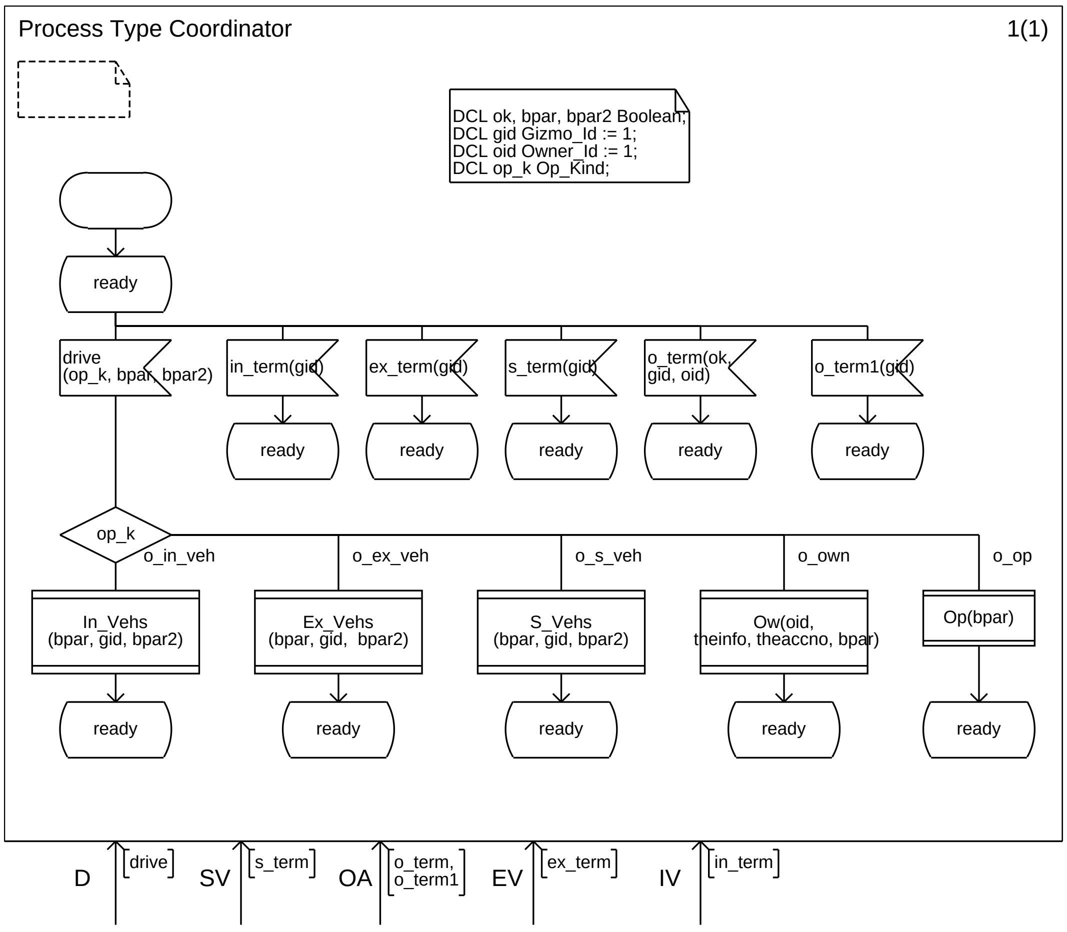


Figure 4.5: Coordinator process

views are shown in Figures 4.6 and 4.7.

The SDL specifications of these two single-user views are composed to form a *multi-user view.* The advantage of dealing initially with single-user views is that they are simpler and give us a place to start when handling large and complex requirements. They provide a separation of concerns; at any one time we focus on a single user'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.

The interaction between user views in our case study is shown in the use case model of Figure 4.8. It should be noted that complex multi-user views may be composed from simpler multi-user views as well as from single-user views. The leaves of the tree are single-user views while all the other nodes represent multi-user views.

It is important that the representation of the constituent user views is clearly visible within the representation of a multi-user view. There are three main ways in which user views interact:

- User views `A` and `B` take place in sequence. This is modelled by the SDL process representing `A` creating the process representing `B` just before `A` terminates.
- The behaviour described by a user view is dependent on a condition set by another user

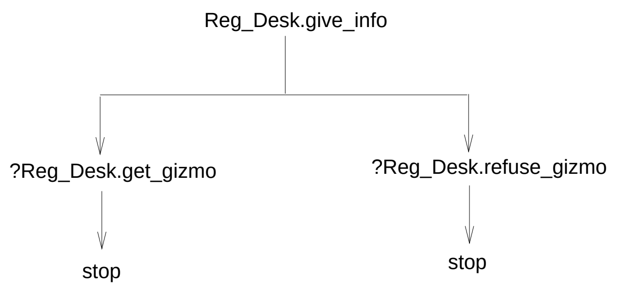


Figure 4.6: Owner registration user view

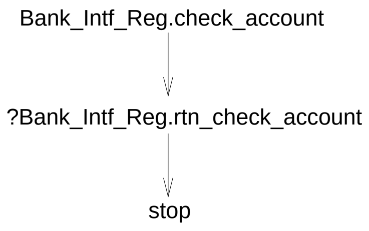


Figure 4.7: Bank enquiry user view

view. For example, we cannot have a vehicle view with a valid gizmo until that gizmo has been registered. To specify this interaction, the `Coordinator` process maintains sets of available and invalid gizmo identifiers which determine the parameters used in the creation of a vehicle view process.

- Two user views interleave as in the case of the bank enquiry and owner registration views. Interleaving is achieved by replacing an SDL process defining a user view by an SDL service definition. The process defining the multi-user view is then composed from the services defining each of the constituent views. As SDL services and processes are both defined using extended finite state machines, the constituent user views are still clearly distinguishable within the composite user view (see Figure 4.9).

The transition from an SDL specification containing single-user views to a user-centred model composed of multi-user views proceeds incrementally. The form of the specification is the same although now the `Coordinator` process maintains sets to record the overall state.

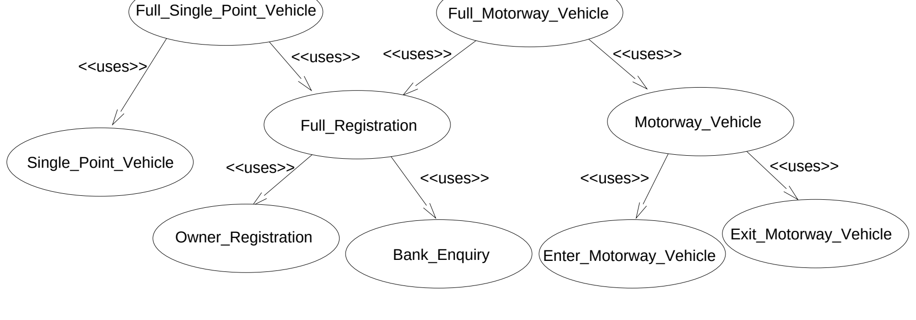


Figure 4.8: Use case model

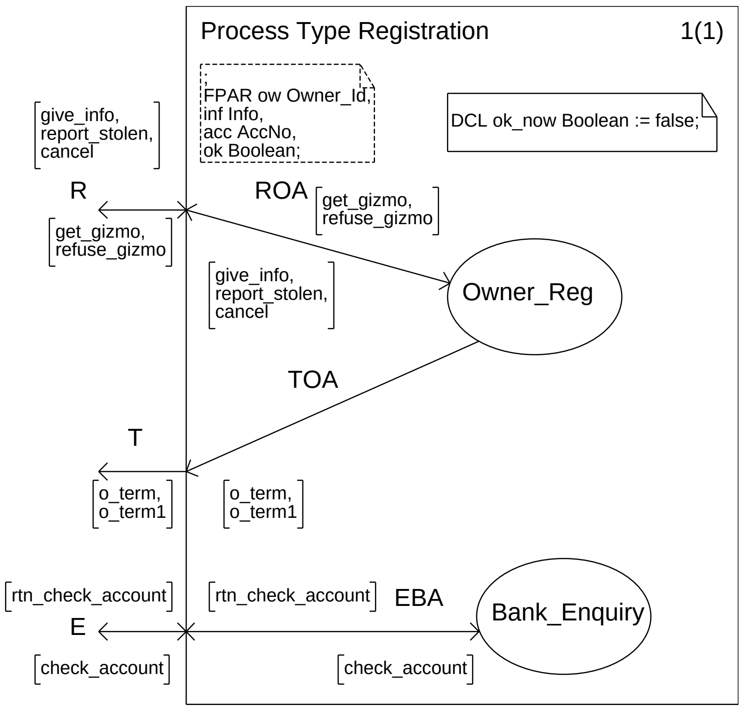


Figure 4.9: A multi-user view composed of two single-user views

These sets are used to determine the values passed during the creation of user-view processes and are updated when the processes are created and when they terminate. The complete user-centred model is shown in Appendix A.

4.2.5 Validate the user-centred model

We first simulate the SDL user-centred model interactively. Interactive simulation is particularly important during the creation of the user-centred model so that we can explore particular execution paths to increase our confidence that the model has the expected behaviour. We then use random walks in the validator to demonstrate that the specification is internally consistent and that all transitions are reached. Now that we have multi-user views, the behaviour offered is much more complex than was the case where we were only considering individual single user-views. We must, for example, now check the situation where we have multiple simultaneous instances of each user view.

Interactive simulation is no longer sufficient. Hence, to validate the user-centred model, we define user scenarios as MSCs which are then used to constrain the behaviour considered by the validator. As the MSCs are representing user scenarios, they only show interactions between the user-centred model and its environment (which here corresponds to the users and the system-centred model). The MSC does not show the interactions between the SDL processes which make up the user-centred model although, of course, the simulator will execute these process interactions in order to satisfy the external behaviour required by the MSC. The MSC describing the registration of a vehicle is shown in Figure 4.10. An owner initiates the user view that results in the registration of a gizmo by sending a `drive` signal with parameters

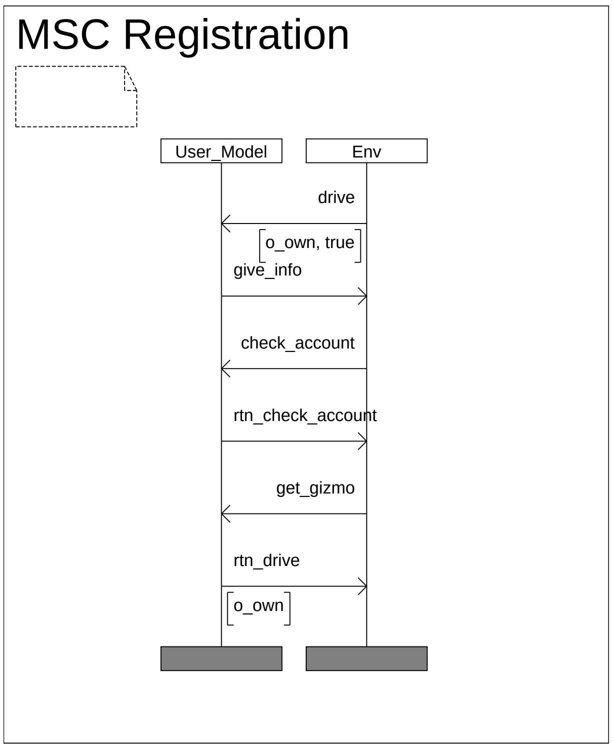


Figure 4.10: Registration top-level MSC

`o_own` and `true`.

We can either use interactive navigation under the control of an MSC or let the validator automatically check that the specification can support a particular behaviour. The SDT validator reports a verified MSC when an SDL specification offers a possible path through the system that is consistent with the MSC. As a specification does not just specify correct behaviour, but must also rule out incorrect one, we also construct MSCs whose behaviour must be rejected by the specification.

Initially, our aim is to understand and to model the given requirements from the point of view of individual users and so it is to be expected that combining single-user views will lead to inconsistencies. It is only at this stage that these inconsistencies need to be resolved. Their resolution may require us to go back to the users for further or revised information.

We cannot guarantee that all inconsistencies will be detected at this stage. Further inconsistencies may be detected and have to be resolved during the construction of the system-centred model. ROOA therefore allows the modelling of inconsistent requirements during the early stages and provides a framework in which the inconsistencies can be detected and therefore resolved.

4.3 Build the system-centred model

4.3.1 Create sequence diagrams

The starting point is the informal requirements, the user views and the initial class diagram. We identify "obvious" classes from the informal requirements and add them to the class

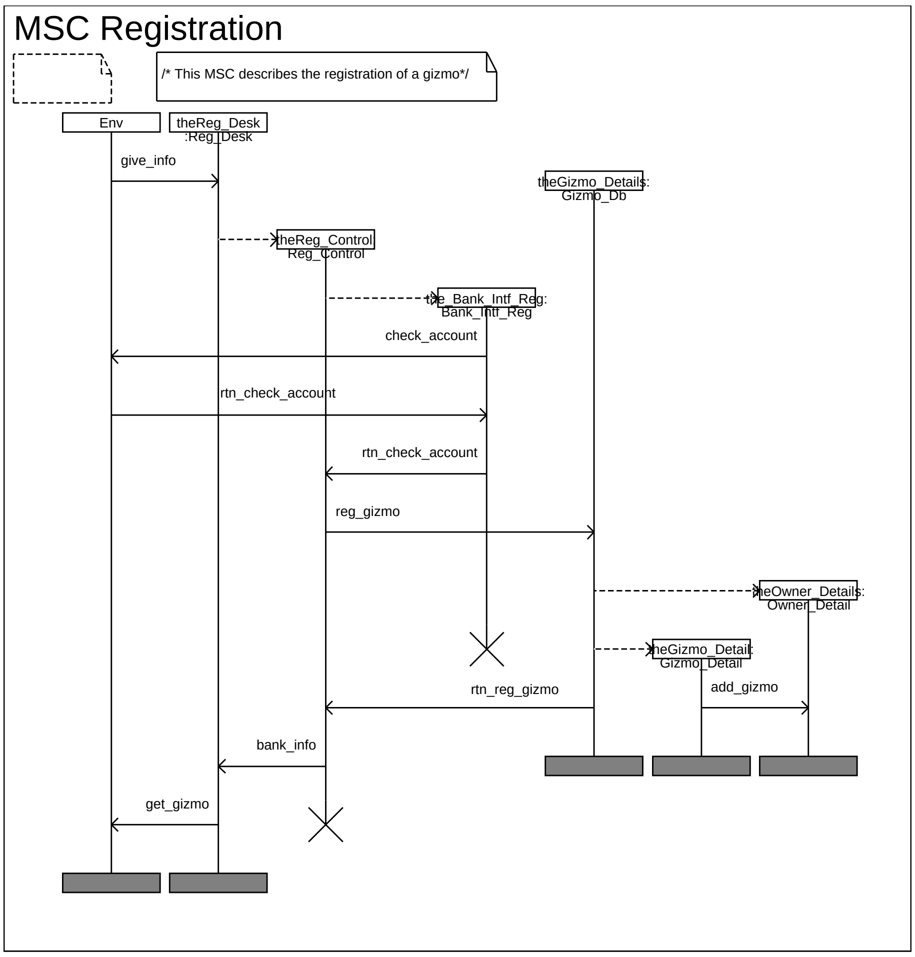

MSC Registration
/* This MSC describes the registration of a gizmo*/
Env
theReg_Desk :Reg_Desk
give_info
theGizmo_Details: Gizmo_Db
theReg_Control: Reg_Control
the_Bank_Intf_Reg: Bank_Intf_Reg
check_account
rtn_check_account
rtn_check_account
reg_gizmo
theOwner_Details: Owner_Detail
theGizmo_Detail: Gizmo_Detail
rtn_reg_gizmo
add_gizmo
bank_info
get_gizmo

Figure 4.11: Registration MSC showing process interactions

diagram.

The user views show interactions between the environment and objects which belong to the interface classes of the system-centred model. The system-centred model also includes control and entity classes which are required to support the external behaviour. To identify these other classes and their interactions, we construct MSCs. The MSCs are more detailed than those used to validate the SDL user-centred model as they show interactions between processes and not just interactions between the system and the environment. The starting event will usually be an event in a user scenario which then triggers a series of interactions between objects internal to the system.

Let us consider the view where a gizmo is registered. As we see from the graphs in Figures 4.6 and 4.7 which represent the two single-user views, an owner gives information to the system through a `Reg Desk` interface object, the system checks the owner's bank account through a `Bank Intf Reg` object and a gizmo is either granted or refused. We must now identify the control and entity objects which are required to support this behaviour. So that several registration requests can be handled at the same time, the `Reg Desk` object creates a control object of class `Reg Control` to handle each registration. Enquiries to the bank are through the `Bank Intf Reg` interface object. On getting a positive reply from the bank, the

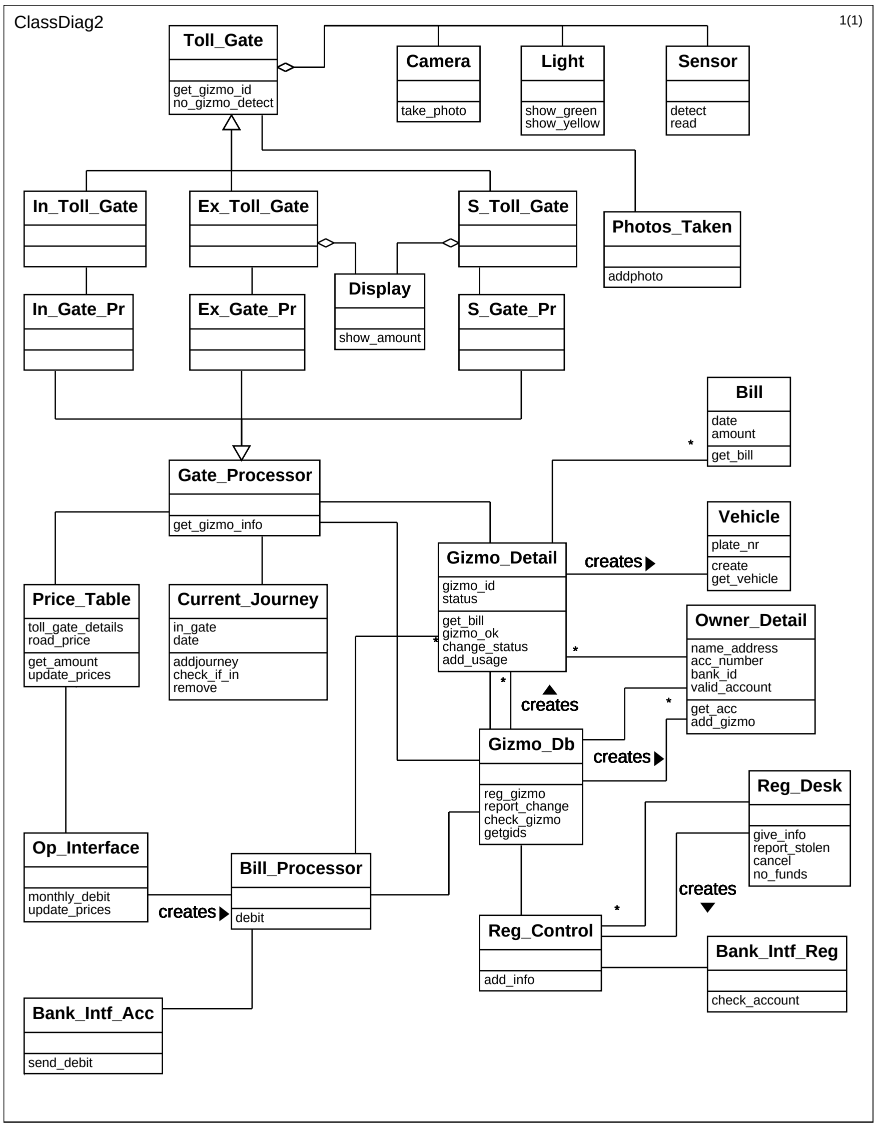


Figure 4.12: Class diagram

`Reg_Control` object sends a request to the `Gizmo_Db` to register the gizmo. The `Gizmo_Db` object creates a new `Gizmo_Detail` object together with an `Owner_Detail` object if one does not already exist. The resulting MSC is shown in Figure 4.11. Its creation has allowed us to identify new services and the new entity classes `Gizmo_Db`, `Owner_Detail` and `Gizmo_Detail` together with the control class `Reg_Control`.

Note that the MSC shown in Figure 4.11 not only has more detail than that shown in Figure 4.10, but the external events are in the opposite direction as we are now considering the system's interaction with the environment while previously we were considering the environment's interaction with the system.

Each multi-user view is considered in turn and detailed MSCs constructed to show the resultant interactions within the system-centred model that is required to support the view. When this causes new classes and services to be identified, they are added to the class diagram. A full class diagram for the road pricing system is depicted in Figure 4.12. The SDT tool supports the creation of a class diagram and links can be set up between items in the data dictionary and the class diagram so that the tool can check that they are consistent with one another.

4.3.2 Identify and specify subsystems

We group logically related classes in the class diagram into subsystems to provide an initial structure for the system. In our example, we identified four subsystems: `Tolls` (with the `Toll_Gate` inheritance hierarchy, its aggregate components and the `Gate Processor` inheritance hierarchy), `Debits` (with `Bill Processor` and its associated interface objects), `Control_Reg` (with `Reg Control`, its interfaces and the entity classes such as `Gizmo Detail`), and `Process_Pass` (with the entity objects `Photos Taken`, `Current Journey` and `Price Table`). The system-centred model is represented by these four interacting subsystems. Each subsystem is itself modelled as a block (see Figure 4.13). We propose that two blocks within a model always communicate through a single channel, but can communicate with the environment via many channels.

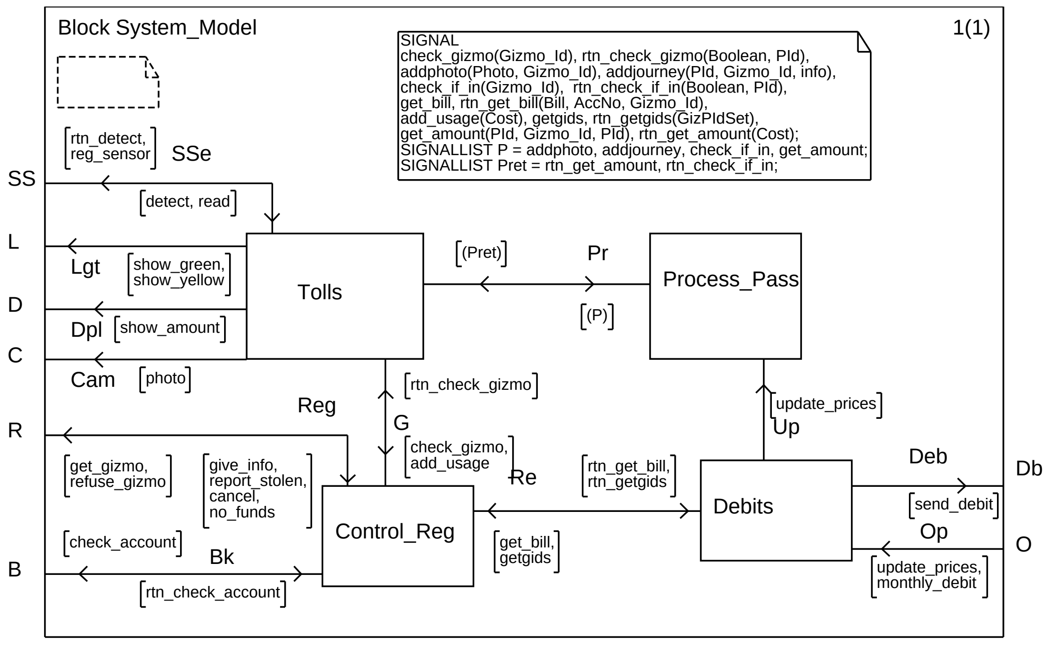

Block System_Model
1(1)
SIGNAL
check_gizmo(Gizmo_Id), rtn_check_gizmo(Boolean, PId),
addphoto(Photo, Gizmo_Id), addjourney(PId, Gizmo_Id, info),
check_if_in(Gizmo_Id), rtn_check_if_in(Boolean, PId),
get_bill, rtn_get_bill(Bill, AccNo, Gizmo_Id),
add_usage(Cost), getgids, rtn_getgids(GizPIdSet),
get_amount(PId, Gizmo_Id, PId), rtn_get_amount(Cost);
SIGNALLIST P = addphoto, addjourney, check_if_in, get_amount;
SIGNALLIST Pret = rtn_get_amount, rtn_check_if_in;
[rtn_detect, reg_sensor]
SSe
SS
[detect, read]
L
Lgt
[show_green, show_yellow]
Tolls
[(Pret)]
Pr
Process_Pass
[(P)]
D
Dpl
[show_amount]
C
Cam
[photo]
[rtn_check_gizmo]
Reg
G
[update_prices]
Up
R
[check_gizmo, add_usage]
Re
[rtn_get_bill, rtn_getgids]
Deb
Db
[get_gizmo, refuse_gizmo]
[give_info, report_stolen, cancel, no_funds]
Control_Reg
Debits
[send_debit]
Op
O
[check_account]
Bk
[get_bill, getgids]
[update_prices, monthly_debit]
B
[rtn_check_account]

Figure 4.13: System-centred model

Block `Tolls` is further decomposed into blocks representing the different kinds of toll gate. Figure 4.14 shows block `S Toll` which corresponds to the `S Toll Gate` aggregate and its components. The figure includes instances of the components `Sensor`, `Display`, `Light` and `Camera` which are modelled as processes.

4.3.3 Specify classes

We start by specifying the interface classes and then proceed to the control and entity classes. The interface classes and their interaction with the environment have already been determined by the user-centred model. Each signal in the SDL process definition of a user view has a corresponding signal in the SDL process definition of an interface class.

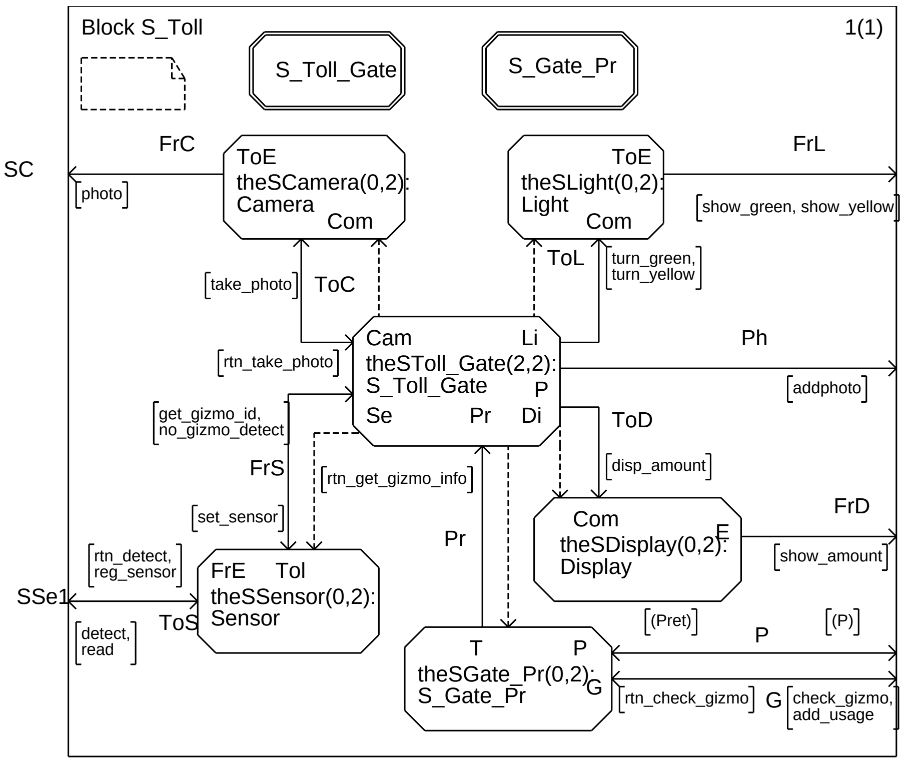


Figure 4.14: Composition of S Toll

Similarly, each class in the class diagram has a corresponding process definition and each offered service a corresponding signal in the SDL specification. SDT allows us to check that the two models are consistent and uses the information in the class diagram to semi-automate the construction of the SDL process definition.

The MSCs used to help identify the classes in the class diagram show the interactions between objects of these classes. This is a major help in determining the required process behaviour.

SDL supports inheritance. For example, in our class diagram, we have inheritance between the superclass `Toll Gate` and the subclasses `S Toll Gate`, `In Toll Gate` and `Ex Toll Gate`. Figure 4.15 shows the definition of the abstract superclass `Toll Gate`. The inputs `get gizmo id`, `rtn_get_gizmo_info` and the start transition are defined as virtual, which means that the associated transitions may be redefined in the subclasses. The process identifiers `li`, `di`, `ca` and `se` are given values in the subclasses.

In Figure 4.16, we define the subclass `S Toll Gate` by redefining the virtual transitions and adding an extra gate to deal with `Display`. The rest of the behaviour is inherited from the superclass.

Let us consider the behaviour of the subclass `S Toll Gate`. It is an aggregate with four components: `Display`, `Camera`, `Light` and `Sensor`. During the initialization step, `S_Toll_Gate` creates its components and notes their PIds. As the creation is in terms of the component instances (process sets) defined within block `S Toll`, the creations cannot be specified in the superclass. It then enters state `wait` where it waits for a signal from its sensor. Two possibilities are specified in the superclass: the sensor reports a vehicle with a gizmo (signal `get_gizmo id`) and passes its gizmo identifier to the toll gate, or it reports a vehicle which

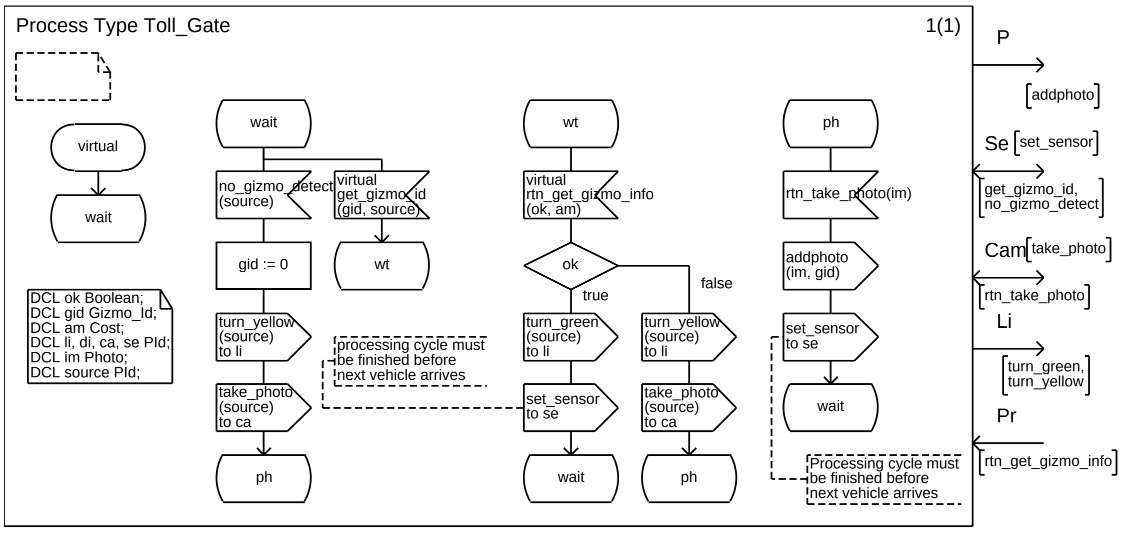


Figure 4.15: Toll abstract superclass

does not have a gizmo (signal `no gizmo detect`). When a vehicle with a gizmo is detected, a control process of class `S Gate Pr` is created to initiate the action required to check whether or not the gizmo is valid. The result is given by the redefined signal `rtn get gizmo info`. When a vehicle with no gizmo has been detected, the actions associated with the transition are sending a signal to the light to turn yellow (this corresponds to calling the offered service `turn_yellow` in class `Light`) and a signal to the camera to take a photograph (corresponding to calling the offered service `take photo` in class `Camera`). The process then enters state `ph` where it waits for a new input `rtn take photo` which gives it the photo image. This input initiates another transition in which the image is sent to process `Photos Taken` (corresponding to calling the offered service `addphoto` in class `Photos Taken`) and the sensor is reactivated to deal with the next vehicle.

When we specify the system-centred model, it is not unusual for incompatibilities to appear in the expectations of different users. This may require us to go back to the users for clarification. The complete system-centred model is shown in Appendix B.

4.3.4 Build and validate the system-centred model

Initially, we use interactive navigation with either the simulator or validator to explore interesting execution paths so that we can increase our confidence that the expected behaviour is offered. The detailed MSCs that aided the creation of the system-centred model can be used to guide the navigation to ensure that their behaviour has, in fact, been modelled. The validator can then demonstrate that the specification is internally consistent. Due to the large number of possible paths through the system-centred model, a larger number of random walks, with each explored to a greater depth than was needed with the user-centred model, is required to demonstrate that all transitions are executed at least once.

The user-centred model has already been validated against the requirements using a set of high-level MSCs which showed the interaction between user views and their environment (i.e. the system). We now replace the informal task of validating the system-centred model with respect to the requirements by the formal task of verifying that it offers the behaviour

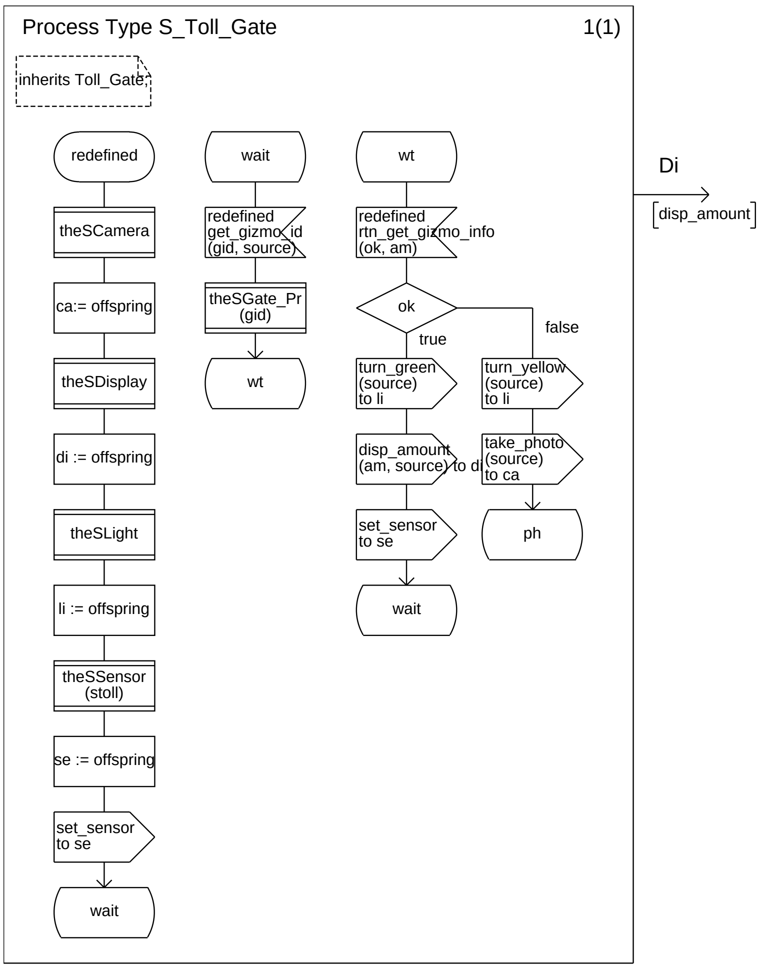


Figure 4.16: S_Toll subclass

expected by the user-centred model. We construct a high-level MSC for the system-centred model corresponding to each scenario used in the validation of the user-centred model. These high-level MSCs only show the interaction between the system and its environment although, of course, for them to be satisfied, the validator must execute the necessary interactions between the interface, control and entity objects that are required to support the external behaviour.

Where the user-centred model accepts a behaviour, that behaviour must also be accepted by the system-centred model. Where the user-centred model rejects a behaviour, that behaviour must also be rejected by the system-centred model.

4.4 The combined model

An SDL specification assumes that its environment behaves reasonably and obeys constraints imposed by the specification. However, these constraints can only be imposed if we model (important parts of) the environment explicitly. As the user-centred model specifies what the users expect from the system, and the system-centred model specifies the system's interaction with the environment, we can compose the two models to enable a complete sequence of interactions between the environment and the system.

Conceptually, the user-centred model interacts with the interface objects of the system being specified. The SDL specification of the user-centred model is therefore expressed in terms of it sending and receiving signals from the processes which constitute the system-centred specification. However, the user-centred SDL specification in fact sends signals to, and receives signals from, its environment; there are no actual process instances to receive and return signals. Similarly, the system-centred model receives signals from, and returns signals to, its environment. Again, there are no actual process instances in its environment. We cannot use the value of `sender` in a simulation to ensure that a dialogue occurs between a particular user in the environment and a particular object in the system as the user-centred and system-centred models each model only one side of the dialogue. Nevertheless, we specify both the user-centred and the system-centred models as if their environment was composed of distinct process instances. Then, when we compose the user-centred and system-centred models, complete sequences of interactions between the system and the environment can be modelled explicitly, with the value of `sender` being used in both models to ensure that signals are sent to the correct instance in the other model. Demonstrating that the combined model is internally consistent, and offers the same behaviour as its components, is a powerful test that the two component models correctly specify the same system.

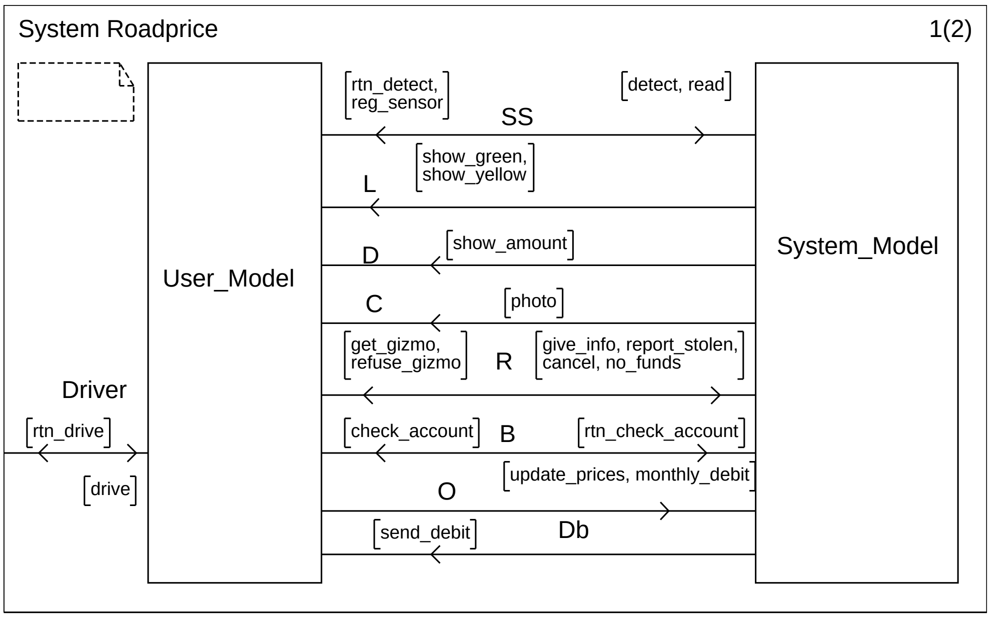


Figure 4.17: The combined system

The SDL specifications of the user-centred and system-centred models each consist of a single block which interacts with the SDL system environment. The SDL system representing the combined model contains these two blocks as shown in Figure 4.17. The `Driver` channel remains between the environment and the user-centred model. The set of MSCs used to validate the user-centred and system-centred models are easily modified so that they can be used to validate the combined model. Our normal rule is that blocks interact through a single channel, but the user-centred and system-centred models interact with each other through as many channels as existed in their original interaction with their environment.

Ideally, the user-centred and system-centred models are used unchanged in the combined model. There is one circumstance where we may want to make a change. The SDT toolset does not allow `PId`s to be passed as parameters to and from the environment while they may be passed between the user-centred and system-centred components of the combined model. Passing `PId`s can be important in setting up dialogues and ensuring that the correct process instances are accessed. In our combined model, we have changed the `Owner Registration` view so that it passes the `PId` of the `Bank Enquiry` view. That way the system-centred model can ensure that it sends the `check account` signal to the correct view. The system-centred model is unchanged.

Chapter 5

Comparison with LOTOS

Our previous work in this area used LOTOS and the result was ROOA. We have found that the replacement of LOTOS in ROOA by SDL is straightforward and that the application of the method is basically unchanged, with the same tasks being applied in the same order. The differences are primarily in the detail of how the two languages are applied within a ROOA task. The differences are concerned with:

- the ease of defining data types,
- the specification of multi-user views,
- the validation process,
- the structuring of the formal specification,
- the tool support,
- the effect of the different approach to process synchronization.

Data types

Although data typing in SDL and LOTOS is based on the same principles, they are much easier to use in SDL than in LOTOS where most data types have to be constructed from first principles. SDL has built in types such as `Integer`, `Character`, `Boolean`, records (`structs`), has predefined generators such as `Array`, `String`, `PowerSet` and `Bag`, and allows enumeration types to be created. As we are dealing with the analysis phase, the new types we create from first principles are simple with only symbolic values and few, if any, operators. The ability to create `structs`, enumeration types and sets of values is very useful.

Multi-user views

Multi-user views are specified in LOTOS using multiway synchronization [6] instead of a `Coordinator` process. New LOTOS processes were defined to specify the constraints imposed by the interaction between two views. The LOTOS processes defining the single-user views are then interleaved and composed in parallel with the constraint processes to form a multi-user view.

In SDL, the `Coordinator` process maintains sets to represent the current state. These are used to control the creation of multi-user view components and to define their parameters.

Validation

When the specifications were expressed in LOTOS, the validation process used scenarios expressed in LOTOS, rather than introducing a new notation (i.e. MSCs) as is the case with SDL. Also, event synchronization in LOTOS is symmetric; there is no notion of a sending event and a receiving event. This allows us to verify that the user-centred and the system-centred models offer the same behaviour. In SDL, we verify that the behaviour required by the user-centred model is that offered by the system-centred model. To show that the system-centred model offers the required behaviour, it must satisfy the same set of MSCs used in validating the user-centred model; only now the direction of the signals is reversed.

During validation with LOTOS, the scenarios acted as the environment and ensured that it behaved reasonably. However, the MSCs do not provide this control when validating an SDL specification and important parts of the environment had to be modelled explicitly in SDL. This means that the combined model plays a more important part in an SDL specification than it does in the LOTOS version.

Structuring

SDL has explicit constructs such as blocks which help structure a specification. We feel that this leads to the construction of an SDL specification being top-down whereas, in LOTOS, the processes were created first and their combination into behaviour expressions occurred later.

With LOTOS, ROOA constructs an object communication diagram (OCD) to show the graphical structure of the eventual LOTOS specification and to help in the construction of the LOTOS behaviour expression. In SDL, the equivalent of the OCD are the block diagrams which diagrammatically show the interaction either between a set of blocks (subsystems) or between a set of process instances (objects).

Tool support

A major advantage of SDL is that it has object-oriented features built in. Toolsets such as SDT support an object-oriented design method and this results in better support for ROOA than with LOTOS tools. SDT also allows links to be set up between entities in the data dictionary, class diagram and SDL diagrams. This supports both forward and backward traceability. Forward traceability is supported as information in one diagram can be used to help create subsequent diagrams and conformance of the diagrams can be checked. In an iterative method it is important that the diagrams remain consistent as changes are made and that we can trace back to the origin of any inconsistency or error.

Process synchronization

When we model message passing in LOTOS, an object cannot send a message unless another object is willing to receive it, i.e. we have synchronous communication.

In SDL, as we have asynchronous communication, an object can send a message (i.e. a signal) when the intended receiving object (process) is not ready. This can lead to several problems.

With asynchronous communication, the fact that a signal has been sent does not guarantee that it has been accepted. Similarly, when one process creates another, there is a delay before the new process comes into existence, executes its start transition and is ready to receive a signal. This can require the new process sending a signal to its parent to confirm that it now exists and is ready to receive signals.

Also, when validating a specification which contains a set of processes, if a signal is sent to the set, a process is chosen at random. With the SDT tool, it is possible that the chosen process has terminated or is not able to receive the signal even when there are processes which are able to receive the signal. This can lead to signals being implicitly consumed.

Another related point is that, with structured events in LOTOS, value matching of any of the parameters of an event can be used to control the synchronization. In SDL, the only way that we can specify that one of a set of processes is to be chosen is by explicitly using its `PId`. We therefore have much tighter control over process interaction in LOTOS while asynchronous communication in SDL leads to a more loosely coupled and flexible structure.

Chapter 6

Conclusions

The main goals of ROOA are:

- to clarify and understand the requirements, i.e. to help get the requirements right,
- to create and validate a formal specification of the informal requirements.

Informal requirements are usually expressed in terms of user needs. They almost invariably contain inconsistencies, ambiguities and omissions. Therefore, it is important to understand correctly the exact needs of a user (or a client) before any code starts to be written. We believe that the best (and least expensive) way to achieve this is by adding formality to the analysis process.

We have investigated how a formal specification can be obtained from informal requirements in a systematic way. The result of this work is the Rigorous Object-Oriented Analysis (ROOA) method.

ROOA produces executable specifications that can be prototyped to identify inconsistencies, omissions and ambiguities very early in the development process.

A formal requirements specification may be used as the starting point for a formal development trajectory. However, the process of creating a formal specification from informal requirements is of major value even when the rest of the development process is not formal. It forces the specifiers to be precise and to consider all combinations of circumstances; they cannot hide in vague generalities.

To simplify the difficult task of constructing a formal object-oriented specification (a system-centred model) from informal requirements, we first build a user-centred model. This model helps us build the system-centred model and simplifies the problem of validating it against the requirements.

Our previous work used LOTOS. Here we have shown how SDL can be integrated into ROOA to represent both models. We have found that the differences in using SDL instead of LOTOS in ROOA was in the detail of multi-user views, the validation process and tool support. The basic approach remained unchanged.

SDL is already widely used in industry for design. Here we have demonstrated its suitability within requirements analysis. Our work therefore complements previous uses of SDL.

Bibliography

[1] Z.100: Specification and Description Language SDL, ITU-T, June 1994.

[2] T. Bolognesi and E. Brinksma, Introduction to the ISO Specification Language LOTOS, *Computer Networks and ISDN Systems*, **14**(1), 25-59, 1987.

[3] E. Brinksma (ed). *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, ISO 8807, 1989.

[4] A. Moreira and R. Clark, Combining Object-Oriented Analysis and Formal Description Techniques, *8th European Conference on Object-Oriented Programming: ECOOP '94*, pp. 344-364, LNCS 821, Springer-Verlag, 1994.

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

[6] R. Clark and A. Moreira, User Centred Models, *Formal Methods on Object-based Open Distributed Systems (FMOODS'97)*, pp. 215-230, Chapman Hall, 1997.

[7] R. Clark and A. Moreira, Constructing Formal Specifications from Informal Requirements, *Software Technology and Eng. Practice (STEP'97)*, pp. 68-75, IEEE Press, 1997.

[8] D. de Champeaux, P. America, D. Coleman, R. Duke, D. Lea and G. Leavens, Formal Techniques for OO Development, *OOPSLA'91: ACM SIGPLAN Notice*, **26** (11), 166-170, 1991.

[9] A. Hall, Seven Myths of Formal Methods, *IEEE Software*, **7**(5), 11-19, 1990.

[10] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima and C. Chen, Formal Approach to Scenario Analysis, *IEEE Software*, **11**, March, 33-41, 1994.

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

[12] S. Somé, R. Dssouli and J. Vaucher, Towards an Automation of Requirements Engineering using Scenarios. *Journal of Computing and Information*, **2**, 1070-1092, 1996.

[13] I. Jacobson, *Object-Oriented Software Engineering.* Addison-Wesley, Reading Massachusetts, 1993.

[14] Telelogic SDT 3.4 Methodology Guidelines Part 1: The SOMT Method, 1998.

[15] D. Sinclair, G.Clynch and B. Stone, An Object-Oriented Methodology from Requirements to Validation, *Proc. OOIS'95*, Springer-Verlag, 1996.

[16] R. Braek and O. Haugen, *Engineering Real Time Systems*, Prentice Hall, 1993.

[17] R. Reed, Methodology for Real Time Systems, *Computer Networks and ISDN Systems*, **28**(12), 1685-1701, 1996.

[18] J. Kuusela and E. Kettunen, Integrating SDL and Object Oriented Analysis through OMT/SDL, *SDL'93*, pp 89-103, North Holland, 1993.

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

[20] Z.120: Message Sequence Charts (MSC), ITU-T, Oct 1996.

[21] R. Braek, O. Haugen *et al*, SISU Integrated Methodology, SISU Report L-2001-7, SISU, Norway, Oct 1996.

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

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

[24] T. Bolognesi, E. Najm and P. Tilanus, G-LOTOS: A Graphical Language for Concurrent Systems, *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, pp 391-437, 1995.

[25] Telelogic SDT 3.4 Reference Manuals, November 1998.

[26] R. Clark and A. Moreira, Using a formal user-centred model to build a formal system-centred model, *Computing Science and Mathematics TR 140*, University of Stirling, 1997.

Appendix A

The user-centred model

A.1 Overall model

The block representing the SDL specification of the user-centred model and all the diagrams defined within that block are given in this appendix.

The user-centred model is defined as a single block. In the combined system, shown in Figure 4.17, that block together with the block representing the system-centred model form the specification.

Block `User Model` consists of a `Coordinator` process together with the processes representing the user views.

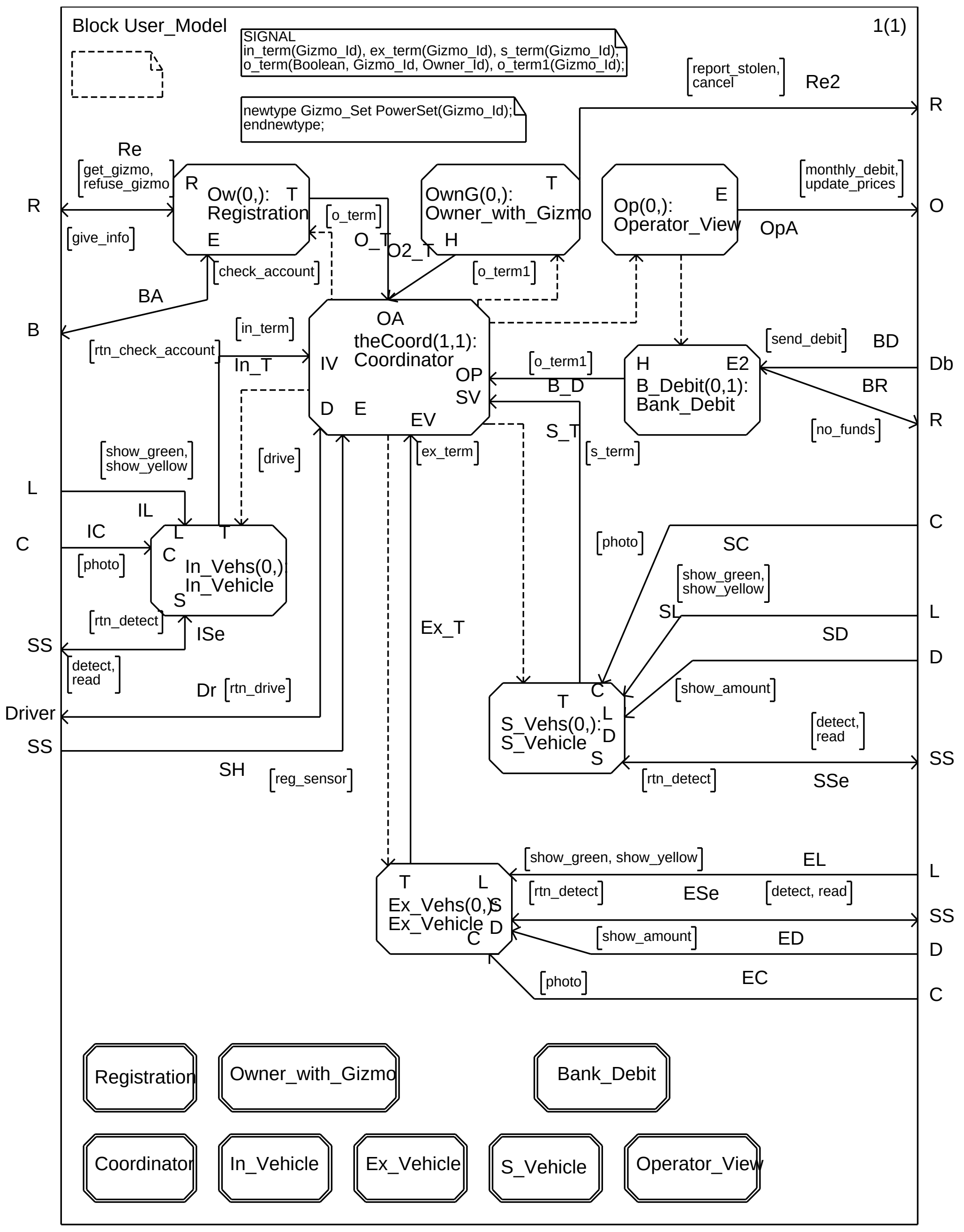


Figure A.1: User-centred model

A.2 The coordinating process

Process `Coordinator`, shown in Figure A.2, controls the execution of the user views. The available toll gate sensors report their existence by `reg_sensor` signals. User views are initiated by a `drive` signal from the environment which determines which view is to be created and executed. There are three main situations: passage of a vehicle through a toll gate, registration of a vehicle and monthly debits. Passage of a vehicle is subdivided into three views depending on the kind of toll gate being passed. A gizmo identifier of zero is interpreted as failure by a sensor to read the gizmo.

Various sets are maintained by `Coordinator` so that the environment can maintain knowledge of the system state, in particular the status of the different gizmos. These sets are used to initialise the parameters used in view creation and are updated when a view is created or when it terminates. Termination of a view is reported to `Coordinator` by it sending a *prefix*`_term` signal.

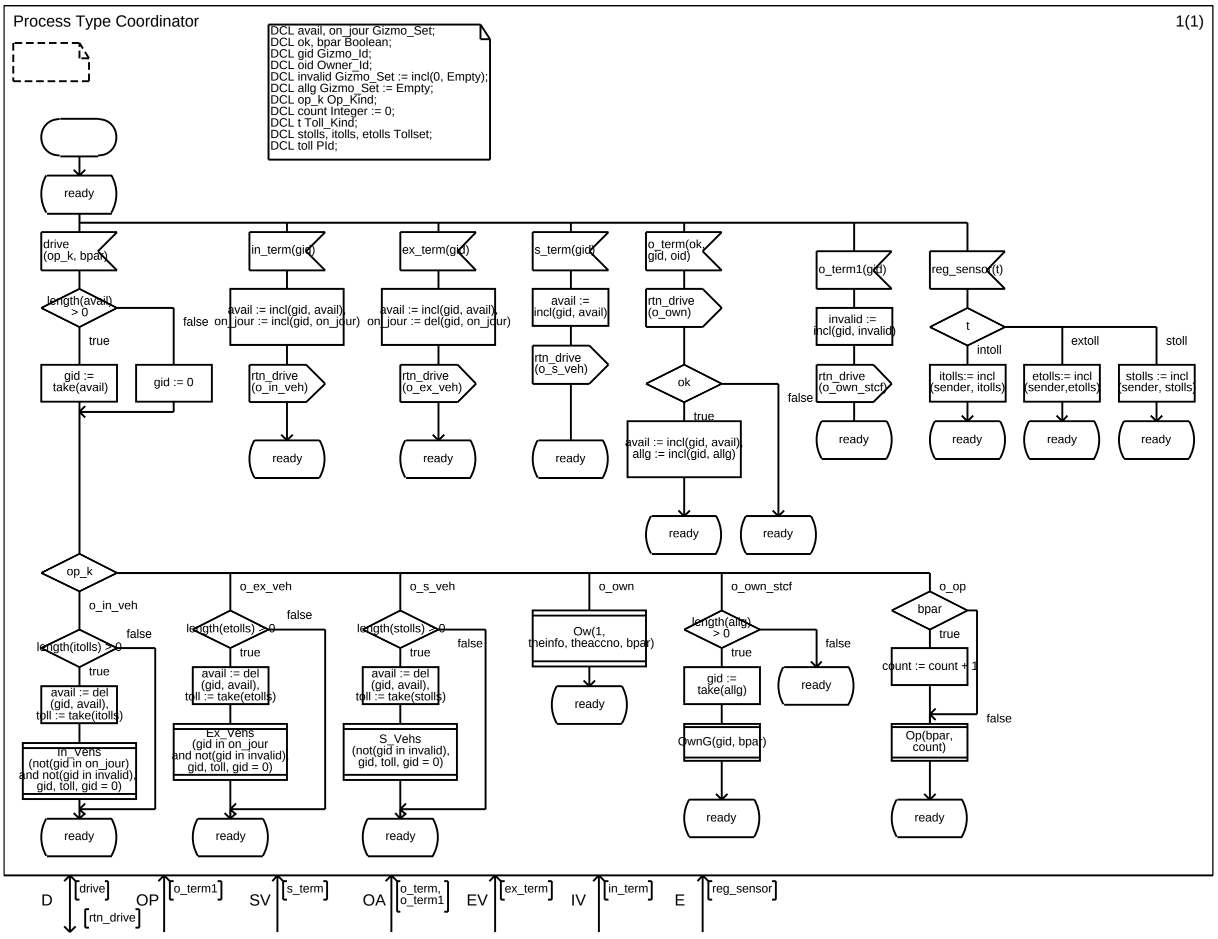

Process Type Coordinator
1(1)
DCL avail, on_jour Gizmo_Set;
DCL ok, bpar Boolean;
DCL gid Gizmo_Id;
DCL oid Owner_Id;
DCL invalid Gizmo_Set := incl(0, Empty);
DCL allg Gizmo_Set := Empty;
DCL op_k Op_Kind;
DCL count Integer := 0;
DCL t Toll_Kind;
DCL stolls, itolls, etolls Tollset;
DCL toll PId;
ready
drive (op_k, bpar)
length(avail) > 0
true
false
gid := take(avail)
gid := 0
in_term(gid)
avail := incl(gid, avail) on_jour := incl(gid, on_jour)
rtn_drive (o_in_veh)
ex_term(gid)
avail := incl(gid, avail) on_jour := del(gid, on_jour)
rtn_drive (o_ex_veh)
s_term(gid)
avail := incl(gid, avail)
rtn_drive (o_s_veh)
o_term(ok, gid, oid)
rtn_drive (o_own)
ok
avail := incl(gid, avail), allg := incl(gid, allg)
o_term1(gid)
invalid := incl(gid, invalid)
rtn_drive (o_own_stcf)
reg_sensor(t)
t
intoll
extoll
stoll
itolls:= incl (sender, itolls)
etolls:= incl (sender,etolls)
stolls := incl (sender, stolls)
op_k
o_in_veh
o_ex_veh
o_s_veh
o_own
o_own_stcf
o_op
length(itolls) > 0
avail := del (gid, avail), toll := take(itolls)
In_Vehs (not(gid in on_jour) and not(gid in invalid), gid, toll, gid = 0)
length(etolls) > 0
avail := del (gid, avail), toll := take(etolls)
Ex_Vehs (gid in on_jour and not(gid in invalid), gid, toll, gid = 0)
length(stolls) > 0
avail := del (gid, avail), toll := take(stolls)
S_Vehs (not(gid in invalid), gid, toll, gid = 0)
Ow(1, theinfo, theaccno, bpar)
length(allg) > 0
gid := take(allg)
OwnG(gid, bpar)
bpar
count := count + 1
Op(bpar, count)
D
[drive]
[rtn_drive]
OP
[o_term1]
SV
[s_term]
OA
[o_term, o_term1]
EV
[ex_term]
IV
[in_term]
E
[reg_sensor]

Figure A.2: Coordinator process

A.3 Vehicle views

The vehicle views are created with four parameters: `gid` gives the gizmo identifier, `sensor_id` gives the identifier of the relevant sensor, `valid` determines whether or not the gizmo is valid and `fail` determines whether or not the gizmo fails to be read. The values of these parameters are determined by `Coordinator` using the information it holds about the status of the different gizmos. We have three vehicle views: Single-Point-Vehicle (Figure A.3), Enter-Motorway-Vehicle (Figure A.4) and Exit-Motorway-Vehicle (Figure A.5).

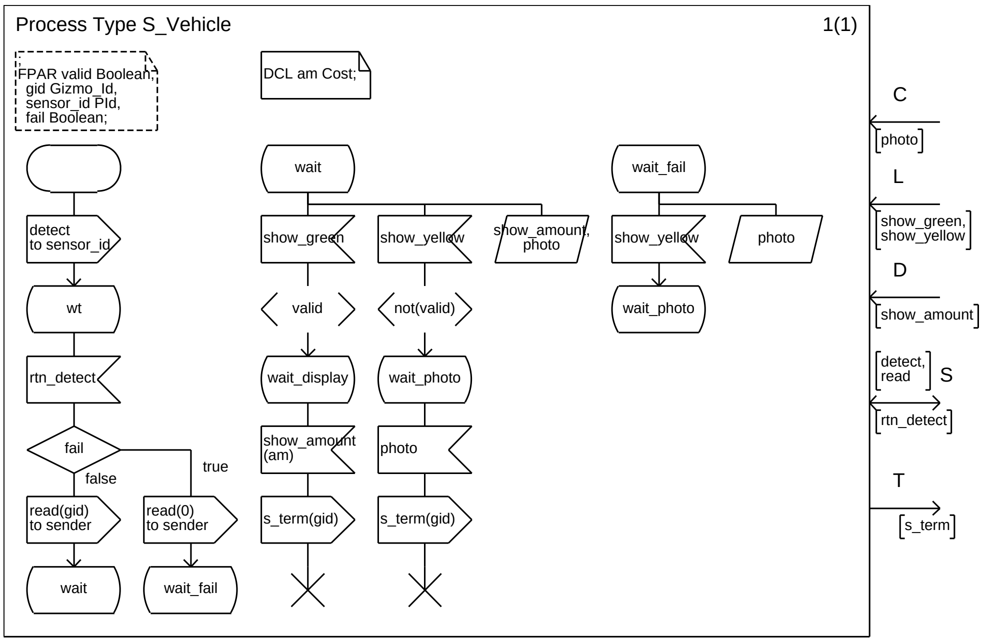


Figure A.3: Single-Point-Vehicle view

The SDL specifications of the user views are created directly from the graphs representing them. A difference is that the signals `photo` and `show_amount` are saved if they arrive too early.

The processes representing the vehicle views send the signals `s_term`, `in_term` or `ex_term` to indicate that they have finished execution.

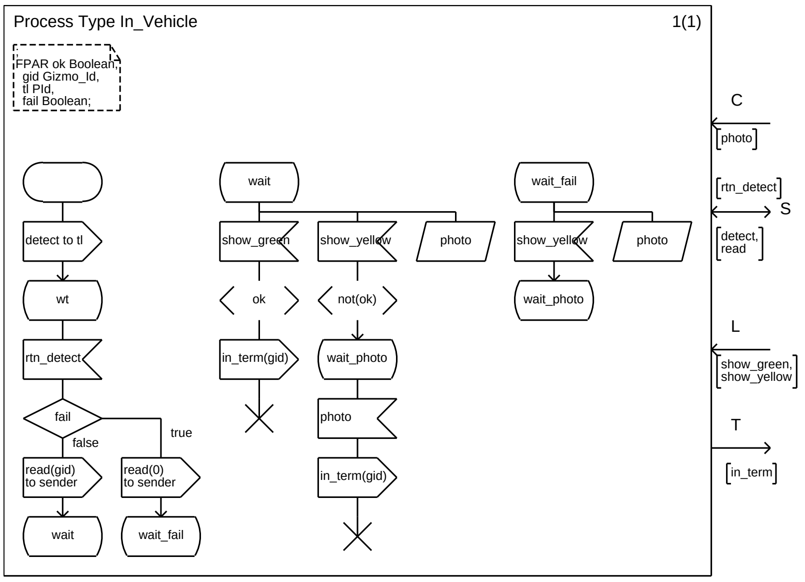


Figure A.4: Enter-Motorway-Vehicle view

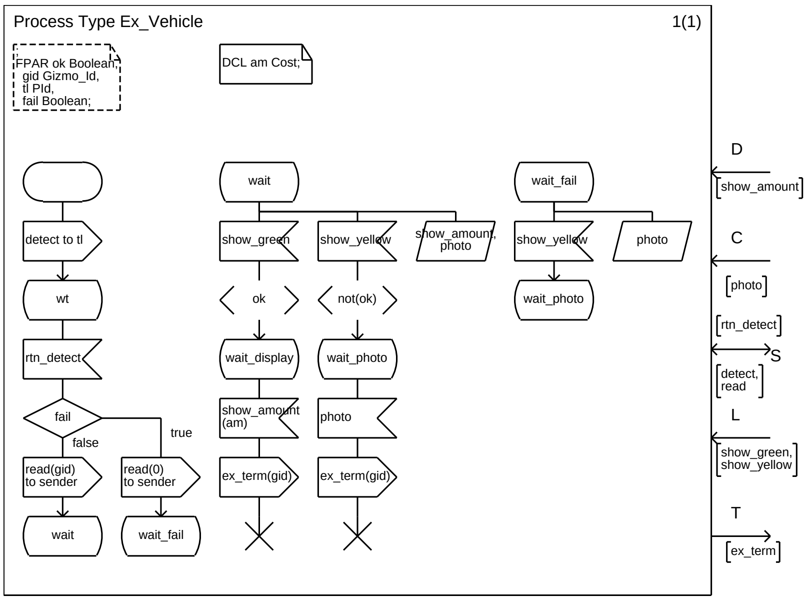


Figure A.5: Exit-Motorway-Vehicle view

A.4 Registration

The registration view consists of a process containing two SDL services (see Figure A.6).

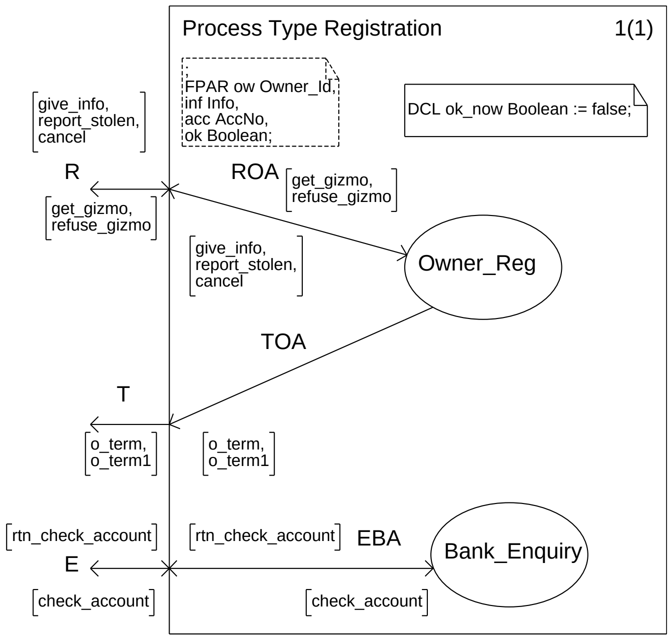

Process Type Registration
1(1)
FPAR ow Owner_Id, inf Info, acc AccNo, ok Boolean;
DCL ok_now Boolean := false;
give_info, report_stolen, cancel
R
get_gizmo, refuse_gizmo
ROA
Owner_Reg
TOA
T
o_term, o_term1
rtn_check_account
E
check_account
EBA
Bank_Enquiry

Figure A.6: Registration view

The service `Owner Reg` (Figure A.7) represents the user view where an owner tries to register a vehicle and get a gizmo. The other, `Bank Enquiry` (Figure A.8), is where the owner's bank account is checked.

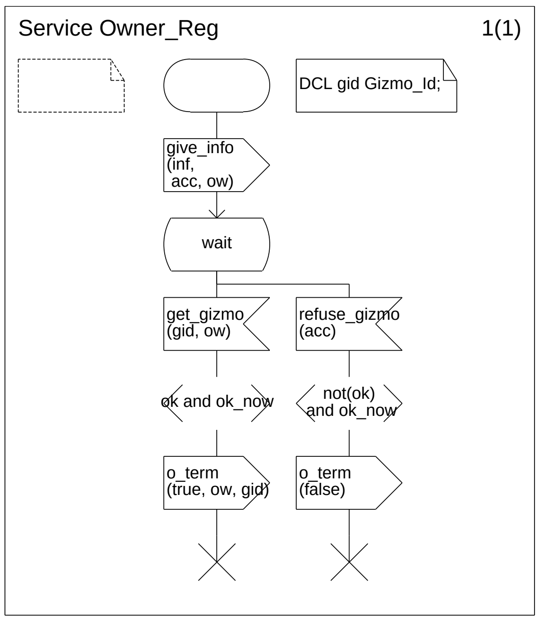


Figure A.7: Owner registration user view

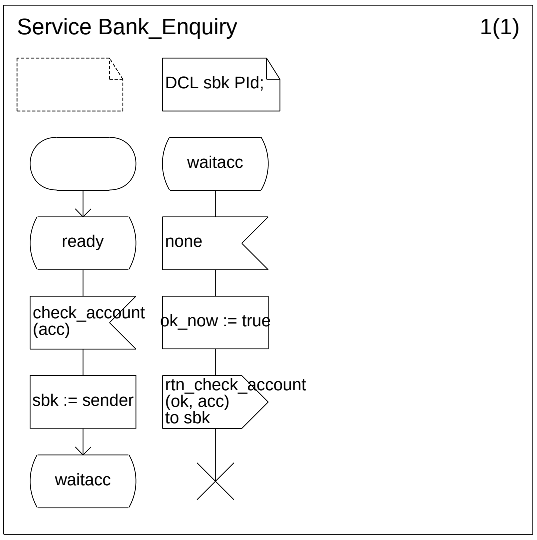


Figure A.8: Bank enquiry user view

The two services communicate through the variable `ok_now`. When the view is created, the start transition of `Owner_Reg` causes a message to be sent to the system giving information about the owner. The system then enquires at the bank (the response is handled by `Bank_Enquiry`) to check that the owner's account is satisfactory. `Owner_Reg` is then informed whether or not a gizmo is to be granted. It can only receive this signal once `ok_now` is `true`.

Once an owner has a gizmo, we have the view where he or she may cancel the gizmo or report that it is stolen. This view is dealt with in a separate process of class `Owner_with_Gizmo` (Figure A.9).

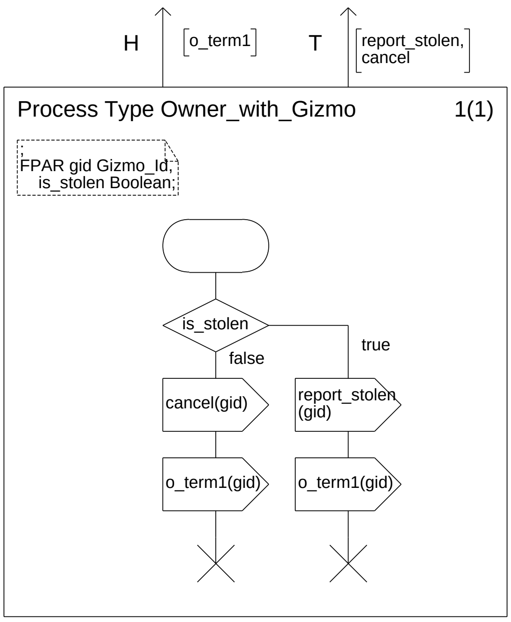


Figure A.9: Error report owner user view

A.5 Monthly debits

The operator view, represented by process `Operator View` (Figure A.10) either prompts the system to deal with the monthly debits by sending a `monthly debit` signal or updates prices by sending an `update price` signal. It then terminates unless this is the first time it has received a `monthly debit` signal, in which case a `Bank Debit` process (Figure A.11) is created to handle responses from the system.

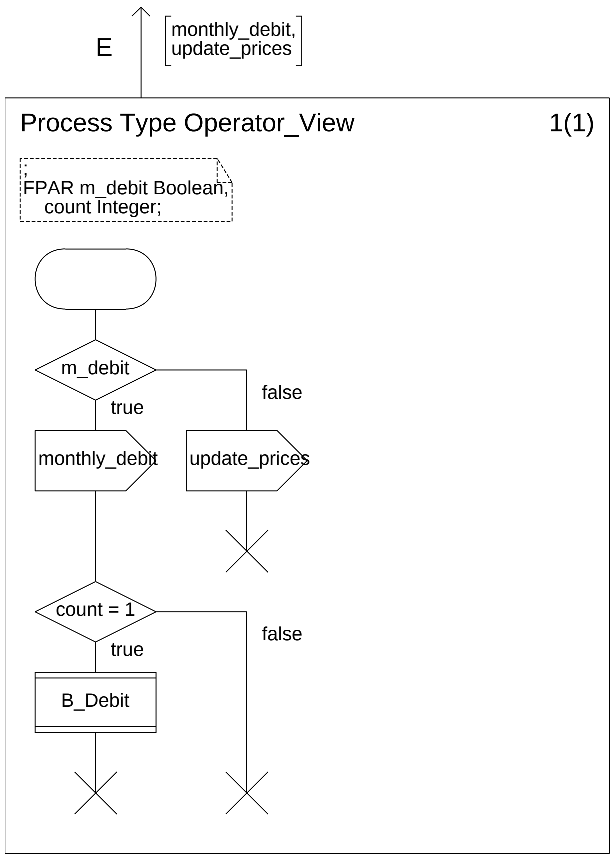


Figure A.10: Operator user view

The Bank_Debit process nondeterministically decides whether the account has funds. If there are no funds, a no_funds signal is sent to the system so that the information on the gizmo can be updated and Coordinator is informed via an o_term1 signal so that it knows that the associated gizmo is no longer valid.

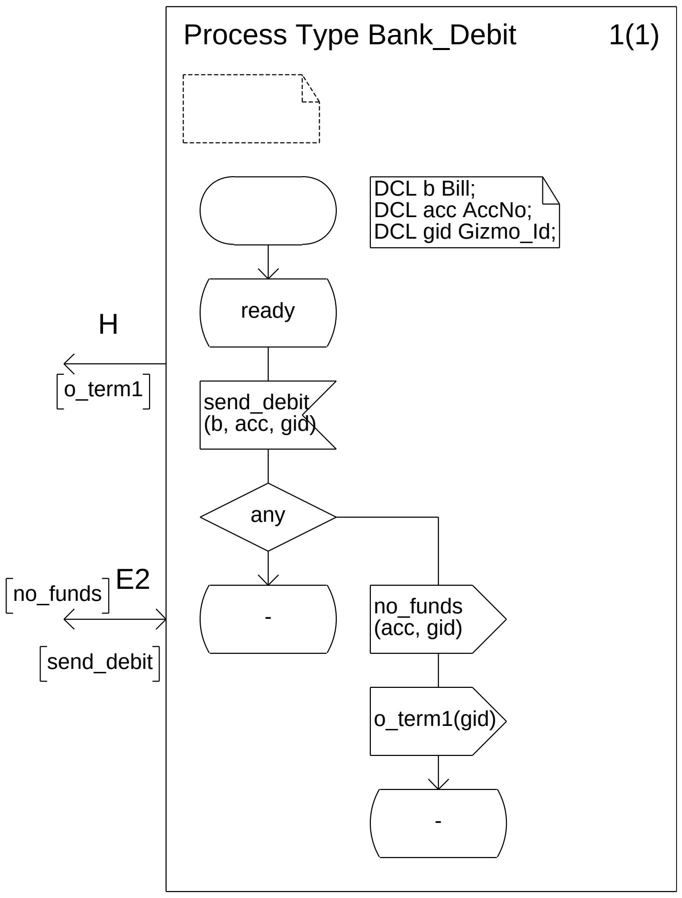


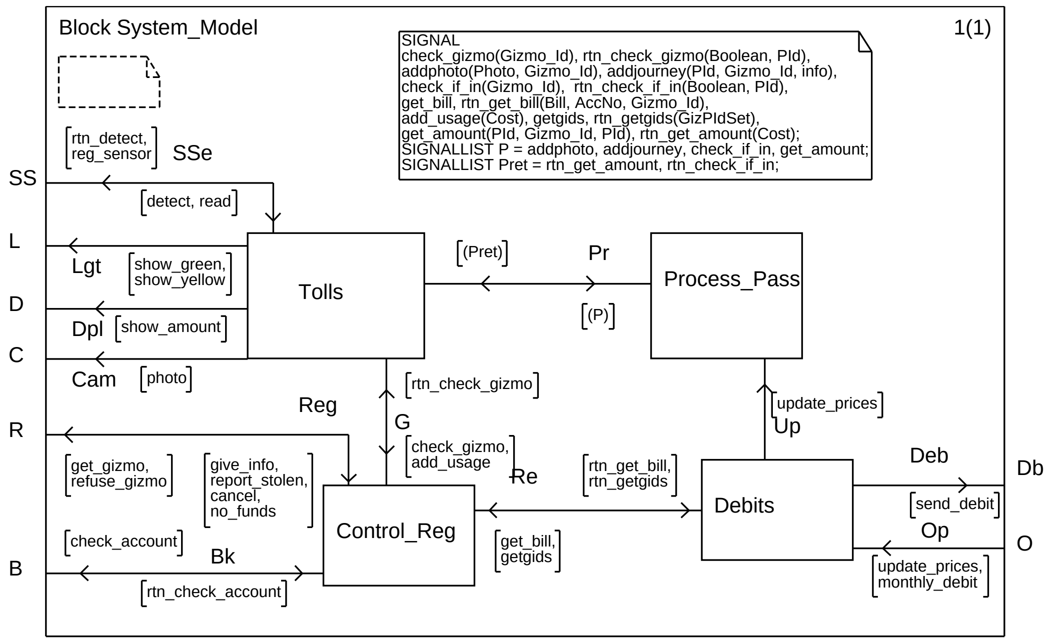
Figure A.11: Bank debit user view

Appendix B

The system-centred model

B.1 Overall model

The SDL specification of the system-centred model is represented as a single block (shown in Figure B.1). It contains four subsystems. This block is unchanged when it is used as one of the two components of the combined model shown in Figure 4.17.



Block System_Model
1(1)
SIGNAL
check_gizmo(Gizmo_Id), rtn_check_gizmo(Boolean, PId),
addphoto(Photo, Gizmo_Id), addjourney(PId, Gizmo_Id, info),
check_if_in(Gizmo_Id), rtn_check_if_in(Boolean, PId),
get_bill, rtn_get_bill(Bill, AccNo, Gizmo_Id),
add_usage(Cost), getgids, rtn_getgids(GizPIdSet),
get_amount(PId, Gizmo_Id, PId), rtn_get_amount(Cost);
SIGNALLIST P = addphoto, addjourney, check_if_in, get_amount;
SIGNALLIST Pret = rtn_get_amount, rtn_check_if_in;
rtn_detect, reg_sensor
SSe
SS
detect, read
L
Lgt
show_green, show_yellow
Tolls
(Pret)
Pr
Process_Pass
(P)
D
Dpl
show_amount
C
Cam
photo
rtn_check_gizmo
Reg
G
update_prices
Up
R
check_gizmo, add_usage
Re
rtn_get_bill, rtn_getgids
Deb
Db
get_gizmo, refuse_gizmo
give_info, report_stolen, cancel, no_funds
Debits
send_debit
Op
O
check_account
Bk
Control_Reg
get_bill, getgids
update_prices, monthly_debit
B
rtn_check_account

Figure B.1: System-centred model

B.2 Block Tolls

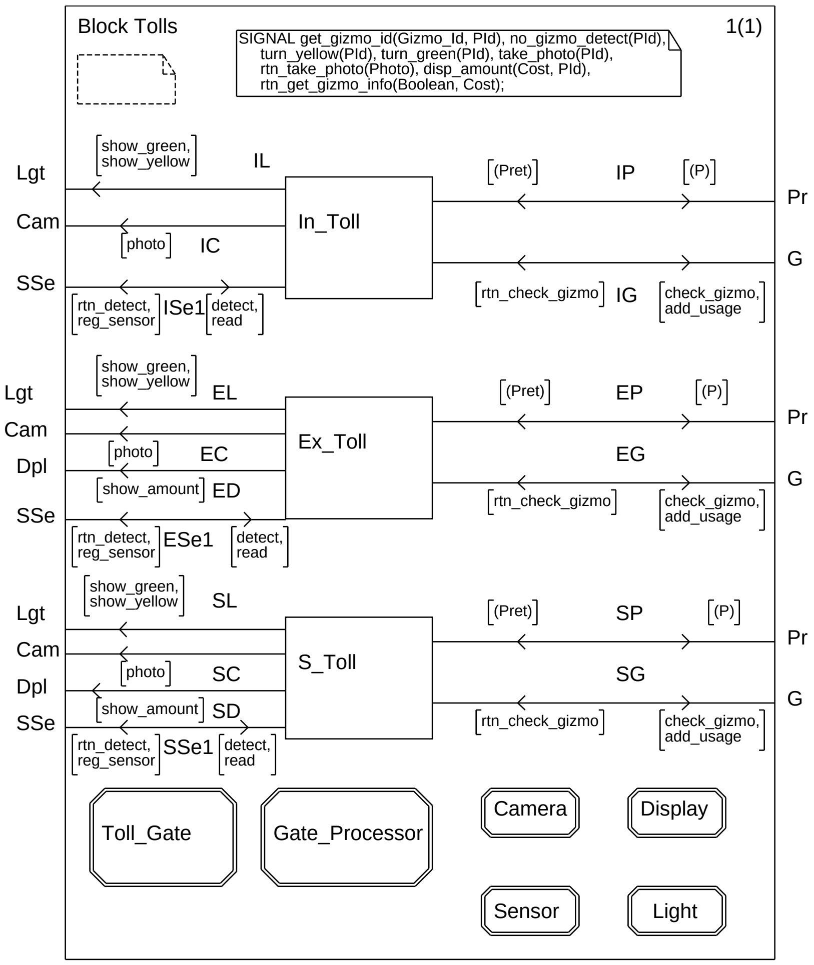


Figure B.2: Block Tolls

Block `Tolls`, shown in Figure B.2, contains separate blocks for each of the three kinds of toll gate. The process types `Toll Gate`, `Gate Processor`, `Camera`, `Sensor`, `Display` and `Light` are all defined here so that they are available in all three blocks.

Process `Toll Gate`, shown in Figure B.3, is an abstract superclass, which is specialised to form each of the three toll gate subclasses. The start transition is virtual as each of the subclasses must create the instances of the `Camera`, `Sensor` and `Light` components (and, in the case of `S Toll Gate` and `Ex Toll Gate`, an instance of `Display`) which make up the toll gate aggregate. The identifiers `ca`, `se` and `li` are `PId`s which are initialised in the start transition of the subclasses and used by a toll gate so that it can send signals to its components.

Process `Gate Processor`, shown in Figure B.4, is also an abstract superclass, which is specialised for each kind of toll gate. `Toll Gate` is an interface class. `Gate Processor` is a control class and an object of that class is created by a toll gate to carry out the processing and communication with the rest of the system necessary to handle a vehicle passing through the toll gate. When it has finished its task, it dies.

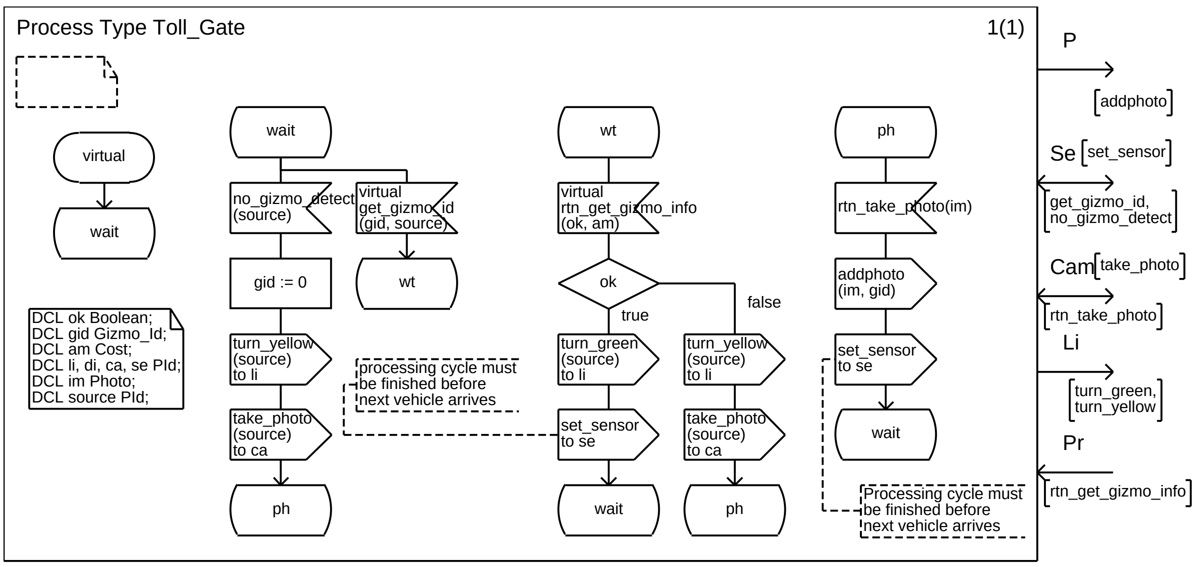

Process Type Toll_Gate
1(1)
virtual
wait
DCL ok Boolean;
DCL gid Gizmo_Id;
DCL am Cost;
DCL li, di, ca, se PId;
DCL im Photo;
DCL source PId;
wait
no_gizmo_detect (source)
virtual get_gizmo_id (gid, source)
gid := 0
wt
turn_yellow (source) to li
take_photo (source) to ca
ph
processing cycle must be finished before next vehicle arrives
wt
virtual rtn_get_gizmo_info (ok, am)
ok
true
false
turn_green (source) to li
set_sensor to se
wait
turn_yellow (source) to li
take_photo (source) to ca
ph
ph
rtn_take_photo(im)
addphoto (im, gid)
set_sensor to se
wait
Processing cycle must be finished before next vehicle arrives
P
[addphoto]
Se [set_sensor]
[get_gizmo_id, no_gizmo_detect]
Cam [take_photo]
[rtn_take_photo]
Li
[turn_green, turn_yellow]
Pr
[rtn_get_gizmo_info]

Figure B.3: Toll abstract superclass

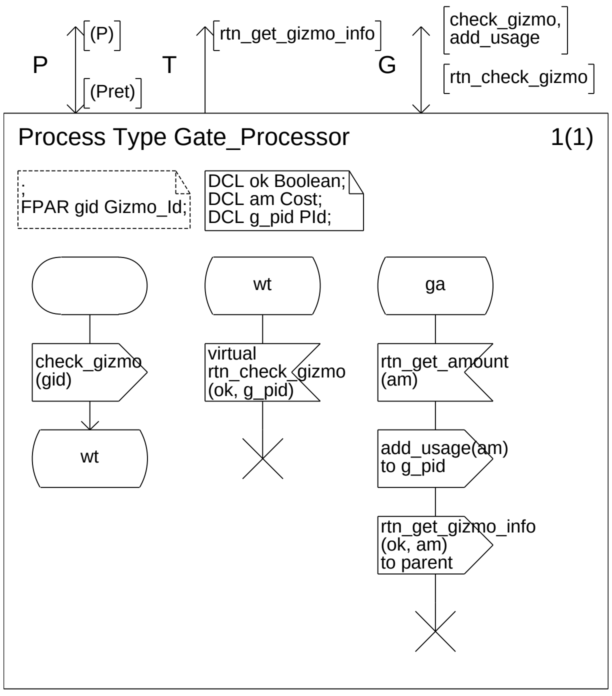


Figure B.4: Gate processor abstract superclass

B.2.1 Single toll

Block `S_Toll` (Figure B.5) shows instances of the single toll gate subclasses `S Gate Pr` (Figure B.6) and `S Toll Gate` (Figure B.7) together with the `Camera`, `Display`, `Sensor` and `Light` components.

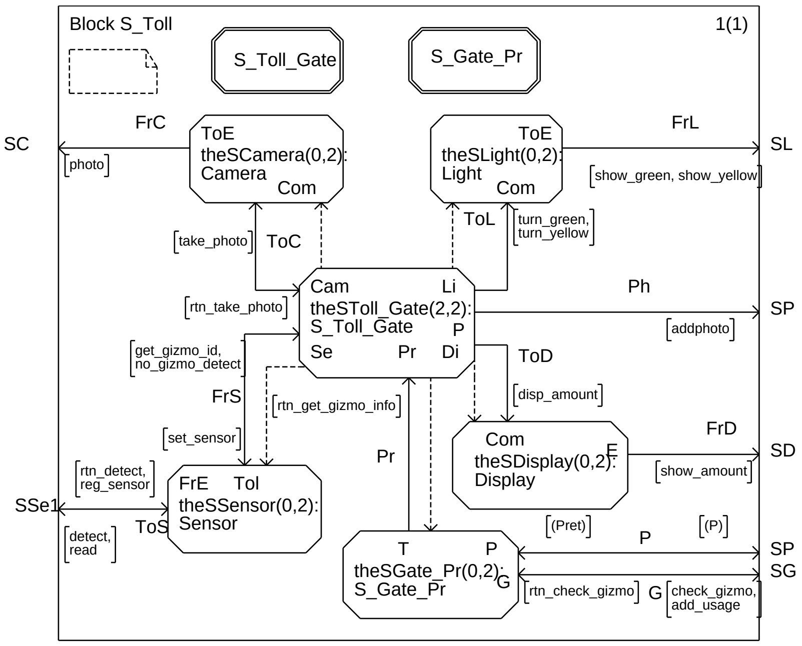


Figure B.5: Composition of `S Toll`

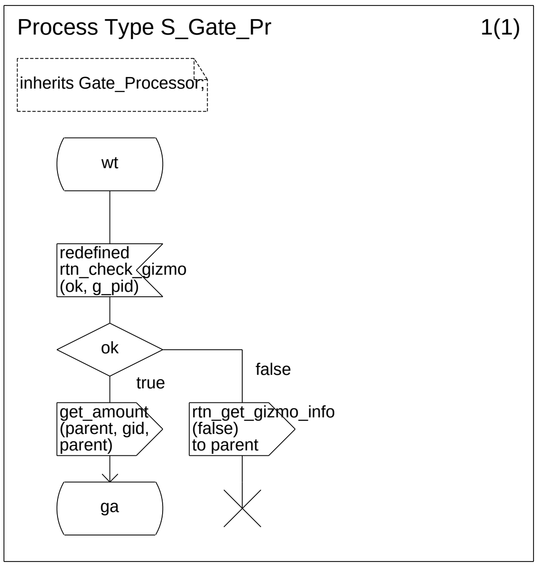

Process Type S_Gate_Pr
1(1)
inherits Gate_Processor;
wt
redefined rtn_check_gizmo (ok, g_pid)
ok
true
false
get_amount (parent, gid, parent)
rtn_get_gizmo_info (false) to parent
ga

Figure B.6: Processing S Toll Gate events

There is a 1:1 association between an `S Toll Gate` object and an object of each of the classes `Camera`, `Display`, `Sensor` and `Light`. The block shows that we have two instances of

S_Toll_Gate which are created statically while all the other objects are created dynamically. The behaviour of S_Toll_Gate is defined to dynamically create one instance of each of its components. This is done within the start transition of process S_Toll_Gate.

Process S_Toll_Gate is a subclass of Toll_Gate and process S_Gate_Pr is a subclass of Gate_Processor. As the Toll_Gate superclass does not have a display, an extra gate is added to process S_Toll_Gate.

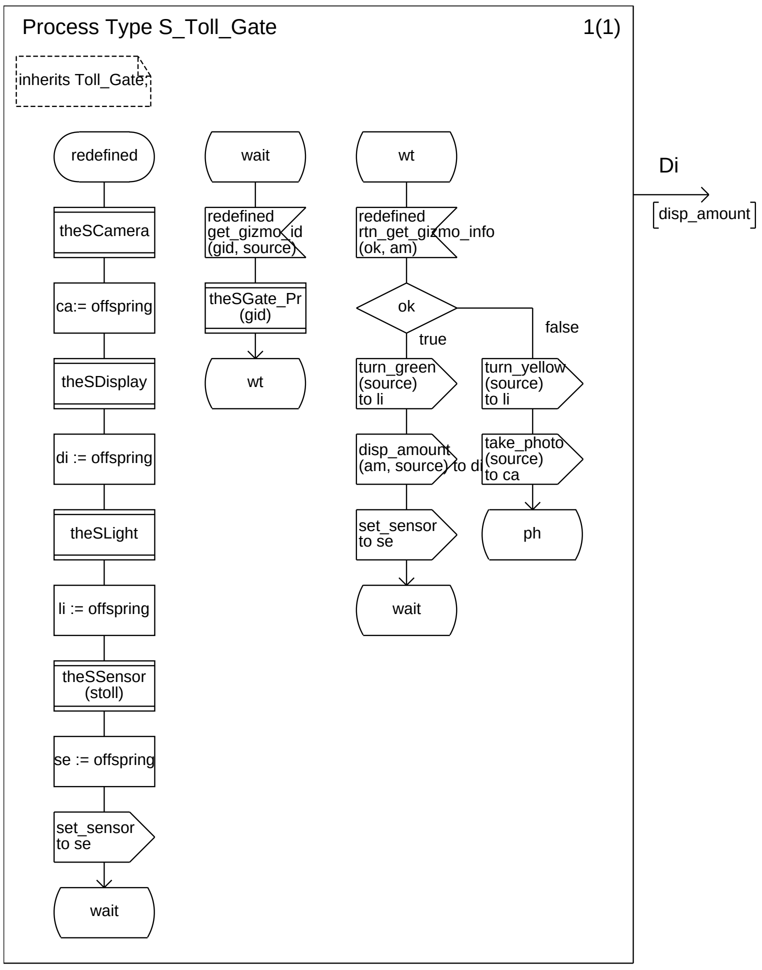


Figure B.7: Composition of S_Toll_Gate

B.2.2 Entry toll

Block `In_Toll` (Figure B.8) shows instances of the entry toll gate subclasses `In_Gate_Pr` (Figure B.9) and `In_Toll_Gate` (Figure B.10) together with their `Camera`, `Sensor` and `Light` components.

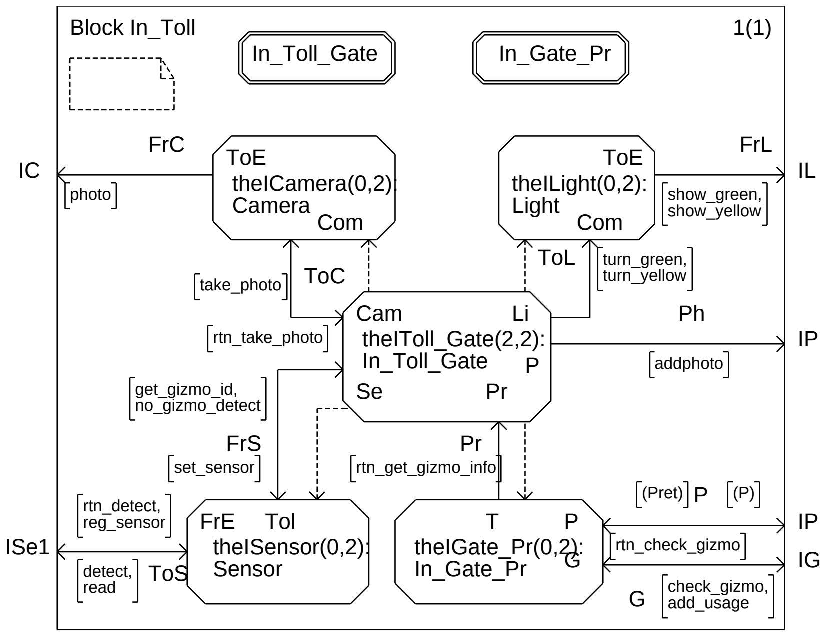


Figure B.8: Composition of `In Toll`

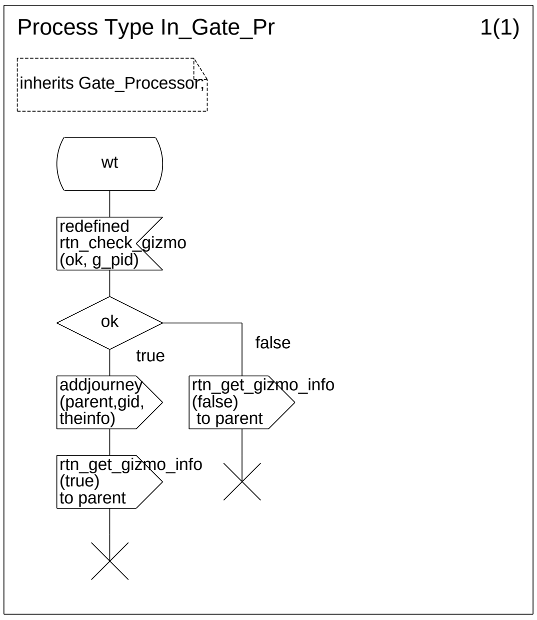

Process Type In_Gate_Pr
1(1)
inherits Gate_Processor;
wt
redefined rtn_check_gizmo (ok, g_pid)
ok
true
false
addjourney (parent,gid, theinfo)
rtn_get_gizmo_info (false) to parent
rtn_get_gizmo_info (true) to parent

Figure B.9: Processing In Toll Gate events

The block shows that we have two instances of `In Toll Gate` which are created statically while all the other objects are created dynamically. The behaviour of `In Toll Gate` is defined to dynamically create one instance of each of its components. This is done within the start transition of process `In Toll Gate`.

Process `In_Toll_Gate` is a subclass of `Toll Gate` and process `In Gate Pr` is a subclass of `Gate_Processor`.

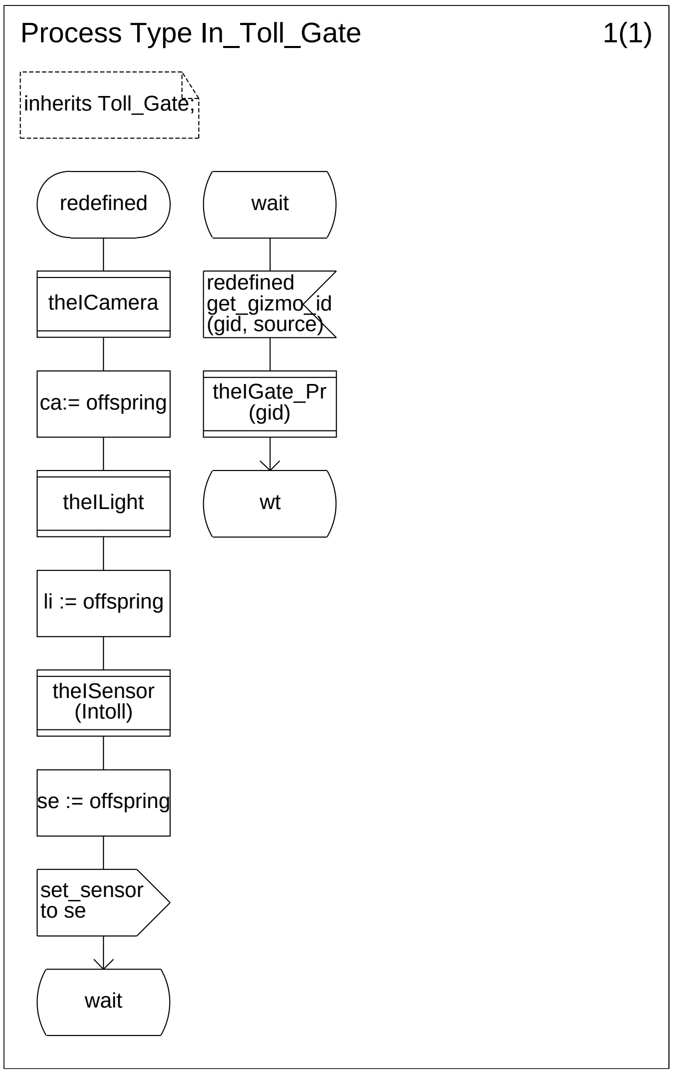


Figure B.10: Composition of `In Toll Gate`

B.2.3 Exit toll

Block `Ex_Toll` (Figure B.11) shows instances of the exit toll gate subclasses `Ex Gate Pr` (Figure B.12) and `Ex Toll Gate` (Figure B.13) together with their `Camera`, `Sensor` and `Light` components.

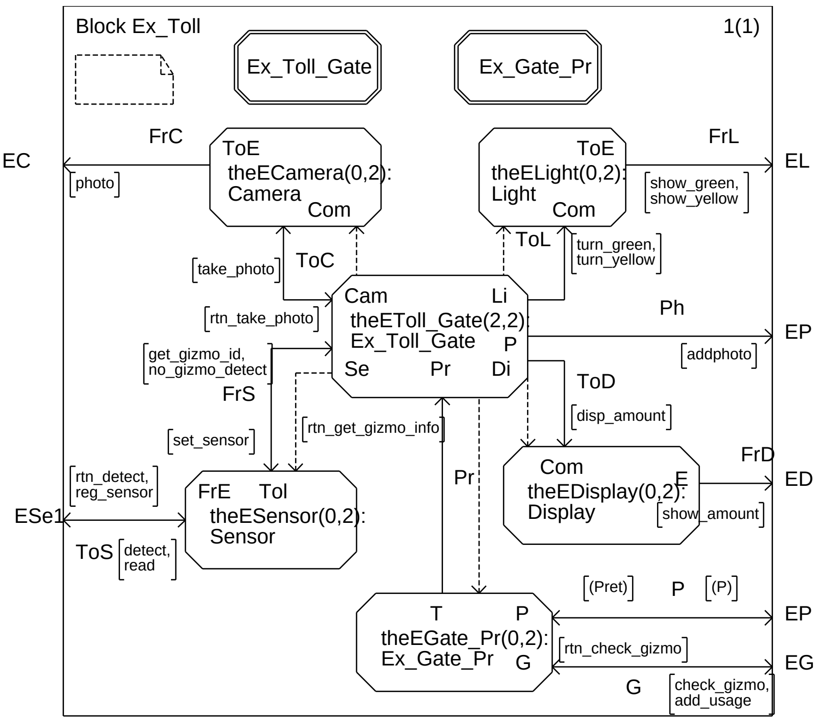


Figure B.11: Composition of `Ex Toll`

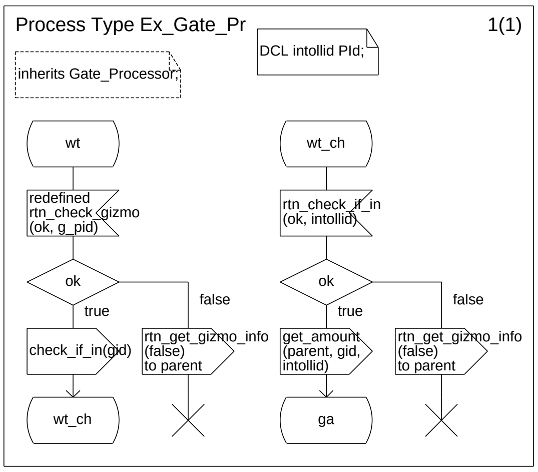

Process Type Ex_Gate_Pr
1(1)
DCL intollid PId;
inherits Gate_Processor;
wt
redefined rtn_check_gizmo (ok, g_pid)
ok
true
false
check_if_in(gid)
rtn_get_gizmo_info (false) to parent
wt_ch
wt_ch
rtn_check_if_in (ok, intollid)
ok
true
false
get_amount (parent, gid, intollid)
rtn_get_gizmo_info (false) to parent
ga

Figure B.12: Processing Exit Toll Gate events

The block shows that we have two instances of `Ex Toll Gate` which are created statically while all the other objects are created dynamically. The behaviour of `Ex Toll Gate` is defined to dynamically create one instance of each of its components. This is done within the start transition of process `Ex Toll Gate`.

Process `Ex_Toll Gate` is a subclass of `Toll Gate` and process `Ex Gate Pr` is a subclass of `Gate_Processor`. As the `Toll Gate` superclass does not have a display, an extra gate is added to process `Ex Toll Gate`.

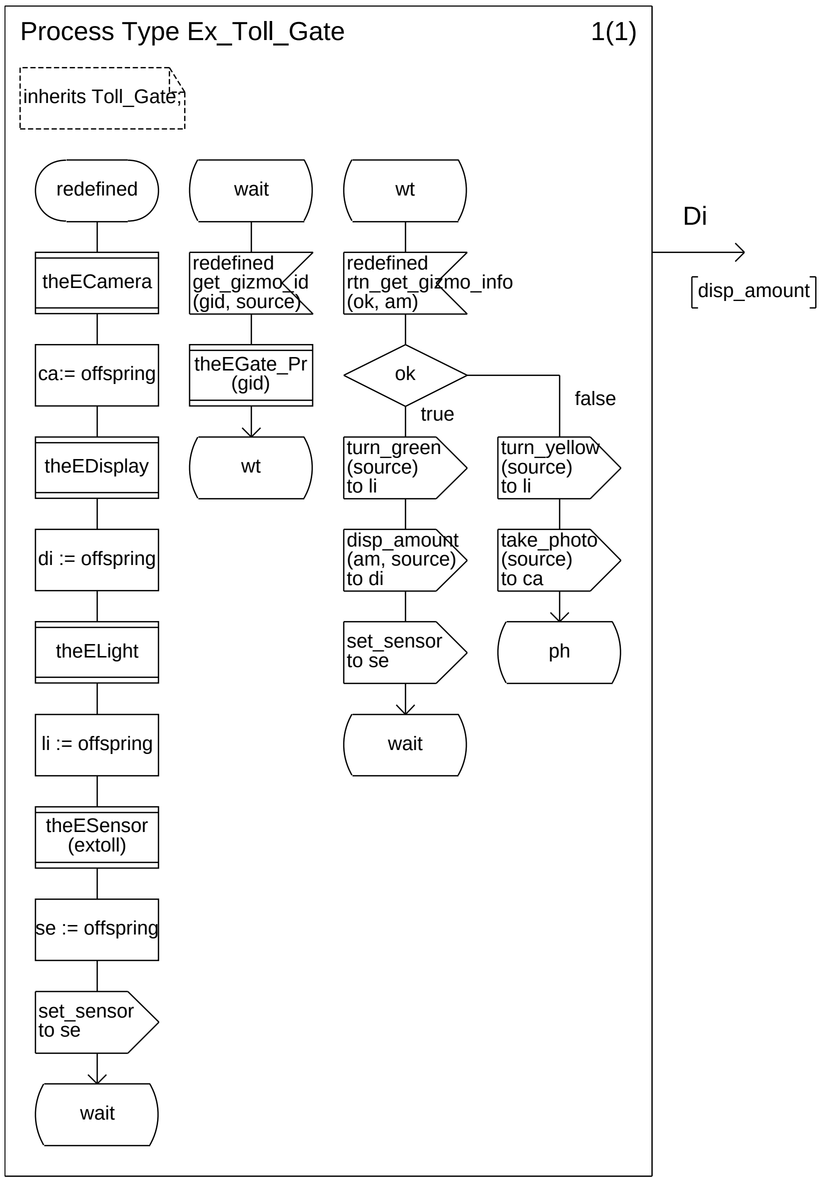


Figure B.13: Composition of `Exit Toll Gate`

B.2.4 Toll components

The toll components are `Camera`, `Display`, `Sensor` and `Light`. Their process definitions are represented in Figures B.14, B.15, B.16 and B.17, respectively.

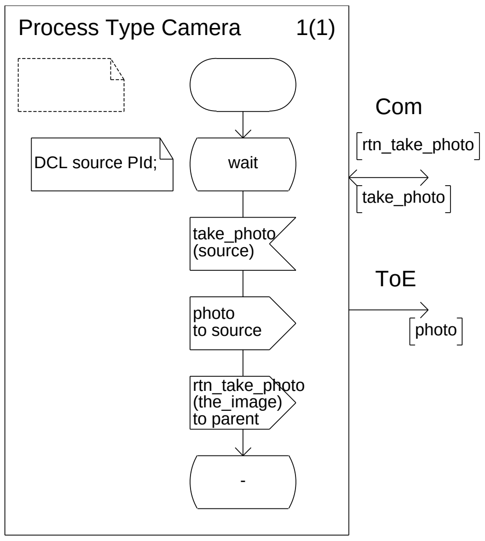


Figure B.14: Camera

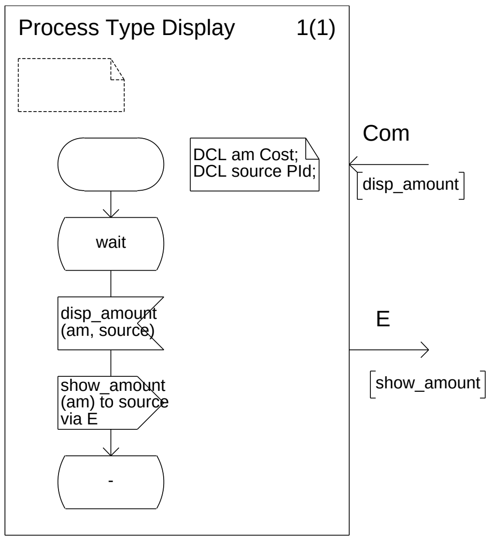


Figure B.15: Display

The sensor is the only component that is accessed by the environment. When a sensor is created, it sends a message to the environment to indicate that it is available to receive signals. The sensor uses synchronous communication with `detect` (it receives a `detect` signal and returns a `rtn_detect` to ensure that the `read` signal is sent from the same source as the `detect`. The `set_sensor` signal from the toll gate ensures that new `detect` signals are only handled once the previous `detect` has been dealt with.

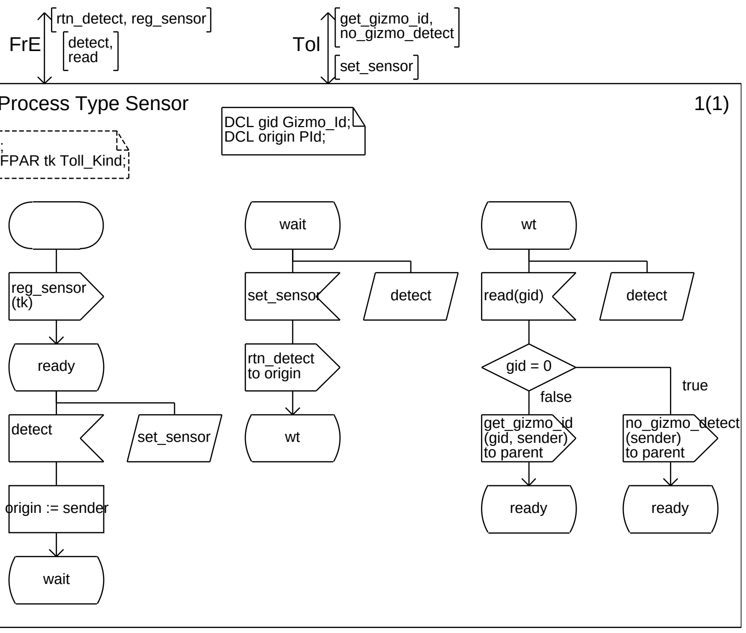


Figure B.16: Sensor

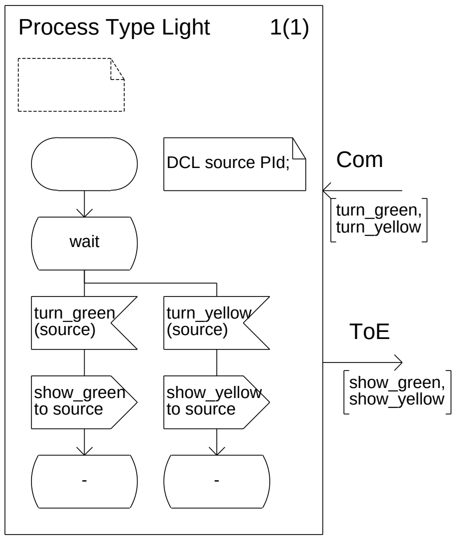


Figure B.17: Light

B.3 Processing passing a toll

The block `Process pass`, depicted in Figure B.18, contains objects which hold information about vehicles that have passed though toll gates.

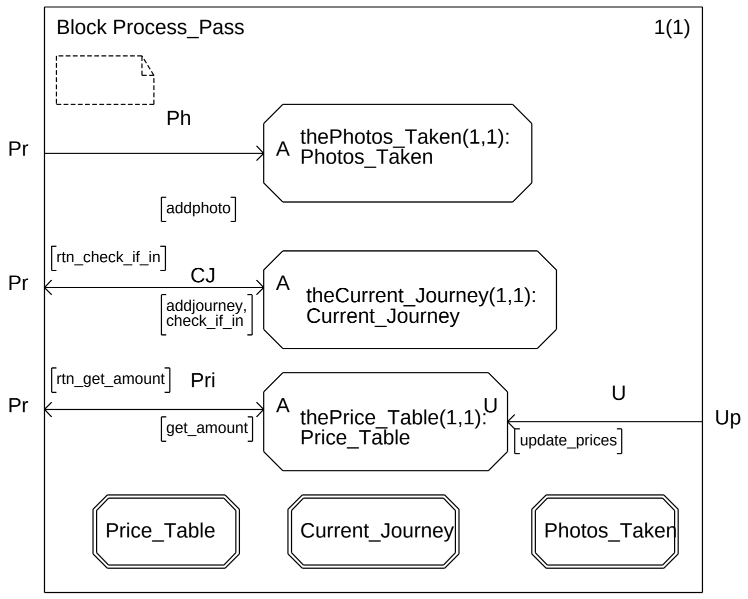


Figure B.18: Block to process passing toll gate

The interface to process `Price Table` (Figure B.19) gives the necessary information so that different prices can be returned. However, as our interest is in the interface to classes rather than their internal details, it always returns the same value.

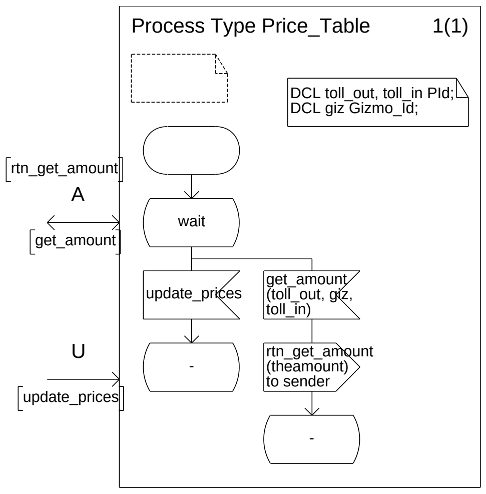


Figure B.19: Price table

There is a single instance of process `Current Journey` (Figure B.20) which maintains the set of gizmo identifiers which have entered, but not yet left a motorway. Again, as our interest is in understanding the problem rather than carrying out detailed calculations, it does not store the identity of the entry toll gate.

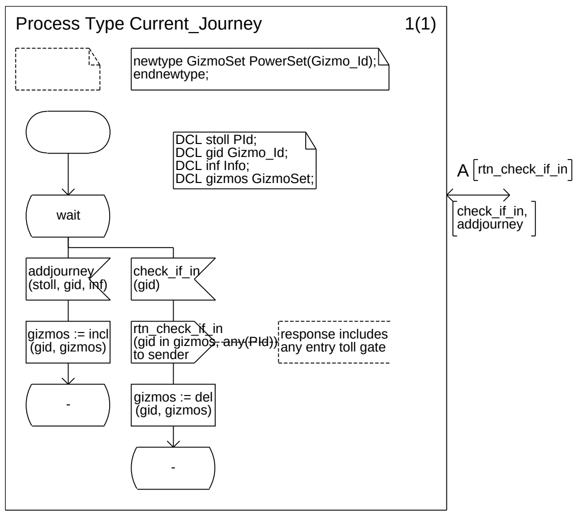

Process Type Current_Journey
1(1)
newtype GizmoSet PowerSet(Gizmo_Id);
endnewtype;
DCL stoll PId;
DCL gid Gizmo_Id;
DCL inf Info;
DCL gizmos GizmoSet;
wait
addjourney (stoll, gid, inf)
check_if_in (gid)
gizmos := incl (gid, gizmos)
rtn_check_if_in (gid in gizmos, any(PId)) to sender
response includes any entry toll gate
gizmos := del (gid, gizmos)
-
-
A
rtn_check_if_in
check_if_in, addjourney

Figure B.20: Current journey

There is a single instance of process `Photos Taken` (Figure B.21). It maintains a bag of photos containing the gizmo identifiers of those who have invalidly gone through a toll gate.

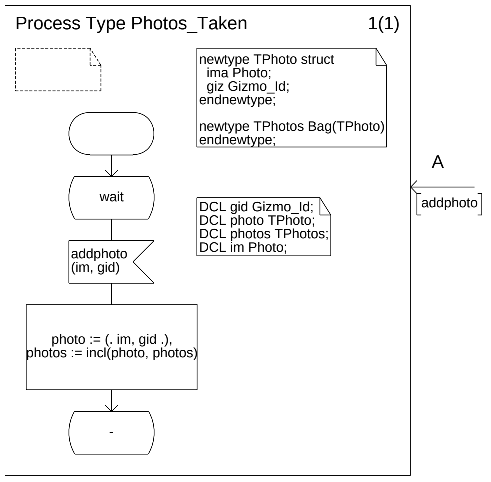


Figure B.21: Photos taken

B.4 Registration

The block Control Reg (Figure B.22) deals with the registration of a gizmo. There are two objects which are concerned with interaction with the environment: Reg Desk (Figure B.23) and Bank_Intf Reg (Figure B.28).

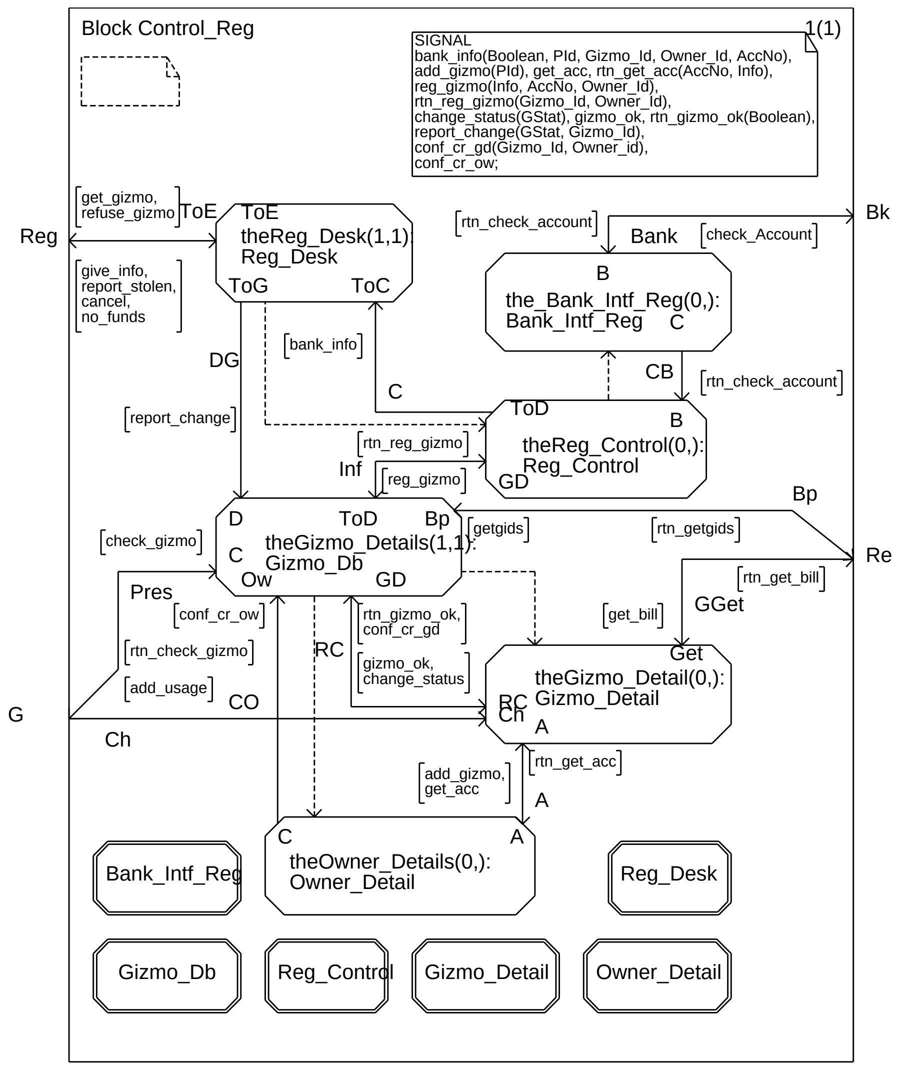

Block Control_Reg
1(1)
SIGNAL
bank_info(Boolean, PId, Gizmo_Id, Owner_Id, AccNo),
add_gizmo(PId), get_acc, rtn_get_acc(AccNo, Info),
reg_gizmo(Info, AccNo, Owner_Id),
rtn_reg_gizmo(Gizmo_Id, Owner_Id),
change_status(GStat), gizmo_ok, rtn_gizmo_ok(Boolean),
report_change(GStat, Gizmo_Id),
conf_cr_gd(Gizmo_Id, Owner_id),
conf_cr_ow;
Reg
get_gizmo, refuse_gizmo
ToE
give_info, report_stolen, cancel, no_funds
theReg_Desk(1,1): Reg_Desk
ToG
ToC
DG
bank_info
C
report_change
rtn_check_account
Bank
check_Account
Bk
B
the_Bank_Intf_Reg(0,): Bank_Intf_Reg
CB
rtn_check_account
ToD
theReg_Control(0,): Reg_Control
GD
rtn_reg_gizmo
Inf
reg_gizmo
Bp
D
ToD
theGizmo_Details(1,1): Gizmo_Db
Ow
getgids
rtn_getgids
Re
check_gizmo
Pres
conf_cr_ow
rtn_check_gizmo
add_usage
CO
rtn_gizmo_ok, conf_cr_gd
gizmo_ok, change_status
RC
get_bill
rtn_get_bill
GGet
Get
theGizmo_Detail(0,): Gizmo_Detail
Ch
A
G
add_gizmo, get_acc
rtn_get_acc
theOwner_Details(0,): Owner_Detail
Bank_Intf_Reg
Reg_Desk
Gizmo_Db
Reg_Control
Gizmo_Detail
Owner_Detail

Figure B.22: Registration block

The interface objects communicate with an object of the control class Reg Control (Figure B.24) which acts as the central controller. A separate instance of this class is created to deal with each registration.

Class Gizmo_Db (Figure B.25) provides the static interface to the information about owners or gizmos and maintains the set of gizmo and owner identifiers. Objects of this kind are sometimes known as *traders*. It dynamically creates instances of Gizmo Detail (Figure B.26) and Owner_Detail (Figure B.27) which is where the actual information is held. Objects of both classes Gizmo Detail and Owner Detail confirm to Gizmo Db when they have executed their start transition and so are ready to receive signals. Gizmo Detail is not created until the

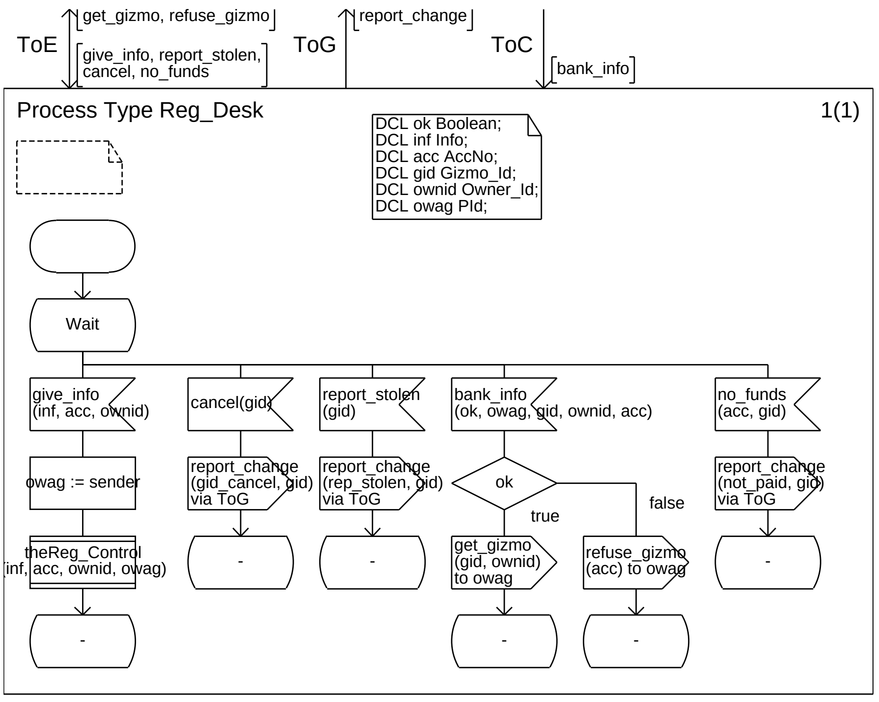

ToE
get_gizmo, refuse_gizmo
give_info, report_stolen, cancel, no_funds
ToG
report_change
ToC
bank_info
Process Type Reg_Desk
1(1)
DCL ok Boolean;
DCL inf Info;
DCL acc AccNo;
DCL gid Gizmo_Id;
DCL ownid Owner_Id;
DCL owag PId;
Wait
give_info (inf, acc, ownid)
cancel(gid)
report_stolen (gid)
bank_info (ok, owag, gid, ownid, acc)
no_funds (acc, gid)
owag := sender
report_change (gid_cancel, gid) via ToG
report_change (rep_stolen, gid) via ToG
ok
true
false
report_change (not_paid, gid) via ToG
theReg_Control (inf, acc, ownid, owag)
get_gizmo (gid, ownid) to owag
refuse_gizmo (acc) to owag
-

Figure B.23: Registration desk

existence of the relevant `Owner Detail` object has been confirmed and so is ready to receive an `add_gizmo` signal.

SDT does not allow `PId`s to be passed as parameters from the environment. Owner and gizmo identifiers are therefore held externally as integers and `Gizmo Db` holds the conversion table between these integers and the `PId`s of the `Gizmo Detail` and `Owner Detail` processes.

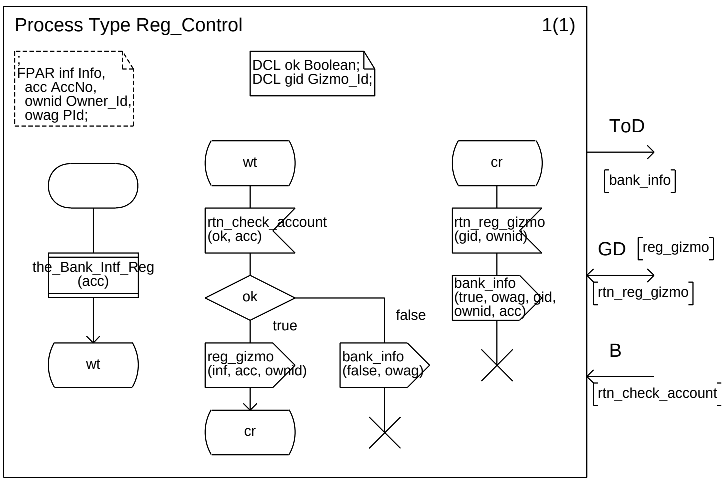
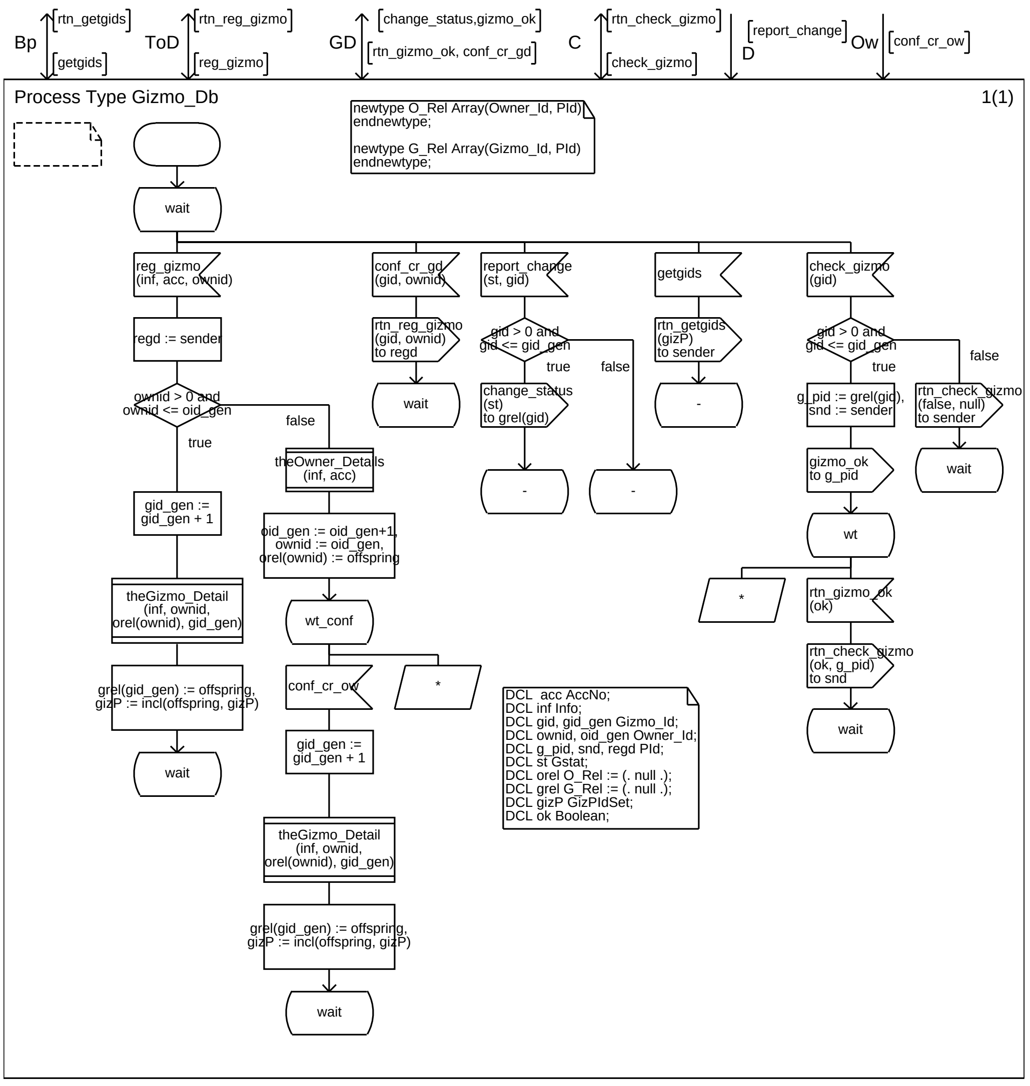


Figure B.24: Registration control

By saving the PId of the process sending the give info signal, we can ensure that Reg_Desk sends a get gizmo or refuse gizmo signal to the correct owner. In the system-centred model, it is the environment which sends the give info signal and so this is only conceptual. However, in the combined model this ensures that communication is between the correct process instances.



Bp
rtn_getgids
getgids
ToD
rtn_reg_gizmo
reg_gizmo
GD
change_status,gizmo_ok
rtn_gizmo_ok, conf_cr_gd
C
rtn_check_gizmo
check_gizmo
report_change
D
Ow
conf_cr_ow
Process Type Gizmo_Db
1(1)
newtype O_Rel Array(Owner_Id, PId)
endnewtype;
newtype G_Rel Array(Gizmo_Id, PId)
endnewtype;
wait
reg_gizmo (inf, acc, ownid)
regd := sender
ownid > 0 and ownid <= oid_gen
true
false
gid_gen := gid_gen + 1
theGizmo_Detail (inf, ownid, orel(ownid), gid_gen)
grel(gid_gen) := offspring, gizP := incl(offspring, gizP)
wait
theOwner_Details (inf, acc)
oid_gen := oid_gen+1, ownid := oid_gen, orel(ownid) := offspring
wt_conf
conf_cr_ow
*
gid_gen := gid_gen + 1
theGizmo_Detail (inf, ownid, orel(ownid), gid_gen)
grel(gid_gen) := offspring, gizP := incl(offspring, gizP)
wait
conf_cr_gd (gid, ownid)
rtn_reg_gizmo (gid, ownid) to regd
wait
report_change (st, gid)
gid > 0 and gid <= gid_gen
true
false
change_status (st) to grel(gid)
-
-
getgids
rtn_getgids (gizP) to sender
-
check_gizmo (gid)
gid > 0 and gid <= gid_gen
true
false
g_pid := grel(gid), snd := sender
gizmo_ok to g_pid
wt
*
rtn_gizmo_ok (ok)
rtn_check_gizmo (ok, g_pid) to snd
wait
rtn_check_gizmo (false, null) to sender
wait
DCL acc AccNo;
DCL inf Info;
DCL gid, gid_gen Gizmo_Id;
DCL ownid, oid_gen Owner_Id;
DCL g_pid, snd, regd PId;
DCL st Gstat;
DCL orel O_Rel := (. null .);
DCL grel G_Rel := (. null .);
DCL gizP GizPIdSet;
DCL ok Boolean;

Figure B.25: Gizmo interface

In process Bank_Intf_Reg, we want to ensure that check account is sent to the correct bank view. In SDL, we can do this by sending the appropriate PId as a parameter of give_info. However, SDT does not allow PIds to be passed as parameters to or from the environment and so this is not possible in the system-centred model, but it is possible in the combined model where the the signals are internal.

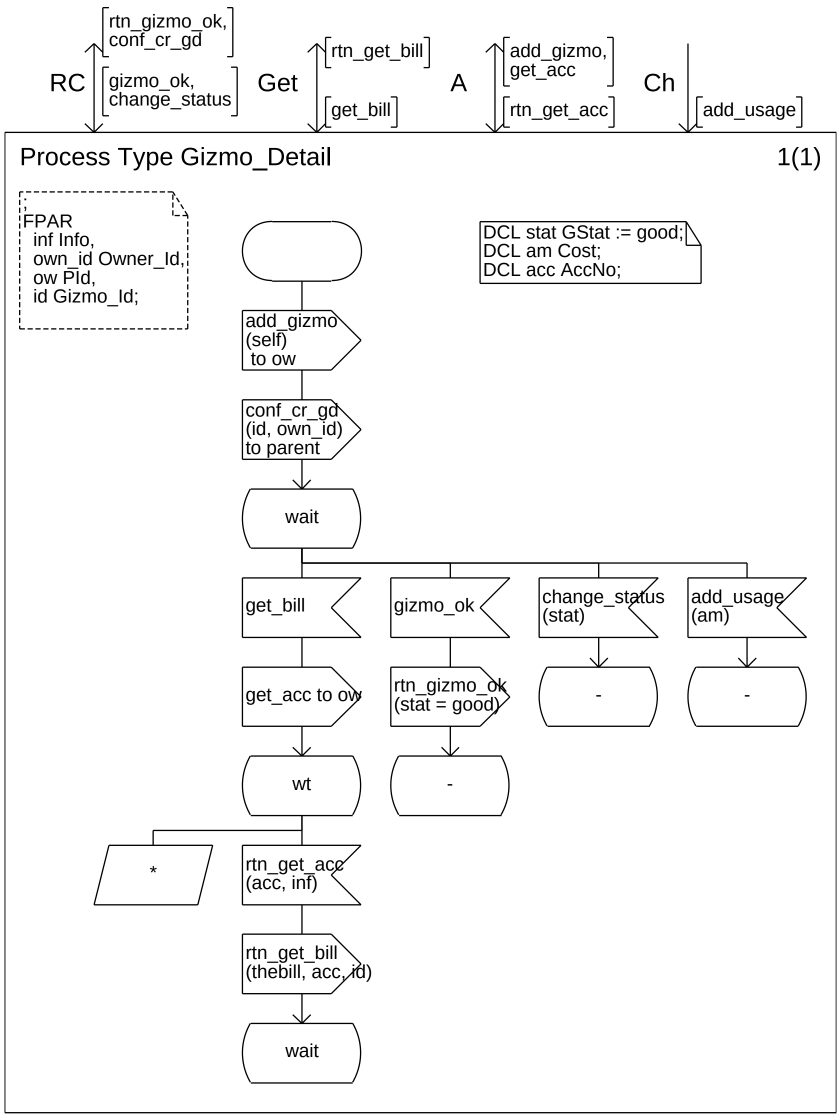


Figure B.26: Gizmo detail

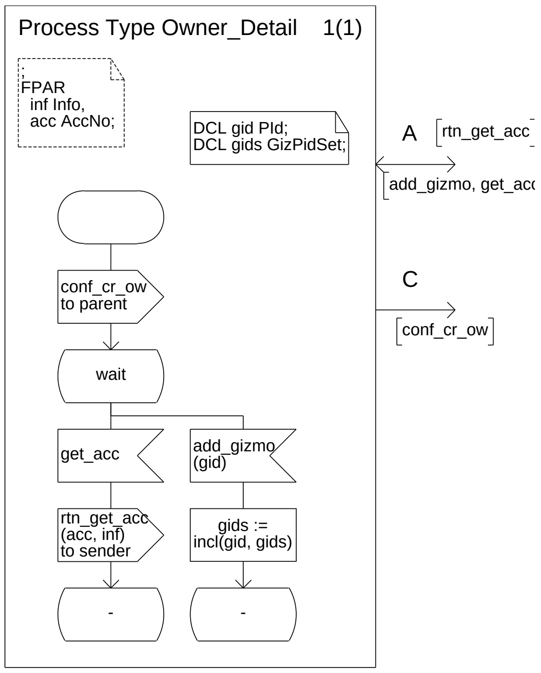


Figure B.27: Owner details

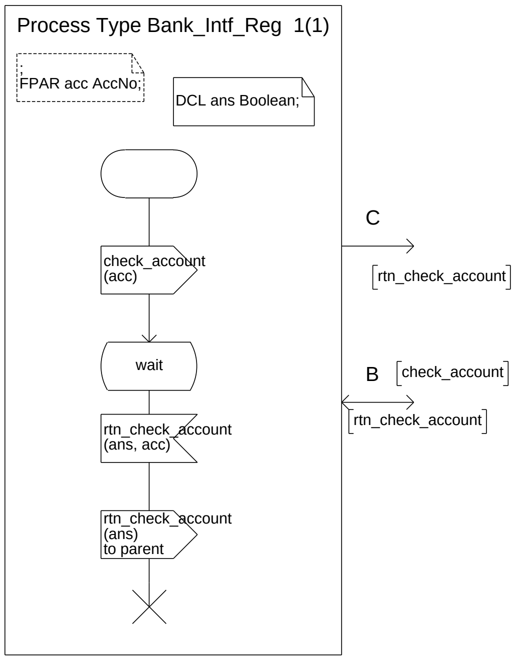


Figure B.28: Bank interface

B.5 Debits

Block Debits (Figure B.29) consists of two interface classes Op Interface (Figure B.30) and Bank_Intf_Acc (Figure B.31) and a control class Bill Processor (Figure B.32).

Op_Interface receives messages from the operator. The most significant is monthly debit which initiates the sending of bills to all owners who have used the system in the last month. It achieves this by creating a Bill Processor control object which handles the actual processing. Bill Processor gets the set of gizmo identifiers from Gizmo Db and deals with them one at a time sending the debits to the relevant bank accounts through the Bank Intf Acc interface. Once it has finished, Bill Processor reports back to Op Interface. This ensures that only a single instance of Bill Processor can exist at any one time.

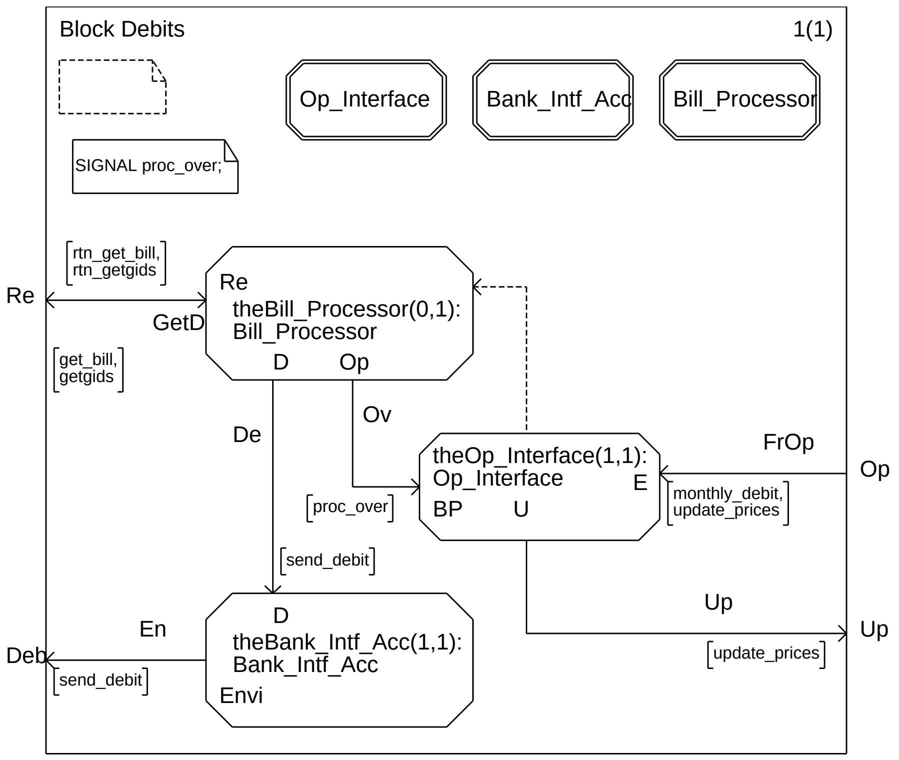


Figure B.29: Debits

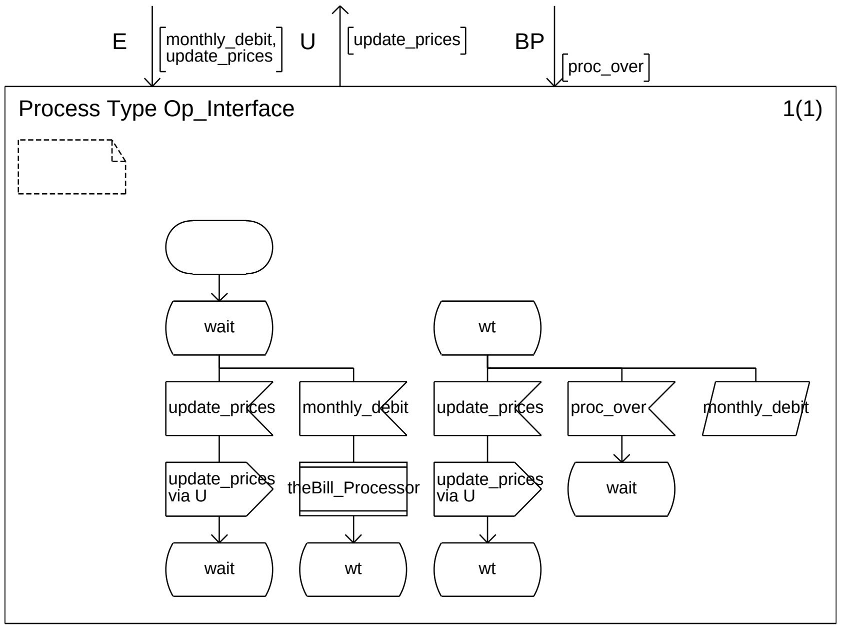


Figure B.30: Operator interface

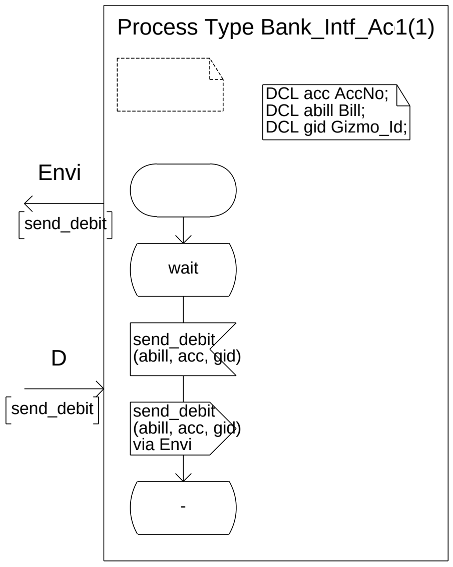

Process Type Bank_Intf_Ac1(1)
DCL acc AccNo;
DCL abill Bill;
DCL gid Gizmo_Id;
Envi
send_debit
wait
D
send_debit
send_debit (abill, acc, gid)
send_debit (abill, acc, gid) via Envi
-

Figure B.31: Bank interface

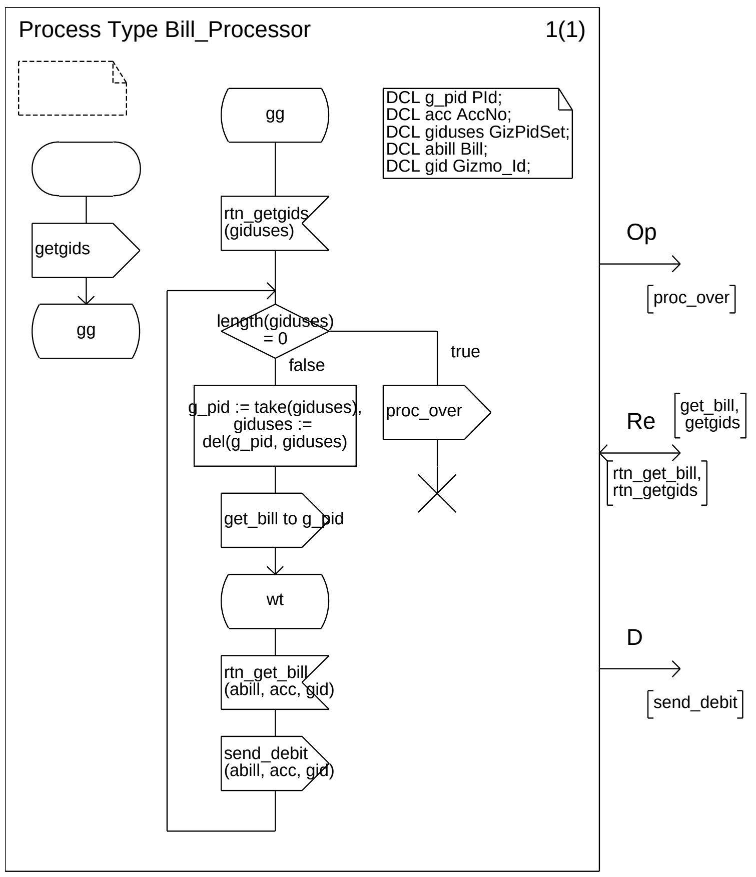


Figure B.32: Bill processing