A UML-Based Method for the Commanded Behaviour Frame

C. Choppy
LIPN, Université Paris XIII
99 Avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Gianna Reggio
DIS - Università di Genova
Via Dodecaneso 35, 16146, Italy
reggio@disi.unige.it

Abstract

In this paper we show how to assist the development for problems fitting the Commanded Behaviour frame, using UML with a well-founded approach, and we illustrate our ideas with a lift case study.

1 Introduction

The problem frames introduced by Mickael Jackson in [11, 12] provide diagrams, notations and concepts to describe frequently met problem structures of different kinds, and thus are a nice way to help to start structuring the problem under study. Trying to match one of the problem frames provided by Jackson [12] to the problem under study yields some clarification on the kind of system that is to be developed (note that failing to match a problem frame may also shed some light on the work to be done). As Jackson puts it [11], once the problem frame fitting is achieved then the corresponding appropriate development method should be available.

We found this idea very interesting and worked with it in several directions, in relationships with formal specifications [7, 8] and with UML [9]. In both cases, for some basic problem frames, our work shows how to associate to the different parts of the problem frame some specification detailed schema.

This workshop on problem frames gives us the opportunity to show how these approaches can be combined on the Commanded Behaviour Frame.

In [7], we provided formal specification skeletons in the CASL and CASL-LTL specification languages [2, 14] associated with problem frames given in [11] (the Translation/JSP frame, and the Information System - IS - frames). In [8], we developed a general formally grounded specification methodology associated with the CASL and CASL-LTL languages, and briefly showed our to relate it with some basic problem frames given in [11] (Translation/JSP, IS, Control). In [9], our motivation was to show how to relate problem frames of [12] with a well founded methodology for UML descriptions, and we worked on the Transformation, the Commanded Information and the Required Behaviour frames.

In this paper, we turn our attention to the Commanded Behaviour Frame, and show how to assist the development for problems fitting this frame, using UML with a well-founded approach.

There are some drawbacks with the use of UML [15]. While it provides a nice variety of constructs, they are redundant, and it may be difficult to choose which are appropriate. There are no means to fully insure the consistency between the different views used to build a model, and, moreover, the UML semantics is both informal and problematic. However, in [3, 4], it has been shown that UML may be used in a development method in a quite precise, structured and well-founded way, avoiding most of its typical problems (e.g., only a subset of UML with a semantics that may be formally given is used).

Here, taking advantage of the formal treatment made for some problem frames in [7], we propose for this problem frame a development method using UML together with precise guidelines for the users, by tailoring the general one proposed in [3, 4]. These help to reduce the development time and prevent loosing time searching how to model the various aspects. This lead us also to discover how to properly handle, using UML, many kind of applications that do not easily fit inside the standard business case/use case approach, as proposed by many UML-based methods (e.g., RUP [13] and COMET [10]).

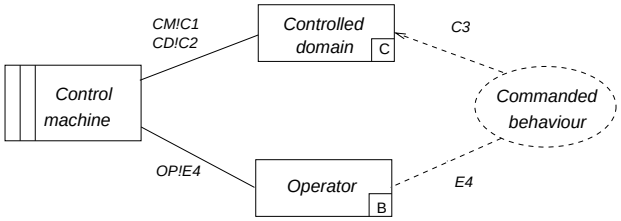
Following [3, 4], the development of an application requires to produce a *Model of the Domain* of the application (i.e., those aspects of the real world that are relevant for the application to be developed for providing a solution to the problem under consideration), then a *Requirement Specification*, and finally a *Design Specification*, each of them being a UML model with a precise structure. Here, we present how to produce them for the particular case of

the Commanded Behaviour Frame using the various parts of the frame, introduced in Sect. 2.

2 The Commanded Behaviour Frame

The problem frame

Jackson [12] describes this problem frame (sketched in the diagram below) as follows. "There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly."



Let us recall that the problem frames distinguish the domain parts (drawn as rectangles), the requirements (as a dashed oval), and the design (as a double striped rectangle), which are connected through interfaces. Events (or phenomena) occur at the interfaces.

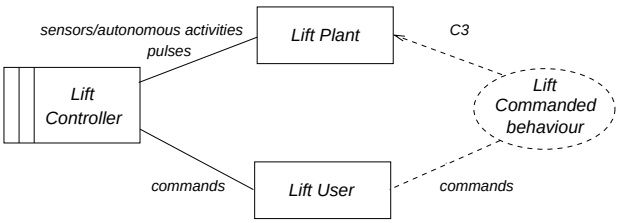
The C in the frame diagram indicates that the domain *Controlled domain* must be causal. The machine is always a causal domain (so an explicit C is not needed). The B stands for "biddable" and is used for domains that are people. The phenomena *E4* are the operator commands.

Running Example: the Lift System

The lift case study is used throughout this paper to illustrate our approach.

A lift system consists of a *lift plant* (that is the cabin, the motor moving it and the doors at the various floors), some software automatically controlling the lift functioning (the *controller*), and the people using it (the *users*). The controller monitors the lift plant by means of *sensors*, which communicate the status of its various components (e.g., there is a sensor detecting the position of the cabin), and directs its behaviour by means of *orders* (e.g., it can order to open/close the doors).

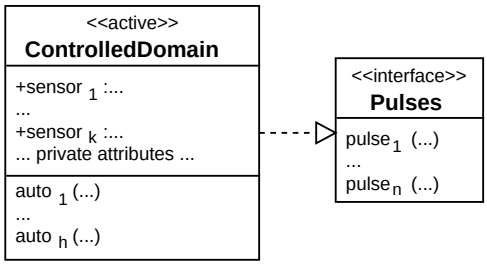
The lift matches the Commanded Behaviour frame as follows. The *Operator* is the lift user (further denoted by *User*), the *Control machine* is the lift controller (that is the software controlling the lift), and the *Controlled domain* is the lift plant (that is the cabin, the motor, and the doors at the floors), further denoted by *LiftPlant*.



3 Domain Model

The UML model

The domain model in the case of the Commanded Behaviour Frame concerns the *Controlled domain* and is described by a UML model that includes the following fragment.[1]



The *Controlled domain* is equipped with some sensors, which are modelled by the public attributes $\mathsf{sensor}_1, \ldots, \mathsf{sensor}_k$)[2], and is controlled by sending it some pulses, which are modelled by the operations of the interface Pulses. However, a *Controlled domain* may change its state and the way it works even if it does not receive a pulse (for example, when a part breaks down or when some external entity acts over it). These "autonomous" activities are modelled by means of calls of the operations $\mathsf{auto}_1, \ldots, \mathsf{auto}_h$.

To describe the state of the *Controlled domain* other private attributes may be used. Moreover, some sensors may signal a value derived by those of other attributes. In this case the UML model should contain invariant constraints defining those sensors in terms of the other attributes.

The behaviour of the class *Controlled domain* is then modelled by a statechart (named *Controlled domain behaviour*)

– whose events are either time events or call events built by the operations $\mathsf{pulse}_1, \ldots, \mathsf{pulse}_n, \mathsf{auto}_1, , \ldots, \mathsf{auto}_h$,

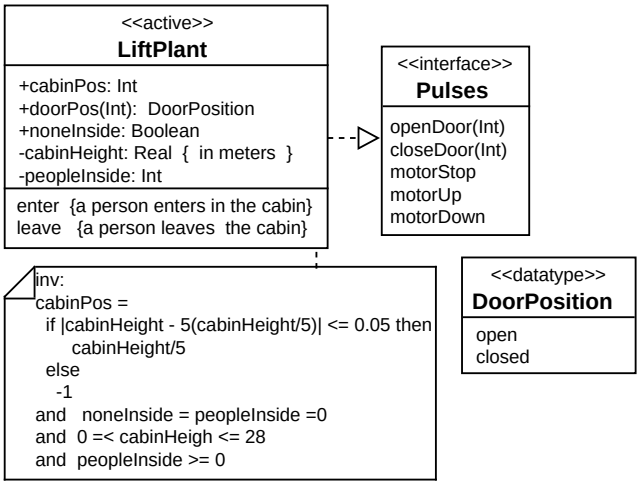
– and whose conditions and actions examine and update its attributes.

[1] A UML interface (keyword ≪interface≫) is a named set of operations. A dashed arrow from a class C to an interface I denotes that C will call the operations of I, whereas a dashed arrow with a solid head from I to C denotes that C will realizes I.

[2] The fact that the sensors may break down is modelled by assuming that they may contain some special values corresponding to failures.

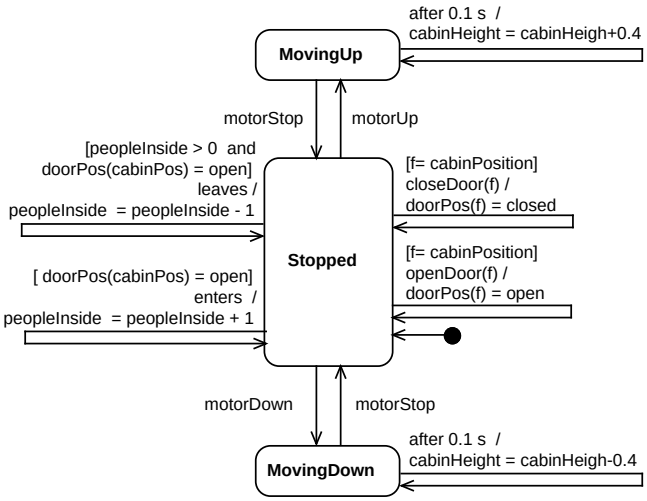
Lift Case Study: Domain Model

The *Controlled domain* in the lift case study is the lift plant, and is modelled by the class LiftPlant presented below.



In the lift there is a sensor revealing the position of the cabin (the floor number when it is at a floor, -1 when it is between two floors), the position of the doors at the various floors (open or closed) and if there is someone inside the cabin. Two private attributes record the actual height of the cabin from the ground and how many persons are inside the cabin. The pulses may require to open or close the door at some floor, and to stop, move up or down the motor that moves the cabin.

The behaviour of the class LiftPlant is described by the following statechart.



Note how the above diagram shows that the doors can be opened/closed only when the cabin is at the corresponding floor, and that people may enter/leave the cabin only when the cabin is at some floor with open door; thus these security features will be not under the responsibility of the software controller. The fact that following the problem frame approach the developer is obliged to explicitly consider and describe the existing parts of the real world interacting with the system to be developed is one of the most valuable aspect of this approach.

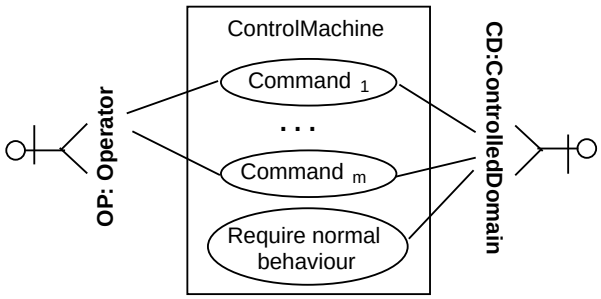
4 Requirement Specification

4.1 The UML Model

The requirement specification corresponds to the parts of the frame *Commanded behaviour* and *Operator*.

In this case use cases are suitable to summarize the requirements, thus the Requirement Specification is a UML model consisting of a use case diagram, a class diagram and various use case descriptions, one for each use case appearing in the diagram.

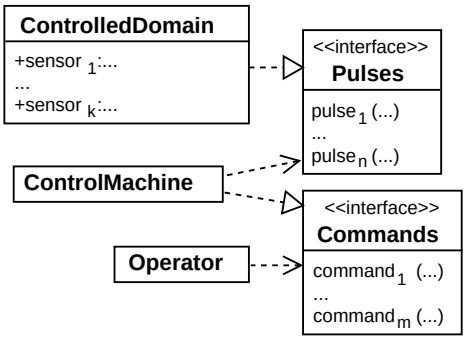
Use Case Diagram



The *Control machine* is the application to develop, and the use cases $\mathsf{Command}_1, \ldots, \mathsf{Command}_m$ correspond to the possible commands sent by the *Operator* to the *Control machine*. The last use case, Require normal behaviour, models the fact that usually also in absence of commands from the *Operator* the *Control machine* must ensure a certain kind of behaviour; obviously, if the *Control machine* must act only as a consequence of a command, this use case may be dropped. Recall that following [3, 4] we consider as actors all those entities interacting with the application, and not only those taking advantage of its services; thus also the Controlled domainis an actor.[3]

Class Diagram This class diagram, corresponding to the Context View of [3, 4], should contain the following fragment.

[3] In [3, 4] a different icon (parallelogram) is used for these "secondary" actors.



This diagram introduces three classes modelling the three corresponding parts of the problem frame and shows which are the mutual interfaces. We have that the ControlMachine interacts with the Controlled domain by sending it pulses (interface Pulses) and by accessing the "sensor" attributes, and with the Operator by receiving its commands (interface Commands); it is assumed that all commands are always correct and acceptable, and thus no error messages should be sent back from the *Control machine* to the *Operator*.[4]

Use Case Description In [7] the *Commanded behaviour* is described in terms of desired properties on the behaviour of the *Controlled domain*, whereas [12] speaks of rules; but, to describe properties in UML is problematic. Indeed, the constraints (invariants for classes and pre-post for operations), written for example in OCL, can express a very limited form of properties on the behavior of a class. In the literature there are several proposals for extending OCL with temporal combinators (see, e.g., [6, 16]), but none of them has either reached the current practice or has been accepted as a standard. Thus, we do not think that it will be possible to propose a sensible UML-based method where the commanded behaviour will be modelled using properties.

On the other hand, the standard techniques to describe use cases focussing on showing the possible scenarios of interactions between the application to be developed and the actors (using natural language, or UML sequence/ collaboration diagrams, or statecharts associated with the class modelling the application, as proposed by [3, 4]) are not very suitable in this case. Indeed, each use case has just a unique trivial scenario "The operator sends $\mathsf{command}_i$ to the *Control machine*, which in turns causes the *Controlled domain* to behave in the following way . . . ".

We think that, instead, a more appropriate way is to present the commanded behaviour by means of a statechart associated with the class Controlled domain[5] such that

– the states are the same of the statechart *Controlled domain behaviour* included in the Domain Model (see Sect. 3),

– the events are either timed events or change events concerning its attributes,

– the conditions concern only its attributes,

– and the actions are updates of its attributes.

Indeed, the required behaviour of the *Controlled domain* is usually expressed in terms of *activities* to be done

– at certain time, they will be triggered by the timed events,

– or when something changes inside itself (e.g., due to failures or to acts by external entities), they will be triggered by the change events.

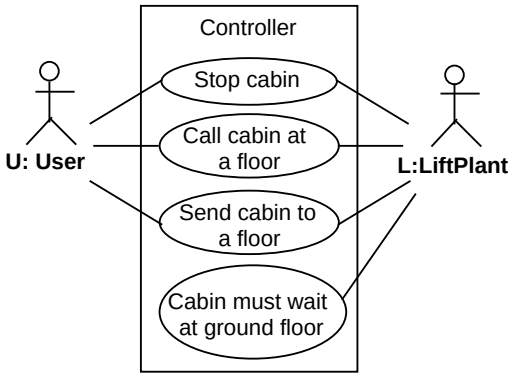
Moreover, these activities consists of modification of the internal state of the *Controlled domain*.

Whenever the commanded behaviour depends on the actual situation of the *Controlled domain* we need to give a statechart, having the form describe above, for each state (of the statechart modelling the *Controlled domain*, see Sect. 3) in which the command has some effect.

The use case Require normal behaviour is described by statecharts, similar to those used for the other use cases, modelling the behaviour that the *Control machine* must guarantee in absence of commands sent be the *Operator*, and by a set of invariant constraints presenting the safety conditions that the *Control machine* must always guarantee.

4.2 Lift Case Sutdy: Requirement Specification

Use case diagram

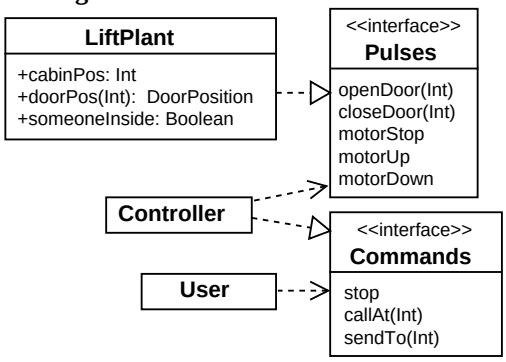


The lift user may ask to the cabin to stop (pressing some button inside the cabin), send the cabin to some floor pressing again a button inside the cabin, and call the cabin to a floor by pressing a button at the door of this floor. When no one is using the lift, then cabin should go back to the ground floor.

[4] Variants of this frame may be developed, where commands may be refused in some situation and the *Operator* may make mistakes in sending the commands.

[5] Similarly [12] proposes on an example to use generic state machines.

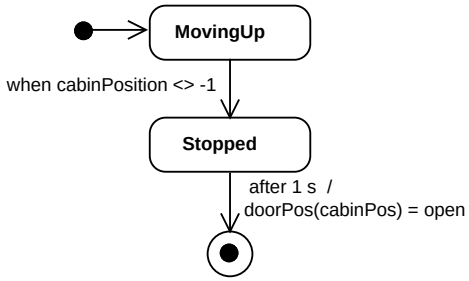
Class diagram

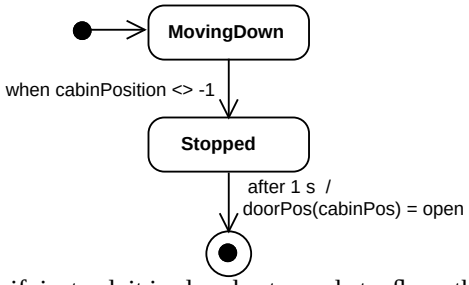


This diagram depicts the context of the lift controller (class Controller) and how it may interact with the entities in such context (by sending the pulses to the *LiftPlant* and reading its sensors) and by receiving the commands from the *User*).

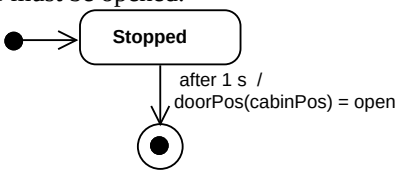
Use case descriptions

Stop cabin The stop command requires that
– if the cabin is moving, it must reach the nearest floor and then the corresponding door is opened (recall cabinPosition = -1 when the cabin is between two floors),

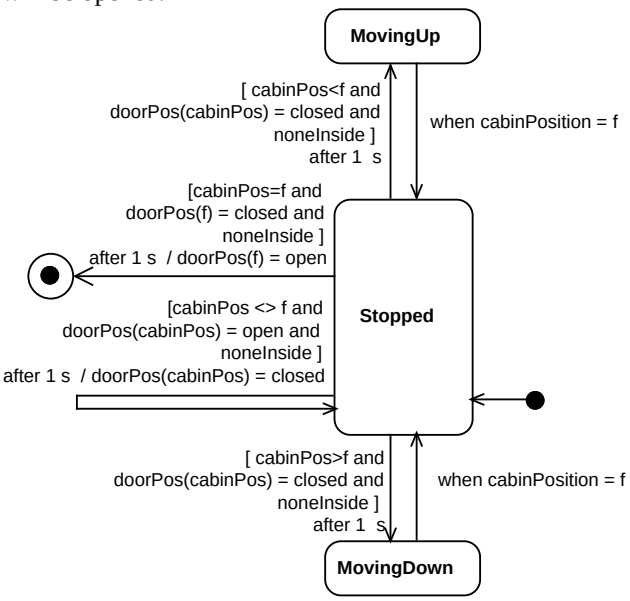




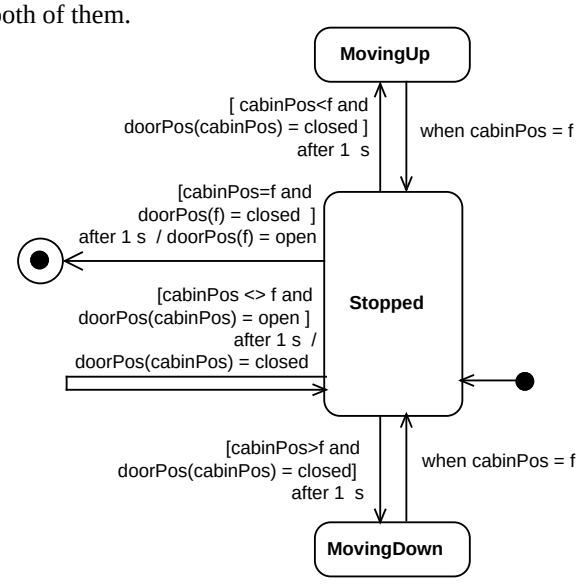
– if, instead, it is already stopped at a floor, then the door at that floor must be opened.



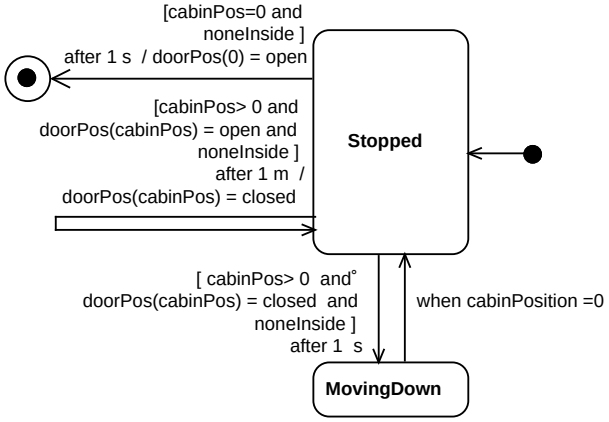
Call cabin at a floor Assume that the called floor is f. This command is ignored whenever the cabin is not stopped, or when it is already at the called floor with open doors. Otherwise, it requires that when no one is inside the cabin, the door at the floor where the cabin is must be closed, and the cabin have to move to f; after the door at f will be opened.



Send cabin to a floor Assume that the floor to whom the cabin is sent is f. This command is ignored whenever the cabin is not stopped or it is already stopped at f with open door. Otherwise, it requires that the door at the floor where the cabin is must be closed, and the cabin have to move to f; after the door at f will be opened. There is a little subtle difference with the previous use case (the former requires that no one is inside the cabin before it can start to move), but the proosed method allows to precisely describe both of them.



Cabin must wait at ground floor When no one is using the lift for 1 minute, the cabin must be sent to the ground floor and the door at that floor opened.



5 Design Specification

5.1 The UML Model

The design effort for a problem matching the Commanded Behaviour Frame consists in developing the *Control machine*. In general that task may be quite complex because many different possible design choices are possible. For example, the *Control machine* may

– be a simple synchronous process that in any cycle reads all the sensors and sends pulses depending on the values read beforehand and on the received command;

– be a simple process that waits to receive a command, then reads the sensors and sends the appropriate pulses, and goes back to waiting for commands;

– have a complex architecture, where some components take care to receive the commands, others read the sensors and elaborate the read values, and others determine and send the pulses.

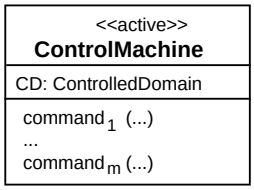
Moroever, the commands may be processed in a sequential way one after the other or in parallel, and policies may be defined to solve conflicts when many different commands are received and some of them have to wait to be processed.

Thus we will provide a family of patterns in the sense of [1] for the *Control machine* design, each one based on a set of coherent design choices, and accompanied by the schematic form of the UML model needed to present it. Here we present the pattern for a simple case.

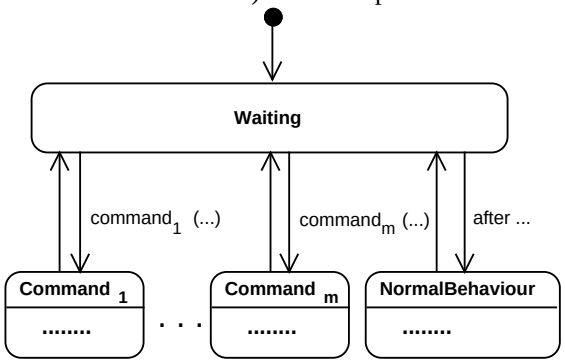
Asynchronous Simple Control Machine

The design of the *Control machine* proposed by this pattern assumes that it is a simple process receiving the commands and processing them one after the other. While it is processing a command it cannot receive another one, and when it is waiting for a command to take care of it should ensure the required normal behaviour. This pattern may be applied only when the description of the use case Required normal behaviour does not include safety constraints.

In this case the *Control machine* is simply modelled by an active class whose behaviour is defined by a statechart. Thus, the design specification is a UML model containing



(the operations $\mathsf{comm}_1, \ldots, \mathsf{comm}_m$ are those of the Commands interface, and the attributes CD refers to the actual *Controlled domain*) and a statechart (associated with the class ControlMachine) with a shape as below



where the subcharts enclosed in any of the sequential hierarchical states (depicted by dots in the above picture) are such that

– their events are only change events on the sensors of CD and timed events (while handling a command another one cannot be received),

– their conditions are on the attributes of *Control machine* and on the sensors of CD,

– their actions may update the attributes of *Control machine* and call the operations of the Pulses interface,

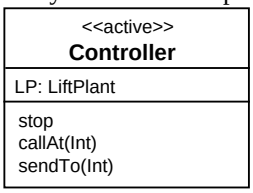
– have a unique initial substate and at least one final substate.[6]

The *Control machine* modelled by the above statechart waits for commands, when one is received it is processed sending pulses to the *Controlled domain* so as to obtain the commanded behaviour, and then goes back to wait; when no command is received for some given time, it sends pulses to the *Controlled domain* so as to obtain the behaviour required in the normal situation, and then goes back to wait.

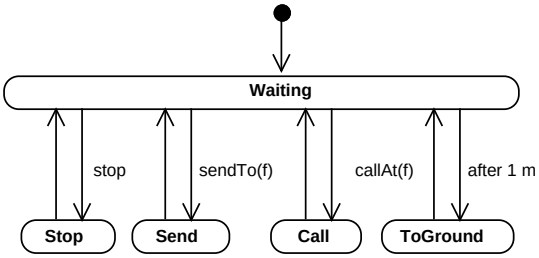
[6] Recall that a transition entering a structured state activates its initial substate, and that an unlabelled transition leaving a structured state will be fired whenever a final substate becomes active.

5.2 Lift Case Study: Design Specification

For the simple case of the lift we propose a design following the pattern “Asynchronous Simple Control”.

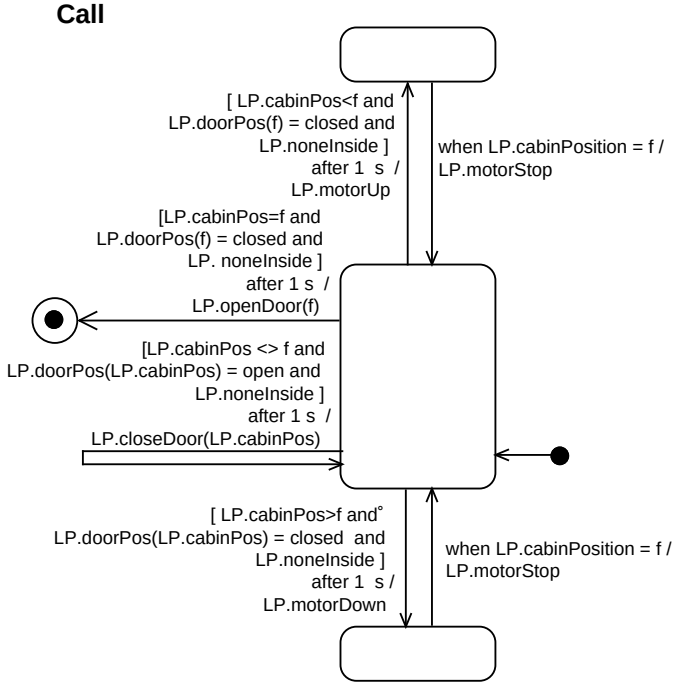


The behaviour of the class Controller is given by the following hierarchical statechart.

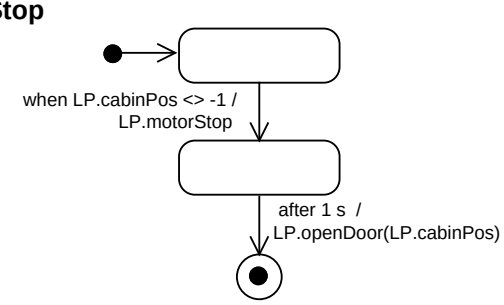


The refinement of the various states are given below.

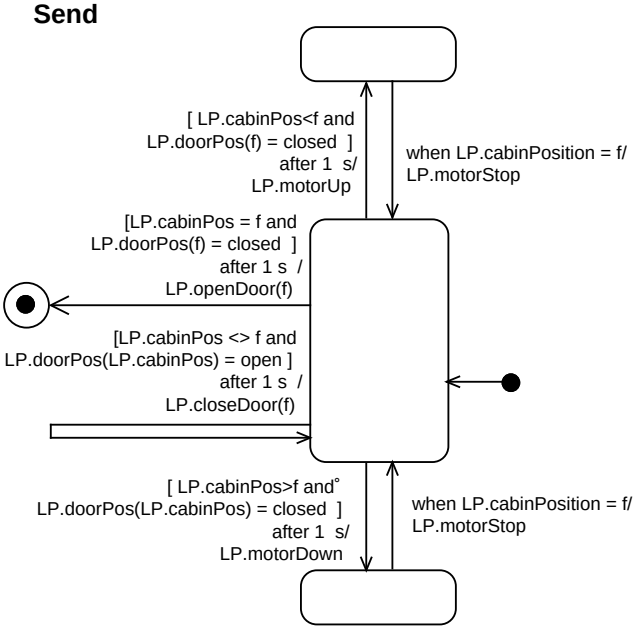
Call



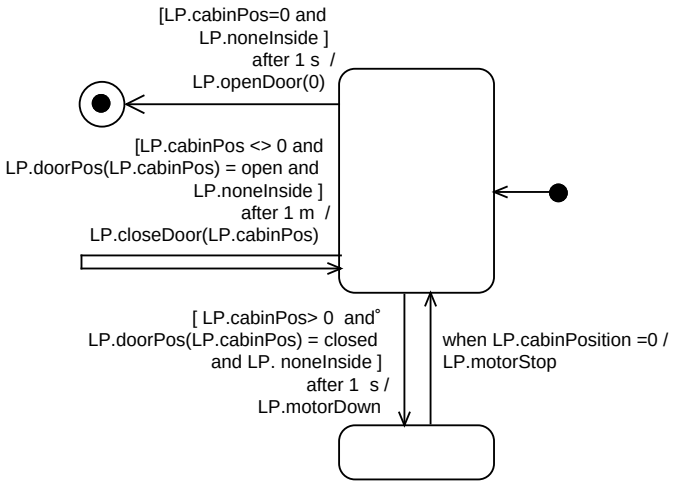
Stop



Send



ToGround



6. Conclusions

In this paper we have presented a software development approach for systems fitting the Commanded Behaviour Frame that combines the use of the UML notation, the use of the structuring concepts provided by the problem frames, together with our methodological approach for well-founded methods.

While the problem frame provides a first overall structure for problems, our method shows, for each development phase, how to use appropriate UML constructs.

There are, to our knowledge, no similar approaches in the literature, proposing a method for the combined use of problem frames and UML.

However, some work was done to associate formal specifications with problem frames, as we did in [7], and D. Bjorner in [5]. In [9], we show how to relate problem frames with a well founded methodology for UML descriptions, and we worked on the Transformation, the Commanded Information and the Required Behaviour frames. Here, we took advantage of our experience in [7] to propose a well-founded way to use UML for the Commanded Behaviour Frame.

We think that, with respect to the use of standard UML-based methods, our method is more efficient since the user does not need to loose time to devise the better way to use UML, deciding among the various constructs to use and in which way, thus (s)he may concentrate on the relevant aspects of the development instead of the questions related to the use of UML.

Let us note that our precise form of the various statecharts used for domain, requirement and design may be at the basis of methods to help guarantee correctness and other good properties we plan to investigate.

We plan to provide similar methods for the other basic problem frames, such as Required Information, Workpieces, and we see no particular problems to do that.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[2] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL : the Common Algebraic Specification Language. *T.C.S.*, 286(2):153–196, 2002.

[3] E. Astesiano and G. Reggio. Towards a Well-Founded UML-based Development Method. In A. Cerone and P. Lindsay, editors, *Proc. of SEFM '03*, pages 102–117. IEEE Computer Society, Los Alamitos, CA, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ AstesianoReggio03g.ps`.

[4] E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, Proc. 9th Monterey Software Engineering Workshop, Venice, Italy, Sep. 2002.*, number 2941 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2004. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03f.pdf`.

[5] D. Bjorner, S. Kousoube, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In M. Hinchey and L. ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997*, pages 263–270. IEEE CS Press, 1997.

[6] J. Bradfield, J. K. Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE 2002*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2002.

[7] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 14th International Workshop WADT'99*, number 1827 in Lecture Notes in Computer Science, pages 106–125. Springer Verlag, Berlin, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[8] C. Choppy and G. Reggio. Towards a Formally Grounded Software Development Method. Technical Report DISI–TR–03–35, DISI, Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio03a.pdf`.

[9] C. Choppy and G. Reggio. Using UML for Problem Frame Oriented Software Development (Complete Version). Technical Report DISI-TR-04-11, DISI – Università di Genova, Italy, 2004. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio04b.ps`.

[10] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML*. Addison-Wesley, 2000.

[11] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

[12] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2001.

[13] Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Tp 165, 8/01, 2001.

[14] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34, DISI – Università di Genova, Italy, 1999. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll99a.ps`.

[15] UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.

[16] P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In M. Broy and A. V. Zamulin, editors, *Proc. PSI 2003*, number 2890 in Lecture Notes in Computer Science, pages 351–357. Springer Verlag, Berlin, 2003.