

# Specification of a Gas Station using a Formalism Integrating Formal Datatypes within State Diagrams

Christian Attiogbé, Gwen Salaün  
IRIN  
Université de Nantes  
{attiogbe,salaun}@irin.univ-nantes.fr

Pascal Poizat  
LaMI - UMR 8042 CNRS  
Université d'Évry Val d'Essonne  
poizat@lami.univ-evry.fr

## Abstract

*In this paper, we propose a generic approach for integrating datatypes expressed using formal specification languages within state diagrams. Our main motivations are (i) to be able to model dynamic aspects of complex systems with graphical user-friendly languages, and (ii) to be able to specify in a formal way and at a high abstraction level the datatypes pertaining to the static aspects of such systems. The dynamic aspects may be expressed using state diagrams (such as UML or SDL) and the static aspects may be expressed using either algebraic specifications or state oriented specifications (such as Z or B). Our approach introduces a flexible use of datatypes. It also may take into account different semantics for the state diagrams. We herein focus on a case study to demonstrate the pragmatism of our approach.*

**Keywords:** Formal Methods Integration, State Diagrams, Algebraic Specifications, Z, B.

## 1 Introduction

The joint use of formal and semi-formal specification languages is a promising approach, with the objective of taking advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. In this paper, we propose an approach dealing with this issue. It enables one to specify the different aspects of complex systems using an integrated language.

Static and functional aspects are specified using static formal specification languages (algebraic specifications [4], state oriented languages such as Z [15] or B abstract machine notation [1]). This makes the verification of specifications possible but also the description of datatypes at a very high abstraction level. The flexibility we propose at

the static aspects specification level enables the specifier to choose the formal language that is the more suited to this task: either the one (s)he knows well, the one with tools, or the one that makes the reuse of earlier specifications possible.

Dynamic aspects (*i.e.* behaviour, concurrency, communication) are modelled using state diagrams. Our proposal is generic. Different dynamic semantics may be taken into account, hence our approach may be used for Statecharts [11], for the different (yet growing number) of UML state diagrams semantics [14, 13, 17, 3, 16, 12], and more generally for any state / transition oriented specification. In our approach, the specification is control-driven: the dynamic aspects are the main aspects within a specification and state how the static aspects datatypes are used. On a larger scale, our work deals with the formal specifications integration and composition issues, where we yet have some general results [2]. At the global specification level, our approach also addresses the consistency of the static and dynamic parts.

This paper is structured as follows. In Section 2, we summarize the formal foundations of our approach: syntactic extensions used to integrate formal datatypes within state diagrams and insights on the integration semantics. The main part of this paper, Section 3, is devoted to the illustration, in a pragmatic way, of how our approach may be used to specify a real system (a gas station). To end, Section 4 concludes the paper and presents some perspectives.

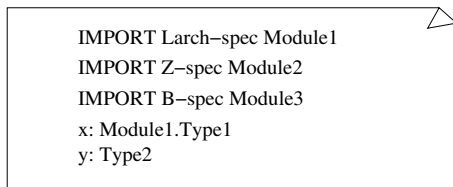
## 2 Formal Foundations of the Combination

In this section we give an introduction to the formal aspects of our integration of formal datatypes within state diagrams. We first present the syntactic extensions for this integration, and then give insights into its semantics. The reader can refer to [7] for a comprehensive presentation.

**Syntax.** We here focus on the syntactic extensions we add to state diagrams to take into account the formal datatypes integration. We advocate for a control-driven ap-

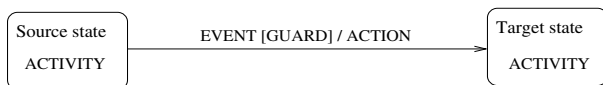
proach of mixed specification. This means that the main part of a specification is given by the dynamic aspects modelling (behaviours and communications). The static aspects are encapsulated within the dynamic aspects. We first add module importation and local variables declaration extensions to state diagrams. Both are done using *data boxes* (Fig. 1), a kind of UML-inspired note which is usually used to give additional information to a diagram in a textual form. The `IMPORT` notation is introduced to indicate which data modules are imported as well as the language used to write their contents (algebraic specifications, Z schemas, B machines, or other). Such a language is called a *framework* in our approach and is used to define the specific evaluation functions which enable us to evaluate the state diagrams data embeddings.

Variables are also declared and typed in the data boxes. Since modules often contain several type definitions, and since types with the same name may be defined in two different modules, the type of a variable may be prefixed with the name of a module in order to avoid conflicts.



**Figure 1. Local declarations of data into state diagrams**

The general form of a state diagram transition is given in Figure 2.



**Figure 2. State diagram transition**

Datatype extensions (*data expressions*) may therefore appear in both states and transitions (Tab. 1). In states, our extensions correspond to activities (actions such as entry/exit or local modifications, and events). In transitions, our extensions enable one to (i) receive values in events and then store these values into local variables, (ii) guard transitions with data expressions, (iii) send events containing data expressions and (iv) make assignments of data expressions to the local variables. The last two take place in the action part of transitions. Possible extended activities within states are not directly taken into consideration in our approach. However, such activities can be viewed as a specific case of transitions between states.

Data expressions may be either variables, terms for algebraic specifications or operation applications for state oriented specifications (such as Z or B). As far as the formal language for the static aspects is concerned, the only constraint is to have some well-defined evaluation mechanism. Our approach makes possible the joint use of several static formal languages at the same time. However, a strong mix of constructs from several languages (such as importing a Z module within an algebraic specification, or using algebraic specification variables in a Z operation application) is prohibited in order to avoid possible semantic inconsistencies. Our goal is neither to advocate for such complex combinations, nor to develop a mean to solve such inconsistencies. As a simple way to detect them, we develop a *meta-type* concept using meta-typing rules. Terms which are not meta-typed (*i.e.* inconsistent) will not be able to be used in the dynamic rules. Finally, we highlight the fact that the meaning of static specifications has to be consistent, and we assume this working hypothesis for the definition of semantic rules. In Table 1, we summarize our transitions extensions.

Part of label	Kind of interaction	Example with data
EVENT	reception	$evt(x_1:T_1, \dots, x_n:T_n)$
GUARD	guard	$pred(t_1, \dots, t_n)$
ACTION	emission	$rec \wedge evt(t_1, \dots, t_n)$
ACTION	local modification	$x:=t$

**Table 1. Links between state diagrams and data**

Then, in Table 2, we give an abstract syntax for these extensions. The `VAR`, `TYPE`, `PREDICATE` and `DATA-TERM` nonterminals correspond respectively to variable names, type names (sorts), predicate and data expressions.

LINK	::=	[EVT] [ [GUARD] ] [ / ACT+ ]
EVT	::=	evt-name [ (VAR-DECL+ ) ]
VAR-DECL	::=	VAR: TYPE
GUARD	::=	PREDICATE
ACT	::=	EMISSION   ASSIGNMENT
EMISSION	::=	[VAR] ^ evt-name [ (TERM+ ) ]
ASSIGNMENT	::=	VAR := TERM

**Table 2. BNF Grammar of links**

**Semantics Overview.** Our goal is to give a formal semantics to state diagrams extended with formal datatypes in a way that has been presented in the syntactic part. We do not aim at formalizing some specific kind of state diagram, which has already successfully been done, see [14, 13, 17, 3, 16, 12, 11] for example. We rather aim at being able to reuse different existing state diagram seman-

tics. Therefore, our semantics is constructed in such a way that generic constructs may then be instantiated for a specific kind of state diagram. In our semantics we will for example deal with properties such as “the event pertains to the input event collection of the state diagram”, which could thereafter be instantiated for a specific state diagram language into “the event is the first element of the input queue associated with the state diagram process”.

Using this generic approach, we preserve a very general description of extended state diagrams. In this way, all kinds of state diagrams and their underlying semantics could be considered. The meaning of the state diagrams extension is expressed using an operational semantics. Therefore, the semantics of a non-extended state diagram has to be given in terms of a Labelled Transition System (LTS) which is currently always the case. We may then define precisely the meaning of the integration of datatypes into dynamic diagrams using *extended states* evolutions and *evaluation functions*. Extended states are made up of the states from the non-extended state diagram semantics, environments memorizing the data managed by extended state diagrams, and input/output events collections. Evaluation functions are deduced from the abstract datatypes definitions and are used to interpret the data embedded in extended state diagrams. The possible evaluation of a term  $t$  into a value  $v$  with respect to an environment  $E$  is denoted in the sequel by  $E \vdash t \triangleright v$ . Suffixes are used to express the framework of the evaluation, *e.g.*  $\triangleright_Z$  denotes the evaluation function in the  $Z$  framework.

Our semantics is given using rules which may be decomposed into: meta-typing rules (badly meta-typed terms may not be evaluated), action evaluation rules (describing the effects of actions on extended states), dynamic rules (formalizing the individual evolution of an extended state diagram), open systems rules (describing the effect of putting extended state diagrams within an external environment), and global system and communication rules (putting things altogether).

### 3 Case Study: A Gas Station

In this section, we illustrate our approach on a simple but realistic example: a gas station. We successively study the requirements, the analysis and the modelling of both static and dynamic aspects.

Our approach is generic, therefore we first have to choose the static specification languages and the kind of state diagrams (*i.e.* their syntax, their semantics and the kind of communication) we want. As far as the datatypes are concerned, we will use both algebraic specifications (LP) and state oriented specifications (Z). We will use UML state diagrams to model the dynamic aspects of our case study. These choices lead to instantiations of the generic

evaluation mechanisms and abstract dynamical concepts we use in our approach to give a semantics to such an integration.

**Evaluation Functions.** We have to define evaluation functions for each static language we use. As far as LP is concerned we may use term rewriting with an appropriate reduction order (**noeq-dsmpos**) [10], that is the evaluation function  $\triangleright_{LP}$  is defined as:  $E \vdash s \triangleright_{LP} s' \Leftrightarrow s[E] \rightsquigarrow_R^* s'$  where  $R$  is the set of rewrite rules deduced from the abstract definitions. Concerning  $Z$ , the obtaining of an evaluation function is a little bit more complex. The idea to define the  $Z$  evaluation function is to consider LTSs associated with  $Z$  specifications. This is possible since  $Z$  follows a state oriented approach. Let  $zS$  be a  $Z$  specification defined with a state schema  $SSch$ , an initialization schema  $SInit$ , and a set of operation schemas. We may then define the associated LTS. Its set of states ( $STATE_{zS}$ ) corresponds to the  $SSch$  semantics state space. The set of initial states of the LTS is the subset of  $STATE_{zS}$  with elements that satisfy the predicate of  $SInit$ . Finally, each operation schema predicate, used to relate the bindings of two states, defines a set of transitions labelled by operation applications. The set of transitions of the LTS ( $TRANS_{zS}$ ) is the union of all these transitions. The evaluation function  $\triangleright_Z$  is then defined as:  $E \vdash t \triangleright_Z s' \Leftrightarrow \exists s \subseteq E . s \xrightarrow{t} s' \in TRANS_{zS}$ .

**Instantiation of the Dynamic Abstract Concepts.** The first thing is to choose an appropriate semantics for UML state diagrams that we will thereafter be able to extend using our approach. Appropriate means that this semantics has to be operational and given in terms of a LTS. We chose Jürjens formalization [12] as it is one of the more comprehensive semantics for UML state diagrams. Then, we have to instantiate the generic communication mechanisms and communication constraints of our approach. For short we use an asynchronous communication with event/message collections associated to the components using queues, see [6] for more detail we cannot give here by lack of place.

In the sequel, we will use UML state diagrams extended with LP and  $Z$  datatypes in conformity with these instantiation choices.

#### 3.1 Requirements and Analysis

**Requirements.** The case study concerns the specification of a self service gas station with credit card payment. The necessary transactions for authentication and effective payment (data transmission to a card management centre) are out of the scope of this case study. There are three pumps at the gas station. Each pump delivers a type of fuel: two-star fuel, lead-free fuel and diesel. The gas station has a control board equipped with a card reader for payments, several button keys for options selection, a numerical keyboard

for inputs, a control screen and a ticket printer. The station enables the user to enter his/her card number via the numerical keyboard, to select the fuel type and to choose or not the ticket printing. The gas station manages all its equipments and furnishes to users the desired quantity of the fuel they selected.

**Analysis and Working Hypotheses.** From these requirements, we clarify the functionalities of the gas station considered here as the main system: users identification and authentication, fuel delivering and payment. We make explicit the necessary conditions to ensure these functionalities and the implied interactions with the environment. According to this analysis, we identify and model several components. We distinguish two main parts in the analysis. A static part is devoted to the datatypes of the system, and a dynamic part deals with the system evolutions and communications.

**Static Part.** To deliver fuel to users without sudden interruption during the distribution, the station must have a sufficient quantity of each type of fuel when it is put on service. We assume that there are several tanks. A unique tank is associated to each pump, that means to each type of fuel. Two different pumps are associated to two different tanks. A pump is off (*i.e.* out of service) when the quantity of fuel in the associated tank is less than a threshold to be defined. In such a case, the station cannot deliver this type of fuel before a refill of the tank. To guarantee smooth working, the system should know at any time the tank status. By the way, this state should be maintained according to the achieved operations (fuel delivering, fuel refilling). To compute the amount to be paid and print the ticket, the system must know the price by litre for each type of fuel and the delivered quantity. A record of this information is necessary. To summarize, we have to specify tanks, fuels and pumps to model the static part. Cards are not treated because authentication and payment are out of the scope of the case study.

**Dynamic Part.** The behaviour of the station is triggered by the user. When the user inserts its card, an authentication procedure takes place. If the authentication succeeds then the user has to select a fuel type and validate or correct his/her choice. These interactions put emphasis on a *card manager* component. According to the user's choice, the station initializes the screen, displays the price by litre and the maximal quantity of fuel which can be delivered. Then the user must disengage the pump and serve himself/herself. The amount to be paid and the quantity are simultaneously displayed during the delivering. At the end of the delivering, when the user engages the pump, the station prints the ticket if the user had selected this option. All these interactions reveal a component we call *pumps manager*. It exchanges information with the user. We assume that when

a pump is on, it guarantees the delivering of the associated fuel. For this reason, the station checks for the tank status after each service and either the pump remains on or is set off if there is not enough fuel. Thus, we identify a *tanks manager* component which has interactions with the pumps manager.

We design the global system as a concurrent composition of the three main independent components: a card manager, a pumps manager and a tanks manager. Each of them is described in the sequel. Other subsystems are considered external (users and a subsystem that provides continuously the quantity of fuel and the corresponding amount to be paid for a distribution).

### 3.2 Static Aspects Modelling

In this case study, we specify the datatypes with both LP and Z. LP is used to model natural and real numbers. Z enables us to specify datatypes for pumps and tanks. We give here (most of) the Z specifications used in the case study dynamic part diagrams. The complete static specification has been analyzed with Z/EVES for static correctness (type checking and syntax analysis) and static semantics (domain control).

**Data Modelling for the Tank Management.** Several Z schemas are used to model the tank management. For each tank, we record the fuel type, the minimal and maximal quantity the tank may contain, and its current quantity. There is a one-to-one function between the pumps (*pumpsId*) and the fuel types (*ZFuel*). Various operations are described to specify the update of the tank contents.

$$pumpsId == \{1, 2, 3\}$$

Pumps are modelled in our system as natural numbers. A pump is identified using a unique natural number.

$$ZBool ::= tt \mid ff$$

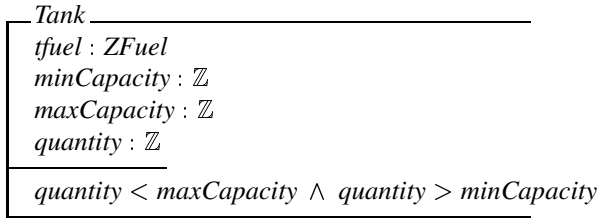
The values *tt* (true) and *ff* (false) characterize the Boolean type.

$$ZFuel ::= twostar \mid leadfree \mid diesel$$

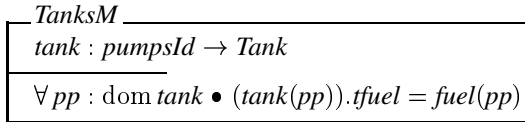
The enumeration of the three fuel types enables us to specify the fuels.

$$\left| \begin{array}{l} fuel : pumpsId \mapsto ZFuel \\ fuel(1) = twostar \\ fuel(2) = leadfree \\ fuel(3) = diesel \end{array} \right.$$

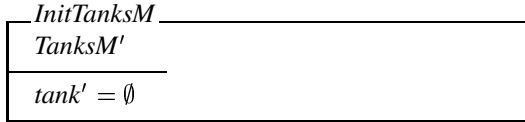
Each pump delivers a unique fuel type. For this purpose, we use the *fuel* function. Therefore, given a fuel type, we can get the associated pump with the reverse function (*fuel<sup>~</sup>*).



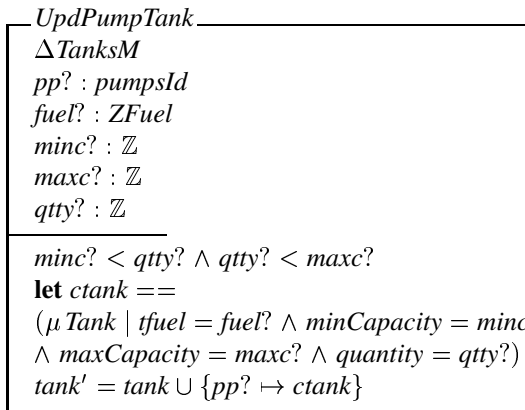
A tank is characterized by its fuel type, its minimal and maximal capacity and its current quantity of fuel.



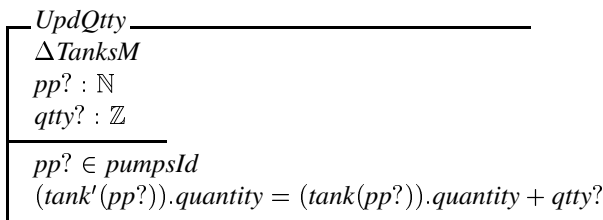
This schema allows us to manage all pumps in our system. A tank is associated to each pump.



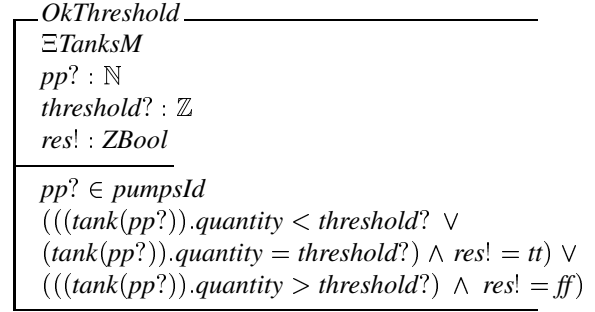
In this initialization schema, we start with an empty function for tank. This state variable is initialized and updated by the *UpdPumpTank* operation. We will also present the *UpdQty* operation which updates the remaining fuel quantity of each pump and the *OkThreshold* operation which checks the fuel quantity.



This operation initializes the tank associated to a given pump.



This operation updates (*i.e.* increases or decreases) the fuel quantity of a given pump. The quantity can be negative to decrease or positive to increase the quantity. The *pp?* and *qtty?* input variables can be replaced by the diagrams local ones using the *UpdQty*[*pp/pp?;nqt/qtty?*] notation. Similarly, the output variables of operation schemas are replaced by local variables to retrieve operation results.



This operation checks the fuel quantity in a tank with regard to a set threshold. The result is positive if the remaining quantity in the indicated pump (*pp?*) is greater than the indicated threshold (*threshold?*).

These data specifications will thereafter be used to model the behaviours of the gas station components. Accordingly, we gather all these specifications in a data module called *Z-GSdata*. Indeed, we can import this module so as to access predefined types as well as the defined types, state schemas and operations schemas. Note that only one diagram uses the initialization schema in order to have only one instance of the system state. The other diagrams use operation schemas and data sent through message parameters. For the sake of readability, in diagrams, we use special definitions for predefined types: *ZInt* ==  $\mathbb{Z}$  and *ZNat* ==  $\mathbb{N}$ .

### 3.3 Dynamic Aspects Modelling

The behaviours of the card manager, the pumps manager and the tanks manager are modelled using UML state diagrams extended following our generic approach. In the managers dynamic models, we take into account the scheduling of events and actions and the possible interactions between the various managers. The models also establish links with previously described data. Datatypes appear in guards (*e.g.* [pstate=tt]), as event parameters (*e.g.* updPumpState(pstate: ZBool)), in actions (*e.g.* /PumpsManager^activatePump(ap)) and in data boxes where variables have types imported (*e.g.* IMPORT Z-SPEC Z-GSdata) from LP or Z modules. In this case study, there is no type name overloading. Therefore, there is no need to prefix the variables types by the module name to avoid confusion.

Let us now give details on the models we obtained, starting with the **card manager** (see Fig. 3). It models events

which permit to insert a card, input the code and select the desired fuel. Afterwards, it interacts with the pumps manager so as to deliver the fuel. Several interaction termination cases are possible: the authentication fails (`error`), the delivering is correctly terminated (`endDelivering`) and the ticket is printed if the user has selected this option (`printTicket`), or the delivering has not been possible (the pump is not disengaged within sixteen seconds) and the interactions terminate with error (`pumpError`). Two different kinds of events are used in the diagrams. Basic ones have no parameters. They express either an internal evolution of the state diagram (for example `insertCard` and `inputCode`) or an interaction between diagrams (for example `/CardManager^pumpError`). Extended ones can bear parameters (e.g. `askTicket(b: Zbool)`). They express interactions with other diagrams. Note that we use external events that correspond to interactions between the system and its environment, particularly with the user which triggers the pump functioning. Examples of such events are `insertCard` in the card manager (see Fig. 3) and `distribution` in the pumps manager (see Fig. 4).

The diagram of the **pumps manager** (see Fig. 4) uses a `pumpsThreshold` variable to model the minimal fuel quantity needed to maintain the pump on; it corresponds to a set threshold. The computation of the amount to be paid (`amount`) is achieved by an external subsystem and the result is obtained via the `receiveAmount` event. This is also the case for the distributed quantity (`servedQtty`). The specification of this subsystem could be simply achieved by considering a state diagram with a local data type handling couples made up of a *fuel* and the corresponding *price per litre*. This is abstracted away with the `receiveAmount` event. The pumps manager ensures the fuel distribution by interacting with the card manager and with the tanks manager. When a distribution is initiated, the user has sixteen seconds to disengage the pump and to serve itself. After this delay an error is indicated. The pumps manager knows the maximum fuel quantity a user can get; it can then interrupt the distribution (`endDistribution`). The card manager interacts with the user for a service, for example through the `insertCard` and `askTicket` events. During a distribution, there are a lot of synchronizations between the pumps manager and the card manager so as to update the control screen (`updScreen(qt:ZInt, mt:Real)`) with the fuel quantity and the corresponding amount. The pumps manager engages the `/CardManager^updScreen(servedQtty, amount)` event and the other does the `updScreen(...)` event. Such an extended UML state diagrams interaction meets our general communication mechanism [6], hence it is completely formalized. These interactions with the user are continued until the end of the distribution (`endDistribution`) and the pump engagement

(`engage`). The pumps manager then locks the pump and synchronizes itself with the tanks manager to update the fuel quantity (`decreaseQtty`). Finally, it communicates with the card manager to end the distribution (`endDelivering`), the threshold is checked (`/TanksManager^checkThreshold`) and the pump state is updated (`/CardManager^updPumpState`).

The **tanks manager** uses the `Z-GSdata` module and the `pstate` variable (pump state) as local data. The tanks manager interacts with the pumps manager (`decreaseQtty`) and with the environment in order to fill tanks with fuel (`increaseQtty`). It also achieves the tanks update operations (`UpdQtty`) and the fuel level tests (`checkThreshold`).

We show in this case study how to practically integrate formal datatypes within state diagrams. Here we used datatypes formalized using Z and LP, and behaviours and communications using extended UML state diagrams. However our approach is generic and other languages can be taken into account.

## 4 Conclusion and Perspectives

Our goal is to propose specifier-friendly integrated languages for mixed specifications, and more generally an integration method suited to the specification of mixed complex systems. We choose to combine state diagrams with formal specification languages devoted to abstract datatypes (algebraic specifications or state oriented specifications such as Z and B). This joint use, in a formal and integrated framework, of a semi-formal notation for dynamic aspects with formal languages for static aspects enables one to take advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. For a comprehensive comparison with related works, the reader may refer to [6].

A first perspective addresses verification issues. If it is yet possible to verify the aspects taken separately, it is important to be able to verify the global system. We are working on the translation of our generic approach for integrated mixed languages and its semantics into higher order logic tools such as PVS [8]. We also have developed a tool dedicated to the animation of specifications combining CCS with abstract datatypes. This tool, ISA [5], is quite generic over the datatype (*i.e.* static) language which is concerned. The next step will then be to extend ISA in order to deal with several dynamic specification languages. Finally, a generalization of our approach represents an interesting challenge in order to be able to combine different formalisms based on the integration of formal datatypes within state / transition systems (such as SDL [9]).

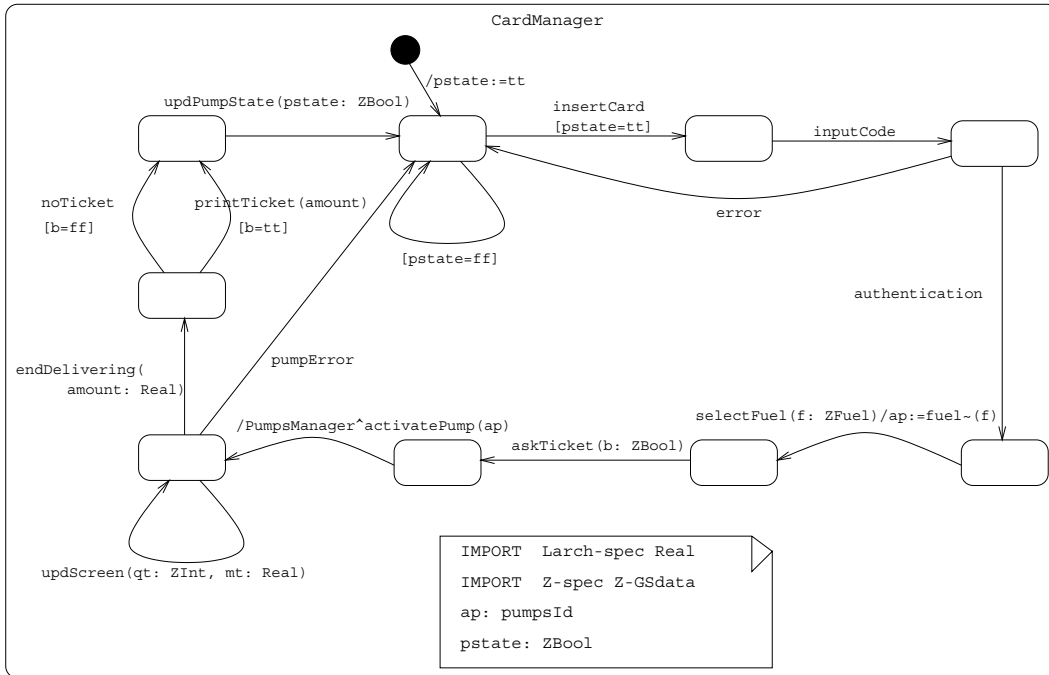


Figure 3. Card manager diagram

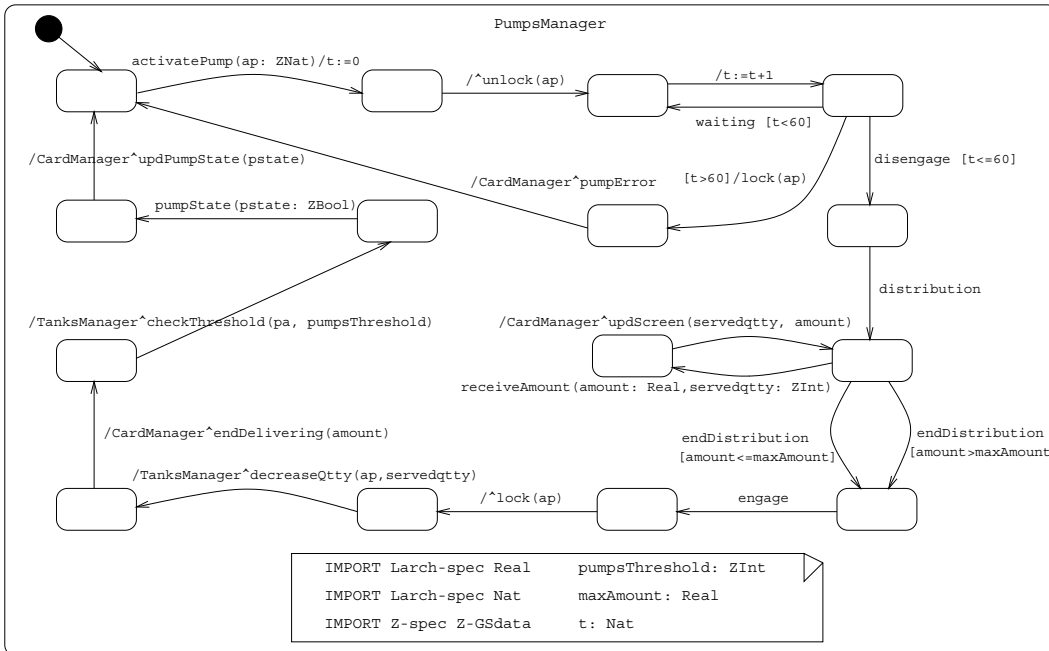


Figure 4. Pumps manager diagram

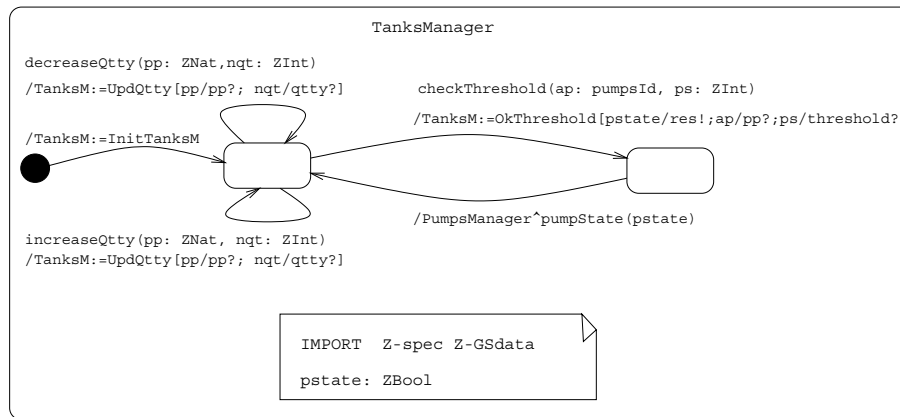


Figure 5. Tanks manager diagram

## References

- [1] J. R. Abrial. *The B-Book*. Cambridge University Press, 1996.
- [2] M. Allemand, C. Attiogbé, P. Poizat, J.-C. Royer, and G. Salaün. SHE'S Project: a Report of Joint Works on the Integration of Formal Specification Techniques. In *Proc. of the Workshop on Integration of Specification Techniques with Applications in Engineering (INT'02)*, pages 29–36, France, 2002.
- [3] D. B. Aredo. Semantics of UML Statecharts in PVS. In *Proc. of the 12th Nordic Workshop on Programming Theory (NWPT'00)*, Norway, 2000.
- [4] E. Astesiano, H.-J. Krewski, and B. Krieg-Brückner, editors. *Algebraic Foundations of System Specification*. Springer-Verlag, 1999.
- [5] C. Attiogbé, A. Francheteau, J. Limousin, and G. Salaün. ISA, a Tool for Integrated Specifications Animation. Available at <http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/ISA/isa.html>.
- [6] C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. Technical Report 83-2002, University of Evry, October 2002. Available at <http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/papers/APS8302.ps>.
- [7] C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Data Types within State Diagrams. In *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE'03)*, Poland, April 2003. Springer-Verlag. To appear.
- [8] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, USA, 1995. Computer Science Laboratory, SRI International.
- [9] J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.
- [10] S. J. Garland and J. V. Guttag. A Guide to LP, the Larch Prover. Technical Report, Palo Alto, California, 1991.
- [11] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [12] J. Jürjens. A UML Statecharts Semantics with Message-Passing. In *Proc. of the 17th ACM Symposium on Applied Computing (SAC'02)*, pages 1009–1013, Spain, 2002. ACM Inc.
- [13] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, *Proc. of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 331–347, Italy, 1999. Kluwer Academic Publishers.
- [14] J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *Proc. of the International Conference on the Unified Modelling Language: Beyond the Standard (UML'99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445, USA, 1999. Springer-Verlag.
- [15] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
- [16] M. van der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th International Conference on the Unified Modelling Language (UML'01)*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421, Canada, 2001. Springer-Verlag.
- [17] D. Varró. A Formal Semantics of UML Statecharts by Model Transition Systems. In A. Corradini and H. Krewski, editors, *Proc. of the 1st International Conference on Graph Transformation (ICGT'2002)*, volume 2505 of *Lecture Notes in Computer Science*, Spain, 2002. Springer-Verlag.