Validation of UML models thanks to Z and Lustre (category 1)

Sophie Dupuy Lydie du Bousquet

Laboratoire LSR-IMAG,
BP 72, 38402 Saint Martin d'Hères cedex, France;
{ Sophie.Dupuy, Lydie.du-Bousquet }@imag.fr

Abstract: Graphical notations such as UML are very popular thanks to their simplicity and their intuitive aspect. Nevertheless their lake of precise semantics limitates the possibility of the specification validation. So we propose here to translate some of the UML models into Z and Lustre formal specifications in order to use a theorem prover and a test generator to validate the models. This approach is presented on the "cash-point" service case study proposed during the world Formal Method congress 1999 tool contest.

1 Introduction

Generally software engineers use graphical models like those of the *Unified Modeling Language* (UML [3]) to describe a system. These models offer intuitive notations which favor the communication within the specification team and with the customers. But the semantics of these notations are not always precise so that their use can lead to ambiguities and misunderstandings of the specifications. They also limit the potential and the tools to validate the models.

In order to offer a more efficient framework to validate system specifications, it has been proposed to integrate semi-formal graphical notations with formal ones ([18, 10, 15, 17, 14]). Formal methods offer mathematical notations which have a precise semantics that removes ambiguities and offers a potential for reasoning and automation. Formal reasoning can be used to detect inconsistencies in specifications and tools can take advantage of the formal semantics to support the reasoning process but also to help to synthesize efficient implementation or test suites.

So we choose an approach that takes advantage of the benefits of each notation: the graphical methods serve for the development of the system specification while the formal ones are used for the system validation thanks to their tools. Our approach is based on the translation of the semi-formal specifications into formal ones and on the use of the tool support of formal notations. So the system is first described by a graphical intuitive model before being made precise and validated thanks to a formal specification. We have chosen to develop this approach with UML which is now a standard for object-oriented modeling notations. It enables designers to express models from different points of view, their data structures and operations (static part of the model) as well as their behaviors (dynamic part). For the validation of both static and dynamic parts of UML models, we translate the UML description into appropriate formal languages and use their tools to process the validation (Fig. 1).

The static part of a UML model is composed of the class diagram. In knowledge systems and data base systems, it is frequent to express some constraints on the classes and relations, and then to check them. To do that, languages such as Z [20], B [1] or VDM [2] are quite appropriate because their concepts are close to the UML class diagram ones and they have tool support both for proving and animating specifications. Among these languages, we choose Z for its visibility and its adequation to the constraint language (OCL [21]) proposed to complete UML models. For these reasons, to validate the static part, we generate Z formal specification with proof obligations. *This is done automatically* thanks to one of our tool, named RoZ [8]. Then we use the Z-EVES theorem prover [19].

To validate the dynamic part of the model, several methods are classically used (animation/testing, model-checking, theorem proving). The animation and test techniques can be used if the system is executable, model-checking and theorem proving require the model to be small enough so that the verification can be done in a limited space or time. We decided to extract an executable specification from the dynamic part of the model, so that animation or testing techniques can be applied. To do that, we translate statecharts diagrams into Lustre specifications [4], which are validated by our testing environment, Lutess [6].

In the paper, we illustrate this approach for modeling and validation on a case study, and we underline the sorts of help that formal specifications can bring to validate the system models. The case study is the world Formal Method 1999 congress contest where exhibitors were invited to specify a "cash-point" service with their own formalism(s), and to process that specification with their tool(s). In the following, we detail our approach and we illustrate it with the contest example [7].

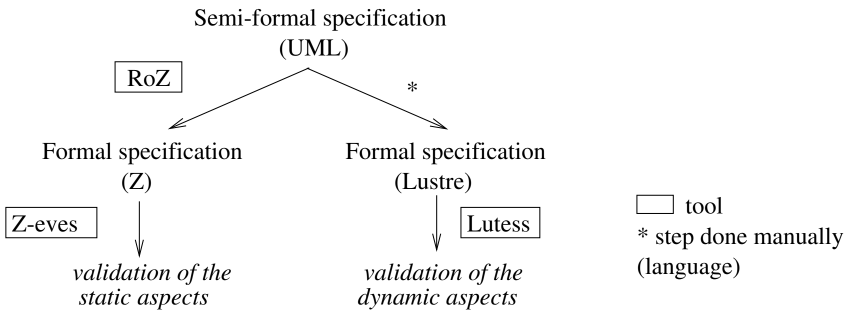

Figure 1: The approach

2 The "Cash-Point" Service

We present here a brief summary of the "cash-point" service, a complete description being given in [7].

For the “cash-point” service, there are tills which can access a central resource containing the detailed records of customers’bank account (Fig. 2). A till is used by inserting a card and typing in a personal identification number (PIN) which is encoded by the till and compared with the code stored on the card. A card can be “illegal” if it has been stolen or if it has a default. Then “illegal” cards are kept by the till. Otherwise after successfully identifying themselves to the system, customers can view the balance of their accounts, withdraw cash or ask for a statement of their account to be sent by post.

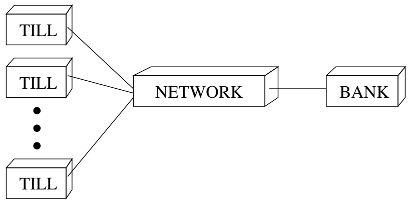

Figure 2: Physical architecture of the system

Information on accounts is held in a central database that may be unavailable. In such case, viewing the balance of an account may not be possible. If the database is available, any amount up to the total of the account can be withdrawn (if it does not exceeded a fixed daily limit on withdrawals). This informal description can be completed by concurrent or time constraints. For instance, it can be added that a transaction initiated by a customer must be completed.

3 Elaboration of the UML specification

The UML specification is the central point of our approach. First it permits to better understand requirements and to propose a model of the system to build. Secondly, it is the starting point for the generation of more formal specifications. So this part gives an overview of the UML models developed for the cash-point service, by focusing on the statement clarifications and the choices made during the development of UML models.

3.1 Requirement capture

We start the requirement capture by identifying the functionalities of the system. According to the statement, the cash-point offers at least three functionalities:

1. view of the balance of an account,
2. make a withdrawal of cash,
3. ask for statement of the account to be sent by post.

We add to these functionalities the possibility to cancel any of them. In UML, they are represented by use cases which interact with the system actors. In the cash-point service, the customer is the only actor. Moreover all the functionalities start by an identification step. We choose to express it

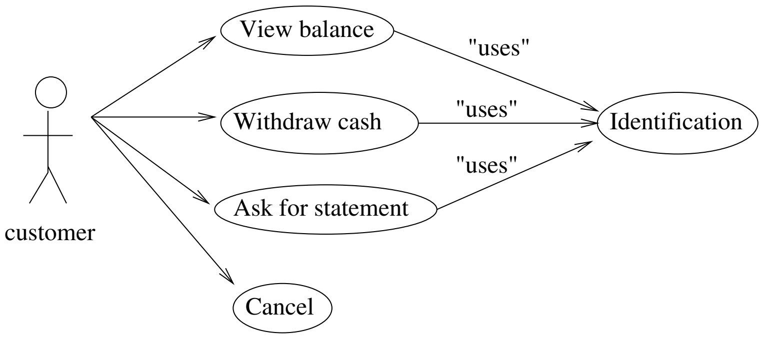

Figure 3: Use cases for the cash-point service

by a use case “Identification” which is used by the other functionalities. So the use case diagram of figure 3 represents the requirements for the cash-point service.

In order to describe each use case, we develop activity diagrams which express what activities are undertaken (by which object) to realize a use case. For brevity sake, we do not present here these diagrams which can be found in [7], but we detail the questions brought up by their elaboration. The description of the activity diagrams helps us to clarify the following points:

1. Is a card always readable?
   No a card can be damaged and then unreadable.
2. What happens if the card is damaged?
   The card is kept by the till and a message informs the customer about this.
3. What should be done if the database is unavailable?
   A message informs the customer that the database is unavailable and his/her card is given back by the till.
4. What should be done to avoid the use of stolen cards?
   The till checks if the card is stolen before asking to the customer the card PIN. This was decided so as to reduce the use of stolen cards and to gain access to the account.
5. How many times is the code tested?
   The customer has three attempts to enter the correct code of his/her card.
6. Can the identification be cancelled after the entry of a wrong code?
   Yes.
7. What happens if the till does not have enough cash for the withdrawal or if the maximum daily withdrawal is reached?
   The till gives the card back to the customer.
8. Is the cash given with the card?
   No the customer must get his/her card to have the cash.

The requirements being better understood, we develop UML models to describe the system structure and its behavior.

3.2 The static part of the cash-point service

From requirements capture, we produce UML models representing the static and the dynamics part of the cash-point service. The static part is described by an UML class diagram which is composed of four classes:

- “BANK” which represents the database of the bank,
- “TILL” which represents the bank tills,
- “ACCOUNT” which represents the customer accounts,
- “CARD” which represents the customer cards.

The bank is composed of accounts, tills and cards. Moreover the association “BankInvalidCards-Rel” describes the invalids cards. An account can have zero or many cards and a card is linked to one account or zero if the card is stolen. A card can be inserted in at most one till at a given moment.

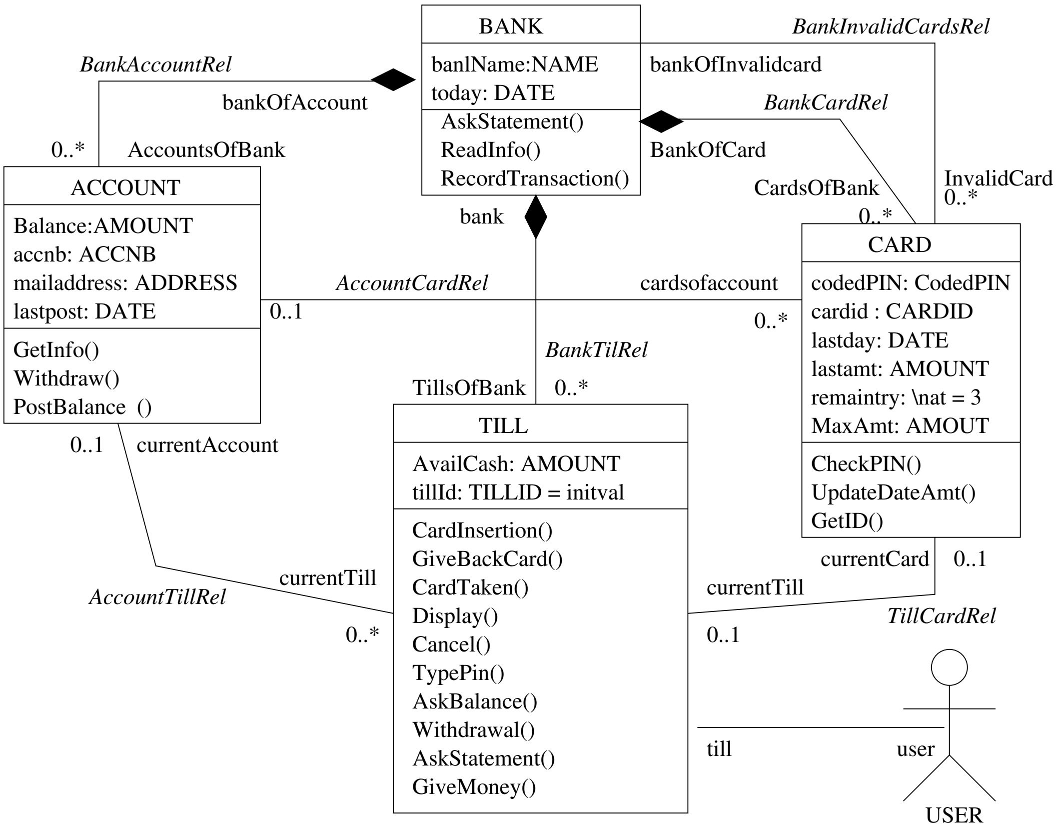

Figure 4: Class diagram for the cash-point service

This model does not express the full specification of the application data: to be complete the data structure must satisfy more than the static constraints of the class diagram. So we add to the graphical model the following constraints:

1. The amount withdrawn during a day must be greater than zero but less than the maximum amount permitted.

2. The PIN must be coded/decoded by a given function.
3. The balance of an account must be greater or equal to zero.
4. The attribute “cardid” is a key of “CARD”.
5. The last day of an card (which represents the last date the card was used) must be less or equal to today.
6. If the till is linked to a card and an account, the account must the one corresponding to the till.

Then to describe the static part of the cash-point service, we have created a class diagram complemented by static constraints. We have chosen to express these constraints in Z in order to make them unambiguous. Nevertheless, we cannot know if the annotations are mutually inconsistent or inconsistent with the model. For instance, does the operation to update the amount withdrawn preserves the constraint that the amount must not exceed a daily limit?

3.3 The dynamic part of the cash-point service

For each class of the model, we have to associate a statemachine. A statemachine is a specification of the sequence of states that an object goes through in response to events during its life, together with its responsive actions. It is usually represented as an automaton. The graphical representation of a statemachine in UML is called “statechart”. The original concept was invented by David Harel [13].

A statemachine (or a statechart) is composed of states and transitions. For instance, Fig. 5 shows a simplified specification of the till behaviors, expressed with a hierarchical statechart. The diagram 5(a) depicts the higher level diagram.

There are different sorts of states in a statechart: simple, initial ($\bullet$), final ($\odot$), branching ($\diamond$) or composite (INIT and OP in 5(a)). The transitions are labeled. The label is composed of three parts representing the calling event, a guard and the responsive actions (*Event [Guard] / Actions*). A guarded transition occurs only if the guard is evaluated to true.

Informally, the statechart 5(a) represent a user session at the till. It is composed of two parts: the user authentication (state INIT) and the treatment of the user operation (state OP). The state INIT corresponds to the requirements (1) to (6) given in section 3.1. The state OP corresponds to the requirements (7) and (8). One condition to deliver cash is that there is enough money (AvailCash) in the till (requirement 7). Moreover, the transition exiting from the state $W21$ indicates that the card is given back before the cash.

The whole system behavior is obtained by the composition of the statemachines of the system objects and actors. One should notice that two communication modes can be considered between the objects: asynchronous (for instance, message passing with queue) or synchronous (for instance, call of a method). At the specification level, it is not necessary (and probably dangerous) to specify the communication modes between the objects. This choice should be done at the implementation level, when the system architecture is definitely fixed.

3.4 Conclusion

At this point, the development of the UML models has permitted to better understand the cash-point requirements and to propose a first specification of the system. To validate this specification,

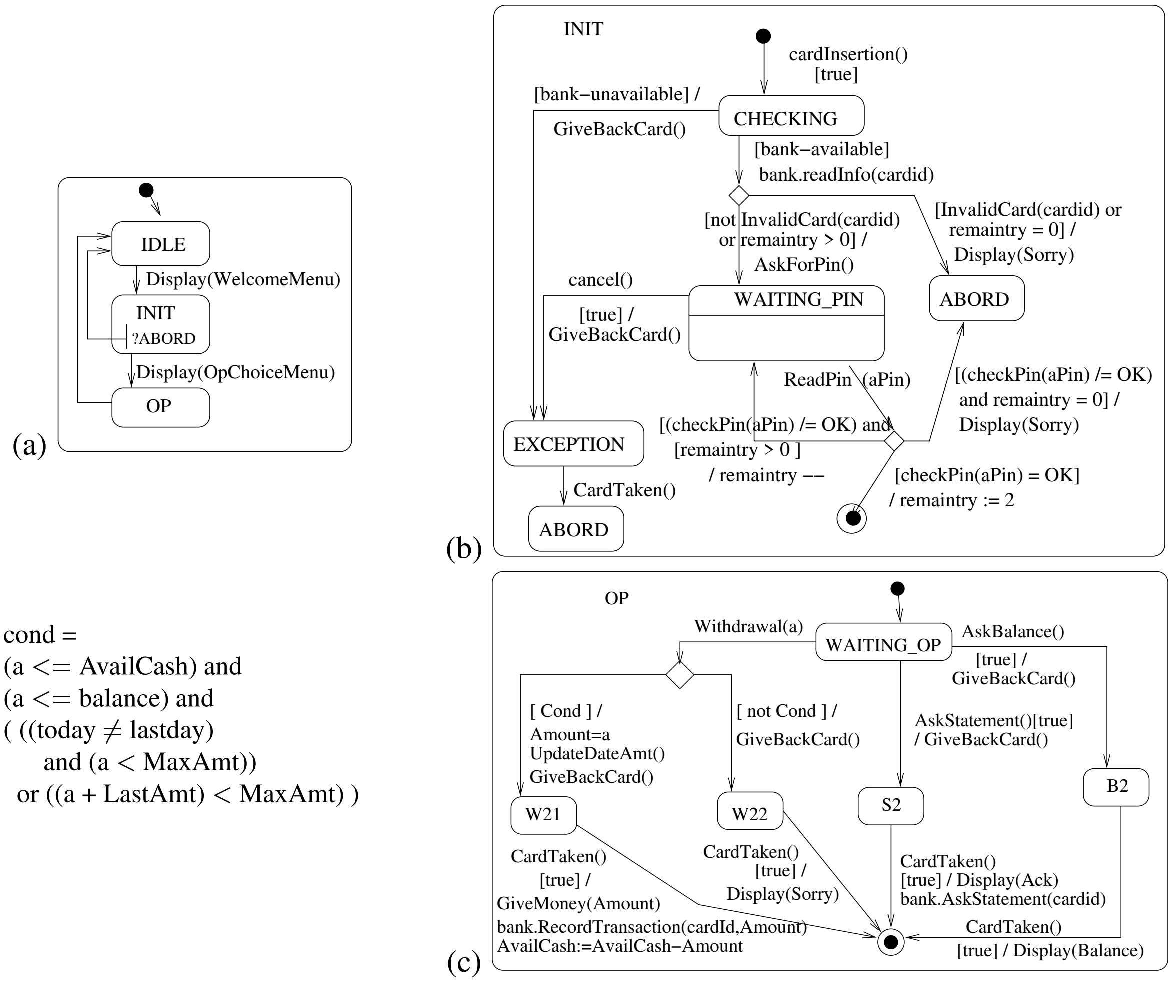

Figure 5: Till behaviors depicted by statecharts

we translate it into formal specification: the class diagram is automatically mapped into a Z specification that is used with a theorem prover while the statechart diagram gives rise to a Lustre model which is validated thanks to a test tool.

4 Validation of the static part

To validate the static part of the cash-point service system, we first use a tool, RoZ, to generate the Z specification corresponding to the class diagram and its static constraints. Then RoZ generates proof obligations that are used in the ORA theorem prover, Z-EVES, to check the mutual consistency of the diagram and its constraints.

4.1 Generating Z specifications

The generation of the Z specification corresponding to the static description of the cash-point service is realized in two steps: first the class diagram gives rise to Z specification skeletons which are complemented by annotations in a second step. The Z specification skeletons are produced according to some translation rules described in [9]. For instance, in UML, a class has two aspects: it is an object factory and an object tank (i.e. the collection of objects of the class present in the system). So a formal representation of a class must distinguish object existence and class attributes. In Z, a class is mapped into two schemas. A Z schema is a data specification structure composed of two parts: the declarations which constitute the local variable lexicon and the predicates expressing constraints on these variables. For the "CARD" class, the factory aspect is represented by the "CARD" schema which contains the declaration of the attributes while the "CardExt" schema introduces the "Card" variable describing the set of existing cards. The attributes of "CARD" are "codedpin" which is the pin number of the card, "cardid" that represents the card identity, "lastday" which is the last date where the card was used, "lastamt" which is the total amount withdrawn during the last day and "remaintry" which is a counter for the number of code attempts.

$$
\begin{array}{|l}
\textit{CARD} \\
\hline
\textit{codedpin} : \textit{CodedPIN} \\
\textit{cardid} : \textit{CARDID} \\
\textit{lastday} : \textit{DATE} \\
\textit{lastamt} : \textit{AMOUNT} \\
\textit{remaintry} : \mathbb{N} \\
\hline
\ldots \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{CardExt} \\
\hline
\textit{Card} : \mathbb{F}\,\textit{CARD} \\
\hline
\ldots \\
\hline
\end{array}
$$

These skeletons must be completed by annotations like the constraints which correspond to the predicate part. For instance, among the constraints presented in Sec. 3.2, the first and the third ones are related to the attributes of "CARD" . The first one specifies that the daily amount must be greater than zero but less than the maximal amount authorized. It is related to the "lastamt" attribute and it is represented by two Z expressions: $\textit{lastamt} \geq 0$ states that the last amount must be greater than zero and $\textit{lastamt} < \textit{MaxAmount}$ expresses that it must be inferior to the maximal amount. For the constraint on the codedpin, let us call "codefct2" a function between a pin and a coded one. Then the second constraint "the PIN must be coded/decoded by a given function" is expressed in Z by $\textit{codedpin} \in (\operatorname{ran} \textit{codefct}2)$. The three Z expressions representing the constraints on the "CARD" attributes complement the Z skeleton of "CARD":

$$
\begin{array}{|l}
\hline
\textit{CARD} \\
\textit{codedpin} : \textit{CodedPIN} \\
\textit{cardid} : \textit{CARDID} \\
\textit{lastday} : \textit{DATE} \\
\textit{lastamt} : \textit{AMOUNT} \\
\textit{remaintry} : \mathbb{N} \\
\hline
\textit{lastamt} \geq 0 \\
\textit{lastamt} < \textit{MaxAmount} \\
\textit{codedpin} \in (\operatorname{ran} \textit{codefct}2) \\
\hline
\end{array}
$$

Following the same principle of annotations, a complete specification of the class operations can be produced. This permits to fully automate the production of Z specifications from an annotated class diagram. This is realized by our tool, RoZ [8] which is an extension of the Rational Rose$^{(TM)}$ environment to integrate UML and Z notations. It translates the UML constructs and merges them with the diagram annotations (constraints and operation specification) expressed in Z. The Rose environment is used to build the UML models which are complemented by formal annotations in Z. The annotations are expressed into forms. A form should contain all the information mandatory to complete Z skeletons. So each form corresponds to a kind of annotation and it is attached to the class diagram element that it complements.

For the cash point service, the class diagram is edited in the usual Rose environment (Fig. 6). The constraints related to the attributes of “CARD” are expressed in Z style in the forms corresponding to these attributes. For instance, the constraint specifying that the amount withdrawn during a day must be greater than zero and inferior to the maximal amount authorized is expressed in the form related to the “lastamt” attribute.

The class diagram gives us the structure of the corresponding Z specification while the constraints in the forms add the details. So the “CARD” class is translated into a Z schema which contains the attribute declarations. This schema is complemented by the constraints on the amount and on the coded pin. So RoZ automatically produces the “CARD” schema given previously. Similarly, RoZ proposes forms to write the predicates corresponding to the operation pre and post-conditions so that it can also produce automatically the operation specifications.

So for the FM’99 contest, RoZ generated the complete Z specification corresponding to the information contained in the cash-point service class diagram and its forms. This specification is composed of:

- eight Z schemas for the four classes “BANK” , “ACCOUNT” , “CARD” and “TILL” ;
- seven schemas for the seven associations ;
- twenty operation schemas corresponding to the twenty operations.

4.2 Using a prover to validate the static part

At this point, we have produced a Z specification and we can start to investigate the properties of our model. In particular we want to check that the class diagram and its constraints are not

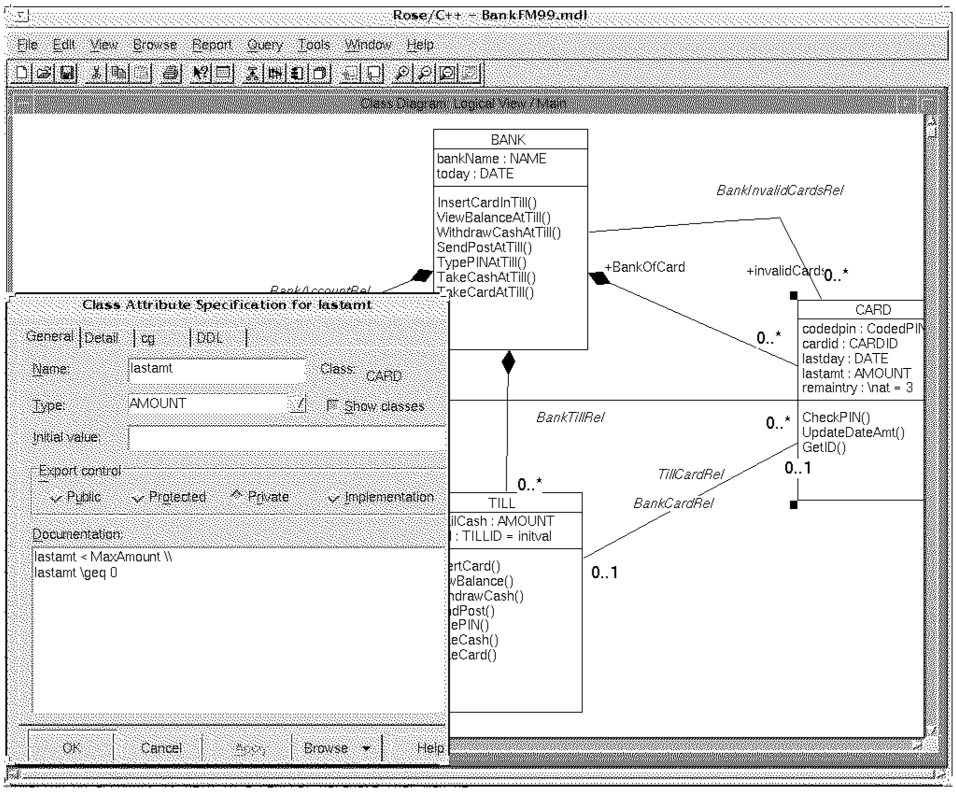

Figure 6: Rose interface

inconsistent or that the constraints are not mutually inconsistent. Then to validate the specification, we propose to prove that the operations do not violate data integrity imposed by the class diagram and its added constraints. So we "design" guards which will be evaluated before the operation execution and which can avoid to execute an operation violating the constraints. Each guard is validated by proving that it is actually a pre-condition of the operation.

Let us consider the operation to update the amount withdrawn during the current day. "CARDUpdateDateAmt" has as input parameters the date ("d?") and the amount to withdraw ("a?"). The first predicate states that the amount "a?" must be greater than zero. The two following ones express the modification of the amount withdrawn at the date "d?". If "d?" is the day of the last withdrawal, "a?" is added to the last amount. Otherwise no withdrawal has been realized during the day ; the last amount takes the values of "a?" and the date of the last withdrawal becomes "d?" . The other attributes of "CARD" keep their values.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CARDUpdateDateAmt}} \\
\hline
\Delta \mathit{CARD} \\
d? : \mathit{DATE} \\
a? : \mathit{AMOUNT} \\
\hline
a? > 0 \\
d? = \mathit{lastday} \Rightarrow \mathit{lastamt}' = \mathit{lastamt} + a? \land \mathit{lastday}' = \mathit{lastday} \\
d? \neq \mathit{lastday} \Rightarrow \mathit{lastamt}' = a? \land \mathit{lastday}' = d? \\
\mathit{codedpin}' = \mathit{codedpin} \\
\mathit{cardid}' = \mathit{cardid} \\
\mathit{remaintry}' = \mathit{remaintry} \\
\hline
\end{array}
$$

"CARDUpdateDateAmt" changes the amount withdrawn during a day, but it does not guarantee that the constraint on the maximal amount to withdraw by day. In Z, this constraint is implicitly included in the operation by $\Delta \mathit{CARD}$. In order to avoid to execute "CARDUpdateDateAmt" when the last amount is greater than the limit authorized, we propose to identify a guard for this operation. Here the chosen guard is composed of several conditions. If some withdrawal has already by done at the date "d?", the total amount of this day must be inferior to the maximal amount ($\mathit{lastamt} + a? < \mathit{MaxAmount}$); otherwise no amount has been withdrawn yet and the amount to withdraw must be inferior to the maximal amount ($d? \neq \mathit{lastday} \land a? < \mathit{MaxAmount}$). The theorem "CARDUpdateDateAmt_Pre" states that this guard is a condition strong enough to logically imply the weakest pre-condition of "CARDUpdateDateAmt", given by the Z operator "pre ".

theorem CARDUpdateDateAmt_Pre
$\forall \mathit{CARD};\ d? : \mathit{DATE};\ a? : \mathit{AMOUNT}$
$\mid (\mathit{lastamt} + a? < \mathit{MaxAmount} \lor (d? \neq \mathit{lastday} \land a? < \mathit{MaxAmount}))$
$\bullet \ \mathrm{pre}\ \mathit{CARDUpdateDateAmt}$

In fact, the proof obligations to validate operation guards are constructed according to the following form [16]:

theorem
$\forall \mathit{State}, i? : \mathit{IN} \mid \mathit{guard}(\mathit{State}, i?) \bullet \mathrm{pre}\ \mathit{Op}$

This theorem expresses that for an operation "Op" which modifies the schema "State" and has "i?" as input parameter, the proposed guard "guard(State,i?)" implies the weakest pre-condition of "Op" ($\mathrm{pre}\ \mathit{Op}$). Using this framework, the RoZ tool can generate the proof obligation to validate each operation guard. Actually it allows the designer to record a guard for each operation and produce the corresponding proof obligation. For the "UpdateDateAmt" operation, its specification is contained into Rose forms as we have shown previously for the constraints. For instance, its guard is registered in RoZ in the "Pre conditions" field of its form (Fig. 7). This allows RoZ to produce the "CARDUpdateDateAmt" schema and the "CARDUpdateDateAmt_Pre" theorem given above.

Finally the Z-EVES prover (from ORA [19]) is used to discharge the proof obligations. Z-EVES features both interactive and automatic mode, automatic mode being used for simple theorem. Actually the automatic mode can also be used as a help to simplify theorems. For the

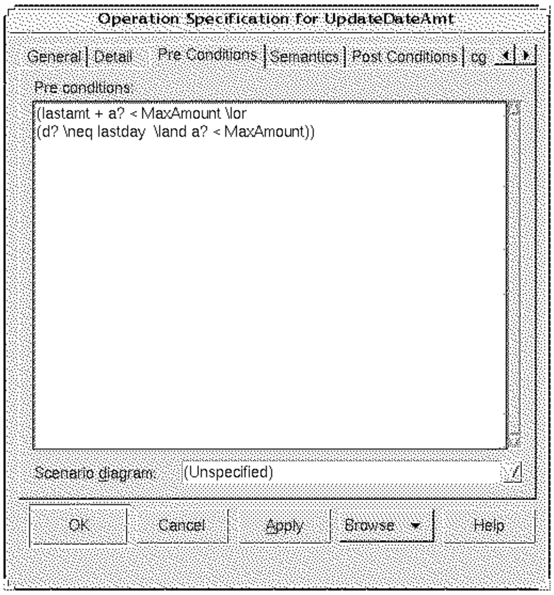

Figure 7: The"UpdateDateAmt" form

"cash-point" service, we used Z-EVES to validate the guards of fifteen operations. The use of Z-EVES helps us to correct our specification by finding mistakes or by helping us to make precise some annotation.

The tool helps us three different ways. First, it helps us to precise the semantics. Second, when a constraints is difficult to write, the tool help us to write the formulae. For the operation specifications, Z-Eves has helped us to identify the two cases (whether the day of the withdrawal is the current date) of the "CARDUpdateDateAmt" operation and of its guard. Finally, we discover some mistakes. For instance, we have forgotten that the balance of account can be equal to zero, and we also have forgotten to specify the case that the last amount of a card must be equal or greater to zero. For the operation specifications, Z-Eves has helped us to identify the two cases (whether the day of the withdrawal is the current date) of the "CARDUpdateDateAmt" operation and of its guard.

5 Validation of the dynamic part of UML model

Thanks to the dynamic part validation, we want to give the user some information helping him to decide whether his UML model behaves according to his hopes. In order to do that, we propose to animate the statechart views of the model. The animation technique consists in considering the system under test as a black box, to simulate its environment behavior, and to check if its reactions are the ones that are expected.

As an example, let us study the validation of a system composed of the bank and one till (Fig. 8(a)). The system environment is composed of one user.

To simulate the environment behavior, one has to determine what are the possible user actions, and when it is possible to do them. Basically, the user actions are: insert a card or take the card off, type a cancel order, type a pin, and ask for a withdrawal, for an account balance, or for statement of the account to be sent by post. Let us consider that it is always possible to type a command on the keyboard. It is from the responsibility of the till to determine when a command should be taken into account. For the actions "insert card" and "take the card off", it is a little bit different, because "physically", it is not possible do to them whenever one wants. If there is no stolen or damage cards, it is only possible to take the card off after one has been introduced, and after the till has given it back. If there is stolen or damage cards, it is possible to introduce a new card in the till, only after a "sorry" message. Figure 9 describes a model for such an environment (for the clarity of the figure, the connections between the user and the till statechart are not fully described).

The validation activity consists in firing the transition of the user statechart, and observes whether the till reactions are those which were expected. For instance, let us fire the transition "OpWithdrawal" at the initial state of the user statechart. If the till specification reaction is to give some money, then there is clearly a problem.

During the contest, we use Lutess testing tool to automatically animate the UML model. Lutess is a tool which produces automatically and dynamically test data with respect to some environment constraints of the program under test [6]. Lutess requires an executable program under test and a description of the environment in Lustre. Lustre is a programming language, which can be seen also as a past temporal logic [4].

To obtain an executable program from the UML model, several solutions can be studied. During the FM contest, we choose to translate UML statecharts in a Lustre program. This step was done manually but systematically. This was long and error prone. Currently, we study some solutions to automate the translation.

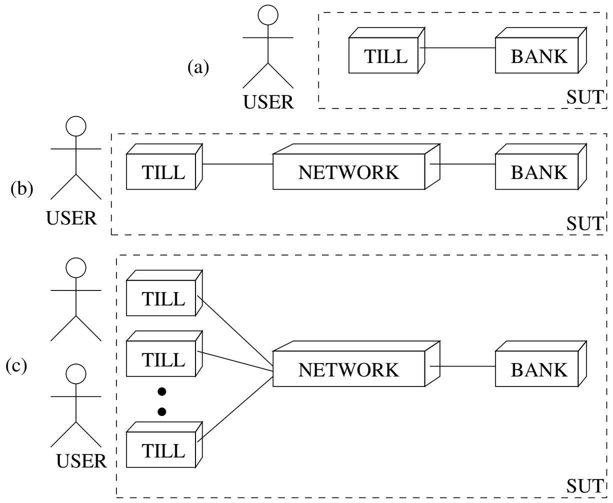

Figure 8: Example of test configuration for incremental validation

To validate a system, it is generally possible to proceed in an incremental way (see Fig. 8). This approach facilitates the error diagnosis. For the "cash-point" service , we validate first a system

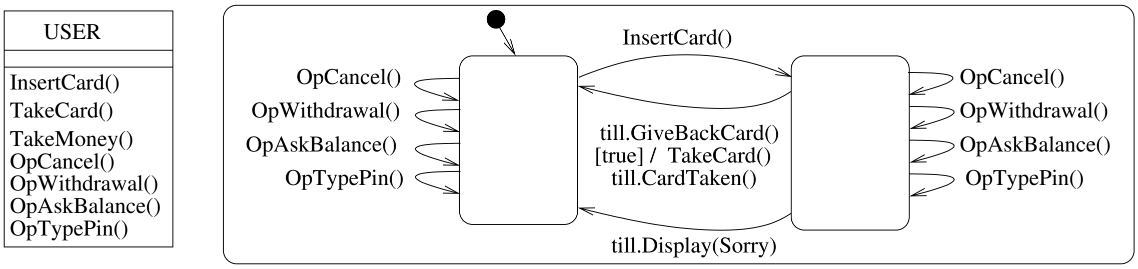

Figure 9: An environment model

composed of one till and the bank (Fig. 8(a)). Next, we introduced the network (Fig. 8(b)), and then several other Tills (Fig. 8(c)).

During the contest, the testing phase allows is to discover several incorrect or missing behaviors. For instance, it is possible to detect that in the statechart depicted figure 5(c), it is not possible to cancel the choice of an operation (contrary to the informal requirements).

The validation phase also shows that our first UML model was not robust enough at the database level: the database took all messages from a till into account even those corrupted by the network. This point was due to incompatible implicit assumptions which were done during the statechart conception. When the bank statechart was built, it was supposed (implicitly) that all the messages were correct. However, the validation phase shows explicitly that (1) the network could corrupt the messages, and (2) that the tills could generate incomplete messages, or send a same message several times. This allows us to adjust the security verifications (at different level of the system).

6 Conclusion

In this paper, we describe how semi-formal language (with UML) and formal ones (Z, Lustre) can be mixed to elaborate and validate a specification from informal requirements. Our approach is based on a translation process from UML specification into Z and Lustre ones. We illustrate it with the "cash point" service problem which was proposed as a contest during the world congress on Formal Methods, in September 1999.

This case study has shown that our approach enables us to better understand the system requirements and to improve our specifications in a progressive way. First, the development of UML specifications adds precision to the initial problem statement. Then the translation into formal specification makes the system specification more precise and provides the basis for their validation thanks to tools supporting the formal notations. The use of RoZ and the Z-EVES prover helps to validate the static part of the system by checking the consistency of the UML class diagram and its annotations. For the dynamic part, test or animation (here with Lutess tool) allows to discover erroneous or missing cases of behaviors. The main help provided by our tools concerns the writing of correct specifications which could also be used to find mistakes in the requirements document. So UML specifications give rise to translations into various formal languages whose choice depends on the properties to check.

The contest has shown the interest of the approach by showing us its benefits. Nevertheless the validation of the models still requires to have a deep knowledge into formal languages and their tools. Then a first perspective of our work could be to develop a domain knowledge that could help in using formal tools. For example, we have noticed that proofs have often the same forms for the same type of constraints. So it seems possible to propose some standard proofs for standard operations like attribute modification or add/suppression of an object, and to add them as proof strategies in RoZ (like in PVS [5]).

We could also try to take advantage of the formalization in Z or Lustre by exploiting other tools supporting these languages. For example, we could simulate Z specifications or use a model checker such as Lesar [12] to prove Lustre ones. The use of such a model-checker helps to prove that with the specified behavior, the model could never do forbidden actions. And if it is possible, the model-checker returns a counter-example.

Finally considering the utility of the translation of UML models into Z or Lustre specifications, it could be interesting to extend it to other UML models. In particular, it could be interesting to exploit the information given by UML scenarii to propose test series. This could be another way of considering and validating the information given informally in UML.

Following the principle of doing partial translation into the more appropriate formal language, we could also try to exploit other formal languages or other tools to validate UML models. For instance, UMLAUT is interesting for the validation of the dynamic part . It is a UML model editor and transformation framework, developed by IRISA [14]. It allows to produce automatically an executable program. Moreover, it is connected to the CADP library [11], which provides several validation tools (animating tool, testing environment, model-checker...). So our future work could be to connect RoZ (for the validation of the static part of a UML model) and UMLAUT (for the validation of the dynamic part).

References

[1] J.R. Abrial. *The B-Book.* Cambridge University Press, 1996.

[2] D.J. Andrews, H. Bruun, B.S. Hansen, P.G. Larsen, N. Plat, et al. *Information Technology — Programming Languages, their environments and system software interfaces — Vienna Development Method-Specification Language Part 1: Base language*. ISO, 1995.

[3] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language- User Guide*. Addison-Wesley, 1998.

[4] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL), Munich*, pages 178–188. ACM, 1987.

[5] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to pvs. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, USA, April 1995.

[6] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*. ACM, May 1999.

[7] S. Dupuy and L. du Bousquet. "Cash-Point service": a multi-formalim approach for specification. technical report PFL, IMAG - LSR, Grenoble, France, 1999.

[8] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ : a Tool for Integrating UML and Z Specifications. In *12th Conference on Advanced information Systems Engineering-CAiSE'2000*, volume 1789 of *Lecture Notes in Computer Science*, Stockholm, Suède, 2000. Springer-Verlag.

[9] S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. Vers une intégration utile de notations semi-formelles et formelles : une expérience en UML et Z. *L'Objet, numéro thématique Méthodes formelles pour les objets*, 6(1), 2000.

[10] R. France, JM. Bruel, M. Larrondo-Petrie, and M. Shroff. Exploring the Semantics of UML type structures with Z. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 247–260, Canterbury, UK, 1997. Chapman and Hall, London.

[11] Hubert Garavel. Open/cæsar: An open software architecture for verification, simulation, and testing. In *Proceedings of the First Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS 1384, Springer Verlag, 1998.

[12] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Programming Language LUSTRE. *IEEE Transactions on Software Engineering*, pages 785–793, september 1992.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[14] Jean-Marc Jézéquel, Alain LeGuennec, and Francois Pennaneach. Validating distributed software modelled with UML. In *Proc. Int. Workshop UML98, Mulhouse, France*, June 1998.

[15] K. Lano and S. Goldsack. Intregrated Formal and Object-Oriented Methods: The VDM++ Approach. In A.Bryant and L.Semmens, editors, *Method Integration Workshop*, Electronic Workshop in Computing, Leeds, Mars 1996. Springer-Verlag.

[16] Y. Ledru. Identifying pre-conditions with the Z/EVES theorem prover. In *Proc. of the 13th Int. Conf. on Automated Software Engineering*. IEEE, 1998.

[17] E. Meyer and J. Souquières. A systematic approach to transform OMT diagrams to a B specification. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems - FM'99*, volume 1708 of *Lecture Notes in Computer Science*, pages 875–896, Toulouse, France, 1999. Springer-Verlag.

[18] F. Polack, M. Whiston, and K. Mander. The SAZ Project: Integrating SSADM and Z. In *Intenational Symposium Formal Methods Europe*, Odense, Danemark, Avril 1993.

[19] M. Saaltink. The Z/EVES system. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–88, Reading, UK, april 1997. Springer-Verlag, Berlin.

[20] J.M. Spivey. *The Z notation*. Prentice-Hall International, 1994.

[21] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, 1998.