An. Şt. Univ. Ovidius Constanţa Vol. 10(1), 2002, 35–48

VISUAL REPRESENTATION OF FORMAL SOFTWARE SPECIFICATION

Crenguţa Bogdan

Abstract

Formal software specification has long been propagated as a way to increase the quality and reliability of software. However, the process of construction of a formal specification remains a hard activity for the most modelers, because they should use mathematical techniques, which those they aren't familiar.

In this paper we will see that it is easier to reason upon formal specifications if we translate them into visual diagrams. The diagrams form some kind of semiformal specifications, which have the advantage of easier readability and understandability.

1 Introduction

A specification is an abstract description of a system with its properties. The system properties usually include: functional behavior, internal structure, temporal behavior, performances, and real-time and security constraints.

Carrying out a specification assumes a specification process, itself an integral part of the software process. Any specification method presumes a specification language, the main tool for building specifications.

According to the specification language used, the specifications are classified in:
- semiformal and
- formal.

A semiformal specification uses graphical techniques such as: data flow diagrams (DFD), entity-relationship diagrams (ERD) and class diagrams, as

Key Words:
Mathematical Reviews subject classification:
Received: June, 2001

well as some kind of structured specification techniques written in natural language such as: event-response specification or contracts.

A formal specification is a specification written in a formal specification language. The goal of any formal specification is to obtain a unambiguous and precise description of the software system, as a base for the system subsequent specification refinements, verification and validation. The formal specification emphasizes what is expected from the system, rather than how it will be realized.

A formal specification language must be itself formally defined that is with both a formal syntax and a formal semantics. This is why mathematical techniques are used for the specification building and handling. Formal specifications and their languages make heavily use of discrete mathematics and the first-order logic concepts such as: set, relation, Cartesian product, predicate, etc.

2 A case study

Let us consider the following problem:

> A safety software system for housing areas:
> - enables its owner to configure it during installation,
> - controls through its sensors the environmental areas against fire and burglary and
> - interacts with the owner through the keypad of the system control panel. During installation, the system programming and configuration is carried out by using the numerical and functional keys of the control panel. Each sensor in the system is identified by a number and a type (fire or burglary). The system stores two passwords (each of them is up to six digits) used for the system activation/deactivation and a telephone number used for emergency calls when an alarm event arises. The system polls the fire sensors for M seconds, then disables them for other N seconds. After that, the system resumes the same cycle. The Ready indicator is lighting when the sensors are polled and is turned off when the sensors are disabled.
>
> When a sensor indicates an event, the system launches an alarm signal. After K seconds, the system calls the security forces office and provides it with information about the event nature and location.
>
> In order to log in, the owner enters the password and presses Enter. If any of the six digits is wrong the password is ignored by the

system and the sensors's state doesn't change. The owner can cancel the password with the Clear key.

If a sensor is activated, the system is armed and the Armed indicator on the control panel is turn on. In the case of the sensor deactivation, the Armed indicator is turn off.

Moreover, the system logs all events. Each logged event is characterized by its type and the date and time of its occurrence. The event list can be delivered by pressing the MEM key.

3 Z Specification of the System

The Z language ([1], [8]) is used for system modeling.

A specification in Z is a system mathematical model. The model description has two parts:
- the system state,
- the system operations.

The state specification describes the system entities, their types, the relations between them and the restrictions on entity properties. In order to specify a system, Z uses the set theory for the definition of the entities and their types, and the mathematical logic for the description of the constraints on the entity properties.

An operation represents a change of the system state. It may also supply some outputs. The operation specification defines the relation between the system's states before and after the operation execution. It also defines how the output values depend on entries and the initial state. Each operation in the system model is described with the next pattern:
- entries,
- start state,
- end state,
- outputs.

An entry is an input variable of the operation. Its type and preconditions describe it. An output is an output variable of the operation.

To structure a specification, Z introduces a new construction: schema. There are state schemas and operation schemas. A schema contains a declaration part and a predicate part. In the declaration part introduces the static component of the specification, while the predicate part introduces the dynamic component, that is relations and constrains on the entities declared in the first part.

In our safety system we identify as basic types the following entities:
$[ControlPanel, SystemParameters, Sensor, Alarm, Event, SupervisorService]$

SecuritySystem

$$\begin{array}{l}
contain : \mathbb{P}_1 Sensor \\
memorize : \mathbb{P} Event \\
configure : ControlPanel \rightarrowtail\!\!\!\!\twoheadrightarrow SystemParameters \\
control : Sensor \twoheadrightarrow ControlPanel \\
release : Sensor \leftrightarrow Alarm \\
associate : Event \rightarrowtail\!\!\!\!\twoheadrightarrow Alarm \\
signal : Sensor \rightarrow ControlPanel \\
obtaine : Event \rightarrow SystemParameters \\
resolve : Event \rightarrow SupervisorService \\
\hline
dom\ (resolve) = memorize \\
dom\ (release) \subseteq contain
\end{array}$$

Figure 1: Z state schema of security system

3.1 The State Schema of System

A state schema for the safety system is shown in the Figure 1. The specification in the Figure 1 models the system state with the following variables:
- *contain* represents the set of all the sensors of the system;
- *memorize* represents the set of all shown events;
- *configure* represents the system parameters configuration: the passwords and telephone numbers, inserted with the control panel;
- *control* represents the control of the sensors through the control panel;
- *release* represents the fact that each sensor releases an alarm;
- *associate* describes that any event corresponds a alarm ;
- *signal* models the fact that each sensor displays events on the control panel;
- *obtain* models the fact that each event associates a phone number from the system parameters;
- *resolve* models the fact that the supervisor service has been called and received information on the event.

In this specification, *memorize* and *contain* represent entity sets. All other variables represent relations and functions. For example, the control variable is a total surjection from *Sensor* to *ControlPanel*, because the system has a control panel that controls all the system sensors.

Two multiplicity constraints are presented in the second section of the specification in Figure 1. In the next subsection, we present how the type of the used functions/relations determines them.

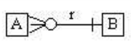
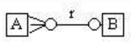
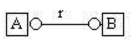
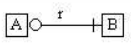
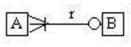
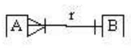
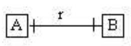
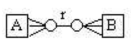
Z Relation	Predicate	ERD Relation
r $A \nrightarrow B$ partial	dom (r) $\subseteq \wp$ A ran (r) $\subseteq \wp$ B	
r $A \rightarrow B$ total	dom (r) $= \wp$ A ran (r) $\subseteq \wp$ B	
r $A \rightarrowtail\!\!\!\!+ B$ partial injection	dom (r) $\subseteq \wp$ A ran (r) $\subseteq \wp$ B	
r $A \rightarrowtail B$ total injection	dom (r) $= \wp$ A ran (r) $\subseteq \wp$ B	
r $A \twoheadrightarrow\!\!\!\!+ B$ partial surjection	dom (r) $\subseteq \wp$ A ran (r) $= \wp$ B	
r $A \twoheadrightarrow B$ total surjection	dom (r) $= \wp$ A ran (r) $= \wp$ B	
r $A \rightarrowtail\!\!\!\!\rightarrow B$ bijection	dom (r) $= \wp$ A ran (r) $= \wp$ B	
r $A \leftrightarrow B$ relation	dom (r) $\subseteq \wp$ A ran (r) $\subseteq \wp$ B	

Figure 2: Translation rules from Z schemas to ERD

3.2 Representating Static Aspects of a Z Specification with an ERD

Let us translate the state schema in an ERD (Entity-Relationship Diagram) ([6]). In this translation we will follow the rules:
- the state schema name is also a name of an entity type, that is the system is an entity type in ERD;
- variables representing entities are translated in ERD as entity types and connected to the system entity with relations;
- free types are translated into super types. Their values are subtypes;
- variables representing relations between entities are translated in ERD as relations or type indicator. Multiplicity constrains for associations are determined by the type of functions or relations used to define the variables and constraints. The table from Figure 2 resumes these translation rules, presented also in ([3]). The ERD in Figure 3 is obtained from the table and the state schema in the Figure 1.

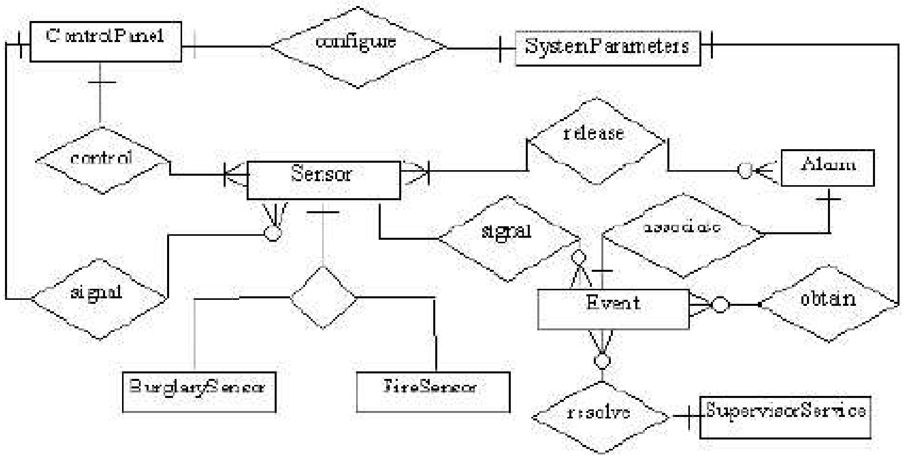


Figure 3: An ERD for Z specification

```
── Sensor ──────────────────────────────
│ number : ℕ
│ type : SensorType
│ state : SensorState
└───────────────────────────────────────
```

Figure 4: Type schema for the set of sensors

3.3 Operation Schemas of the System

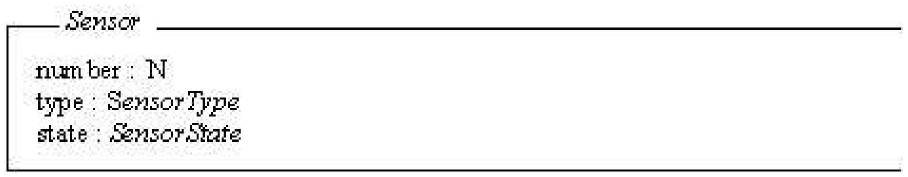
In the following, we model the sensor's state and type through the free types: $SensorState ::= active|inactive, SensorType ::= fire|burglary$.
In this case we can represent the set of sensors with the next type schema from the Figure 4.

To specify the parameters of the system, we define
$Digit == 0..9$, and $seq_6[Digit] == \{s \in seqDigit | \#s = 6\}$ the set of the sequences with six digits.

We can use arrays to memorize the passwords and phone numbers:
$Array = [array : seqData]$, where $Data = [value : seq_6[Digit]]$.

Now we can define the $SystemParameters$'s type schema from the Figure 5.

The process of checking of a password contains in a checking digit after digit conformably with a password memorized by the system. First of all, we


Sensor

number : ℕ
type : *SensorType*
state : *SensorState*

Figure 5: Type schema for the set of sensors

should see which password the owner chooses, after the first digit introduced. That's why we define the free type:
$Boolean ::= false | true$ and the type schemas:
$DigitsNumber = [nr : 0..6]$,
$DigitsNumberInit = [DigitsNumber' | nr' = 0]$ and those from the Figure 6. The checking of the first digit is described through the operation schema from the Figure 7.

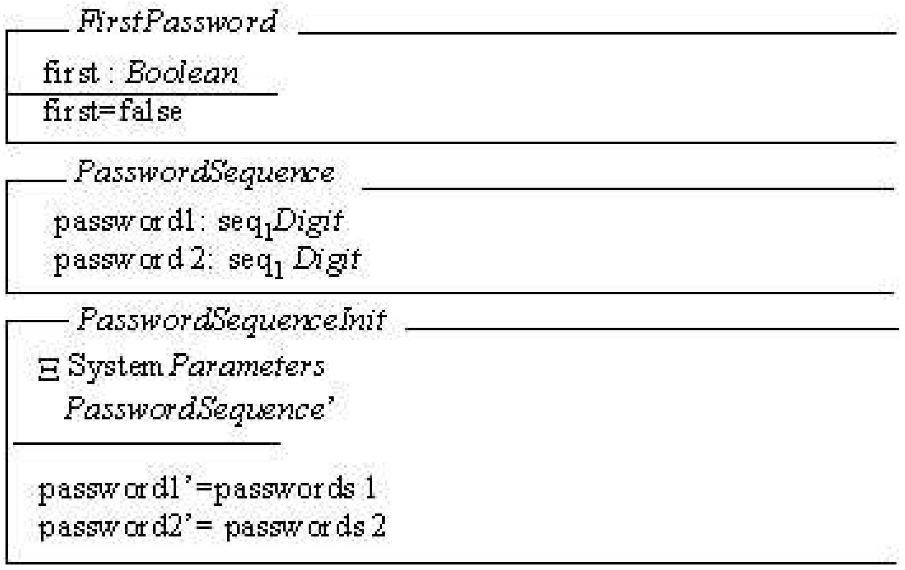
For the next digits, the checking is done in the same way. The operation schema that specifies the update of the sensor's state is in the Figure 8.

To treat the errors we define the free type $Report ::= ok | invalidePassword$ and the type schemas from the Figure 9.

In conclusion, we can combine these operation schemas to describe the operation $ControlPassword$:
$ControlPassword == (CheckFirstDigit \bigwedge FirstDigitOK \bigwedge CheckNextDigit \bigwedge$
$\bigwedge NextDigitOK \bigwedge UpdateSensorState) \bigvee Error$

3.4 Representing Dynamic Aspects of a Z Specification

To represent graphically the operation schemas of a Z specification, we will construct a DFD diagram ([6]) following the next rules:
- the operation schemas will be represented by data processes,
- the type schemas that treat boolean values and contain outputs parameters, which are values of the any free type, will be shown by the control flows,
- the sets specified through the type schemas will be data stores and the schema's variables will be data flows to/from stores, and
- basic types will be actors.
By example, the DFD for the $ControlPassword$ operation schema is shown in Figure 10.



```
FirstPassword
first : Boolean
first=false

PasswordSequence
password1: seq1Digit
password2: seq1 Digit

PasswordSequenceInit
Ξ SystemParameters
PasswordSequence'

password1'=passwords1
password2'= passwords2
```

Figure 6: Type schemas for FirstPassword, PasswordSequence and Password-SequenceInit

4 Object-Z Specification of the System

The Object-Z language allows a more complete and appropriate specification of systems. An Object-Z specification describes a system as a collection of interacting objects. Each object has a structure and a behavior.

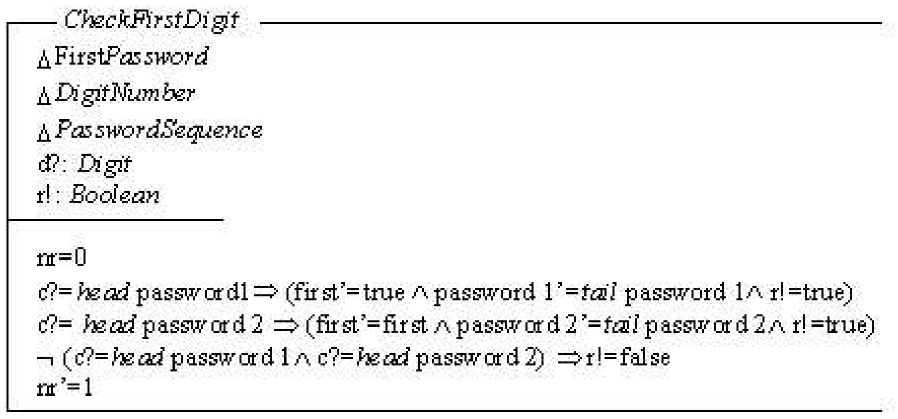
In general, an Object-Z specification consists of a number of class schemas ([2], [5]). A class schema captures the object-orientation notion of a class and encapsulates a state schema and a set of operation schemas. These operations may affect state variables.

Syntactically, a class schema is constituted from the elements in the Figure 11.

The specification of the formal generic parameters allows describing of the abstract data types.

Inheritance (simple and multiple) is described by including the names of the inherited classes within the inheriting class. In this case, the type, constant and schema definitions in the inherited class are merged with those declared explicitly in the inheriting class.

The type and constant definitions have the same syntax as the global type and the constant definitions in Z. However, their scope is limited to the class

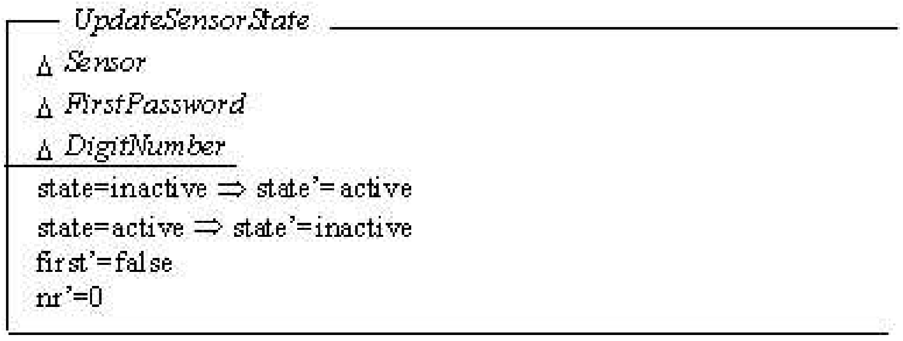


CheckFirstDigit

Δ FirstPassword
Δ *DigitNumber*
Δ *PasswordSequence*
d?: *Digit*
r! : *Boolean*

nr=0
c?= *head* password1 $\Rightarrow$ (first'=true $\wedge$ password 1'=*tail* password 1$\wedge$ r!=true)
c?= *head* password 2 $\Rightarrow$ (first'=first $\wedge$ password 2'=*tail* password 2$\wedge$ r!=true)
$\neg$ (c?=*head* password 1 $\wedge$ c?=*head* password 2) $\Rightarrow$ r!=false
nr'=1

Figure 7: Operation Schema for CheckFirstDigit



UpdateSensorState

Δ *Sensor*
Δ *FirstPassword*
Δ *DigitNumber*

state=inactive $\Rightarrow$ state'= active
state=active $\Rightarrow$ state'=inactive
first'=false
nr'=0

Figure 8: Operation schema for UpdateSensorState

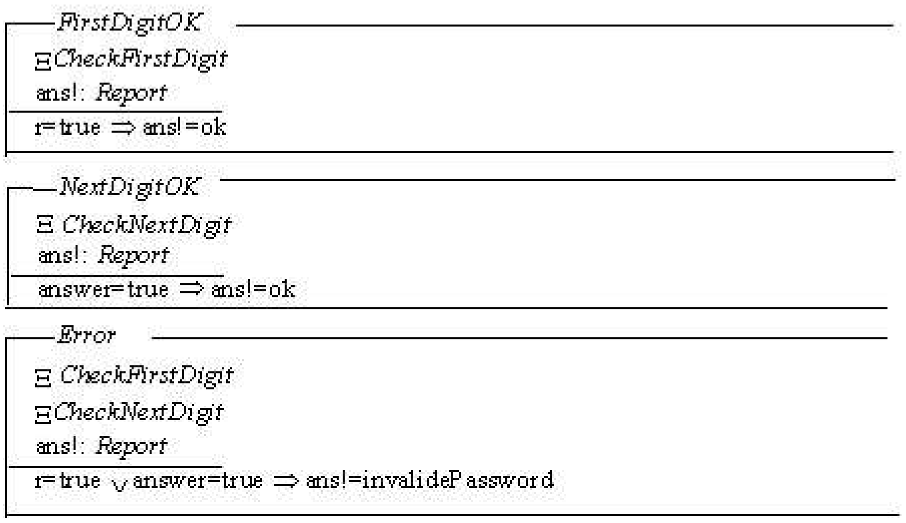
in which they are declared.

The state schema is a Z state schema except that it has no name associated with it. The declarations of the state schema are referred to as the state variables and the predicate as the state invariant.

The initial state schema is named INIT and shows the initial state of class's objects.

Operation schemas are Z operation schemas, excepting that they use the D operator to show the variables which values are modified when the operation is executed on the object of class.

A history invariant is a temporal predicate that constrains the behavior of the classes's objects.



FirstDigitOK
Ξ *CheckFirstDigit*
ans!: *Report*
r=true $\Rightarrow$ ans!=ok

NextDigitOK
Ξ *CheckNextDigit*
ans!: *Report*
answer=true $\Rightarrow$ ans!=ok

Error
Ξ *CheckFirstDigit*
Ξ *CheckNextDigit*
ans!: *Report*
r=true $\vee$ answer=true $\Rightarrow$ ans!=invalidePassword

Figure 9: Type schemas to verify the errors

Following the definition of our system, we construct o class for each data type also obtained in the Z specification, but now, the classes contain operations too, operations that should be specified in Z using operations schemas.

Now, the class PasswordController has responsibility that the control panel controls the sensors.

From the Object-Z specification of the system, we give in the Figure 12 only the class schemas of *SystemParameters* and *PasswordController*.

4.1 Visualizing the Object-Z Class Schemas

To construct a UML class diagram ([4], [7]) from an Object-Z formal specification, we may follow the next rules:
- every class schema will be a UML class, where the schema variables will be the attributes of the UML class. INIT schema corresponds to the class constructor and the operation schemas correspond to the UML class operations,
- the entries of the operation schema are formal parameters of the corresponding operation of the UML class and the output type gives the return type from the operation signature,
- the predicate part of every operation schema may be specified in note elements associated to UML class and can be expressed using the formal language OCL, which is part of UML language,
- if a class schema use a variable with type another class schema, then in the

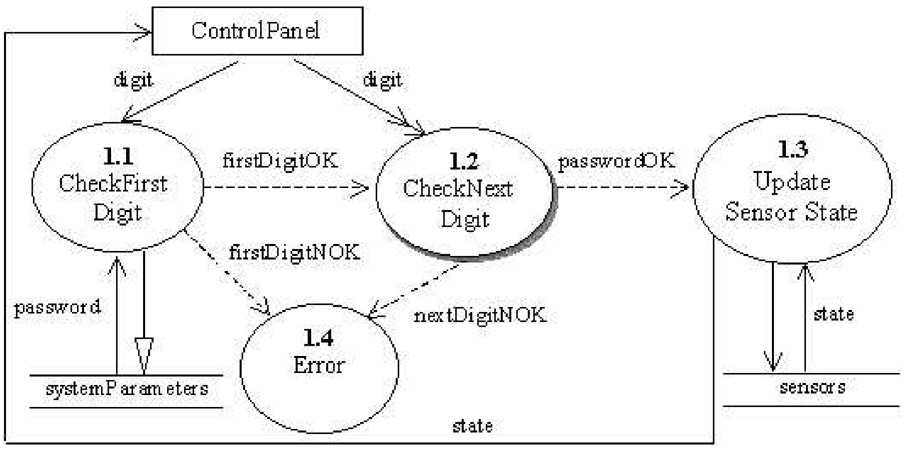


Figure 10: A DFD for Object-Z Specification

UML class diagram will be an association between the corresponding UML classes, with the name deriving from variable's name.

Applying the above rules for the system Object-Z specification of our system, we obtain the UML class diagram in the Figure 13.

5 Conclusions

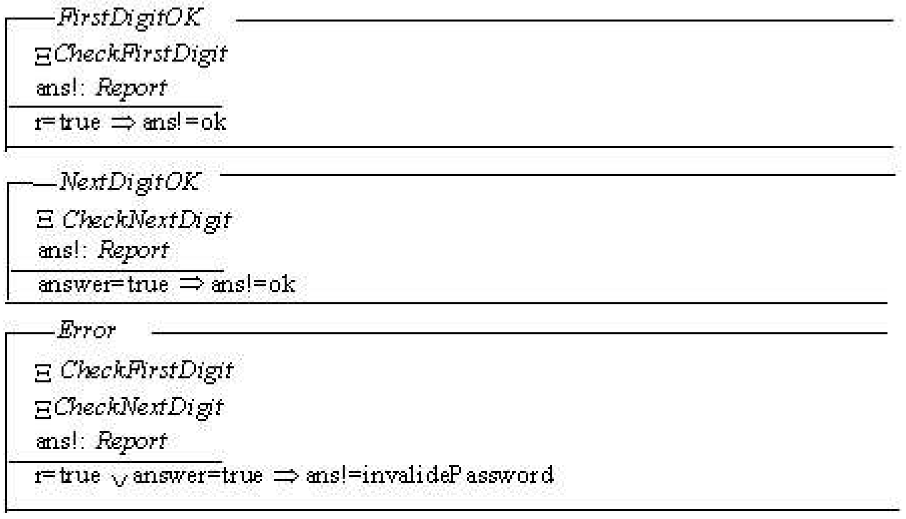
The paper presents the construction of graphical, semiformal specifications from formal specifications written in Z and Object-Z. The mapping has been done for a case study, by using translation rules informally specified for each language.

For the structured formal language Z, we mapped the Z specification of the system state in an entity-relationship diagram, a conceptual technique used in the structured analysis.

To construct an ERD, we have analyzed the semantic relations between the variables declared in the state schema and the syntactic constructs ERD. As result, the basic types were translated into entity types and the variables into the ERD relations and sub/super type indicator, respectively.

The dynamic aspects of Z specification are mapped in a DFD, that is in processes, data repositories, data flows, control flows, and actors specifications.

Representing an Object-Z specification with a UML class diagram was an almost linear mapping process, because Object-Z is an object-oriented language, so it's basic construct is class. The class's attributes and operations



```
┌─FirstDigitOK ─────────────────────
│ ΞCheckFirstDigit
│ ans!: Report
├──────────────
│ r=true ⇒ ans!=ok
└───────────────────────────────────

┌─NextDigitOK ──────────────────────
│ Ξ CheckNextDigit
│ ans!: Report
├──────────────
│ answer=true ⇒ ans!=ok
└───────────────────────────────────

┌─Error ────────────────────────────
│ Ξ CheckFirstDigit
│ ΞCheckNextDigit
│ ans!: Report
├──────────────
│ r=true ∨ answer=true ⇒ ans!=invalidePassword
└───────────────────────────────────
```

Figure 11: Type schemas to verify the errors

have been specified through the state schema and operations schemas, respectively.

In conclusion, we consider that formal specifications may be documented with diagrams, which provide a visual way for reading and understanding them easier.

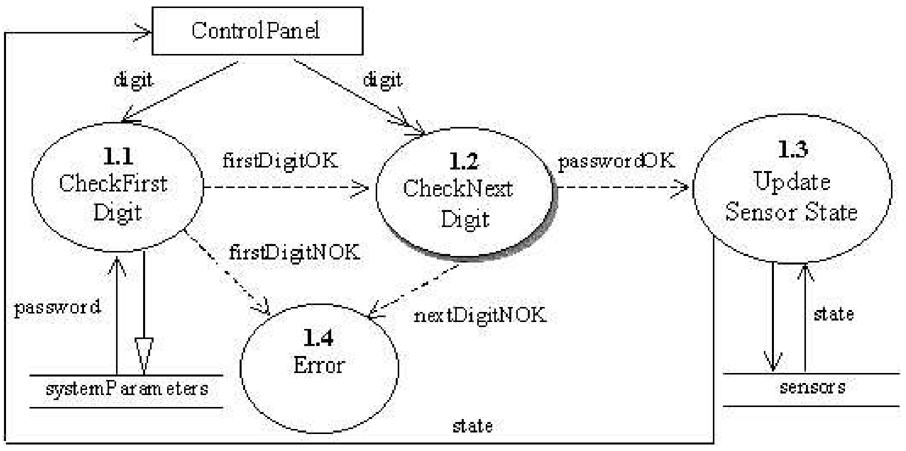
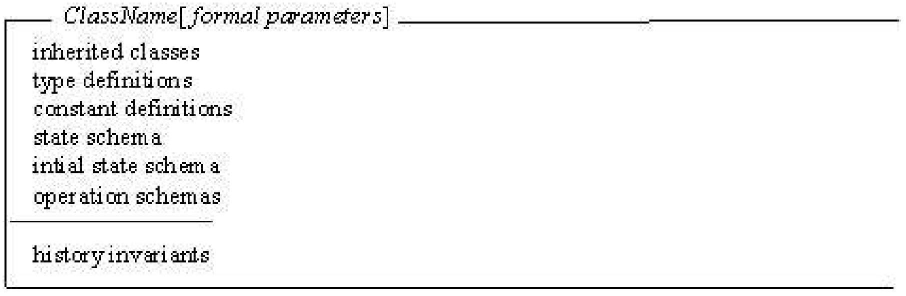


Figure 12: A DFD for Object-Z Specification



ClassName[*formal parameters*]
inherited classes
type definitions
constant definitions
state schema
intial state schema
operation schemas

history invariants

Figure 13: The syntax of Object-Z class schema

References

[1] J. P. Bowen, *Z: A Formal Specification Notation*, In Marc Frappier and Henri Habrias (eds.), Software Specification Methods: An overview U-sing a Case Study, Springer-Verlag, 2001.

[2] R. Duke, P. King, G. Rose, G. Smith, *The Object-Z Specification Language: Version 1*, Technical Report No. 91-1, The University of Queensland, 1991.

[3] S. K. Kim, D. Carrington, *Visualization of Formal Specifications*, Technical Report No. 99-47, The University of Queensland, 1999.

[4] J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language. Reference Manual*, Addison-Wesley, 1999.

[5] G. P. Smith, *An Object-Oriented Approach to Formal Specification*, PHD Thesis, The University of Queesland, 1992.

[6] L. D. Şerbănaţi, *Analiza şi proiectarea structurată a sistemelor*, course in Romanian, 1998.

[7] L. D. Şerbănaţi, *Curs de ingineria programării. Modelarea orientată spre obiecte*, course in Romanian, 1997.

[8] J. Woodcock, J. Davis, *Using Z. Specification, Rafinement and Proofs*, Prentice Hall, 1999.

"Ovidius" University of Constanta,
Faculty of Mathematics and Informatics,
8700 Constanta,
Romania
e-mail: cbogdan@univ-ovidius.ro