3 BACKGROUND: Notations

"A good notation has a subtlety and suggestiveness which at times make it almost seem like a live teacher."

[Bertrand Russell, in Introduction to L. Wittgenstein's Tractatus Logico-Philosophicus, 1922]

3.1 Introduction

This chapter shifts the focus from the distinguishing aspects of the domains that make up the thesis' research space to particular details of the two specification languages, Z and UML, that provide the notational basis for the modelling approach proposed in this thesis. Presentations of the main features of both Z and UML are included, and the salient components of the two notations are illustrated by short examples. A look at the larger family of Z-based languages and a brief introduction of Z++ (the OO variant of Z employed in our approach) are also included and UML's support for modelling RTS is examined. In order to illustrate UML an Automatic Camshaft Testing System (ACTS) inspired from our previous work on developing software for an automobile manufacturing company is employed as a "recurrent theme" from which short examples are extracted. Observations regarding current directions of exploiting the modelling power of UML are also presented.

3.2 Z and Flavours of Z

The formal specification language Z has been developed by the Programming Research Group at the Oxford University Computing Laboratory from the influential work of Jean-

Raymond Abrial [Abrial80] and has been used by various industrial organisations all over the world. Numerous variants of Z have been proposed over the years, including object-oriented alternatives. A model-oriented formal language based on set theory and first order predicate logic, Z is undoubtedly one of the most successful formal specification notations in terms of its acceptance by the software development community. The specialised literature contains numerous accounts of software and hardware products that have employed Z as a formal instrument for specification, for instance IBM's Customer Information Control System (CICS) [Nix88], Inmos transputers [Barrett89], Tektronix oscilloscopes [Delisle90], Bellcore's PLAN system for planning and administrating feeder loop networks [Morgan94], and Lloyd's Register's COBOL parser [Neil98]. More on the industrial use of Z can be found in [Gerhart94] and [Bowen95b] and a comprehensive set of pointers to Z resources is available at "The World Wide Web Virtual Library: The Z Notation" [Zed01]. Authoritative books on the syntax and semantics of Z are [Spivey92] and [Wordsworth92], while very good texts on the application of Z in practice are [Barden94] and [Jacky97]. Object-oriented versions of Z are comprehensively surveyed in [Stepney92a, Stepney92b] and amply illustrated in [Lano94a], while the most complete reference for Z++ is [Lano95].

This section continues with a summary overview of the main features of Z, including its types, predicates, relations, functions, schemas, and schema calculus. These features provide the foundation on which all Z variants have been built, including the object-oriented alternative Z++ employed in our dissertation. Variants of Z are then briefly surveyed in Subsection 3.2.2, followed by a succinct introduction of Z++ in Subsection 3.2.3. The presentation of Z++ is kept to a minimum here since the notation is further detailed in Chapter 6, in conjunction with the proposed formalisation process of UML constructs. Z++ is also briefly presented in Appendix A.

3.2.1 The Z Notation

In essence, a Z specification consists of a number of schemas, which describe both the static and the dynamic aspects of the system. The static properties of the system are captured in the

collection of possible states of the system and in the invariant relationships that must be satisfied as the system transitions from state to state. The dynamic properties of the system are modelled by the system's operations, the relationship between their input and outputs, and the changes in the system state. Schemas provide the necessary support for modularisation and refinement, allowing the developer to specify pieces of the software product separately and then relate and combine them using the rules of Z schema composition. Refinement is supported, abstract specifications being transformed into equivalent concrete schemas that contain additional details. This powerful mechanism for composition and refinement accounts for the notation's successful application to larger projects. Schemas are expressed formally and the effect of each operation is described abstractly using first order predicate logic expressions, but the entire Z specification can (and normally should) include textual annotations, natural language descriptions that clarify the meaning of the rather arid mathematical statements.

The remaining part of this Subsection summarises the main features of Z, as described in [Spivey92] and [Wordsworth92], and illustrates them with several short examples. The notation style follows the one of [Barden94].

3.2.1.1 Sets, Types, and Predicates

Z's set theory is a typed set theory, which means that every value in the specification is assigned to a type. From simple, basic types, it is possible to define more complex types via three ways of type composition: set types, Cartesian product types, and schema types. A type can be introduced in Z by a given set (or basic type), declared by writing its name in the given set brackets. For instance:

```
[LIGHT]
```
(3.1)

introduces the given set with the name `LIGHT`, declared to be a type covering values of a specific kind, used as "atomic" entities in a given application (in this case, we need to

describe the various lights that can be turned on or off in an apartment). As already mentioned, each variable declared in Z must have a type, for instance:

```
|  balconyLight: LIGHT                                          (3.2)
```

A set can be defined by set enumeration, that is by listing its members (of the same type) in order, separated by commas, and enclosed in brackets. For instance, possible kinds of room of interest in a particular application can be written:

```
{livingroom, bedroom, bathroom, kitchen, den, hall, extra}      (3.3)
```

A name can be given to a set introduced via set enumeration by using syntactic equivalence, specified by the == symbol, for instance:

```
ROOMKIND == {livingroom, bedroom, bathroom, kitchen,
             den, hall, extra}                                  (3.4)
```

A set with just one member is called a singleton set, while a set with no members is an empty set or a null set. The equality of two sets means that both sets have exactly the same members, so for two sets A and B with members of different types, both the `A = B` and `A ≠ B` notations are not well-formed because comparison is not possible between such sets. Typical set operations such as union (∪), intersection (∩), difference (\), and Cartesian product (×), as well as relationships such as subset (⊆), strict subset (⊂), and membership (∈) can be applied. The cardinality of a finite set `A` is denoted `#A`, while the set of all subsets of set `A` is denoted `ℙA` (powerset of `A`). The set of integer numbers is denoted `ℤ` and its type is `ℙℤ`. Similarly, natural numbers are in the set `ℕ`, which has the type `ℙℕ`. These two types are included by default in any specification and need not be introduced formally. Types with smaller number of values can be introduces using data definition, specified by the data definition operator (::=), for instance:

```
STATUS ::= on | off                                                    (3.5)
```

Another way of defining a set is by set comprehension, in the form:

```
{D | P • E}                                                            (3.6)
```

where D is a declaration, P a constraint imposed on values, and E an expression denoting the terms. The expression (3.6) denotes the set of values of term E for everything declared in D that satisfies the constraining predicate P. For instance, if TEMPERATURE == ℤ then:

```
{t: TEMPERATURE | t ≥ 0 ∧ t < 20 • t}                                  (3.7)
```

gives the set of all positive temperatures that are less than 20-degree Celsius.

More complex structures can be defined using schema types, for instance, assuming LENGTH and WIDTH are already introduced (e.g., LENGTH == ℕ and WIDTH == ℕ) then:

```
┌─── Room ──────────────────────────────────────────────────┐
│kind:                ROOMKIND                              │
│dimension:           LENGTH x WIDTH                        │   (3.8)
│temperature:         TEMPERATURE                           │
│lights:              ℙLIGHT                                │
└───────────────────────────────────────────────────────────┘
```

The components of composite type variables can be accessed using the dot notation. In the above case if libraryRoom is a Room variable then the specification can make use of libraryRoom.kind, libraryRoom.dimension, etc.

In Z, predicates provide formal ways of expressing the meaning conveyed by declarative sentences of the natural language. It is possible to build more complex predicates from simpler ones by using the connectives of the first-order logic: negation (¬), conjunction (∧), disjunction (∨), equivalence (⇔), and implication (⇒). In addition, the universal quantifier (∀) and the existential quantifier (∃) are available, as well as the unique existential quantifier

($\exists_1$), which indicates the fact that there is a single item satisfying a certain property. An example of a predicate, describing a situation that requires turning a heater on, is the following (assume that a `Heater` type, a `roomHeater` variable of `Heater` type, and the `comfortLevel` constant have been defined, and note the single quote decoration that indicates the "after-state" value of `roomHeater.status`):

```
(roomHeater.status = off) ∧
   (libraryRoom.temperature < comfortLevel)
      ⇒ roomHeater.status′ = on                    (3.9)
```

3.2.1.2 Relations, Functions, and Sequences

A relation is a set of ordered pairs. In a relation `R`, the first member of the pair belongs to a set `X`, while the second member of the pair to a set `Y` (`X` and `Y` need not be different). The notation for the relation `R` is:

```
| R : X ⟷ Y                                        (3.10)
```

where `X` is the from-set and `Y` is the to-set of relation `R`. To indicate the fact the `x∈X` and `y∈Y` are in relation `R` the following notation is used:

```
|  x ⟼ y ∈ R  or, equivalently,  x R y            (3.11)
```

The generic definition:

```
|  X ⟷ Y == ℙ(X x Y)                              (3.12)
```

describes all possible relations that can be defined from `X` to `Y`. For a given relation `R` defined as in (3.10), the domain of `R`, denoted `dom R`, is the set of first members of all pairs in

the relation, while the range of R, denoted ran R, is the set of second members of all pairs in the relation. They are defined, respectively, by the following expressions:

$$\mid \{x:X \mid \exists y \in Y \bullet x \longmapsto y \in R\} \tag{3.13}$$

$$\mid \{y:Y \mid \exists x \in X \bullet x \longmapsto y \in R\} \tag{3.14}$$

Two relations R and S, defined as $R : X \longleftrightarrow Y$ and, respectively, $S : Y \longleftrightarrow Z$ can be subjected to relation composition, denoted $R \fatsemi S$ and defined formally as:

$$\mid \{x:X ; z:Z \mid (\exists y \in Y \bullet (x \longmapsto y \in R \land y \longmapsto z \in S)) \bullet x \longmapsto z\} \tag{3.15}$$

A number of operators are useful when working with relations. Assuming that M is a set of members from the domain type X and N is a set of members from the range type Y, then the domain restriction operator $\lhd$, which confines relation R to those pairs whose first members are in the set of interest M is defined as:

$$\mid M \lhd R == \{x:X; y:Y \mid x \in M \land x \longmapsto y \in R \bullet x \longmapsto y\} \tag{3.16}$$

The range restriction operator $\rhd$ is defined symmetrically:

$$\mid R \rhd N == \{x:X; y:Y \mid y \in N \land x \longmapsto y \in R \bullet x \longmapsto y\} \tag{3.17}$$

It is also possible to use the domain subtraction operator $\lhd\!\!\!-$, which restricts the relation to those pairs whose first members are not in the set M. This operator is defined as:

$$\mid M \lhd\!\!\!- R == \{x:X ; y:Y \mid x \notin M \land x \longmapsto y \in R \bullet x \longmapsto y\} \tag{3.18}$$

Finally, the range subtraction operator ⩤ restricts the relation to those pairs whose second members are not in the set N. The range subtraction operator is defined as:

$$| \quad R \rhd N == \{x:X ; y:Y \mid y \notin N \land x \mapsto y \in R \bullet x \mapsto y\} \tag{3.19}$$

A function is a particular case of relation, in which each member of the from-set can be in relation with at most one member of the to-set. This is expressed formally by the following generic definition, which defines all possible functions from X to Y:

$$| \quad X \pfun Y == \{f: X \rel Y \mid (\forall x:X; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\} \tag{3.20}$$

In Z there are predefined symbols for partial functions, total functions, partial injections, total injections, partial surjections, total surjections, and bijections. Partial functions correspond to the definition (3.20), since there is no constraint on the from-set –specifically the domain of f may not be the entire from set X. The domain of a total function is the entire from-set of the function. The formal generic definition for total functions is:

$$| \quad X \fun Y == \{f : X \pfun Y \mid \mathrm{dom}\ f = X\} \tag{3.21}$$

An injection has the property that the second members of its pairs are unique. Injections can be partial injections, described by:

$$| \quad X \pinj Y == \{f : X \pfun Y \mid (\forall x_1, x_2 : \mathrm{dom}\ f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\} \tag{3.22}$$

or can be total injections, corresponding to:

$$| \quad X \inj Y == (X \fun Y) \cap (X \pinj Y) \tag{3.23}$$

A surjection has the property that its range is the whole of its to-set. Partial surjections are described by:

$$X \pfun\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Y == \{f : X \pfun Y \mid \mathrm{ran}\ f = Y\} \tag{3.24}$$

while total surjections are defined as:

$$X \fun\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Y == (X \fun Y) \cap (X \pfun\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Y) \tag{3.25}$$

Finally, a bijection is a function both injective and surjective:

$$X \rightarrowtail\!\!\!\!\!\rightarrow Y == (X \rightarrowtail Y) \cap (X \fun\!\!\!\!\rightarrow\!\!\!\!\!\rightarrow Y) \tag{3.26}$$

Since functions are relations, all operators that apply to relations apply as well to functions. In addition, an operator that guarantees that the result is a function and can be used for updating information is the function overriding operator $\oplus$. For instance, with the `[LIGHT]` given set (3.1) and the `STATUS` type (3.5), a partial function that keeps track of the lighting situation in a given environment can be defined as:

$$lightsynopsis : LIGHT \pfun STATUS \tag{3.27}$$

and turning on the particular light `balconyLight` can be described as:

$$lightsynopsis = lightsynopsis \oplus (balconyLight \mapsto on) \tag{3.28}$$

Sequences are particularly useful in software specification. A sequence of values of type `X` is an ordered collection of values and can be defined as a partial function from the natural numbers $\mathbb{N}$ to `X` with the property that its domain is `1 .. n` (where `n` is the length of the sequence). The generic definition that describes all possible sequences of values of type `X` is:

```
|   seq X == {f : ℕ ⤖ X | (∃ n : ℕ • dom f = 1 .. n)}        (3.29)
```

As shown in Table 3.I, where the definition (3.4) is used for `X`, a number of operators permit the extraction of useful information from a sequence `s`, namely `head(s)`, `tail(s)`, `last(s)`, and `front(s)`. Also, a sequence `s` can be reversed by applying the reverse operator `rev(s)` and two sequences `s` and `t` can be concatenated using the concatenation operator `⌢`.

Table 3.I A Summary of Sequences

Item	Description	Example [assume X = ROOMKIND as defined in (3.4)]
finite sequences	seqX == {f : ℕ ⤖ X | dom f = 1 .. n}	s = <hall, kitchen, den, bedroom, bedroom> emptyseq = <>
non-empty finite sequences	$seq_1 X$ = { f : seqX | #f > 0}	t = <livingroom, bedroom, extra, hall, extra>
injective sequences	iseqX == seqX ∩(ℕ ⤔ X)	u = <hall, den, livingroom>
head	head s = s (1)	head s = hall
tail	tail s ={ k : 1 .. #s - 1 • k ⟼ s (k + 1)}	tail s = <kitchen, den, bedroom, bedroom>
last	last s = s #s	last s = bedroom
front	front s = {#s} ⩤ s	front s = <hall, kitchen, den, bedroom>
reverse	rev s = {k : 1 .. #s • (#s – k +1)⟼ s (k)}	rev s = <bedroom, bedroom, den, kitchen, hall>
concatenation	s ⌢ t = s U { k : 1 .. #t • (k + #s) ⟼ t(k)}	s ⌢ t = <hall, kitchen, den, bedroom, bedroom, livingroom, bedroom, extra, hall, extra>

Z includes definitions for non-empty sequences ($seq_1\ X$) as well as for injective sequences ($iseq\ X$). The notion of bag can also be used in Z, with the expression:

$$bag\ X == X \pfun \mathbb{N}_1 \tag{3.30}$$

indicating that items $x_k \in X$ have a number of apparitions $n_k \geq 1$ in the bag.

3.2.1.2 Schemas and Schema Calculus

The fundamental building block for Z specification is the schema, which can be defined as a mathematical description of a part of the system under construction. Z schemas are used for describing both structural and behavioural properties of the system. They can express both the state space and the operations of the systems, and can be interconnected using the mechanisms of Z schema calculus. Thus, they provide support for modularity and composition, allowing the development of large-scale specifications. Schemas also provide the basis for refinement through schema transformation.

A Z schema has a name (used for reference throughout the specification) and consists of two parts: a declaration part, and a predicate part (some examples of schemas are given in Fig. 3.1). In the declaration part schema components are introduced and their types are specified. These components act as local variables that, together with the global variables defined in the system, can be used by the first-order expressions included in the predicate part. These expressions define conditions that must be satisfied by the variables introduced in the declaration part. The association between the names introduced in the declaration part and the types of values that those names denote is referred to as the signature of the schema. The property of the schema is given by the constraints included in the predicate part, together with the predicates implicit in the declaration part. The property of the schema is also called the schema invariant and together with the signature of the schema makes up, when the schema defines the abstract state of an abstract data type, the data space, which describes all possible data states of the abstract data type. Schemas also describe operations and operations

on the abstract state are called abstract operations. When schemas define such operations the following conventions are used:

- undashed names denote the values of the components in the starting state, before the operation;
- dashed names denote the values of the same components in the ending state, after the operation;
- names postfixed by a question mark denote the input values to the operation;
- names postfixed by an exclamation mark denote the output values from the operation.

Predicates that refer only to the input values and to the starting state define the precondition of the schema. The precondition must hold in order for the operation to behave as defined. The remaining predicates included in the predicate part are concerned with input, output and the ending state. They are referred to as the postcondition of the schema and describe the conditions that must be obtained after the operation has behaved as specified. Z provides a number of conventions for schema presentation aimed at reducing the amount of specification shown in a document. These conventions include schema decoration (for systematical inclusion of dashed names associated to a schema, together with the schema's invariant), schema inclusion (which has the result of bringing into the declaration part of the enclosing schema all the declarations of the enclosed schema), the delta convention, which makes use of the symbol Δ and indicates change in the schema's variables, and the xi convention, which uses the symbol Ξ and provides a shorthand notation for situations in which the schema's variables are not changed by an operation. Schemas can be combined using the following schema calculus mechanisms: disjunction, conjunction, negation, implication, quantification, piping, and composition. Both the signatures and the properties of two or more schemas can be combined according to the significance of the logical connectives indicated above. A short example of a Z specification illustrating some of the above conventions is shown in Fig. 3.1 (the example is an adaptation of a small part of the case study presented in [Evans97]).

Description of a robot arm. The robot loads bottles from a conveyor and unloads them at a filling machine. The bottles are loaded and unloaded one at a time. Given type for identifying bottles:

```
[BOTTLEID]
```

The two opposite positions of the robot arm and a type useful in describing the loading process (the robot arm is either unloaded or loaded with an identifiable bottle):

```
ArmPosition ::=   at_coveyor | at_filling
bottleLoaded ::=  loaded ⟪BOTTLEID⟫ | unloaded
```

Composite type describing the robot arm:

```
┌─── Arm ──────────────────────────────────────────
│  position: ArmPosition
│  status: bottleLoaded
└──────────────────────────────────────────────────
```

Robot arm initialisation:

```
┌─── InitArm ──────────────────────────────────────
│  Arm
├──────────────────────
│  position = at_conveyor
│  status   = unloaded
└──────────────────────────────────────────────────
```

Robot arm operations. The delta convention for Load and Unload operations indicates that the Arm state is changed:

```
┌─── Load ─────────────────────────────────────────
│  Δ Arm
│  bottle? : BOTTLEID
├──────────────────────
│  position = at_conveyor
│  status   = unloaded
│  status'  = loaded (bottle?)
│  position'= position
└──────────────────────────────────────────────────

┌─── Unload ───────────────────────────────────────
│  Δ Arm
│  bottle! : BOTTLEID
├──────────────────────
│  position = at_filling
│  status   = loaded (bottle!)
│  status'  = unloaded
│  position'= position
└──────────────────────────────────────────────────
```

Fig 3.1 Partial Z Description of a Robot Arm

In conclusion of this summary presentation of Z we note again that there is much more about this language than presented here and for all the necessary details we refer the reader once more to the sources cited at the beginning of this section.

3.2.2 Z Variants and Tools

Currently, there is an expanding community of Z users and researchers in many parts of the world, with major concentrations in the Great Britain, and significant presence in other countries, most notably Australia, Canada, France, Germany, and the U.S.A. The Internet makes available numerous Z resources, many of them accessible through the Z community's web-site [Zed01], which provides pointers to a variety of materials, including papers, reports, books, and tools. As pointed out from this web-site, as well as from Stepney et al.'s surveys [Stepney92a, Stepney92b] and Lano and Haughton's collection of case studies [Lano94a], various groups have worked on developing extensions to Z, some of the most notable being:

- Object-Z, which provides a construct for classes that encapsulates both state and operation schemas and includes a non-conformant inheritance mechanism (operations in derived classes can be redefined by strengthening the operation's precondition or by re-writing it) [Duke94]. Provisions for specifying the allowable sequences of operations are included, and a temporal logic notation is employed. The formal semantics of classes in Object-Z are based on event histories, and operators for specifying parallel operations are available. This object-oriented variant of Z uses a graphical notation for classes that extends Z's basic construct, the schema, and provides a useful visual aid for the developers;
- Z++, which follows an approach similar to the one taken by the authors of Object-Z for specifying systems in an object-oriented fashion [Lano91, Lano95]. Z++ has class definition and an inheritance mechanisms similar to that of Object-Z but, as indicated in [Stepney92b], while Object-Z is fairly abstract, the design of Z++ has been more influenced by object-oriented programming language constructs. Z++, which also

includes temporal logic support, is further covered in Subsection 3.2.3, in Chapter 6, and in Appendix A of this thesis;

- ZEST (Z Extended with Structuring), developed by British Telecommunications as an object-oriented dialect of Z intended primarily for modelling network structures and open distributed systems [Zadeh96]. ZEST extends the conventional Z language by including a class construct that consists of five distinct clauses (inheritance, interface, axiomatic clause, unnamed schema, and named schemas) and has a syntax similar to that of Object-Z. However, its semantics are significantly different and it has no special provisions for capturing temporal properties of systems;
- OOZE (Object-Oriented Z Environment), which has an algebraic formal semantics based on OBJ3, employs Z's notation and specification style, and consists of a full-fledged environment that includes a database for indexes, dependency relations, and module extensions, as well as support for animating the specifications [Alencar94]. OOZE permits the nesting of schema boxes and incorporates readability enhancements such as separate exception schemas and separate pre- and post- conditions;
- Sum, an extension of the conventional Z oriented towards refinement and translation of formal specifications to Ada [Utting95]. Sum is part of the Cogito formal development environment, which includes among its components a type-checker, a configuration management tools, and the Ergo theorem prover (the inspired header of their web-site is "Cogito, Ergo Sum" [Cogito97]). Sum specifications are translated into constructs of the functional programming language Haskell. This particular language was chosen because its type system is similar to that of Z, its strong typing allows for a fair amount of checking the specifications during editing, and its lazy evaluation increases the chance of termination in the case of some specifications;
- S, designed as a "gentler Z" (hence, its name, suggesting a Z without asperities), a machine readable notation developed at the University of British Columbia in the context of Hughes Aircraft of Canada's complex project CAATS (Canadian Automated Air Traffic System) [Joyce94]. The purpose of the language is to satisfy the requirements of applying formal specifications in large scale industrial projects by overcoming the perceived limitations of Z, specifically difficulty in explaining it to non-expert users and

impossibility of creating Z specifications using a standard text editor. The advantages of S, which has a Z-like syntax and HOL semantics, stem from its relatively simple semantics, entire printable set of characters, and the availability of a proven verification tool, the HOL system;

- Alloy, the "new kid in town," is defined by its originator as "a little language for describing structural properties" [Jackson00b, pp. 1], both sufficiently simple to allow a completely automatic semantic analysis and sufficiently powerful to express complex constraints. The new modelling language, which has its semantic basis taken from Z and a structuring mechanism similar to that of existing object-oriented notations such as UML, is aimed at supporting lightweight formal development of object-oriented systems. Alloy is less powerful than Z but compensates Z's unsuitability for object-oriented modelling with a number of features such as more flexible state declarations and distinct types of schemas. It also has a graphical notation with a textual counterpart that allows building a model entirely textually and, in comparison to UML, is more abstract (since it is based on sets and not classes) and, of course, has precise semantics.

Several well-known Z tools are the already mentioned COGITO, which proposes a methodology and toolset for the formal development of software [Bloesch94]; Logica's Formaliser environment, consisting of a ZEST Specific Formaliser, a Z Specific Formaliser, and a Generic Formaliser that supports the development of grammars for new languages [Logica01]; Z/EVES, a complex analysis system from ORA, Canada, that allows the examination of Z specifications through syntax and type checking, precondition calculation, schema expansion, domain checking, and theorem proving [Meisels97, ZEVES00]; Wizard, a type-checker for Object-Z [Uttig95, Wizard01]; Jia's ZTC and ZANS tools, a type checker and, respectively, an animator for Z specifications [Jia98a, Jia98b, ZANS98, ZTC98]; ZETA, an open environment written in Java that provides an integration framework for editing, analysing, and animating Z specifications [ZETA00], and the more recent Alloy Constraint Analyzer, developed by Daniel Jackson and his colleagues at the Massachussetts Institute of Technology [Alloy00]. Additional information on several other Z tools, specifically ProofPower Z, Zola, CADiZ, and HOL-Z, can be found in an earlier paper by

Steggles and Hulance [Steggles94]. Since a number of newer Z-centred specification approaches are currently being developed it is expected that novel Z tools, supporting these approaches, will emerge in the near future. Some of these recent approaches are described in Chapter 4, where work related to ours is surveyed.

3.2.3 A Glance at Z++

One of the OO extensions of Z, Z++ distinguishes itself through its support for capturing timing properties of systems. Since the language constitutes an integral part of the foundation on which our modelling approach is built, the purpose of this subsection is to provide only a quick look at Z++, its features being described in more detail in Appendix A, which contains a summary overview of Z++, and in Chapter 6, where rules for formalising UML constructs are presented. Here, only the general form of Z++'s most important construct, the class declaration, is given, based on [Lano94e] and [Lano95] (Fig. 3.2).

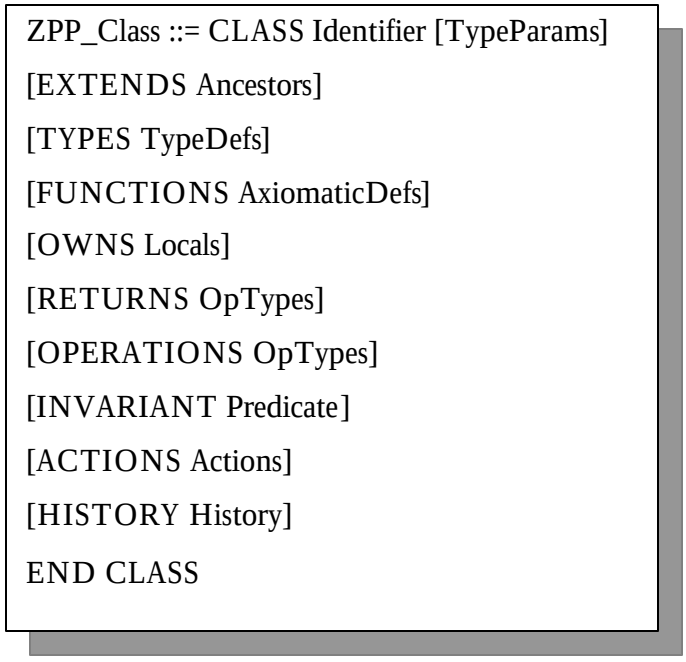


Fig. 3.2 General Form of Z++ Class Declaration

As indicated by Lano and Haughton, in the Z++ class declaration TypeParams represents a list of generic type parameters, EXTENDS specifies the superclasses of the class, TYPES introduces type identifiers used in the declaration of local variables, FUNCTIONS gives a list of axiomatic definitions of constants, OWNS specifies the local variables (attributes), RETURNS lists the operations that do not change the state of the object, OPERATIONS declares the operations that may change the attributes of the object, INVARIANT specifies a predicate that describes the internal state of the object and is guaranteed to be true between executions of the object's operations, ACTIONS defines the class operations that can be performed on objects of the class (the operations are specified using regular Z schema definitions), and HISTORY specifies the admissible sequences of execution for the objects of the class in the form of temporal-logic formulae that make use of operators such as $\Box$ (henceforth), $\bigcirc$ (next), $\diamond$ (eventually), before and until. In essence, it is here, in the HISTORY clause, where Z++'s capability of dealing with temporal aspects of the systems resides. As described in Chapter 6, this clause can also contain RTL (Real-Time Logic) formulae (Jahanian and Mok's RTL is introduced in Chapter 5).

3.3 On UML and Its Capability of Dealing with Time

The Unified Modeling Language (UML) has emerged from the combination of several widely-used, practice-validated object-oriented notations developed over the last decade by a number of prominent authors, primarily from the Booch notation [Booch94], Rumbaugh et al.'s OMT [Rumbaugh91], and Jacobson et al.'s OOSE [Jacobson94]. The fact that Booch and Rumbaugh joined forces at Rational Software Corporation in October 1994, followed by Jacobson in October 1995, created the premises for a standard notation and a common methodology for the object-oriented development community. In order to accommodate the variety of existing object-oriented analysis and design approaches and gain a large industry support, UML has borrowed concepts and notations from several other methods, including Coad and Yourdon [Coad90, Coad91], Shlaer and Mellor [Shlaer88, Shlaer91], Fusion [Coleman94], and Statecharts [Harel87]. In November 1997 version 1.1 of UML has been adopted as object modelling standard notation by the Object Management Group (OMG)

and as of March 2001 minor revisions have been included in versions up to 1.4, the work on the last one being currently in progress. A major revision, version 2.0, is tentatively scheduled for standardisation in 2001 [Kobryn99]. The complete version 1.3 of the language's specification, published in March 2000, is available from the OMG web-site [UML00]. In our opinion, the incorporation of concepts from numerous sources, although justified by the goal of ending "the methods war" by providing a comprehensive, widely accepted modelling language for object-oriented development, requires some "fine-tuning" in practical terms – the generality of the notation and its higher level of diffuseness [Green96] typically necessitating decisions on what to be used, what customisations to be made (what "stereotypes" to be employed), and what to be left out from UML in a particular methodology and/or application. As detailed later in the thesis, for practical reasons we focus in our approach on only a subset of the UML notation and thus minimise the notational alternatives that UML would provide if considered in totality. Here, in Subsection 3.3.1 an overview of UML's general capabilities is presented, while in Subsection 3.3.2 the features of the language that support the modelling of RTS are examined. Although the most authoritative reference for UML is [UML00] for a shorter and less formal description of UML we have relied on [Booch98] as primary reference for the two Subsections that follow, with additional sources consulted [Quatrani98], [Si-Alhir98], [Douglass98] and [Douglass99]. The examination of UML is completed in Subsection 3.3.3 with a look at some of the current research and industrial developments involving UML.

3.3.1 A Bird's Eye View on UML

As indicated by its authors, UML is a "graphical language for visualizing, specifying, constructing, and documenting the artifacts of software-intensive systems" [Booch98, pp. xv]. The notation is primarily designed for supporting the analysis and design phases of the software development process, but it is useful also for the deployment and maintenance of software. UML is supported by the industrial-strength software development environment Rational Rose (latest version 2001), commercially available form Rational Software Corporation [RationalRose01], and has been integrated in a number of other CASE tools,

some of which are mentioned in Section 3.3.3 of the thesis. Recently, a real-time version has also been made available [RationalRoseRT01]. The environment is powerful and supports not only the modelling of large software systems, but also processes such as reengineering and automated code generation. Fig. 3.3 contains an image of the Rational Rose environment (version 2001), showing its main components in the case of a logical view specification: the menu bar, the toolbar, the browser, the palette for class modelling, and the pane with drawing windows (the example shown is of an Elevator system, based loosely on [Dong97b], where an OMT description is given).

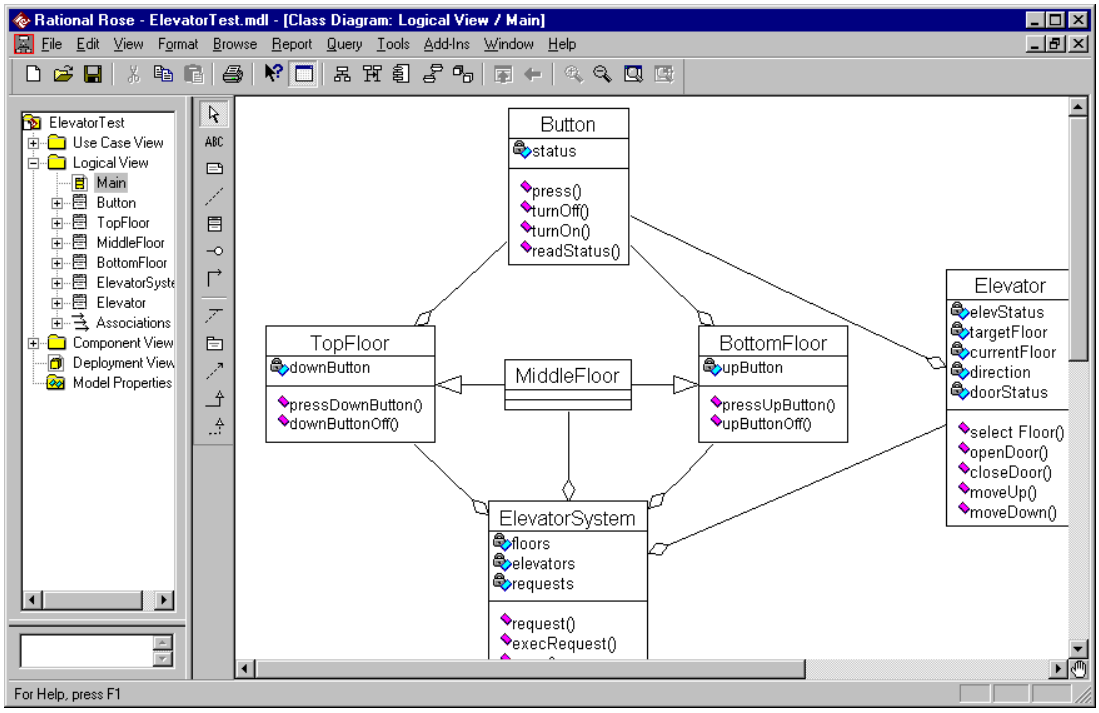
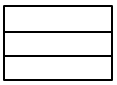
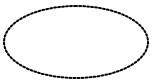
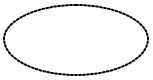
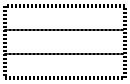
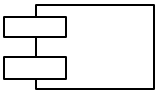
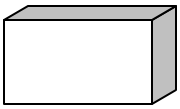
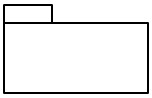


Fig. 3.3 Snapshot of Rational Software Corporation's Rational Rose

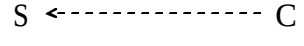
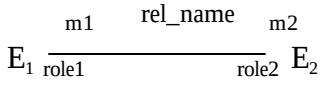
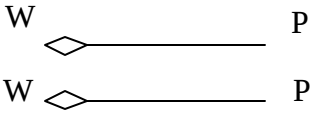
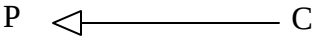
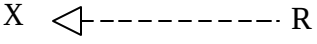
Booch et al. indicate that there are three types of building blocks in the language, namely things (first-class abstractions in UML's conceptual model, alternatively referred to by us as model elements), relationships that interconnect abstractions, and diagrams that present collections of things together with the relationships that exist among these things [Booch98]. A summary of UML basic things (there are also variations for them), further classified as structural, behavioural, grouping, and annotational is presented in Table 3.II, while the

Table 3.II UML Things (Model Elements)

UML Thing (Model Element)	Kind	Description	Symbol
class	structural	represents a set of objects with the same attributes, operations, relationships and semantics	
interface	structural	defines a collection of operations (a set of services) that represent the externally visible behaviour of an entity	
collaboration	structural & behavioural	describes an interaction and includes several cooperating elements with their specific roles	
use case	structural & behavioural	represents a set of sequences of actions that yield an observable result to some actor(s)	
active class	structural	a class whose instances owns threads or processes and thus can initiate control activities	
component	structural	a physical and replaceable part of a system that packages implementation	
node	structural	a run-time computational resource generally having memory and often processing capability as well	
interaction	behavioural	defines a behaviour and consists of a set of messages exchanged among collaborating objects within a particular context	(message, the basic element)
state machine	behavioural	specifies the sequences of states an object or an interaction can go through during its lifetime	details in Subsection 3.3.2
package	grouping	general purpose mechanism for organising model elements in groups	
note	annotational	explanatory item (comments, constraints, etc.)	

fundamental relationships of the language, namely dependency, association, aggregation (with its particular form composition), generalisation, and realisation are succinctly described in Table 3.III (diagrams, the third kind of building blocks, are discussed in more detail later in this Subsection). The authors indicate that besides building blocks the conceptual model of UML consists also of a number of rules for putting together these blocks (such as rules for names, scope, visibility, and integrity) and common mechanisms that are consistently applied within the language (specifications, adornments, common divisions, and extensibility mechanisms).

Table 3.III UML Relationships

UML Relationship	Description	Symbol
dependency	a "using" relationship from a client C to a supplier S ("C uses S" or "C depends on S"); changes in the specification of S may affect the using class C	
association	structural relationship between two model elements that declares a connection between their instances (on the association symbol the name of the relationship, the roles of the two model elements and the multiplicity of their instances can be specified)	
aggregation / composition	aggregation is a particular case of association that specifies a "has-a" relationship between the whole W and its part P; composition is an aggregation with strong ownership and the lifetime of P subsumed to the lifetime of W	
generalisation	relationship between a more general model element (parent P) and a more specific kind of that element (child C); "is-a" relationship	
realisation	a relationship in which the contract stipulated in the model element X is carried out in the model element R	

Since extensibility is one of the most important characteristics of the language the mechanisms that ensure UML remains open-ended, namely stereotypes, tagged values, and constraints are presented in Table 3.IV.

Table 3.IV UML Extension Mechanisms

UML Extension Mechanism	Description	Symbol
stereotype	allows the creation of new building blocks from the existing ones; extends the language by allowing the addition of new, problem-specific model elements	<<stereotype_name>>
tagged value	attaches new information to an existing model element; extends the language by allowing the addition of new properties	{property_description}
constraint	extends the semantics of UML building blocks by adding new rules or modifying the existing ones	{constraint_description}

In essence, the UML allows the modelling of a system through a number of diagrams that capture either the static or the dynamic aspects of the system and can be organised in a number of views, each view being "a projection into the organization and structure of the system, focused on the particular aspect of that system" [Booch98, pp.31]. Static aspects are captured in use case diagrams, which contain actors (an actor being someone or something that externally interacts with the system), use cases, and their relationships; class diagrams, which show classes, interfaces, collaborations, and their relationships; object diagrams, which provide snapshots of instances of classes and their relationships; component diagrams, illustrating the organisation of and the dependencies among software implementation components (a component typically maps to one or more classes, interfaces and collaborations); and deployment diagrams, which show the configuration of nodes and the run-time allocation of components to nodes. Behavioural aspects are described in sequence

diagrams, which depict a succession of messages exchanged among objects and emphasise the time ordering of messages; collaboration diagrams, which are similar to sequence diagrams, but stress the organisation and the roles of objects that send and receive messages; statechart diagrams, which essentially contain states and transitions that describe the event-driven life cycle of objects; and activity diagrams, whose role is to indicate how the control flows from activity to activity within a system. The sequence diagrams and collaboration diagrams, which convey the same information and can be easily transformed one into the other, are referred to commonly as interaction diagrams.

Although with the exception of the use case view the terminology related to views varies from author to author (for instance, [Booch98] speaks of use case, design, process, deployment, and implementation views, [Si-Alhir98] uses the terms use case, structural, behavioural, environment, and implementation views, and [Quatrani98], whose presentation is based on the Rational Rose tool, discusses the use case, logical, process, component, and deployment views) and there are also different nuances involved in the meanings these authors associate to views, it is generally acknowledged that a "4+1" architectural view model allows a comprehensive description of the system. Useful to note, the view concept is not part of UML specification, but the language supports this generally accepted "4+1" view of architecture that facilitates the organisation of knowledge and allows the modelling of the system from various interconnected perspectives. Interestingly, the Rational Rose environment, including its latest edition [RationalRose01], has predefined sections for only four views (use case, logical, component, and deployment) and relies on component diagrams to render the process view employed by Booch et al. and by Quatrani.

In Fig. 3.4 we have set for a "4+1" view solution that associates meanings and diagrams to views in Si-Alhir's way, which emphasises a more traditional demarcation between structure and behaviour [Si-Alhir98]. However, in order to use as much as possible the more prevalent Rational terminology, the names of views have been borrowed from all the references mentioned above.

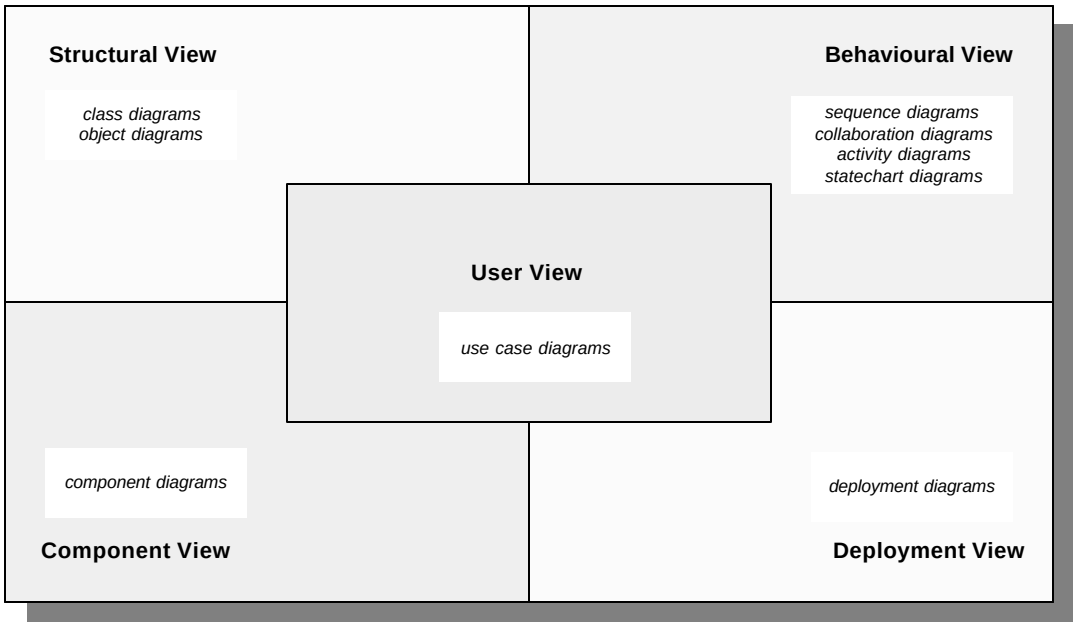


Fig. 3.4 The 4+1 Architectural Views and UML Diagrams That Express Them

In this "4+1 views" architectural model of the system the use case view describes the behaviour of the system as seen by its users. It employs use case diagrams to capture the functionality provided by the system to its external interactors and constitutes the "central perspective" that binds together the different angles under which the system can be scrutinised. The structural view relies on class and object diagrams to describe the system's structural elements and their interconnections. The behavioural view is concerned with the dynamic aspects of the system and uses all four types of behavioural diagrams to capture them. The component view (or implementation view) makes use of component diagrams to capture both the behavioural and static aspects of a system's realisation and shows all the components and files that are needed to assemble the physical system. Finally, the deployment view presents in its associated deployment diagram the nodes that form the hardware topology on which the system executes. Of course, for managing the complexity of a problem each view can be considered separately but the complete "picture" of the system is obtained by interconnecting them. In fact, as pointed out by Booch et al., the views interact inherently, for instance the nodes of the deployment view contain components (defined in the component view) that realise classes (specified in the structural view) and behaviours

(described in the behavioural view), all derived from use cases (captured in the use case view). In our dual-notation specification approach focused on time-constrained systems we are primarily interested in the use-case, structural, and behavioural views, and leave aside details pertaining to the component and deployment view (details are given in Chapter 7).

To keep the description shorter and in agreement with the selection of the UML subset used in our approach only five out of the nine possible kinds of UML diagrams are illustrated below. A use-case diagram (Fig 3.6), a class diagram (Fig 3.7), an object diagram (Fig. 3.8), and a sequence diagram (Fig. 3.9) pertaining to a common "theme", an Automatic Camshaft Testing System (ACTS, Fig. 3.5) inspired from our previous work [Dascalu89, Ionescu93], are presented in this Subsection, while a statechart diagram is included in Subsection 3.3.2.

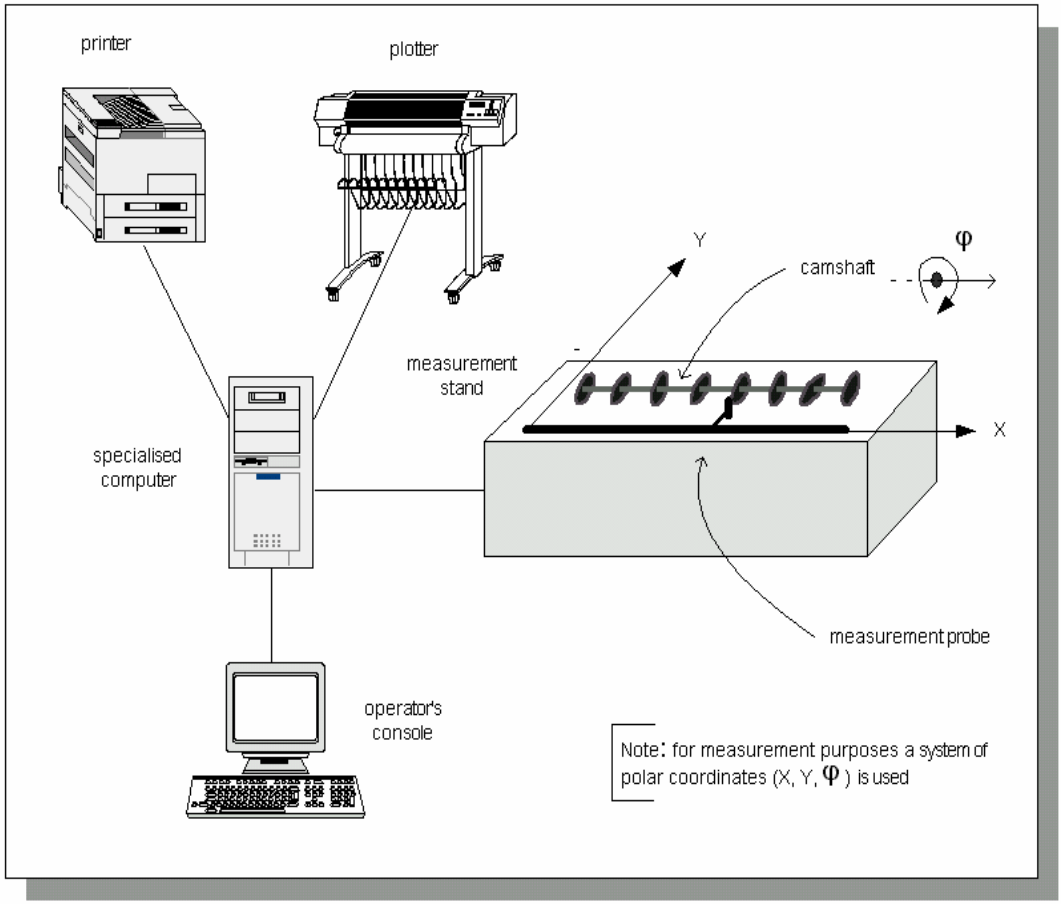


Fig. 3.5 Overview of the Automatic Camshaft Testing System (ACTS)

These diagrams are only intended to illustrate several of the basic UML concepts and not to capture the entire complexity of the automatic camshaft testing problem, so they should be viewed only as small excerpts from a larger specification.

However, in order to provide the necessary context, a short description of the system presented in Fig. 3.5 is necessary. Functionally, what is important to know is that in the ACTS actual profile data (Y values) for each of the N cams (typically N = 8 or N = 12) of an automobile engine camshaft are automatically collected and then a number of associated diagrams ("height," speed, and acceleration) can be drawn and a variety of comparisons can be performed against theoretical values. To achieve this, for each cam the measurement probe is first moved along the X axis to a position that corresponds to the middle of the cam's lateral surface, pushed subsequently against the cam (Y movement) and then the camshaft is rotated a little more than 360^{O} (movement on φ axis) while profile values Y are collected from the cam. The ACTS operator has a number of options, including selective testing of cams (e.g., only cams 1, 4, 5, and 8 are inspected), variable "angular step" for measurement (e.g., 0.5^{O} or 1^{O}), and various formats for test certificates (out of tolerance values only or full diagrams, printed or plotted profiles, etc.).

In the four related UML diagrams the view on ACTS is "sequential" in the sense that the three axis controllers (X, Y, φ) work only one at time and the sensors are polled by the controllers. As shown later in Subsection 3.3.2 a concurrent approach can also be considered with sensors sending signals to the axis controllers and the camshaft being rotated towards a predefined position while the probe is "cruising" on X and Y directions (in principle, simultaneous movements on X and Y axes are not excluded either). In the UML diagrams pertaining to the ACTS specification italics have been used to highlight key elements, many of them introduced in Tables 3.II to Table 3.IV. Besides the references cited at the beginning of this section for more examples of UML diagrams we suggest [TogetherSoft00a], which contains one of the most concise and clear UML tutorials we found during our survey of the notation.

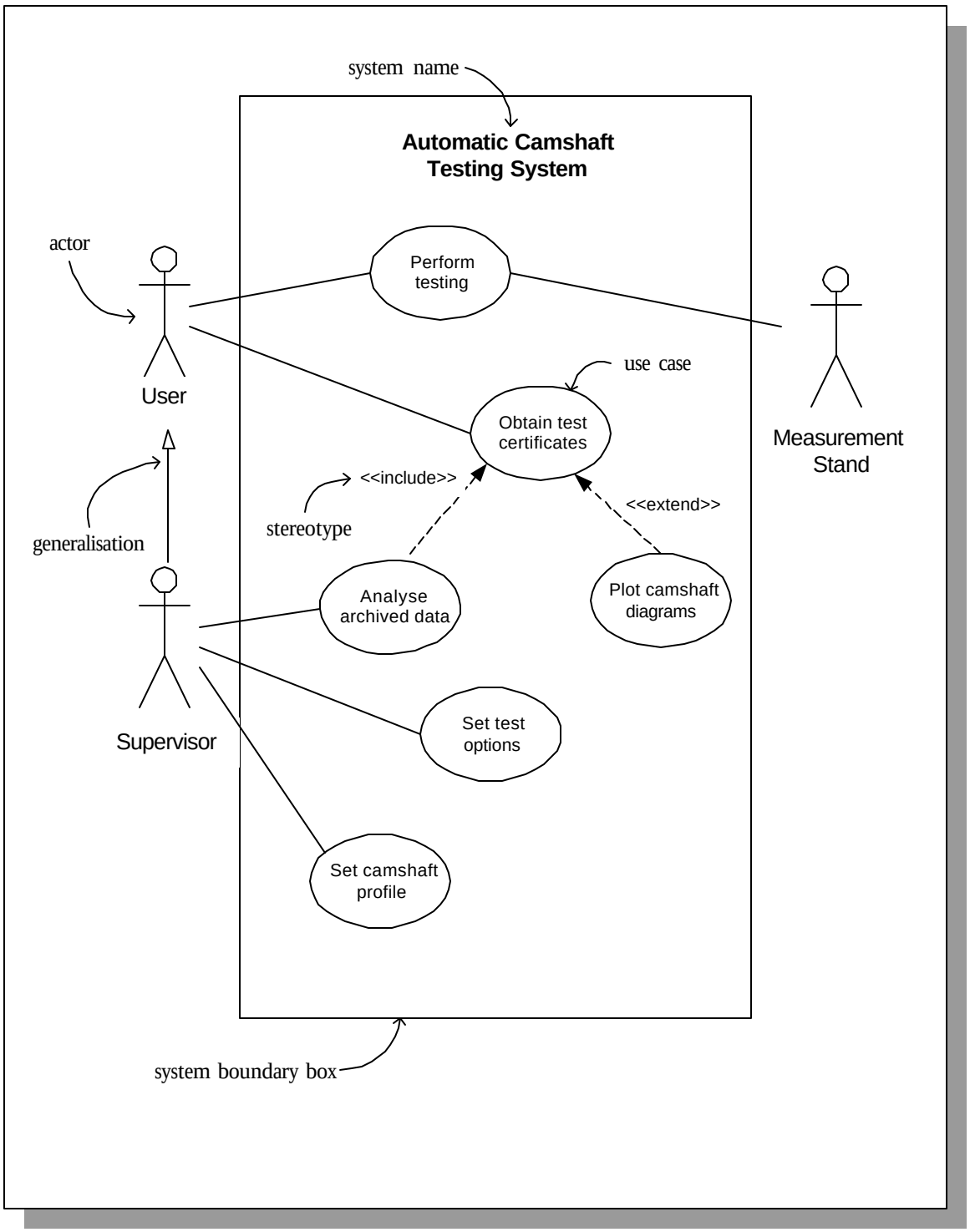


Fig. 3.6 Example of Use Case Diagram: Excerpt from the ACTS Specification

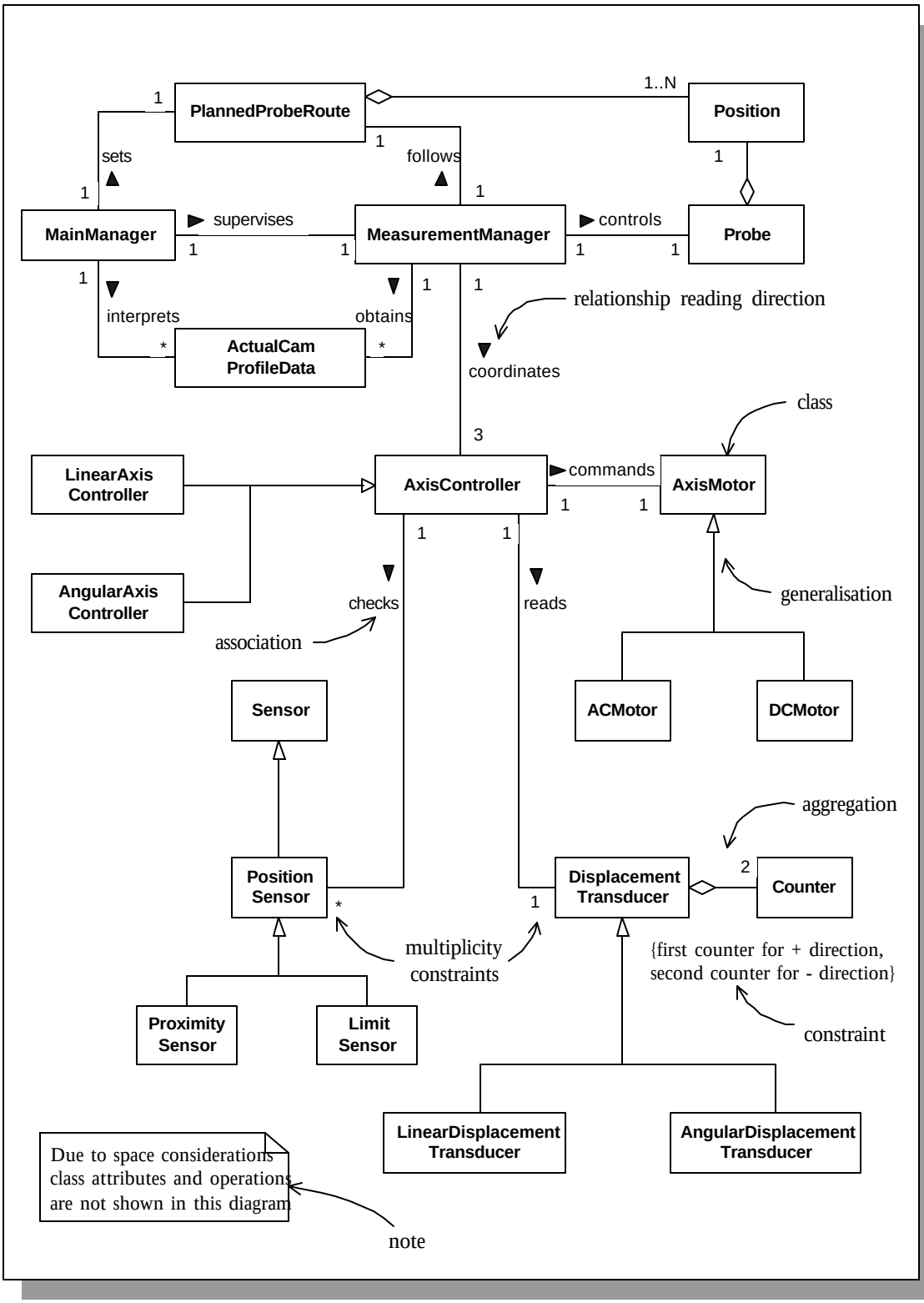


Fig. 3.7 Example of Class Diagram: Excerpt from the ACTS Specification

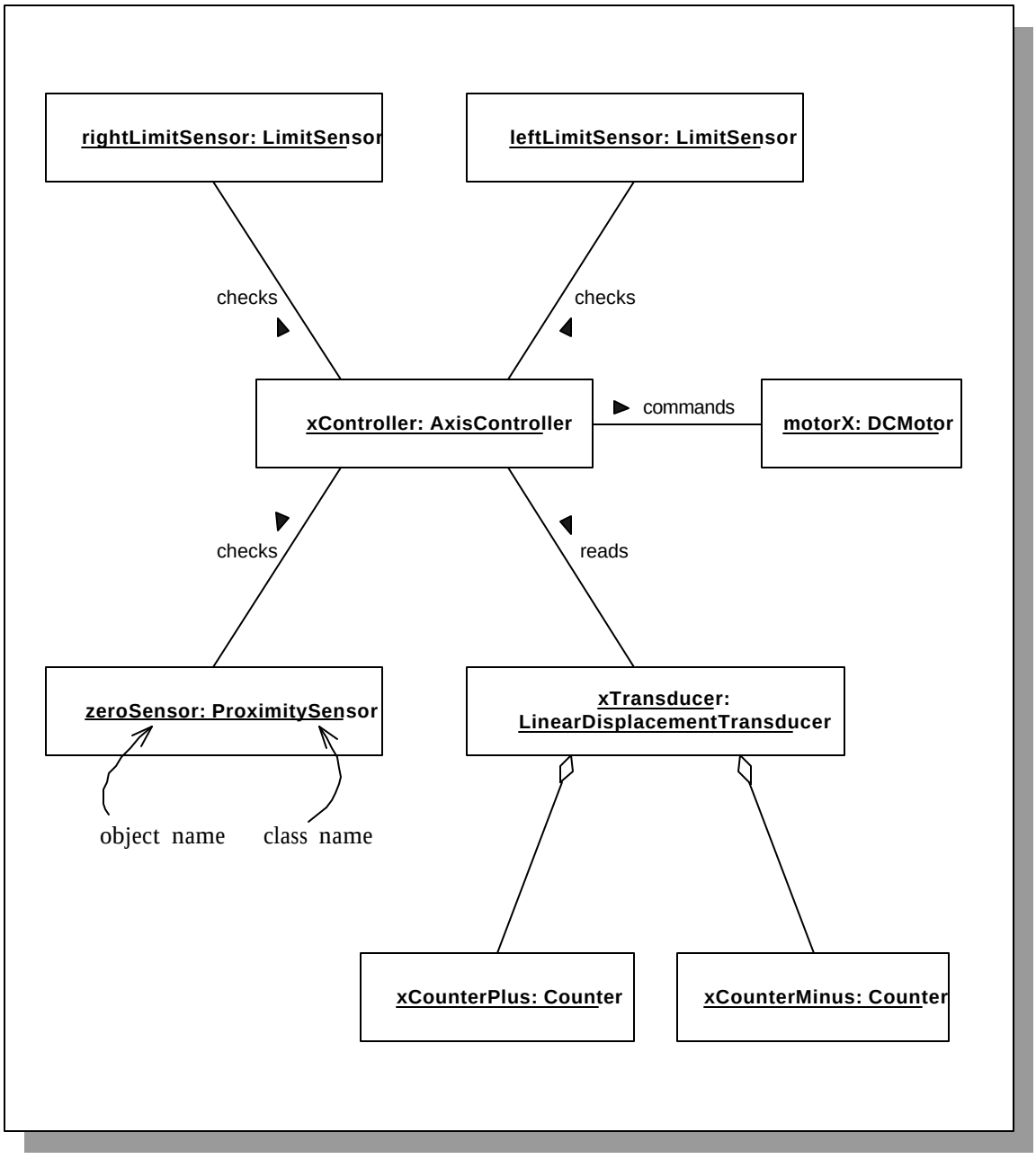


Fig. 3.8 Example of Object Diagram: Excerpt from the ACTS Specification

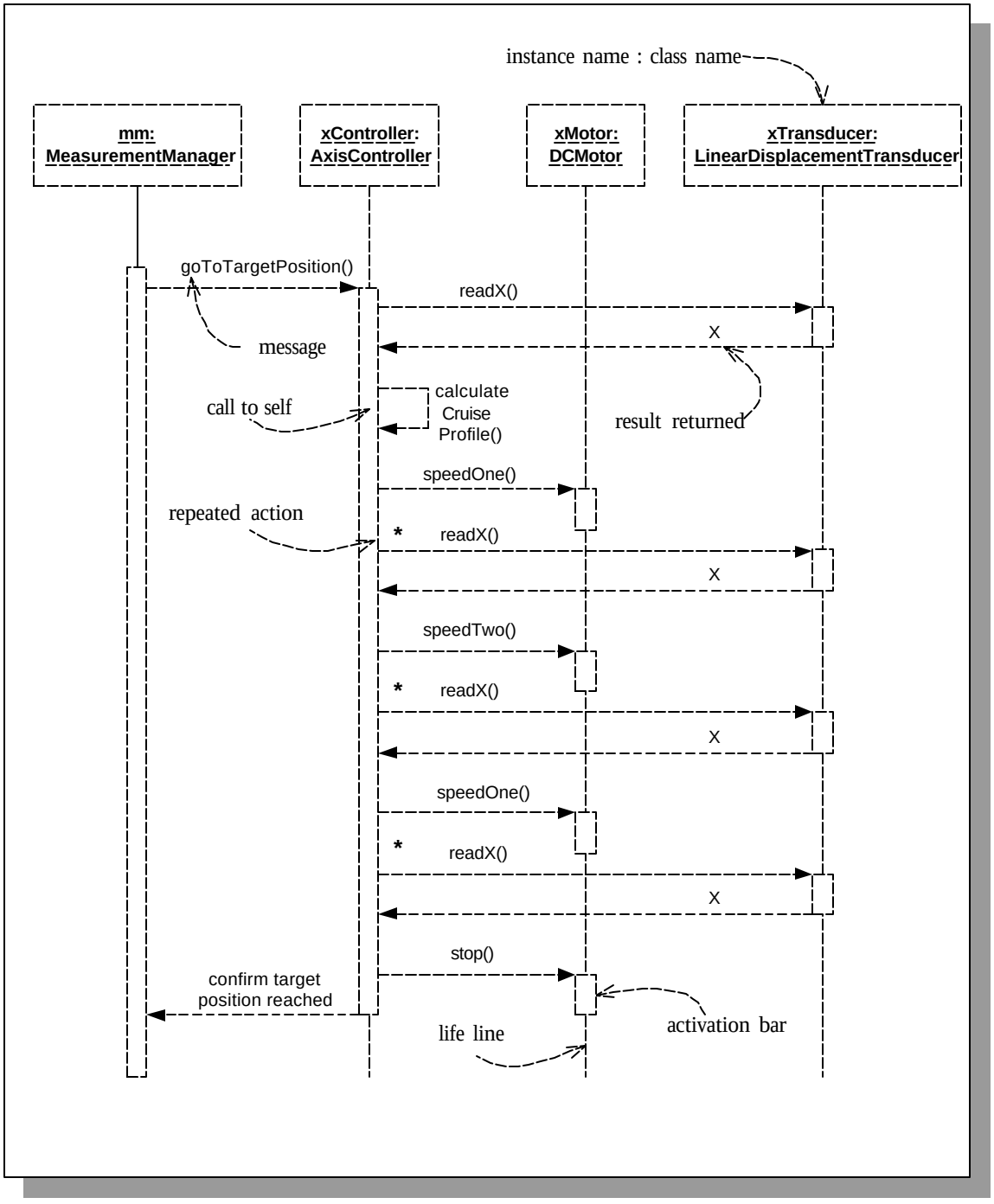


Fig. 3.9 Example of Sequence Diagram: Excerpt from the ACTS Specification

3.3.2 UML Support for Modelling Real-Time Systems

As described in [Booch98], UML provides support for modelling real-time systems via a number of special constructs and mechanisms.

Firstly, there are events and signals. According to Booch, Rumbaugh, and Jacobson, events are "things that happen" [Booch98, pp. 37], each event specifying some occurrence that has identifiable location in time and space. A signal is a particular type of event that represents an asynchronous stimulus communicated between instances of classes. Other types of events are call event, passage of time event, and change event (Table. 3.V). Signals are similar to classes; they can have instances, attributes, and operations, and can be organised in hierarchies. Represented as classes stereotyped with the <<signal>> mark, they are in relation

Type	Description	Symbol
signal event	an occurrence of interest packed as an object and dispatched asynchronously from an object to another	<<signal>>
call event	method invocation from an object to another; synchronous notification from the caller object to the object whose operation is invoked	no special symbol, graphical notations for regular operation invocation apply
passage of time event	event that specifies that a given duration has elapsed	after (duration)
change event	event that indicates the satisfaction of some condition (typically based on the changing of some attribute's value);	when (condition)
	in particular, can be used as a time event, marking the arrival of an absolute moment of time	when (time_value)

Table 3.V Types of Events in UML

"send" with the class operation that dispatches them. As a notational convention, receivers of signals may include in their class symbol, below attributes and operations, an additional compartment showing the list of accepted signals. An important kind of signal is exception, predictably stereotyped with the mark <<exception>>. All types of event can be either internal or external to a system, synchronous or asynchronous (depending on whether or not their sender waits for the receiver's response), and can be multicasted (send to a specified set of receivers) or broadcasted (dispatched to all objects in the system that might be listening). Events other than signals are typically involved in state diagrams only as transition triggers but they can also be modelled as classes. Considering again the ACTS problem used in Subsection 3.3.1 but opting this time for a concurrent solution (several threads of control involved), an example of signal communicated between classes is given in Fig. 3.10.

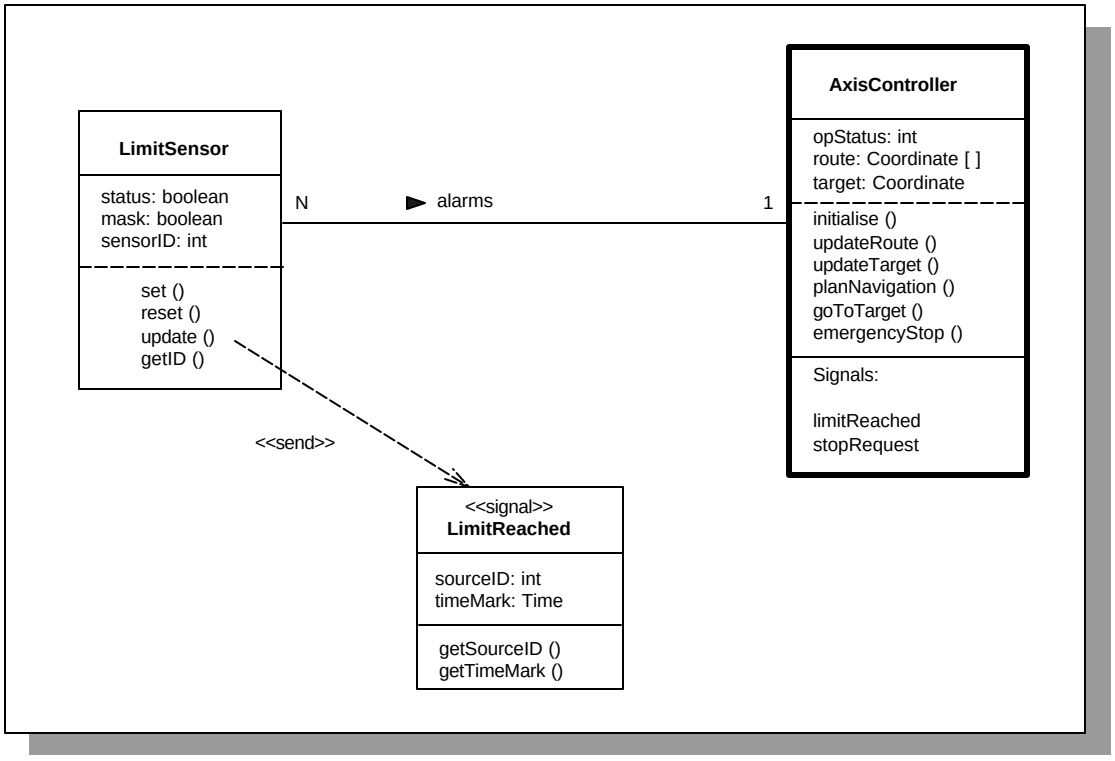


Fig. 3.10 Example of Signal: Excerpt from the ACTS Specification

In this figure, the thick-line class "AxisController" is an active class, as explained below, while "LimitSensor" is a regular class whose objects are within the flow of control rooted in some active class (not included in the diagram).

Secondly, there is UML support for describing processes and threads. While a process represents a heavyweight flow of control that is known to the operating system, has its own address space, and competes with peers on the same node, a thread is a lightweight flow of control that runs in the address space of an enclosing process and competes with peers within this process. In a concurrent system an active class is used to show an independent flow of control, each active object representing a thread or process that can initiate control activity. An active class can be stereotyped either as <<process>> or <<thread>>, is graphically represented with a thick line, and typically contains the extra compartment "signals" in its representation. Communication between thread objects can be achieved using either signals (asynchronous) or call events (synchronous) while process objects usually communicate via message passing (asynchronously) or remote procedure calls (synchronously). As shown in Fig. 3.11, UML includes graphical symbols for representing both asynchronous communication, which has mailbox semantics, and synchronous communication, characterised by rendezvous semantics. Two special cases of rendezvous are also included in Fig. 3.11: timeout rendezvous, meaning that the sender will wait for the receiver to respond to the message up to some preset period of time before aborting the transmission and continuing with its processing, and balking rendezvous, describing the situation in which the sender aborts the communication and continues its processing if the receiver is not immediately ready to accept the message [Booch94]. In UML it is also possible to indicate a critical region by attaching the {concurrent} constraint to operations.

Thirdly, although not restricted to the specification of RTS, finite-state machines and statechart diagrams are of great help for modelling such systems, especially if the behaviour of these systems is event-driven. A finite-state machine serves for representing the lifecycle of objects and contains a set of states and all possible transitions among these states. A state can be viewed as a situation in the life of an object during which it performs some activity, satisfies a specific condition, or simply waits for an event. A stable state is a state in which the object may exist for an identifiable period of time. In its most complete description a state has a name, executes some entry/exit actions, contains internal transitions (which are

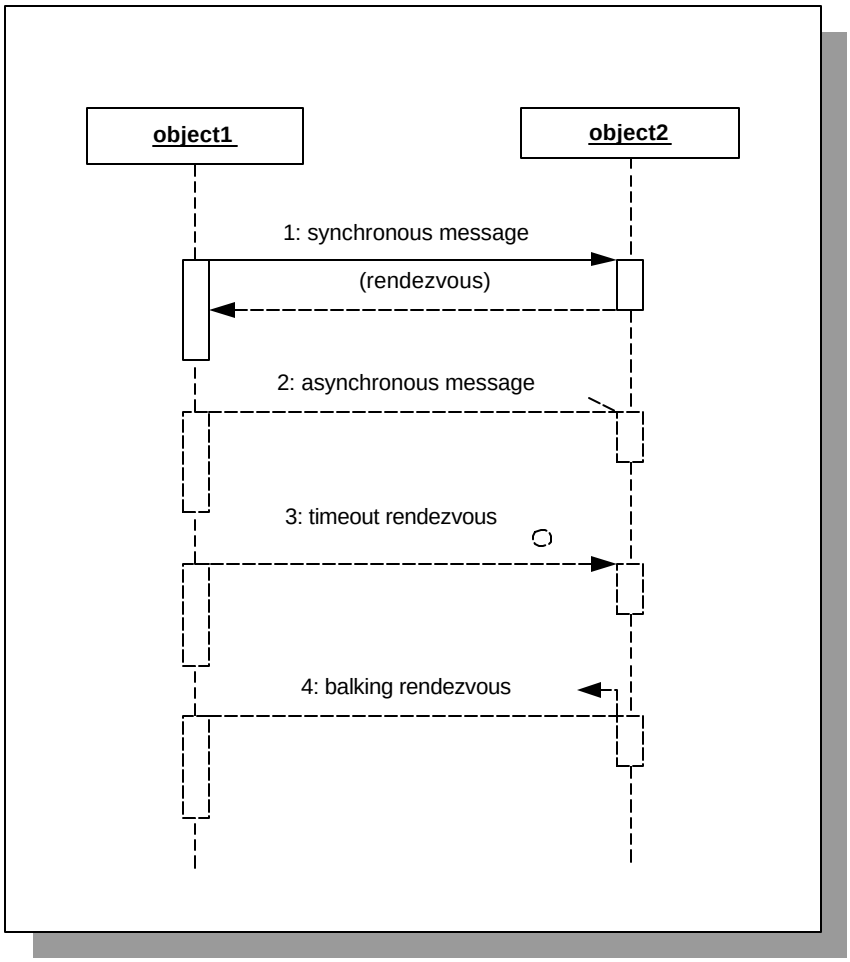


Fig. 3.11 UML Symbols for Synchronous and Asynchronous Communication

handled without causing a state change), has a number of substates (either sequential or concurrent), executes some activities, and declares deferred events (events queued and handled by the object in a different state). A transition, in its most complete form, has five parts: a source state, the state from which the transition originates, an event trigger, the event that makes the transition eligible to fire, a guard condition, a Boolean expression that must be true for the transition to take place, an action, which is an atomic computation and may act directly on the object described by the state machine and indirectly on other objects, and a target state, the state that becomes active as the result of firing the transition. Modelling reactive objects means specifying stable states, events, actions that occur on state changes, conditions for these actions to take place, as well as initial and final states. It is useful to note

that in a Mealy machine actions are attached to transitions, while in a Moore machine actions are attached to states, both approaches being handled properly by the state and transition concepts described above. A statechart diagram (see also Section 3.3.1) shows a state machine, emphasises the flow of control from state to state, and can be used in the context of the whole system, of a subsystem, or of a class. In Figure 3.12 an example of finite state machine is presented, illustrating some of the most commonly used state-transition

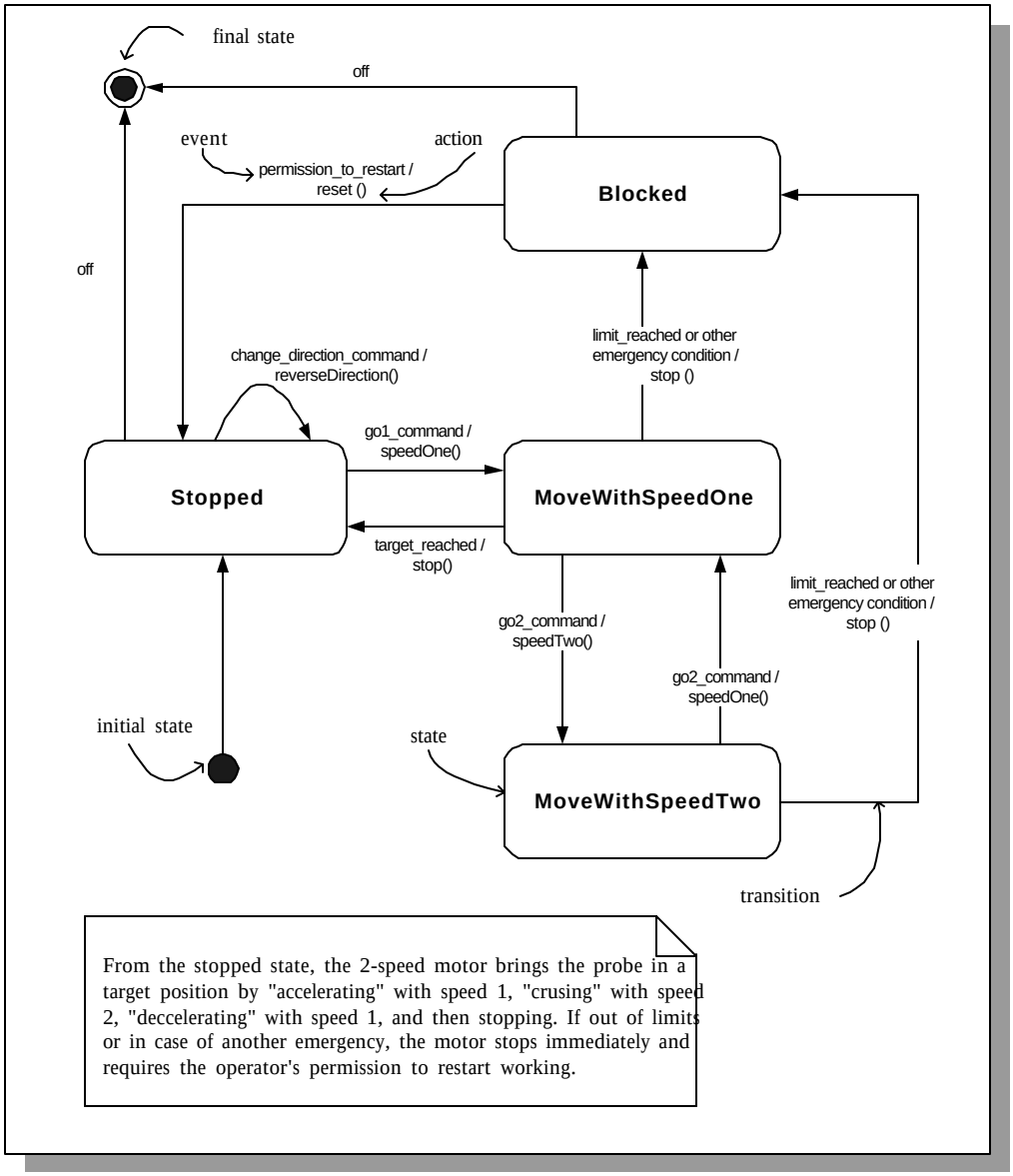


Fig. 3.12 Example of Statechart Diagram: A 2-Speed DC Motor for ACTS' Axis X

elements. The example is again related to the ACTS problem and describes the operation of a 2-speed DC motor that moves the measurement probe on axis X under the supervision of the X axis controller.

Fourthly, modelling RTS can make use of some UML markings and expressions that are dedicated to capturing time and space properties of systems. These are timing marks, time expressions, time constraints, and location tagged values (Table 3.VI). Additionally, semantics tagged values can be attached to operations and a time expression may be used to specify the operation's time complexity: minimum, maximum, and/or average execution time.

Table 3.VI UML Markings and Expressions for Time and Location

Type	Description	Example
timing mark	indicates the time of an occurrence and is denoted by a lowercase letter attached to an event (message)	a: updateRoute() b: goToTarget()
time expression	an expression that evaluates to a time value; it may include the predefined functions for messages startTime, stopTime, and executionTime	b.startTime < a.stopTime + 0.5
time constraint	a statement about relative or absolute value(s) of time; as all other UML constraints it is represented by a string in brackets	{a.executionTime < 2 ms} {every 12 hours}
location tagged value	specification of a component's placement in a node; typically written below the name of the component, it is primarily used in deployment diagrams	{location = RemoteController}

Finally, the modelling of any particular class of systems, including RTS, is supported by UML's extension mechanisms described in Subsection 3.3.1. In particular, stereotypes are of

great value for satisfying the modelling requirements of particular types of application and for supporting specialised methodologies. A prominent example in this respect is Selic's definition of new stereotypes, including <<capsule>>, <<protocol>>, and <<port>> [Selic99b]), as means of expressing the concepts put forth by the ROOM approach.

However, UML alone is not sufficient for rigorous development of RTS and supplementary formalisation is necessary [Evans99, Howerton99, McUmber99, Alagar00]. Solutions in this direction have already been proposed, as indicated in the next Subsection, where several of UML's avenues of development are overviewed.

3.3.3 The UML Promise

As a new and promising language for specifying object-oriented systems, UML has been recently approached from various perspectives by practitioners and researchers. Without claiming that our survey of the latest developments involving UML has exhaustively covered the related literature, we have identified several major directions of exploiting the benefits brought by the notation. These directions, some of which intersect in many of the surveyed reports, can be described as follows:

- Application to the development of industrial systems, from medium-sized to complex. The range of reported applications extends from the construction of an MPEG-4 decoder [Barrios99] to the development of a next-generation AWACS (Airborne Warning and Control System) [Bell99], and encompasses projects such as a wine bottling production line [Becker00], a car radio assembly line [Fernandes00], a gas turbine engine simulation system [Xie99], and a GSM (Global System for Mobile Communication) [Jigorea00];
- Extensions covering special application domains. Equipped with the new device, UML, teams from both industry and academia have recently started to employ its modelling capabilities in areas traditionally more difficult to tackle, such as heterogeneous systems and distributed systems. In many cases, extensions to the notation have been proposed,

from simple additions consisting of a few of stereotypes to more intricate modifications involving both syntax and semantics. For instance, in the [Barrios99] proposal aimed at heterogeneous system design the additions consist of a number of tags attached to classes to indicate the target implementation language and of a number of stereotypes for hardware entities, in the [Fernandes00] tackling of embedded systems the extensions come in the form of tagged values called references associated to use cases for better correspondence with object diagrams, in the [Conallen99a] paper comprehensive stereotype-based extensions to UML that allow the modelling of Web applications are presented, while in the [Price99] treatment of spatiotemporal data modelling a set of supplementary definitions to UML is proposed, resulting in what the authors have termed STUML (Spatio-Temporal UML);

- Support for new development methodologies. In order to accommodate particular aspects of the modelling process within specific application domains or to promote novel software construction approaches not only extensions to UML but also new development methods and methodologies have been devised. Examples of such UML-based methods and methodologies, some of them accompanied by specific extensions of the notation, include D'Souza and Wills' comprehensive Catalysis approach for object and component-based development [D'Souza98], Conallen and Bebick's proposal for modelling Web applications [Conallen99b], Muller's utilisation of UML for database design [Muller98], Cheeseman and Daniels' process for server-side component-based development [Cheesman00], Fernandes et al.'s alternative for modelling embedded systems [Fernandes00], Lu et al.'s UDRE (User-driven Domain-specific Requirements Engineering), focused on the user's involvement in the development process and on the relevance of requirements throughout it [Lu99], and Oldevik et al.'s methodology for developing distributed systems, a methodology that employs RM-ODP (the ISO standard Reference Model for Open Distributed Processing) as a conceptual and architectural framework and UML as a flexible modelling notation "that can be used for virtually anything" [Oldevik98, pp.13];
- Combinations with other notations. There are also situations where simply extending UML is not sufficient for capturing all the aspects of a particular class of application (or

for capturing these aspects in the desired way), yet employing the modelling power of UML still brings significant advantages. In such cases, UML can act as companion to some other language or languages, and play various roles. Examples of partners for UML include IDL (Interface Description Language) in the object-oriented development of distributed systems [Watkins98], Java in the construction of simulation systems [Kortright97], E-DFDs (extended data flow diagrams) in modelling distributed real-time systems [Becker00], and a variety of rigorous languages such as cTLA (a variant of Lamport's Temporal Logic of Actions) [Graw00], $GSBL^{OO}$ (an algebraic specification language based on inheritance [Favre99]), and VHDL [McUmber99] in specification approaches with enhanced formalism. Since combining notations is also the avenue of research we follow in the present thesis, more about combinations of notations, in particular about those involving UML and variants of Z, is presented in Chapter 4, Related Work, of this thesis.

- Strengthening of UML's underpinning formalism and, generally speaking, formalisation of UML. Representatively, the need for concerted efforts in this direction has lead to the formation of a dedicated group, pUML ("precise UML") whose primary objective is "to clarify and make precise the semantics of UML" [pUML01a] and whose membership include well known scientists such as Andy Evans, Robert France, David Harel, Kevin Lano, Stuart Kent, and Bernand Rumpe. The two working groups of pUML focus on building a rigorous meta-model semantics for UML and, respectively, on defining precise semantics for OCL (Object Constraint Language, the standard constraint language of UML [Warmer98]). Approaches taken by pUML members include the proposal of a meta-model semantics for structural constraints in UML [Kent99], the incorporation of rigorous reasoning techniques within UML's component abstractions and representations [Evans98], the development of an axiomatic semantics model for a large part of the notation [Lano98], and the formalisation of key UML constructs [Shroff97] (for a more complete image of the group's research we refer the reader to pUML's list of publications available at [pUML01b]). Sustained work in the same direction of formalising UML has also been ongoing for some time in other centres of research (e.g.,

Alagar and Muthiayen's proposal for Real-Time UML [Alagar00] and Alemán and Alvarez's work on foundations of developing UML model verification tools [Alemán00]);

- Eclectic uses, for instance as supporting notation in an approach aimed at helping students understand the value of precise specifications [Stoecklin98] or in the generation of the OOHyTime meta-model, whose purpose is to facilitate both the understanding and the utilisation of HyTime (Hypermedia/Time-Based Structured Language), a standard for interchanging hypermedia documents [Scott99];
- Tool support. The growing popularity of the modelling language has also been fuelled by the development of a variety of CASE tools that incorporate support for the notation. Besides Rational Software Corporation, with its vanguard tools Rational Rose [RationalRose01] and Rational Rose Real-Time [RationalRoseRT01], a number of other major commercial vendors have already provided the software development community with CASE tools that incorporate support for the UML notation. Among these, major vendors are TogetherSoft Corporation, developers of Together Control Center, Together Enterprise, and Together Solo [TogetherSoft00b], I-Logix, Inc., with its Rhapsody environments for modelling real-time embedded systems [Rhapsody01], Popkin Software, creators of System Architect 2001 [SystemArchitect01], Computer Associates International, Inc., providers of the application lifecycle management tool Paradigm Plus [ParadigmPlus01], and Microsoft Corporation, who have recently acquired Visio Corporation, the developers of Visio, an intelligent diagrammatic editor [Visio00]. However, it is worth noting that providing comprehensive tool support for UML is not easy to achieve. For instance, according to [Bell99] the coverage of the notation need be improved in the case of the two (unnamed) design tools that were used in the AWACS project and, as the author indicates, provided incomplete support for UML.

A number of observations can been drawn from our survey of the recent UML literature. First of all, that the wealth of projects and applications domains making use of the new modelling language as well as the variety of directions from which the language has been approached provide a solid justification for the term general notation associated with UML.

Also, even before the apparition of a dedicated tool such as Rational Rose Real-Time we could note as a fact the significant number of approaches aiming at dealing with complex systems such as real-time embedded systems and distributed systems. This reflects a clear need for notational support for modelling such systems, and it appears that in the near future UML will have a major role to play in the distributed and real-time areas.

The application of UML to specific domains has lead to a number of extensions or adaptations of the notation and in not a few cases a new, custom-made methodology has also been proposed. In fact, extensions to the notation appear to represent the norm rather than the exception in the employment of UML. However, this should not come as a surprise, since UML was built from the beginning with provisions for extension, but the spawning of notations can lead to an "inflated" UML, hard to manipulate efficiently. Also, the number of new methodologies proposed is rather large, and unless the proposal of a new methodology is fully justified, the present methodological gusto may actually backfire, and undermine the very idea of unification behind UML. In truth, however, we should note that many of the new methodologies reported in the literature are necessary to fill the gap between UML's generality and the development idiosyncrasies of specific application domains. Also, the impending "stabilisation" of the Unified Modelling Process will most probably reduce significantly the number of new methodologies that bring only marginal modifications to existing practices.

Finally, we should note the quasi-unanimous acceptance of UML and the fact it is generally perceived as a very useful tool, with luminous future. For instance, Xie et al. are particularly satisfied with the support provided by UML for iterative development and the value of frequent shifts between views, especially between the use-case view and the logical view, which are deemed to accelerate the understanding of the requirements and the developing of new ideas [Xie99]; Barrios and Lopez's study highlights UML's versatility in supporting, at higher levels of abstraction, the design of complex systems that necessitate mixed implementation solutions [Barrios99]; the [Becker00] paper shows that it provides adequate input to a CASHE tool such as SIM2SYS (developed by the authors to support their

methodology); Lu et al. consider that UML's semantics help the clarification of requirements while its diagrammatic notations enhance the "understandability, traceability, verifiability, and modifiability of the requirements" [Lu99, pp. 133]; and Watkins et al. deem it a "rich methodology," well equipped to "express the requirements of large systems" [Watkins98, pp.149].

Of course, not all UML is shining brightly in the limelight, and beyond shortcomings mentioned at the beginning of Section 3.3 and inherent limitations that have triggered a significant number of UML extensions and combinations with other notations there are also some other deficiencies in the language, for example logical flaws in the definition of certain UML concepts, as pointed out by [Simons99] who strongly disputes the soundness of a number of features pertaining to use cases. But the vast preponderance of useful features present in the language, as well as UML's versatility and widespread acceptance have provided us with good reasons to decide to employ it in our approach. Of course, because the magic concoction is still boiling over the fire, we probably have to cool it a bit first and then see if its flavour is the one promised by its tempting aroma. Or, to put it in a different way, we believe that only time and practice will tell us what things deserve to stay.

3.4 Chapter Summary

In this chapter the two specification languages, Z and UML, that pertain to the thesis' research space and further characterise the topic of this dissertation have been presented. Examples of application as well as surveys of both Z and UML landscapes, including descriptions of extensions, research directions, and ways of utilisation, have been included. UML's support for RT software development has been described and Z++, the object-oriented variant of Z used in the thesis has been succinctly introduced, with a view of further detailing it in Chapter 6, where it provides fundamental support for the formalisation approach proposed in this thesis. Remarks on the current expectations raised by UML have also been presented.