6 TRANSLATIONS BETWEEN uml AND z++: FORMALISATION AND DEFORMALISATION

"Poetry is what gets lost in translation."

[attributed to Robert Frost (1874-1963)]

6.1 Introduction

This chapter presents the translation processes between UML models and their corresponding Z++ specifications. Emphasis is placed on the UML to Z++ translation, whose purpose is to increase the rigor of the system's description, but in order to make formal specifications easier to understand during the integrated modelling of the system the reverse translation, from Z++ to UML, is also considered. The first type of translation, alternatively referred to as formalisation, applies both to UML class diagrams, which capture structural aspects of the system, and to UML state diagrams, which describe the system's dynamics. The second translation, alternatively denoted deformalisation, produces UML classes from the information contained in Z++ specifications and thus can be considered "structure-oriented". The focus is on those parts of formalisation and deformalisation that can be performed automatically, a detailed set of translation principles and a translation algorithm based on these principles being presented for each process. The formalisation and deformalisation processes described in this chapter are included in the larger modelling frame of TCS that constitutes the subject of Chapter 7 and their application is illustrated in the Elevator Controller case study presented in Chapter 8.

6.2 Preliminary Remarks

The modelling approach described in this thesis relies on the combined use of UML and Z++. In Chapter 7 details are given on the complete UML/Z++ integrated modelling process proposed in the thesis, a process that consists of a number of activities such as definition of use cases, construction of UML class diagrams, and elaboration of Z++ specifications. In the present chapter the focus is on two key parts of this process, the formalisation and deformalisation activities. Before describing these two activities, which essentially consist of translations between UML models and Z++ specifications, some general observations are necessary.

First, a couple of remarks on terminology. Specifically, in the larger frame of the modelling approach described in Chapter 7 formalisation and deformalisation are denoted activities (or subprocesses), yet for simplicity in the present chapter we refer to them as processes (another possible generic term for formalisation and deformalisation, procedure, was avoided because it appears extensively in the pseudocode description of the algorithms presented later in this chapter). Also, the term translation (from UML to Z++, or from Z++ to UML), used as a substitute for formalisation and, respectively, deformalisation, should be seen as "selective translation" since in both cases only a partial mapping from one modelling space to the other is performed (in the case of deformalisation the term "truncated translation" would be even more accurate since significant informational content is possibly discarded when generating UML constructs from Z++ specifications).

In what regards formalisation, its main role in the approach presented in this thesis is to help both developers and their clients gain a better understanding of the system under construction by increasing the rigour of the system's description. With an accurate insight into the system's desired structure and behaviour those involved in the early stages of the system's development will be able to avoid a significant number of potentially very costly specification errors. Also, since the formalisation process makes precise and amenable to formal reasoning and formal refinement the initially written in UML description of the

system, it opens the door for subsequent formal processing, but aspects regarding formal analysis of specifications and formal refinement of specifications to code are not dealt with in the present thesis.

Guidelines for formalising object-oriented semi-formal models have been proposed by Lano and Haughton in [Lano94c] and by Lano in [Lano95]. They represent the starting point for the semi-formal to formal translation process presented in this chapter but it should be pointed out that Lano and Haughton's work was concerned with the formalisation in Z++ of OMT models, so we have adapted and extended their approach to UML models. Also, in the present approach we have attempted to provide a systematic description of the formalisation, through detailed sets of principles and detailed algorithms, and have additionally tackled the reverse translation from formal specifications to graphical representations, translation that was not considered by Lano and Haughton.

As in the case of Lano and Haughton's work, the approach proposed in this thesis addresses the formalisation of both structural and behavioural aspects of the system. For the latter, the same RTL formalism proposed by Jahanian and Mok is employed but differences exist between the two approaches regarding the details of this employment, as shown in Section 6.4. In practical terms, the formalisation of UML constructs in Z++ consists of two components, formalisation of class diagrams (described in Section 6.3, and concerned primarily with structural aspects of the system), and formalisation of state diagrams (presented in Section 6.4 and dealing with behavioural characteristics of the system). The formalisation of UML models applies only to the core elements of the language (class diagrams, classes, relationships, and state diagrams) but, as shown in studies published by authors who have worked on similar formalisation approaches, these constructs provide good insights into the system and allow formal reasoning about its properties [Lano95, France99, Kim99a].

Additional reference for the formalisation processes described in this chapter has been provided by the work of Kim and Carrington on formalising UML models in Object-Z

[Kim99a, Kim00a, Kim00b]. In particular, their formal Z description of UML class diagram constructs, preliminary to the translation procedure from UML to Object-Z, has served us to better define and organise the rules for well-formed UML class diagrams presented in Subsection 6.3.1.

In what regards the reverse translation, from Z++ specifications to UML constructs, it should be noted that it has a secondary role in the modelling process, its purpose being to make easier the interpretation of the integrated model by developers and users not trained in formal methods. This feature may or may not be used within a particular modelling context, but its inclusion in the proposed approach allows a form of "reverse engineering," from formal specifications to semi-formal graphical descriptions. In practice, it is thus possible to have some Z++ specifications developed first and then their class structure propagated into the UML space. This allows an improved communication between developers skilled in formal methods and developers and users that favour the graphical representation of the system. The deformalisation option is not a common feature in integrated approaches and its practical utility is smaller than that of formalisation. In fact, the only other approach that deals with the reverse propagation of models is Headway System's RoZeLink [RoZeLink99], from which we have borrowed the idea. Nevertheless, the reverse translation suggested in Section 6.5 is significantly distinct from that used in RoZeLink, major differences stemming both from the quite dissimilar OO variants of Z used (ZEST in the case of RoZeLink, and Z++ in our case) and from the particular way the Formaliser structured editor used in conjunction with RoZeLink continually enforces the correct syntax of ZEST specifications [Formaliser01].

Since both formalisation and deformalisation processes can be partially automated we focus in this chapter on those translation operations that can be implemented by a computer program. For each process a set of translation principles is presented first and then, based on these principles, an algorithm that allows the automatic execution of parts of the translation is proposed. A number of issues pertaining to the practical utilisation of the formalisation

and deformalisation algorithms, in particular regarding their combined application, are discussed in Section 6.6.

6.3. Formalisation of UML Class Diagrams in Z+

The first part of formalisation addresses the translation of UML structural constructs to Z++. This formalisation applies to UML class diagrams and to the elements they contain (classes and relationships), the result being a set of corresponding Z++ classes. For the target language of the translation, Z++, it is useful to consider again the general form of a Z++ class, introduced in Chapter 2 and presented in more detail in Appendix A, and to notice that a supplementary clause, `PUBLICS`, has been included in the definition of Z++ classes. This clause allows better specification of member visibility, in the same way the ↾ list of Object-Z classes declares the attributes and operations that are externally accessible through the dot notation [Duke94]. (The introduction of this clause is in agreement with the declared intention of Z++'s authors, who designed the language's syntax "to enable simpler extension of the notation by the addition of new clauses to a class definition" [Lano94d, pp. 138]). During the automatic translation the clauses of Z++ classes are partially filled in according to the information contained in UML class diagrams and then the formal specifications can be enhanced by developers with details of data structures, definition of operations, and more elaborate constraints. In this section, the input considered for the formalisation process is a single class diagram, a discussion regarding the application of the process to a set of class diagrams, as well as to a class or a group of selected classes being presented in Section 6.6.

6.3.1 Rules for Developing Well-Formed Class Diagrams

In order to reliably perform the translation of UML structural constructs into Z++ specifications a number of constraints on the syntactic structures of UML class diagrams must be enforced. These constraints ensure that the UML constructs are syntactically well-

formed and thus can be subjected to automatic translation to Z++. Many of them represent restrictions on the development of UML models that are due to the specifics of the target language of the translation, Z++ (they can be described as "compatibility constraints" between UML and Z++), for instance interfaces and abstract classes are not treated since there are no equivalent constructs for them in Z++ and, if parameters of operations are provided in UML, both the names and the types of parameters must be specified in order to allow the automatic formalisation of operation signatures. Other restrictions represent simplifications of UML in cases in which it has been considered that the burden on the formalisation process would not be compensated in practice by the inclusion of less frequently used features (e.g., only binary relationships are considered).

These constraints, given below in the form of rules for developing well-formed class diagrams, raise indeed the level of rigour required in the UML space and reduce to a certain degree the modelling options of the UML developer. However, this reduction in modelling flexibility is well compensated by the benefits of the more precise descriptions made possible by formalisation. Also, while rather large and detailed, the set of constraints described below is however not exhaustive, its purpose being to avoid the more common modelling errors that would prevent reliable automatic formalisation of class diagrams. In addition, minor constraints such as restrictions on the number of characters used in the names of UML constructs have been omitted for simplicity.

The rules for well-formedness presented in this section have been inspired primarily from [Kim99a], with additional observations drawn from [Lano95]. Many rules have been added (e.g., rules regarding attributes and operations, rules for generic classes) while some have been discarded (association classes are not considered). All rules are commented and organised in a manner intended to facilitate the subsequent description of the translation principles presented in Subsection 6.3.2 and of the formalisation algorithm AFCD (Algorithm for Formalising Class Diagrams) described in Section 6.3.3.

6.3.1.1 Rules for Class Diagrams

The following must be satisfied by each class diagram that is subjected to formalisation:

- The class diagram consists only of classes and binary relationships between classes; (6.1)
- There is a finite number of classes and a finite number of relationships in the class diagram; (6.2)
- Each relationship that belongs to the given class diagram involves two classes that also belong to the given class diagram; (6.3)

The first rule indicates that for formalisation purposes only classes and binary relationships between classes are considered, other structural elements of UML that in general can be included in class diagrams, such as interfaces and multiple relationships, being ignored (these are restrictions generally imposed in other similar formalisation approaches, e.g., [Bruel96], [France99], [Kim99a]). However, in practice, some of the UML constructs that are not subjected to formalisation can still be present in the UML model, but in this case means to extract a representation of the class diagram suitable to formalisation should be devised. In addition, as indicated by rule (6.4) below, the classes can be of three kinds: regular, parameterised, and binding (classes that instantiate parameterised classes [Booch98]). The AFCD algorithm described in Subsection 6.3.3 assumes that rule (6.1) is satisfied, the class diagram that represents the input to AFCD being given as two sets, one of classes, and the other of binary relationships.

Rule (6.2) imposes limitations on the cardinality of the set of classes and, respectively, of the set of relationships that make up a diagram. Included here for the sake of completeness, it can serve for a formal description (e.g., in Z or Z++) of the formalisation algorithm.

Rule (6.3) makes sure that the input provided to AFCD is valid in the sense that no extraneous classes are involved in a relationship that belongs to the input class diagram. In

practice, this rule has an impact on the way two or more class diagrams can be related for translation purposes, as discussed in more detail in Section 6.6.

Some other rules presented later in Subsection 6.3.1 can also be seen as applied to class diagrams, for instance rule (6.37) that prevents more than one generalisation relationship between any two classes, but for presentation reasons they have been described as "rules for relationships," after the description of the rules for classes and the introduction of the kinds of relationships considered for formalisation.

6.3.1.2 Rules for Classes

The following constraints apply to UML classes contained in the class diagram that provides the input of the formalisation process:

- Each class is either a regular class, a parameterised class, or a binding class; (6.4)
- Each class has a name, a finite number of attributes and a finite number of operations; (6.5)
- In addition to name, attributes, and operations, each parameterised class and each binding class has a finite number of class parameters (in the following, the parameters of parameterised classes are denoted formal class parameters while the parameters of binding classes are denoted actual class parameters). Regular classes do not have class parameters; (6.6)
- The name of each regular class is unique within the class diagram; (6.7)
- The name of each parameterised class is the same as the name of its binding classes but is distinct from the names of all other classes that belong to the class diagram; (6.8)
- The name of each binding class is the same as the name of the parameterised class it binds and the name of other binding classes that instantiate this parameterised class, but is distinct from the names of all other classes that

belong to the class diagram; (6.9)

- Each parametrised class and each binding class has at least one class parameter; (6.10)
- Each formal class parameter and each actual class parameter is given only as a name; (6.11)
- Each instantiating class has the same number of parameters as the parameterised class it binds; (6.12)
- Each attribute has a name and, optionally, a type, a visibility, an initial value, and a property; (6.13)
- The name of each attribute of a class is distinct from the names of all attributes and operations that belong to the same class; (6.14)
- The visibility of an attribute is one of the following: public, protected, or private; (6.15)
- The property of an attribute is either changeable or frozen; (6.16)
- Each operation has a name and, optionally, a visibility, a finite list of parameters, a return type, and a property; (6.17)
- The name of each operation of a class is distinct from the names of all operations and attributes that belong to the same class; (6.18)
- The visibility of an operation is one of the following: public, protected, or private; (6.19)
- The property of an operation is either none or query; (6.20)
- Each parameter of an operation has a name, a type, and, optionally, a direction; (6.21)
- The parameters of an operation have unique names within the operation's list of parameters; (6.22)
- The direction of each operation parameter is one of the following: in, out, or inout; (6.23)
- The type of each attribute, class parameter, operation parameter, and the return type of each operation is either a basic type, a class type, or

an array type; (6.24)

- Each formal class parameter denotes a basic type or a class type that is not the type defined by a parameterised or binding class; (6.25)
- The name of the each formal parameter is different from all the names of types used in the class diagram outside the parameterised class to which the formal parameter belongs ; (6.26)
- The name of an actual class parameter is the name of a basic type or of a class type that is not the type defined by a parameterised or binding class. (6.27)

In the above, rule (6.4) specifies the types of classes that are subjected to formalisation. In essence, only the regular UML classes and the UML parameterised classes together with their binding classes are translated to Z++, which also allows parameterisation of classes (the parameterised classes are also referred to as template classes, or as generic classes, while the binding classes are alternatively denoted instantiating classes).

The structure of classes that is considered by the formalisation process is specified in rules (6.5) and (6.6), the former giving the regular class structure while the latter appending the requirement for class parameters in the case of parameterised and binding classes. As in the case of rule (6.2), the requirements for a finite number of items in rules (6.5), (6.6), and (6.17) are included for the sake of completeness. Evidently, the formalisation algorithm will work on a finite input.

Rules (6.7) to (6.9) provide constraints on the naming of classes. In general, within a class diagram the names of classes must be unique, but exceptions to this principle are necessary to accommodate binding of template classes such as `Queue[X]`, which can be instantiated as `Queue[Task]`, `Queue[Patient]`, etc. (this is denoted implicit binding). In UML there is a second way of instantiating parameterised classes, explicit binding, with the name of the binding class different from the name of the template class, but for simplification purposes the formalisation algorithm assumes only implicit binding is used in class diagrams. In practical

terms, to ensure efficient checking of class names, the AFCD will consider as names of generic and binding classes the string formed by concatenating the name of the class with the list of the class' parameters. As such, it is easier to automatically detect that, for instance, the class Queue[Task] is distinct from the class Queue[Patient]. Also, this internal representation is needed for the specification of relationship ends, as indicated in Subsection 6.3.1.3.

Rules (6.10) to (6.12) deal further with the well-formedness of template and instantiating classes. Obviously, the absence of parameters would contradict the concept of parameterised classes, hence rule (6.10), and the matching between formal class parameters and actual class parameters must also be enforced, as stated by rule (6.12). Rule (6.11) limits the format of class parameters to a single name, whose use is further restricted by rules (6.25) and (6.26).

Rules (6.13) to (6.16) are concerned with the well-formedness of attributes. Although the visibility and the property of an attribute are listed as optional in rule (6.14), the AFCD will assign default values for these two components if none is provided (public for visibility and changeable for property). Also, even though Z++ requires types for all the attributes, we decided to allow the AFCD to translate attributes without their types specified in UML, leaving to the developer the task of specifying in Z++, post translation, the missing types of attributes. Rule (6.14) requires unique names for attributes in a given class. Notably, the names of attributes must also be distinct from the names of operations, including inherited operations, a constraint that stems from the specifics of Z++ and from the addition of the PUBLICS clause, which lists attributes and operations without their type. Rule (6.15) specifies the possible kinds of attribute visibility and rule (6.16) gives details about allowable values for attribute property. The inclusion of rule (6.16) serves the formalisation process since the frozen (constant) attributes are included in Z++ in the clause FUNCTIONS while the changeable attributes are specified in the OWNS clause.

Regarding the visibility of attributes and operations addressed by rules (6.15) and, respectively, (6.19), public attributes and operations will be made visible in Z++ by their inclusion in the PUBLICS clause, while private attributes and operations will require the use

of an intermediary class and of a hiding operation applied to this class, as detailed in the formalisation algorithm. Following from the specifics of Z++ and from the introduction of the PUBLICS component in the definition of Z++ class, protected attributes and operations will not require any special treatment.

Rules (6.17) to (6.23) address syntactic aspects of operations. Regarding the uniqueness of an operation name in a class required by rule (6.18), considerations similar to those for rule (6.14) apply. Rule (6.20) has been included to support the translation process to since query operations, which do not change the state of the object, are listed separately (in the RETURNS clause) from the regular operations indicated by the none property (these operations are listed in the ACTIONS clause of the Z++ class). Rules for the parameters of operations are also necessary to help the automatic translation to Z++. In particular, both the name and the type of a parameter are required (6.21), since both are necessary in Z++ for declaring operations and an automatic assignment of parameter names by the AFCD would complicate unnecessarily the translation. Also, unique names for the parameters of an operation are required in Z++ even though they may have distinct types, hence rule (6.22), and the provisions of rule (6.23) are used in specifying the signatures of operations in Z++. If unspecified, the direction of a parameter will be considered in.

Rule (6.24) indicates that three kinds of types are possible for attributes, parameters of template classes, parameters of operations, and the returns of operations. Class types are all the types whose name is identical with one of the names of classes that exist in the class diagram. For practical purposes, the formalisation algorithms will accept names of types given either as T, T[ ], or T[params], where params is a set of class parameters (more details are given in Subsection 6.3.2.1).

Rules (6.25) to (6.27) further restrict the use of class parameter names in order to avoid possible complications when formalising generic classes.

6.3.1.3 Rules for Relationships

The following rules apply to relationships between classes included in the class diagram:

- Each relationship between two classes is either an association, an aggregation, a composition, a generalisation, or an instantiation; (6.28)
- Each association relationship has a name; (6.29)
- Each relationship has two relationship ends; (6.30)
- Each end of a relationship is attached to a class; (6.31)
- Each end of a relationship has one of the following types, depending on the kind of relationship to which it belongs:
  (a) assoc in the case of association;
  (b) aggreg, if the end is attached to the "whole" class of the aggregation, and none if the end is attached to the "part" class;
  (c) comp, if the end is attached to the "whole" class of composition, and none if it attached to the "part" class;
  (d) super, if the end is attached to the superclass of a generalisation, and none if it is attached to the subclass;
  (e) generic, if the end is attached to the parameterised (generic) class of an relationship, and none if it is attached to the binding class; (6.32)
- Each end of a relationship has a multiplicity constraint attached, which is expressed in the form of a finite sequence of ranges

  $$a_1 .. b_1, a_2 .. b_2, \ldots , a_K .. b_K$$

  where:

  $$K > 0,$$

  $$\forall i, 1 \le i \le K, a_i \ge 0, b_i > 0, a_i \le b_i$$

  $$\forall i, 1 \le i \le K-1, b_i < a_{i+1},$$

  and b_K only may be $+\infty$ (denoted *) (6.33)
- The multiplicity of the relationship end that is attached to the "whole" part of a composition relationship is 1; (6.34)

- Both ends of a generalisation have multiplicity 1; (6.35)
- Both ends of an instantiation have multiplicity 1; (6.36)
- Between any two given classes, if more than one relationship exist, the relationships are all either associations or aggregations/compositions; (6.37)
- The names of the associations that involve the same two classes are distinct; (6.38)
- Each generalisation involves two distinct classes; (6.39)
- Each instantiation is between a parameterised class and an instantiating class; (6.40)
- A class cannot be the superclass of any of its ancestors; (6.41)

Rule (6.28) specifies the kinds of relationships considered in the present approach. Compared with the types of UML relationships described in Section 3.3, the dependency and realisation relationships are not included (with the exception of the instantiation version of dependency). Also, it should be noted that the term instantiation relationship is not in the UML vocabulary, but we use it here to describe in a shorter way the dependency relationship between a parameterised class and a binding class.

The names of association relationships are needed for formalising purposes, hence rule (6.29).

Rules (6.30) and (6.31) enforce non-tangling relationships by requiring that each relationship be specified in terms of two relationship ends, each end being attached to a class.

Rule (6.32) specifies constraints on relationship ends for properly formed relationships. It avoids incorrect situations such as a relationship with both ends of type aggregation.

Rule (6.33) gives a general form for the multiplicity constraint attached to a relationship end. This form encompasses all cases normally used in UML, including the multiplicity 1, which can be represented as 1 .. 1, and the notation *, which can be represented 0 .. *.

Rule (6.34) makes sure that unshared containment, characteristic to composition, is properly specified in terms of multiplicity while rules (6.35) and (6.36) do the same for the instantiation of parameterised classes and, respectively, for generalisation.

Rule (6.37) gives the conditions under which multiple relationships between two classes are allowed, while rule (6.38) makes sure that duplicate associations can be mechanically formalised.

Rule (6.39) prevents a class to be its own superclass, while rule (6.40) defines more precisily the instantiation relationship;

Finally, rule (6.41) avoids invalid situations in which a class acts as superclass to one or more of its ancestors. Technically, rule (6.41) incorporates rule (6.39), but the latter was included for increased clarity. The AFCD will detect the existence of cycles in the graph whose nodes are the classes and whose links are the generalisation relationships contained in the input class diagram.

The set of rules for relationships described above need be completed with rules regarding the involvement of generic and binding classes in other types of relationships than instantiation. To keep things simple, the algorithm for automatic translation will assume that invalid situations such as a generic class at the “part” end of an aggregation whose “whole” end is attached to a regular class are resolved by the developer before the algorithm is applied.

6.3.2 Translation Principles for Class Diagrams

The automated translation of UML class diagrams to Z++ specifications follows a number of principles, as described below.

6.3.2.1 Translation of Types

In order to facilitate the mechanisation of the formalisation process restrictions are placed on the use of types, as indicated in rule (6.24). In the UML space the considered types of attributes, parameters of operations, and returns of operations (henceforth collectively denoted UML types) can be expressed in one of the following forms:

(a) In "scalar form" `T`, where `T` is a string identifier denoting either a basic type or a regular class type (the latter means that a regular class with name `T` exists in the class diagram);
(b) In "array form" `T[]`, where `T` is the name of a basic type or a regular class type (note the empty space within the square brackets, meaning that only one dimensional arrays are automatically processed and the information on array bounds, if any, is left to be formalised manually by the developer);
(c) In "generic form" `T[`params`]`, where params is a list of parameters passed to a template class, each parameter in params denoting a basic type or a regular class type (array types and types in generic form are not allowed within params, as indicated by rule (6.27)).

With these restrictions, the mapping of types from UML to Z++ proceeds along the following lines:

- When the UML type is expressed in scalar form `T`, then:
  - if `T` is the name of a recognised basic type, specifically unsigned integer, integer or real then the correponding Z++ type will be, respectively, $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$ (variants such as byte, int, long, double, and float will also be treated as recognised basic types within the above three categories). Constraints on the range of the type, if needed, will be specified by the human formaliser. The Boolean type will be recognised for the returns of operations, but no explicit output variable and no output domain will be associated in Z++ to the operation's return. Also, the type void of an operation's return will be recognised and treated as a type that requires no specification of output domain in the

operation's signature and no specification of output parameter in the operation's definition;

- if T is the name of an existing regular UML class then the Z++ type will also be T;
- if uppercase(T) is the name of an existing given set in Z++, then the Z++ type will be uppercase(T) (by uppercase(X) we denote the string obtained from the identifier X by promoting to uppercase all its lowercase letters, while keeping the others unchanged);
- if T is neither the name of a recognised basic type, nor the name of an existing regular UML class or of an existing Z++ given set, it will be treated as the name of a unrecognised basic type and a new given set will be added in Z++, with the letters of the identifier T written in uppercase, as it is customary in Z. The Z++ type will therefore be uppercase(T);
- if T is used in the context of a parameterised class and it is identical with the name of a formal parameter of the class, the Z++ type will also be T;

- When the UML type is an array type T[], then T will be first checked as described above and then the operator seq will be applied to the Z++ type corresponding to the scalar type T. For instance, the UML type int[] will become seq(ℤ) in Z++ and the UML type Car[] will be mapped either to seq(CAR), if no class with the name Car exists in the class diagram, or to seq(Car), if Car is the name of an existing UML class;
- When the UML type is given in generic form T[params] it will be assumed that the items of the params list represent actual parameters for the generic class T. If these parameters are provided by the formal parameters of the enclosing class, they will be left unchanged, otherwise each parameter P of params will be checked against recognised basic types, existing regular classes, and existing given sets, as outlined previously for types expressed in scalar form. It is possible therefore that a new given set will be created in Z++ if P is neither the name of an existing class, nor the name of a recognised basic type (e.g., the UML type Stack[Book]will lead to the creation of the given set BOOK in Z++ if class Book does not exist in the class diagram).

Since in order to allow an earlier transfer of UML class diagrams to Z++ specifications the UML types can be left unspecified, the formalisation algorithm may produce incomplete definitions for attributes and operations in Z++. This means that after the automatic

translation is performed one of the first tasks of the human formaliser will be to complete the information on types if further development of formal specifications is intended.

6.3.2.2 Translation of Attributes

The following apply when translating to Z++ the attributes of UML classes :

- The names of UML attributes will be used as names for the corresponding Z++ attributes, for instance the attribute size in UML will be mapped into the same name attribute size in Z++;
- The property of the UML attribute will determine the clause in which the corresponding Z++ attribute is placed. Attributes that cannot be modified, declared frozen in UML, will be placed in the `FUNCTIONS` clause of the Z++ class, while all other attributes (changeable) will be included in the `OWNS` clause. Due to Z++'s specifics, it is assumed that frozen attributes are also declared protected, since the constants declared in the `FUNCTIONS` clause of a Z++ class are local to the class and to its subclasses;
- The initial value of the attribute, if provided in UML, will be used as follows:
  - if the attribute is listed as changeable the initialisation of the attribute will be performed using an assignment statement in the init operation of the Z++ class;
  - if the attribute is frozen the initialisation will be performed in the predicate part of an axiomatic box definition that will be included in the `FUNCTIONS` clause;

  It is assumed that the type of the initial value of the attribute is the type of the attribute, which means that for array types the initial values must be given as sequences of the form $\langle v_1, \ldots, v_n \rangle$, $n \geq 0$;
- The visibility of an attribute att of a class C will be treated as follows by the translation algorithm:
  - if the attribute has public visibility the name of the attribute will be appended to the clause `PUBLICS` of the Z++ class C;
  - if the attribute has protected visibility no special measures will be taken since in Z++ all attributes are inherited automatically by the derived classes;

- if the attribute has private visibility it will be appended to the list of hidden features kept by the algorithm for each class. This list, if not empty after the processing of all the attributes and operations of the class, will require a hiding operation applied to the class, as detailed in Subsection 6.3.2.4.
- the type of the attribute will be determined according to the translation priciples for types presented in Subsection 6.3.2.1.

6.3.2.3 Translation of Operations

The following principles apply for translating to Z++ the operations of UML classes:

- The names of UML operations will be used as names for the corresponding Z++ operations, for instance the operation determineTrend in UML will be mapped into the same name operation determineTrend in Z++;
- The property of an op operation of a UML class C will determine the clauses of the Z++ class C in which the signature and the definition of the corresponding Z++ operation op are placed. Operations declared query, which do not change the state of the object, will have their signatures specified in the RETURNS clause of the Z++ class C, while all other operations will have their signatures included in the OPERATIONS clause. For both query and non query operations, definitions specified as indicated below are included in the ACTIONS clause of the Z++ class;
- The parameters of the UML operation op, if any, are processed as follows:
  - the type of each operation parameter will be processed according to the translation principles for types described previously and the Z++ type of the parameter will be added to the Z++ operation's signature according to the direction of the parameter. Specifically, if the direction of the parameter is in then the type of the parameter will be added to the list of input domains, if the direction is out it will be added to list of output domains, and if the direction is inout it will be added to both lists;
  - the name of each operation parameter is used to construct the initial part of the operation's definition in Z++. If the type of the parameter is in the name of the

parameter post-fixed by the symbol ? (denoting an input variable in Z) will be appended to the operation's definition list of input parameters and if the type of the parameter is out, the name of the parameter postfixed by ! (denoting an output variable in Z) will be added to the operation's definition list of output parameters. If the direction of the parameter is inout, both the above operations will be performed;

- The return type of an UML operation, if present and different from void and Boolean, will be first processed according to the principles outlined for types in Subsection 6.3.2.1 and then placed as an item in the list of output domains of the corresponding Z++ operation's signature. If void or Boolean, no action will be taken;
- The visibility of each UML operation will be processed similarly to the visibility of UML attributes. The name of a UML public operation will be included in the PUBLICS clause of the Z++ class in which the corresponding Z++ operation has been created while the name of a private operation will be added to the list of hidden features maintained by the translation algorithm for each Z++ class for the purposes described in Subsection 6.3.2.4. Protected UML operations will not require any special treatment.

6.3.2.4 Translation of Classes

The following apply for automatic formalisation of UML classes in Z++:

- Only regular and generic classes will be translated, no action being necessary for binding classes, which simply instantiate generic classes. In fact, a particular instantiation of an existing generic class may not necessarily correspond to a binding class present in the class diagram (e.g., if the parameterised Producer[X] class exists in the class diagram, a variable can can declared as P:Producer[Car] in a UML class without having the Producer[Car] explicitly drawn in the class diagram);
- The names of UML classes will be used for their corresponding Z++ classes, each regular or generic UML class C being mapped into a class with the same name C in Z++;
- The class parameters of a generic UML class will be listed in the parameter list of the corresponding Z++ class;

- The names of all direct superclasses of a UML class will be listed in the EXTENDS clause of the corresponding Z++ class;
- All the attributes of a UML class will be processed according to the principles described previously in Subsection 6.3.2.2, information being placed in the PUBLICS, FUNCTIONS, OWNS, and ACTIONS clauses of the corresponding Z++ class, as well as in the list of hidden features maintained by the algorithm for the Z++ class. The list of given sets of the Z++ specification will be updated during this process based on the information contained in the types of UML attributes;
- All the operations of a UML class will be processed according to the principles described previously in Subsection 6.3.2.3, information being placed in the PUBLICS, RETURNS, OPERATIONS, and ACTIONS clauses of the corresponding Z++ class, as well as in the list of hidden features maintained by the algorithm for the Z++ class. The list of given sets of the Z++ specification will be updated during this process based on the information contained by the types of operation parameters and the type of operation return;
- After all the classes in the class diagram are processed as described above, the classes C with a non empty list of hidden features will be used for creating hiding classes, prefixed by the symbol H (from Hiding), classes needed for providing the desired visibility of attributes and operations. Specifically, for each class C with hidden features an operation $H\_C \triangleq C \setminus [hidden\_features_C]$ will be included in the Z++ specification and the class H_C will be used instead of C in the EXTENDS list of classes that have C superclass.

6.3.2.5 Translation of Relationships

The relationships included in a class diagram are formalised in Z++ as follows:

- Inheritance relationships (generalisations) are formalised during the translation of classes through the inclusion in the EXTENDS clause of each Z++ class of the names of the class' immediate superclasses;

- Instantiation relationships are formalised during the translation of classes by including the formal parameters of the class in the parameter lists of Z++ classes, as described in Subsection 6.3.2.4;
- Aggregation and composition relationships are formalised by adding to the container class of the relationship an attribute that indicates the contained object or objects. Specifically, if the aggregation or composition is between class W ("whole") and the class P ("part"), then the attribute will be created in class W with a name and a type that depend both on the multiplicity of the "part" end of the relationship, as follows:
  - if the multiplicity is "one," then the attribute will have the name p (the class name in lowercase) and its type will be P. For instance, given a one-to-one aggregation or composition between the classes Radio and Antenna, with Antenna the "part" class of the relationship, then the attribute antenna : Antenna will be created in the class Radio;
  - if the multiplicity is "many," then the attribute will have the name p+"s" and its type will be $\mathbb{P}$P. For instance, considering a one-to-many aggregation or composition between Radio and Button, with Button the "part" class of the aggregation, then the attribute buttons : $\mathbb{P}$Buttons will be created in the class Radio.

  However, if attributes of type P or $\mathbb{P}$P already exist in W, no additional attribute describing the aggregation/composition will be created in W.
- Associations relationships are formalised by creating a Z++ class that describes the association and by including in the System class of the Z++ specification an object of this class, with appropriate constraints attached. More precisely, considering a many-to-many association assoc between classes A and B, then:
  - a class with the name AssocDescriptor will be created in Z++;
  - the attributes instancesOfA of type $\mathbb{P}$A, instancesOfB of type $\mathbb{P}$B and assocInstances of type A $\leftrightarrow$ B will be included in the OWNS clause of the AssocDescriptor class;
  - the constraint dom assocInstances = instancesOfA $\wedge$ ran asssocInstances = instancesOfB will be included in the INVARIANT clause of the AssocDescriptor class;
  - the object theAssocDescriptor of type AssocDescriptor will be included in the OWNS clause of the System class of the specification.

For instance, considering the many-to-many association departs between the classes Flight and Airport, then the class DepartsDescriptor will be created in Z++ with attributes instancesOfFlight: $\mathbb{P}$ Flight, instancesOfAirport: $\mathbb{P}$ Airport, and departsInstances: Flight $\leftrightarrow$ Airport placed in its OWNS clause (the names of the classes are underlined to indicate that instances of associations are created between existing objects of the classes). A single object theDepartsDescriptor of type DepartsDescriptor will also be created in the System class of the Z++ specification.

If the association is one-to-one or many-to-one from A to B than the type of the attribute assocInstances will be $A \rightarrowtail\!\!\!\!\!\!\rightarrow B$ and if the association is one-to-many from A to B the attribute's type will be $B \rightarrowtail\!\!\!\!\!\!\rightarrow A$.

6.3.3 Algorithm for Formalising Class Diagrams (AFCD)

Based on the rules for syntactically well-formed UML class diagrams, classes, and relationships presented in Subsection 6.3.1 and on the formalisation principles described in Subsection 6.3.2, an algorithm for translating the core structural UML constructs into Z++ specifications is given below in a Pascal-like pseudocode. The structure of the algorithm's input as well as the format of the algorithm's output are given first and then the algorithm is detailed in top-down fashion. The code of a Java program that implements the algorithmic contents of ADFC and adapts its data structures for an OO solution is included in Appendix B. Details that have been omitted from the presentation that follows can be found in the code presented in this Appendix. As a matter of convention, in ADFC's pseudocode the basic structuring module employed, the procedure, is specified as follows:

procedure ProcedureName (<inputParams>; <outputParams>) (6.42)

where <inputParams> is a list of parameters given in the form $<ip_1 : T_1, ip_2 : T_2, ..ip_M : T_M>$, with each ip_i, $1 \leq i \leq M$, an input parameter of type T_i, and <outputParams> is a list of the form $<op_1 : T_1, op_2 : T_2, .. op_N : T_N>$, with each op_j, $1 \leq j \leq N$, an output parameter of type T_j. For simplicity, the implicit type of output parameters is considered to be inout, meaning that the calling

module passes them to the procedure, which returns them after execution in a possibly modified form.

6.3.3.1 AFCD Input

The input of the formalisation algorithm is a representation of a UML class diagram, denoted CD, that consists of the tuple (C, R) where C is the set of classes and R is the set of binary relationships between the classes, $R : C \leftrightarrow C$. In terms of the structure, the following are considered:

$$C = \{C_0, .., C_{N-1}\}, N \geq 0 \tag{6.43}$$

with $N = 0$ for the empty set of classes $C = \varnothing$. Similarly:

$$R = \{R_0, .., R_{M-1}\}, M \geq 0 \tag{6.44}$$

Each class c in C has the following format:

$$c = (\text{name}, \text{ctype}, \text{atts}, \text{ops}, \text{cparams}) \tag{6.45}$$

where name is a string identifier and ctype one of the following: reg, para, or bind, while the other components have the form:

$$\begin{aligned} &\text{atts} = \{\text{att}_0, \ldots, \text{att}_{Na-1}\}, \ Na \geq 0 \\ &\text{ops} = \{\text{op}_0, \ldots, \text{op}_{No-1}\}, \ No \geq 0 \\ &\text{cparams} = \{\text{cp}_0, \ldots, \text{cp}_{Ncp-1}\}, \ Ncp \geq 0 \end{aligned} \tag{6.46}$$

Each attribute att in atts has the form:

$$\text{att} = (\text{name}, \text{attype}, \text{vistype}, \text{initval}, \text{property}) \tag{6.47}$$

where name and type are string identifiers, vistype is either public, protected, or private, and property is either changeable or frozen. With respect to initval, this should be a value of type, but the formalisation algorithm does not perform type checking.

Each operation op in ops shown in (6.45) has the form:

$$op = (name, vistype, params, rettype, property) \tag{6.48}$$

where name is a string identifiers, vistype is either public, protected, or private, property is none or query, and params is a set:

$$params = \{p_0, \ldots, p_{Np-1}\},\ Np \geq 0 \tag{6.49}$$

where each parameter p in params has the form:

$$p = (name, ptype, dir) \tag{6.50}$$

with name and ptype string identifiers and dir one of in, out , or inout.

Each class parameter cp in cparams given in (6.46) is a string identifier and attype of (6.47), rettype of (6.48) and ptype a (6.50) are type identifiers given as T, T[ ] or T[tparams], where T is a string identifier and tparams is a list:

$$tparams = (tp_0, \ldots, tp_{Ntp-1}),\ Ntp \geq 0 \tag{6.51}$$

with each $tp_i, 0 \leq i \leq Ntp-1$, a string identifier.

Each relationship R in *R* of (6.44) has the form:

$$R = (name, rend1, rend2) \tag{6.52}$$

where name is a string identifier or the reserved word null and the two ends of the relationship have the structure:

$$rend = (kind, classname, mult) \tag{6.53}$$

with kind either assoc, aggreg, comp, super, generic, or none, the classname given as a string identifier, and mult specified in the form:

$$mult = (a_1 .. b_1, .., a_K .. b_K) \tag{6.54}$$

where K and the range limits a_i and b_i, $1 \leq i \leq K$, satisfy condition (6.33).

6.3.3.2 AFCD Output

The output of the algorithm is a Z++ specification Z = (H, ZC, OC) that consists of a header H that precedes the class declarations, a set ZC of classes, and a set OC of operations on classes that gathers statements that represents operations applied on Z++ classes such as hiding and composition. A statement is considered to be a text consisting of one or more lines built according to the syntax of Z++. For AFCD purposes:

$$H = (\text{GivenSets}) \tag{6.55}$$

meaning that only given sets are placed by the algorithm in the header specification, with:

$$\text{GivenSets} = \{GS_0, ..., GS_{Ngs-1}\},\ Ngs \geq 0 \tag{6.56}$$

where each GS is an uppercase string identifier.

The set of Z++ classes has the form:

$$ZC = \{ZC_0, ..., ZC_{Nz-1}\},\ Nz \geq 0 \tag{6.57}$$

where each ZC has the structure indicated in (6.62). The set of operation on classes is given as:

$$OC = (\text{HidingOperations}) \tag{6.58}$$

meaning that AFCD constructs only hiding operations on classes for inclusion in OC, the form of HidingOperations being:

$$\text{HiddingOperations} = \{HO_0, ..., HO_{Nho-1}\},\ Nho \geq 0 \tag{6.59}$$

where each HO is a Z++ statement.

The form of each ZC in (6.59) is:

$$\begin{aligned} ZC = (&\text{NAME, CPARAMS, EXTENDS, PUBLICS, TYPES , FUNCTIONS, OWNS, RETURNS} \\ &\text{OPERATIONS, INVARIANT, ACTIONS, HISTORY}) \end{aligned} \tag{6.60}$$

which corresponds to the structure of Z++ described in Appendix A. In the above NAME is a string identifier, CPARAMS, EXTENDS and PUBLICS are lists of string identifiers and all the other components of ZC are sets of Z++ statements. Notationally:

$$
\begin{array}{lll}
\text{CPARAMS} = & \{cp_0, \ldots, cp_{Nzcp-1}\}, & Nzcp \geq 0 \\
\text{EXTENDS} = & \{ext_0, \ldots, ext_{Nxt-1}\}, & Nxt \geq 0 \\
\text{PUBLICS} = & \{pb_0, \ldots, pb_{Npb-1}\}, & Npb \geq 0 \\
\text{TYPES} = & \{typ_0, \ldots, typ_{Ntp-1}\}, & Ntp \geq 0 \\
\text{FUNCTIONS} = & \{fun_0, \ldots, fun_{Nfun-1}\}, & Nfun \geq 0 \\
\text{OWNS} = & \{own_0, \ldots, own_{Now-1}\}, & Now \geq 0 \\
\text{RETURNS} = & \{ret_0, \ldots, ret_{Nret-1}\}, & Nret \geq 0 \\
\text{OPERATIONS} = & \{zop_0, \ldots, zop_{Nzo-1}\}, & Nzo \geq 0 \\
\text{INVARIANT} = & \{inv_0, \ldots, inv_{Ninv-1}\}, & Ninv \geq 0 \\
\text{ACTIONS} = & \{act_0, \ldots, act_{Nact-1}\}, & Nact \geq 0 \\
\text{HISTORY} = & \{hist_0, \ldots, own_{Nhis-1}\}, & Nhis \geq 0
\end{array}
\qquad (6.61)
$$

From the AFCD point of view the above corresponds to the external representation of a Z++ class, but for implementation purposes additional components are used for modelling Z++ classes (they make up the "internal representation" of the Z++ class, which facilitates the translation and allows extensions of the algorithm). Specifically, a set of attributes, a set of operations and a list of hidden features are included, as shown in the AFCD code presented in Appendix B.

6.3.3.3 AFCD Pseudocode

The highest level, pseudocode description of the AFCD is given in Fig. 6.1. The input for the FCD procedure is a class diagram, and its output is a Z++ specification. The FCD procedure invokes first the CheckCDSyntax procedure to verify that the rules for well-formed class diagrams are satisfied and, if this is confirmed, proceeds with the translation of UML constructs to Z++ by calling the TranslateCD procedure. The errorFlag variable, visible across the FCD, is used to signal the detection of errors (violations of rules for well-formedness) at all levels of procedure nesting. Specific messages that indicate the kind of the errors detected are issued locally by the lower level procedures.

```
-- Top level UML to Z++ formalisation procedure

procedure FCD(CD:ClassDiagram)
   ZPPS:ZPPSpec;                    -- Z++ specification to be generated
   errorFlag := false;              -- flag to signal well-formedness errors
   begin
      CheckCDSyntax(CD);            -- check correctness of the class diagram
      if (not errorFlag) then
         TranslateCD(CD;ZPPS)       -- and translate only if no errors found
      endif;
      PrintZPPSpec(ZPPS);           -- print to file resulting Z++ specification
   end FCD;
```

Fig. 6.1 The Top Level FCD Procedure

In the following, the CheckCDSyntax procedure is described only through its high-level components, specific details of implementation being provided by the code included in Appendix B. Here, only the rules that involve more than preliminary checks of the input in terms of expected structures and valid items are covered (examples of such preliminary checks include verifying that two relationship ends have been provided for each relationship and checking that the property of an attribute is either changeable or frozen). The TranslateCD procedure is described after the high-level modules of CheckCDSyntax are presented.

```
-- Check the well-formedness of the input class diagram

procedure CheckCDSyntax(CD:ClassDiagram)
   begin
     CheckRelationships(CD);            -- check constraints at relationship level
        if (not errorFlag) then
           CheckAcrossCD(CD);
        end if;
        if (not errorFlag) then
           CheckClasses(CD);            -- check constraints at class level
        end if;
   end CheckCDSyntax;
```

Fig. 6.2 The CheckCDSyntax Procedure

The CheckCDSyntax procedure shown in Fig. 6.2 consists of three categories of checkings, each addressing a context (class, relationship, or class diagram) that corresponds from a notational point of view to the groups of rules presented in Subsection 6.3.1. However, due to practical considerations, the order of contexts has been changed and, as detailed later, the contents of each group of checkings match only loosely the contents of the associated group of rules (although globally all major rules are covered). More precisely, we have taken the approach of checking in a given context those rules that require (almost) exclusively information available in that context. For this reason, a rule such as (6.41) given previously as a relationship rule (a rule preventing a class to be the superclass of any of its ancestors) is verified in the CheckAcrossCD procedure and not in CheckRelationships. Regarding the order of checkings, the validation of the internal contents of classes (CheckClasses procedure), involving the inspection of lower-level structural details, is performed only if the other two categories of tests are passed. Also, the CheckAcrossCD procedure follows the internal checking of relationships since improperly formed relationships would preclude reliable verifications at the class diagram level.

To simplify the pseudocode descriptions that follow, the testing of the errorFlag indicator between procedures is no longer shown, but it should be considered that an error in a given procedure would generally preclude the meaningful execution of the procedures that follow. Thus, if a test fails, the execution of the algorithm will stop. With this approach, the UML developer is required to incrementally improve the well-formedness of the class diagram.

Also, since comments are included in the procedures given below, only brief indications on the correspondence between the FCD's procedures and the rules of well-formedness are given in conjunctions with the components of the CheckCDSyntax procedure.

As shown in Fig. 6.3, the internal verification of relationships consists of five tests, covering, in order, rules (6.32), (6.33), (6.29), (6.34), (6.35), and (6.36). The other rules listed as relationships rules in Subsection 6.3.1 are checked in the CheckAcrossCD procedure, shown in Fig. 6.5.

```
-- Check constraints on the relationships

procedure CheckRelationships(CD:ClassDiagram)
   begin
      CheckRelationshipEnds(CD);          -- verify proper ends of the relationships
      CheckWellFormedMultipicity(CD);     -- verify multiplicity at the two ends
      CheckAssociationsHaveName(CD);      -- verify names are given to associations
      CheckCompMultOne(CD);               -- the whole part of composition and
      CheckRelMultOne(CD,GEN)             -- both ends of generalisation and
      CheckRelMultOne(CD,INST)            -- instantiation must have multiplicity one
   end CheckRelationships;
```

Fig. 6.3 The CheckRelationships Procedure

It is necessary to note that the organisation of tests shown in Fig. 6.3 for CheckRelationships was chosen over the faster alternative depicted in Fig. 6.4 because it allows a clear demarcation of tests and a clear separation of error messages.

```
-- Alternative testing of relationships (not used). Faster, but with no clear separation of messages.

procedure AlternativeCheckRelationships(CD:ClassDiagram)
   begin
      for i = 0 to M-1 do                        -- verify all relationships
         CheckRelationshipEnds (CD.R[i])         -- verify proper ends of the relationship
         CheckWellFormedMultipicity(CD.R[i])     -- verify multiplicity at the two ends
         if (isAssociation(CD.R[i])) then
               CheckAssocHasName(CD.R[i])        -- associations must have names
         end if;
         if (isComposition(CD.R[i])) then
               CheckWholeMultOne(CD.R[i])        -- the whole part of composition
         end if;                                 -- must have multiplicity  one
         if (isGeneralisation(CD.R[i])) then
               CheckRelMultOne(CD.R[i],GEN)      -- both ends of generalisation
         end if;                                 -- must have multiplicity  one
         if (isInstantiation(CD.R[i])) then
               CheckRelMultOne(CD.R[i],INST)     -- and both ends of  instantiation
         end if;                                 -- must have multiplicity  one
      end for;
   end ALternativeCheckRelationships;
```

Fig. 6.4 Alternative CheckRelationships Procedure

More complex verification work is done by the CheckAcrossCD procedure, whose component tests are sequentially ordered based on their possible implications on other tests.

```
-- Check constraints across class diagram

procedure CheckAcrossCD (CD:ClassDiagram)
   begin
      CheckEndRelClassesExist(CD); -- verify existence of classes involved in relationships
      CheckClassNamesUnique(CD);    -- check constraints on names of classes
      CheckDistinctAssocNames(CD); -- distinct names of assocs. between the same two classes
      CheckDuplicateRelationships(CD); -- only assoc and aggreg/comp can be duplicated
      CheckInstantiationEnds(CD);   -- verify instantiation ends attached correctly to classes
      CheckMatchingBindings(CD);    -- classes in an inst. rel. must have same no. of params.
      CheckNoAncestorToSelf(CD);    -- a class cannot be ancestor to itself
   end CheckAcrossCD;
```

Fig. 6.5 The CheckAcrossCD Procedure

The rules verified by the CheckAcrossCD procedure are, in order (6.3), (6.7 to (6.9), (6.38), (6.37), (6.40), (6.41), (6.12), and (6.41).

The last procedure within CheckCDSyntax is CheckClasses, shown in Fig. 6.6, whose role is to ensure the uniqueness of names of attributes, operations, and parameters of operations, as required by rules (6.14), (6.18), and (6.22).

```
-- Check constraints at class level

procedure CheckClasses(CD:ClassDiagram)
   begin
      for i = 0 to N-1 do                        -- verify all classes in the class diagram
         CheckAttributeNamesUnique(CD.C[i]); -- verify names of attrib. within the class
         CheckOperationNamesUnique(CD.C[i]); -- verify names of ops. within the class
         CheckOpParamNamesUnique(CD.C[i])     -- verify names of op. parameters
      end for;
   end CheckClasses;
```

Fig. 6.6 The CheckClasses Procedure

The translation part of the algorithm, coordinated from the CDTranslate procedure is described next (Fig. 6.7 to 6.20).

The top-level procedure CDTranslate performs the major tasks of translating the classes and the relationships (Fig. 6.7). In order to establish the required visibilities of attributes and operations, it also applies hiding operations on classes, an activity that can take place only after both classes and relationships are processed.

The TranslateClasses procedure (Fig. 6.8) subjects to translation all non-binding UML classes by invoking TranslateClass (Fig. 6.9). Here, detailed formalisation work on individual UML classes is performed. Based on the information available in the input UML class a corresponding Z++ class is created, with its "internal representation" filled according to the translation principles presented in Subsection 6.3.2. Esentially, translations of attributes (procedures TranslateAttributes of Fig. 6.10 and TranslateAttribute of Fig. 6.11) and operations (procedures TranslateOperations of Fig. 6.12 and TranslateOperation of Fig. 6.13) are performed first, followed by placement of information in the "externally visible representation" of the Z++ class. This preparation work for external representation is done by PlaceZPPAttributes and PlaceZPPOperations procedures (Fig. 6.16 and 6.17). Details on the processing of operations are shown in the procedures ProcessOPParameters (Fig. 6.14) and ProcessOpReturn (Fig. 6.15), which deal with the translation of the operation's parameters and, respectively, of the operation's return.

Since some of the relationships are implicitly processed during the formalisation of classes, only associations and aggregations/compositions receive special treatment, as indicated by the procedure TranslateRelationships (Fig. 6.18). Details on formalising aggregations and compositions are given in TranslateAggregation (Fig. 6.19), while the translation of association is described by TranslateAssociation (Fig. 6.20).

Further translation details are available from the code included in Appendix B.

```
-- UML to Z++ translation of a class diagram

procedure CDTranslate(CD:ClassDiagram; ZPPS:ZPPSpec)
begin
   TranslateClasses(CD;ZPPS);           -- process classes
   TranslateRelationships(CD;ZPPS)      -- process relationships
   ResolveVisibility(;ZPPS)             -- apply hiding operations on Z++ classes
end CDTranslate;
```

Fig. 6.7 The CDTranslate Procedure

```
-- Translation of classes

procedure TranslateClasses(CD:ClassDiagram; ZPPS:ZPPSpec)
begin
   for i = 0 to N-1 do                          -- inspect all classes in the class diagram
      if(CD.C[i].ctype /= bind) then            -- translate regular and parameterised
         TranslateClass(CD,CD.C[i];ZPPS)        -- classes only (ignore binding classes)
      endif;
   end for;

end TranslateClasses;
```

Fig. 6.8 The TranslateClasses Procedure

```
-- Translation of an individual class

procedure TranslateClass(CD:ClassDiagram,C:UMLClass; ZPPS:ZPPSpec)
   ZC:ZPPClass;                                 -- Z++ class to be created
begin
   AppendClass(C.name; ZPPC, ZC);               -- create corresponding Z++ class
   if (C.ctype==para)then                       -- if UML class is generic transfer formal
      TransferCParams(C; ZC)                    -- class parameters to Z++ class
   endif;
   ProcessParents(CD,C; ZC);                    -- process parents and fill EXTENDS clause
   TranslateAttributes(CD,C; ZPPS,ZC);          -- formalise attributes
   TranslateOperations(CD,C; ZPPS,ZC);          -- formalise operations
   PlaceAttributes(;ZC);                        -- fill FUNCTIONS, OWNS, and ACTIONS
   PlaceOperations(;ZC);                        -- fill FUNCTIONS, OWNS, and ACTIONS
end TranslateClass;                             -- work done on this class
```

Fig. 6.9 The TranslateClass Procedure

```
-- Translation of attributes

procedure TranslateAttributes(CD:ClassDiagram, C:UMLClass;
                              ZPPS:ZPPSpec, ZC: ZPPClass)

begin
   for i = 0 to Na-1 do                              -- inspect all attributes of the class
      TranslateAttribute(CD,CD.atts[i];ZPPS,ZC)      -- and save info in Z++ class
   end for;
end TranslateAttributes;
```

Fig. 6.10 The TranslateAttributes Procedure

```
-- Translation of an attribute

procedure TranslateAttribute(CD:ClassDiagram, att:UMLAtt;
                             ZPPS:ZPPSpec, ZC:ZPPClass)

   zatt: ZPPAtt;                                     -- Z++ attribute to be created
begin
   zatt.name = att.name;                             -- take name,
   zatt.visibility = att.visibility;                 -- visibility,
   zatt.initval = att.initval;                       -- and initial value from UML attribute
   if (att.property == changeable) then              -- determine place of attribute in Z++
       zatt.clause = OWNS                            -- class depending on property
   else
       zatt.clause = FUNCTIONS
   end if;
   if (zatt.visibility == public) then               -- make provisions for attribute visibility
       Append(zatt.name; ZC.Publics)
   else if (att.visibility == private) then
       Append(zatt.name; ZC.HiddenFeatures)
   end if;
   ProcessType(att.type,CD,ZC;ZPPS,zatt.ztype);-- determine type of Z++ att. and
                                                     -- possibly add to given sets of Z++ spec.
   Append(zatt;ZC);                                  -- finally, add attribute to Z++ class
end TranslateAttribute;
```

Fig. 6.11 The TranslateAttribute Procedure

```
-- Translation of operations

procedure TranslateOperations(CD:ClassDiagram, C:UMLClass;
                              ZPPS:ZPPSpec, ZC: ZPPClass)

begin
   for i = 0 to No-1 do                          -- inspect all operations of the class
      TranslateOperation(CD,CD.op[i]; ZPPS,ZC);  -- and save info in Z++ class
   end for;
end TranslateOperations;
```

Fig. 6.12 The TranslateOperations Procedure

```
-- Translation of an operation

procedure TranslateOperation(CD:ClassDiagram, op:UMLOp;
                             ZPPS:ZPPSpec, ZC: ZPPClass)
   zop: ZPPOp;                                   -- Z++ operation to be created
begin
   zop.name = op.name;                           -- take name and
   zop.visibility = op.visibility;               -- visibility from UML operation
   if (zop.visibility == public) then            -- make provisions for operation visibility
       Append(zop.name; ZC.Publics)              -- in Z++ context
   else if (zop.visibility == private) then
       Append(zop.name;ZC.HiddenFeatures)
   end if;
   if (op.property == query) then                -- determine place of operation signature
       zop.clause = RETURNS                      -- in Z++ class depending on property
   else
       zop.clause = OWNS
   end if;
   ProcessOPParameters(CD,op;ZPPS,ZC,zop);       -- process parameters of operation and
                                                 -- possibly add to given sets of Z++ spec

   ProcessOpReturn(CD,op;ZPPS,ZC,zop);           -- process operation return and
                                                 -- possibly add to given sets
   Append(zop;ZC);                               -- finally, add operation to Z++ class
end TranslateOperation;
```

Fig. 6.13 The TranslateOperation Procedure

```
-- Translation of parameters of operations

procedure ProcessOPParams(CD:ClassDiagram,op:UMLOp,ZC:ZPPCLass;
                          ZPPS:ZPPSpec,zop:ZPPOp)
   ztype:Ztype;                                    -- helper variables
   name,dir:String;
begin
   for i = 0 to Npo-1 do                           -- process all operation parameters
      name = op.p[i].name;                         -- take name and
      dir = op.p[i].name;                          -- direction of parameter
      ProcessType(op.p[i].ptype,CD,ZC; ZPPS,ztype);-- determine Z++ type and
                                                   -- possibly add to given sets of Z++ spec
      if (dir == in) then                          -- if direction of parameter is in
         Append (ztype;zop.sign.InputDomain);-- append type to input domain
         Append (name+"?"; zop.def.InputList)-- and decorated name to input list
      else if (dir == out) then                    -- if direction of parameter is out
         Append (ztype; zop.sign.OutputDomain);-- append type to output domain
         Append (name+"!"; zop.def.OutputList)-- and decorated name to input list
      else                                         -- otherwise, direction is inout
         Append (ztype;zop.sign.InputDomain);-- and therefore do both
         Append (name+"?"; zop.def.InputList);
         Append (ztype; zop.sign.OutputDomain);
         Append (name+"!"; zop.def.OutputList);
      end if;
   end for;
 end ProcessOpParams;
```

Fig. 6.14 The ProcessOpParams Procedure

```
-- Interpretation of operation return

procedure ProcessOPReturn(CD:ClassDiagram,op:UMLOp,ZC:ZPPCLass;
                          ZPPS:ZPPSpec,zop:ZPPOp)
   ztype:Ztype;
begin
   ProcessType(op.rettype,CD,ZC; ZPPS,ztype);-- determine Z++ type and
                                                   -- possibly add to given sets of Z++ spec
   if ((op.rettype /= boolean)&&                   -- if type neither boolean nor void
       (op.rettype /= void)) then
      Append (ztype;zop.sign.OutputDomain);   -- append to output domain
      Append ("result!";zop.def.OutputList); -- and append result param. to output list
   end if;
end ProcessOpReturn;
```

Fig. 6.15 The ProcessOpReturn Procedure

```
-- Placement of Z++ attribute descriptions in appropriate clauses
procedure PlaceZPPAttributes( ;ZC:ZPPCLass;)
   stmtA:String;            -- two statements needed per attribute, one for attribute definition
   stmtB:String;            -- the other for intialisation assignment (if an init value is provided)
   initop:ZPPOp;            -- a Z++ operation that may be needed for the initialisation of attributes
   axiomDef:String;         -- representation for the predicate part of a Z axiomatic definition
begin
   for i = 0 to Nza-1 do                          -- process all attributes
      AssembleZPPAttDef(ZC.att[i];stmA);          -- form att. def. from data in Z++ class
      if(ZC.att[i].clause == OWNS) then
         Append(stmA;ZC.OWNS)                     -- place attribute def.  in OWNS clause
         if(ZC.att[i].initval not null) then      -- if initial value exists
            AssembleZPPAttAssign(ZC.att[i]; stmB); --  form assignment statement
            if (initop not in ZC.ops) then
               AddInitOp(;ZC.ops)                 -- create init op. in Z++ class if needed
            endif;
            Append(stmB;ZC.initop.code)           -- and add initialisation assignment to it
         endif;
      else
         Append(addBar(stmA);ZC.FUNCTIONS);       -- place att. def.  in FUNCTIONS clause
         if(ZC.att[i].initval not null) then      -- and if initial value exists
            AssembleZPPAttAssign(ZC.att[i]; stmB); --  form assignment statement
            Append(stmB;axiomDef)                 -- and append it to pred. part of ax. def.
         endif;
      endif;
   endfor;
   Append(schemaPred; ZC.FUNCTIONS);              -- complete Z schema in FUNCTIONS
end ProcessOpReturn;
```

Fig. 6.16 The PlaceZPPAttributes Procedure

```
procedure PlaceZPPOperations ( ;ZC:ZPPCLass;) -- place op. descriptions in clauses
   stmt: String;                  -- statement that can be used for both signature and definition
begin
   for i = 0 to Nzo-1 do                          -- process all operations
      AssembleZPPOpDef(ZC.op[i];stmt);            -- form op. def. from data in Z++ class
      Append(stmt; ZC.ACTIONS);                   -- and place it  in ACTIONS clause
      AssembleZPPOpSign(ZC.op[i];stmt);           -- form op. signature
         if(ZC.op[i].clause == RETURNS) then
            Append(stmt;ZC.RETURNS)               -- and place it either in RETURNS clause
         else
            Append(stmt;ZC.OPERATIONS)            -- or in OPERATIONS clause
         endif;
   endfor;
end PlaceZPPOperations;
```

Fig. 6.17 The PlaceZPPOperations Procedure

```
-- Translation of relationships

procedure TranslateRelationships(CD:ClassDiagram; ZPPS:ZPPSpec)
begin
   for i = 0 to M-1 do                                  -- inspect all relationships
      if (IsAggreg(CD.R[i])or IsComp(CD.R[i])then       -- translate aggregs/comps
         TranslateAggregation(CD.R[i];ZPPS)
      else if (IsAssoc(CD.R[i])) then
         TranslateAssociation(CD.R[i];ZPPS)             -- and associations
      end if;
   end for;
end TranslateRelationships;                             -- gen. and instantiations are
                                                        -- processed during the
                                                        -- translation of classes
```

Fig. 6.18 The TranslateRelationships Procedure

```
-- Translation of aggregation and composition

procedureTranslateAggregation(rel:UMLRelationship; ZPPS:ZPPSpec)

   whole,part: String;          -- names of classes in aggreg/comp relationships
   mp: boolean;                 -- multiplicity of component (one/many as F/T)
   watt: ZPPAtt;                -- attribute to be added to container (by default
                                -- protected, without initial value, and with clause OWNS)
   cmp = "component";           -- constant string used in def. of attributes

begin
   getEndsDescription(rel;whole,part,mp);     -- get  info from relationship
                                              -- and then assign name and type to attribute
   if (!mp) then                              -- depending on the multiplicity of the part class
      Assign(cmp+part,part;watt)                      --  multiplicity of part one
   else
      Assign(cmp+part+"s","ℙ"+part;watt)              --  multiplicity of part many
   end if;
   addAttToZPPClass(watt,whole; ZPPS);        -- add attributes to container class
endTranslateAggregation;
```

Fig. 6.19 The TranslateAggregation Procedure

```
-- Translation of assocation

procedureTranslateAssociation(rel:UMLRelationship; ZPPS:ZPPSpec)

   one,two: String;                -- names of the two classes in association
   zatt: ZPPAtt;                   -- helper ZPP attribute to be added to Z++ classes
                                   -- (protected, without initial value, and with clause OWNS)
   line: String;                   -- local variable
   zcls: ZPPClass;                 -- Z++ class to be created
   dscr: =“Descriptor”;            -- constant strings used in the creation of the new class
   instOf: =“instancesOf”;
   inst: =“instances”;

begin

   zcls.name = rel.name + dscr;  -- the name of new class is derived from the name of assoc.
   getEndsDescription(rel; one, two);    -- get the names of the two classes in association
   formInvariantConstraint(one, two; line);   -- create predicate for INVARIANT clause
   Append(line; zcls.INVARIANT);              -- and append to new class
   AddClassToZPPSppec(zcls; ZPPS);            -- append class to Z++ spec.
   Assign(instOf + one, “ℙ” + one; zatt);
   AddZPPAttToClass(zatt, zcls.name; ZPPS);   -- add first attribute to the new class
   Assign(instOf + two, “ℙ” + two; zatt)
   AddZPPAttToClass(zatt, zcls.name; ZPPS)    -- add second attribute to the class
   FormInstancesType(one,two,ZPPS; stmt)
   Assign(rel.name + inst, line; zatt)
   addAttToZPPClass(zatt, zcls.name; ZPPS);   -- add third attribute
   updateSystemDescriptors(zcls; ZPPS);       -- update descriptors of associations

endTranslateAssociation;
```

Fig. 6.20 The TranslateAssociation Procedure

6.4 Formalisation of UML State Diagrams in Z++

The second part of formalisation is concerned with the translation of UML dynamic constructs to Z++. More precisely, this formalisation applies to UML state diagrams that are associated to individual classes, the result consisting in information appended to the Z++ classes created previously during the formalisation of the structural aspects of the system. As in the case of formalising class diagrams, the focus is on those parts of the translation process that can be automatically performed. The structure of the present section is similar to that of Section 6.3, but instead of a set of rules for syntactically correct state diagrams the expected format of states and transitions is given in a descriptive manner. Also, the Algorithm for Formalising State Diagrams (AFSD) is not presented at the same level of details as AFCD and it does not have an example of implementation included in the thesis' appendices (due to space considerations only the code for AFCD is provided in Appendix B). However, an example of formalising a state diagram is given in Section 6.4.

6.4.1 Constraints on the Contents of State Diagrams

In Subsection 3.3.2 the notions of event, finite state machine and statechart diagram were discussed and the description of states and transitions was given. Compared with that description, the AFCD uses a slightly different version of state machine, some elements being ignored while other are added. In Fig. 6.21 the general form of a transition is presented, showing the modelling elements used in state diagrams that are accepted by the formalisation algorithm (the structure of these elements is reflected in the format of the AFSD's input detailed in Subsection 6.4.3.1).

As can be seen from Fig. 6.21, the AFSD takes into consideration timed transitions, in the sense described in [Lano95], but internal transitions of states (which do not cause state changes) and deferred events (that could be handled by the object in different states) are not dealt with during the mechanised translation to Z++. Also, signal events are omitted but all other possible types of trigger events, namely call event, passage of time event, and change

event, are considered. A further simplification is that composite states are not covered, although their possible treatment is briefly discussed in Section 6.6.

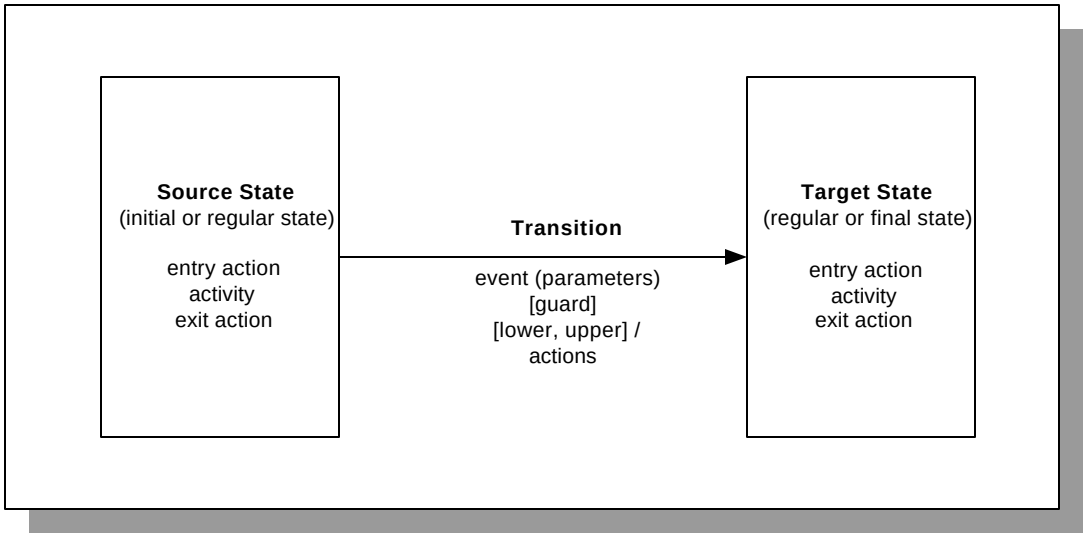

Fig. 6.21 General Form of a State Transition

A state diagram consists of a finite number of states and a finite number of transitions between states. Each state is of one of the following kinds: initial, final, or regular (we introduce the last term to denote a state that is neither initial nor final). Exactly one of the states is the initial state of the diagram, and zero or more final states can be included in the state diagram. Each regular state has a unique name within the state diagram and may contain an entry action, an activity, and an exit action. Initial and final states, which are in fact pseudostates, do not have names and do not contain actions or activities.

Each transition connects a source state to a target state and is either triggerless (automatic transition) or has a trigger event of the kind indicated below. A guard condition that can enable or disable the transition, an additional condition denoted initiation timing condition (expressed as an interval of time [lower, upper]), and a set of actions can optionally be attached to the transition. The source state and the target state of the transition may be the same, and each transition has only one trigger event. The same event, however, may serve as trigger for several transitions. The trigger event is of one of the following kinds: call event, denoted by a

name, passage of time event, specified in the form after (duration), or change event, given as when (condition). A call event may have a number of formal parameters, with types indicated. The guard condition is a Boolean expression that when evaluated as true enables the firing of the transition, provided the object is in the source state of the transition. When not indicated on the transition, the guard condition is assumed to be true. The timing limits lower and upper, if present, indicate the requirements for the transition's initiation time, more precisely after the transition is enabled its execution must be initiated no earlier than lower units of time and no later than upper units of time. The actions attached to the transition as well as the actions and activities included in states are specified as method invocations, using a name and optionally a list of formal parameters, with types indicated (as in the case of the call events, the requirement for explicit types of parameters is needed for automated translation purposes, although usually the types of parameters are not specified in state diagrams). Actions may represent invocations of operations from supplier classes, in which case the name of an object of the supplier class precedes the name of the action (the dot notation is used, for instance in the state diagram for class C an action a.op() denotes the invocation of method op of object a, where a is an object of C's supplier class A). Activities of states are assumed to be operations of the class for which the state diagram was drawn, so the dot notation need not be used (they are methods invoked on self).

Depending on the type of their trigger event, the transitions can be classified as externally invoked if the trigger is a call event or internally invoked if the trigger is a change or passage of time event, or the transition is triggerless. For formalisation purposes triggerless transitions are assimilated to transitions caused by "change events" when(true). Anonymous transitions with guarding condition guard are assimilated to transitions triggered by change events when(guard). Normally, when a change event when(condition) triggers a transition the guard component of the transition should be omitted (included in condition), although the AFSD processes it properly by appending the guard to the condition of the transition. In order to simplify the translation procedure, it is assumed that transitions from the initial state are triggerless, with no guarding condition, execution timing condition, or actions attached.

Also, it is assumed that the same call event appears throughout the entire state diagram with the same formal parameters, including names and types, as do actions and activities.

6.4.2 Translation Principles for State Diagrams

Before detailing the formalisation of the principal components of state diagrams, the states and the transitions, a number of preliminary observations on the approach taken for formalising state diagrams are necessary.

6.4.2.1 General Principles and Terminology

First of all, we need to recall that while a transition has a single trigger event an event may serve as trigger for several transitions (for the time being the point of view is sequential, meaning that at each occurrence time a trigger event triggers a single transition, but the transition it triggers may be different over the lifetime of the object). As pointed out by Kim and Carrington, who cite [Douglass98], each trigger event must have an associated event acceptor operation in the class for which the state diagram has been drawn [Kim00b]. Since an event may trigger more than one transition, this operation may in fact describe several transitions. Because it indicates the effects of the event in terms of transitions triggered and because of notational reasons that will become apparent in Subsection 6.4.2.3, we chose to use the term transit operations for these event acceptor operations.

However, using a single transit operation to cover all transitions possibly triggered by a certain event can be difficult to formalise mechanically, mainly because of the potential complexity of the timing constraints included in the `HISTORY` clause of the Z++ class. In our approach, we resort to the notion of transition signature for avoiding excessively long temporal formulae in the `HISTORY` clause, while keeping reasonably small the number of transit operations associated to a trigger event. The use of transition signatures, defined below, provides an intermediary solution between two opposite alternatives: the alternative of using a transit operation for each trigger event, which may lead to complex formulae, and

the alternative of using a transit operation for each transition, which may lead to a large number of operations included in the Z++ class.

By transition signature we denote the compound resulting from the concatenation of the following components associated to a transition, starting from the source state: the exit actions of the source state, the trigger event of the transition, the guard condition of the transition, the initiation timing constraint of the transition, and the actions attached to the transition (the parameters of events and actions are also part of the signature). In short, the signature of a transition includes all the components of the transition depicted in Fig. 6.1, prefixed by the exit action of the source state of the transition. This signature serves the purpose of identifying transitions that behave similarly but differ in the states they connect, transitions with identical signatures being described by the same transit operation. For example, in the state diagram of Fig. 3.12, reproduced in a simplified form in Fig. 6.29, there are three shared transition signatures, namely "when (limited_reached)/stop()," "goSpeedOne," and "off". (Fig. 6.29 is used in Subsection 6.4.4 for exemplifying the application of the AFSD).

The above definition of transition signature also hints to the fact that while we attach exit actions of states to outgoing transitions and include them in transit operations, the entry actions of the states are not formalised using transit operations. This is further explained in Subsection 6.4.2.2.

For formalisation purposes, a number of additional conventions are introduced, as follows:

- A transition triggered by a call event is said to be a simple transition if its signature consists exclusively of the name of the trigger event and, if provided, of the names and types of the parameters of the event (in other words, the source state of the transition has no exit action and the transition itself has no guard condition, no initiation timing condition, and no actions). The notion of simple transition describes a non-guarded asynchronous method call with no restrictions on initiation time and no appended

actions (examples of such simple transitions in Fig. 6.29 are reverseDirection, goSpeedOne, goSpeedTwo, and off);

- Since several transit operations may be created for the same trigger call event, a basic name for the transit operations associated with the call event is needed. The basic name is the name of the event, for instance if the call event is sendCharacter(c: char) the basic name for the transit operations will be sendCharacter, and if more than one transit operation will be created, they will be denoted $sendCharacter_1$, $sendCharacter_2$, etc. (an exception applies if one of the transit operations describes the simple transition associated with the event –in this case the name sendCharacter will be used for it, without an index appended). To distinguish between the operations that model the transitions and the event that triggers the transitions, in the Z++ specification the name of the event will be prefixed by ω, for instance the call event in the case described above will be denoted ωsendCharacter.

A note on the creation of Z++ operations describing transitions, actions, and activities is also necessary. During the formalisation of the state diagram, when such an operation is to be created, an operation with the same name may exist as the result of previously applying the AFCD. In this case, it is no longer necessary to create another operation, but an error message will be generated if the input and output domains, as well as the input and output lists of the existing operation do not match the ones that would be generated for the new operation.

6.4.2.2 Translation of States

The formalisation of states proceeds as follows:

- An enumerated type CState will be created in the TYPE clause of the Z++ class C corresponding to the UML class associated to the state diagram (e.g., a type DisplayState in the class Display). The elements of this type are the names (in lowercase) of the regular states included in the state diagram plus the names $final_K$, $K \geq 1$, generated incrementally for each final state present in the state diagram (final states are included here for the sake of

completeness, although they appear rarely in RTS). In addition, an attribute state of type CState denoting the current state of the object will be created in the OWNS clause. The state attribute, local to the class, will not be listed in the PUBLICS clause of the Z++ class;

- The name of the target state of the transition outgoing from the initial state will be used as initial value for the state attribute. The initialisation of state will be performed in the init operation of the Z++ class;
- The names of the regular states and the generated names of the final states will be used to construct predicates in the HISTORY clause of the Z++ class, along the lines proposed in [Lano95]. Specifically, the following categories of predicates will be generated: permission predicates, definition of transition effects, and reachability properties. Delay, duration, and other timing constraints will also be included in the HISTORY, and the names of the states will be used in these constraints as well, as detailed later in the description of translations of state actions, state activities, and transitions (the last category of HISTORY predicates, describing mutual exclusion properties, involves only the names of transitions). For the first three categories of predicates, the following apply:
  - the permission predicates relate transitions with their source states and will be given in the form

    $$(\underline{transit\_operation} \Rightarrow state = sourcestate_1 \vee \ldots \vee state = sourcestate_N);$$

  - the predicates describing the effect of transitions relate transitions with their target states and will be given as

    $$(\underline{transit\_operation} \Rightarrow \bigcirc(state = targetstate_1 \vee \ldots \vee state = targetstate_M));$$

  - the predicates for reachability indicate the relationships between source states and their outgoing transitions, and will be specified in the form

    $$(state = sourcestate \Rightarrow \underline{transit\_operation}_1 \vee \ldots \vee \underline{transit\_operation}_P)$$

  The names of regular and final states will be placed accordingly in the above predicates, as will be the names of transit operations created as detailed in Subsection 6.4.2.3.
- The entry action of each state, as well as the activity of the state will be formalised as local operations of the Z++ class, if not already declared otherwise in the class. The principles of translating UML operations described in Subsection 6.3.2.3 apply here as well, the names and the types of the parameters of the actions and activities being

processed in the same way the names and the parameters of operations are processed by the AFCD. A distinction occurs however if an entry action represents the invocation of a method on an instance of a supplier class. Since the class of this supplier object is not specified in the format of entry actions, no generation of operation will take place and no verification will be made to ensure that the method invoked actually exists, but a reminder in the generated Z++ specification will be included as a comment (e.g., // >> check invocation heater.raiseTemp(delta) is valid <<). This remainder will help the specifier to complete the formalisation of the state diagram after the AFSD is applied. If an operation with the same name already exists in the Z++ class as the result of previously applying the AFCD no action will be taken, the idea being that entry actions and activities may be operations already declared in the UML description of the class included in the class diagram provided as input to the AFCD. Temporal specifications on the entry action and the activity of the state will be appended in the `HISTORY` clause of the Z++ class as follows:

- if the entry action entry_action(params$_E$) exists in state S, where params$_E$ are the names of the action's parameters, then the predicate

$$\forall i \in \mathbb{N}_1 \bullet \uparrow (\text{entry\_action}(\text{params}_E), i) = \clubsuit((\text{state} = S) := \text{true}, i)$$

  will be added to indicate that the entry action initiates its executions as soon as the state is entered;

- if the entry action entry_action(params$_E$) is followed by an activity activity(params$_A$), where params$_A$ are the names of the activity's parameters, then temporal chaining between the two will be indicated as

$$\forall i \in \mathbb{N}_1 \bullet \downarrow (\text{entry\_action}(\text{params}_E), i) = \uparrow(\text{activity}(\text{params}_A), i)$$

  meaning that the termination of the entry action coincides with the initiation of the activity;

- if the state has only activity (params$_A$) but no entry action, the predicate

$$\forall i \in \mathbb{N}_1 \bullet \uparrow (\text{activity}(\text{params}_A), i) = \clubsuit((\text{state} = S) := \text{true}, i)$$

will be included to indicate that the state's activity commences its executions as soon as the state is entered.

For both the entry action and the activity the precondition state = S will be added to the definition of the operations that describe them;

- The exit actions of the states will be covered by transit operations created to formalise translations, as described in the next subsection.

6.4.2.3 Translation of Transitions

Each transition will be formalised using a transit operation declared in the OPERATIONS clause and defined in the ACTIONS clause of the in the Z++ class. As previously stated, a transit operation describes several transitions with the same signature. Differences exist between the formalisation of externally invoked transition (transition whose triggers are call events) and internally invoked transitions (transitions triggered by change or passage of time events), as follows:

- If the transition is triggered by a call event denoted call, then for formalisation purposes the basic name of the transit operation will be call and the event itself will be denoted ωcall. For each such transition:
  - an operation call with the signature included in the OPERATIONS clause and definition included in the ACTIONS clause of the Z++ class will be created using the information provided by the parameters of the event ωcall for defining the input and output domains of the operation's signature and the input and output lists of the operation's definition. The name of this operation will be included in the PUBLIC clause of the class;
  - if this is the only transition in the state diagram triggered by ωcall, or if all the transitions triggered by ωcall have the same signature, then the above is the only transit operation associated with ωcall. Information extracted from the transitions that have the same signature will be appended to the Z++ class as follows:
    - if the guard condition guard is specified then a predicate of the type

$$(\text{enabled}(\text{call}) \equiv (\text{state} = S_1 \vee \ldots \vee \text{state} = S_K) \wedge \text{guard})$$

will be included in the `HISTORY` clause of the Z++ class. In this predicate the states S_1, ... , S_K are the source states of the transitions that share the same signature. Since the well-formedness of the guard condition is not verified, a reminder for the human specifier to check the condition will be included as a comment, in the form // >> check condition [guard] is well-formed <<; The inclusion of this predicate in the `HISTORY` clause allows further specification by the human formaliser of detailed temporal constraints regarding the execution of transition, for instance in the case of a transit operation that corresponds to a single guarded transition it is possible to write

$$\begin{aligned}
&(\text{enabled}(\text{call}) \equiv (\text{state} = \text{sourcestate}) \wedge \text{guard}) \wedge \\
&\forall i \in \mathbb{N}_1 \bullet \exists j, j_1, j_2 \in \mathbb{N}_1 \bullet ((\text{state} = \text{sourcestate}) \wedge \text{guard}) \odot \clubsuit (\omega\text{call}, j) \wedge \\
&\clubsuit (\omega\text{call}, j) = \rightarrow (\text{call}, i) \wedge ((\text{state} = \text{sourcestate}) \wedge \text{guard}) \odot \uparrow (\text{call}, i) \wedge \\
&\downarrow(\text{call}, i) = \clubsuit ((\text{state} = \text{sourcestate}) := \text{false}, j_1) \wedge \\
&\downarrow(\text{call}, i) = \clubsuit ((\text{state} = \text{targetstate}) := \text{true}, j_2)
\end{aligned}$$

The above indicates the conditions under the operation call is enabled, shows that the enabling condition holds at the time of the j-th occurrence of the trigger event ωcall and that the operation is requested as soon the trigger event occurs. It also indicates that the enabling condition still holds at the initiation of the operation and details the change of state at the termination of the operation (the assumption is that sourcestate and targetstate are distinct, otherwise the last two lines should be omitted);

- if specified, the timing condition [lower, upper] will be used for including in the `HISTORY` clause the predicate

$$\forall i \in \mathbb{N}_1 \bullet \text{fires}(\text{call}, i) \Rightarrow \text{lower} \leq \text{delay}(\text{call}, i) \leq \text{upper}$$

which indicates that the execution of call initiates sometime between lower and upper units of time after the request for execution is made;

- in the definition of the operation call predicates relating the source state with the target state of all transitions covered by the operation will be included in the form

  $$(state = sourcestate_1 \wedge state' = targetstate_1) \vee \ldots \vee (state = sourcestate_K \wedge state' = targetstate_K)$$

  unless there is only one target state involved, in which case the inclusion of the predicate state' = targetstate will suffice (conditions on source states will be included in permission and reachability predicates);
- the state exit action and the actions attached to transitions are formalised as class operations declared in the `OPERATIONS` clause and defined in the `ACTIONS` clause of the Z++ class. These operations, which are local to the class, will have their invocations appended in sequence in the definition of the call operation (the order is the exit action first, followed by the actions attached to transitions in the order they are written on the transitions).

- if there are several distinct signatures for the transitions triggered by $\omega call$, then for each distinct signature a transit operations will be created in the `OPERATIONS` clause and defined in the `ACTIONS` clause of the Z++ class. These operations will be declared public. If one of the transition signatures is the signature of a simple transition, then the corresponding transit operation is the call operation created previously, the remaining operations being named $call_1$, $call_2$, etc. If there is no simple transition signature among the signatures of transitions triggered by $\omega call$, then the names of the operations will be $call_1$, $call_2$, etc.;
- for each transit operation $call_K$ ($K \geq 1$), information extracted from the transitions that have the same signature will be appended to Z++ class in the manner described above for processing guards, initiation timing constraints, and source and target states. However, the state exit action and the actions attached to the transitions are appended in the following order to the body of the transit operation: state exit action first, followed by the invocation of the simple operation call, and then by the actions attached on transitions, in the order they are specified on transitions. Operations for

state exit action and transition actions are created in the Z++ class in the way described previously;

- If the transition's trigger event is a change event when(condition) then the formalisation proceeds in a way similar to the one described for transitions triggered by call events, the difference being that no operation for the simple transition is created and that internal (spontaneous) transit operations with the name τ_k, $k \geq 1$, will be generated incrementally, one for each group of transitions that have the same signature. These internal operations are local to the class, therefore their names will not be included in the PUBLICS clause of the Z++ class. The condition of the event will be appended to the guard condition of the transitions, if any, and will be used in the above given formulae in the place of guard;

- If the transition's trigger event is a passage of time event after(time_expression) then the formalisation is similar to that of transitions triggered by change events, internal transit operations with the name τ_k being generated incrementally by the algorithm for each group of transitions that have the same signature. The only difference resides in the way the temporal condition is handled. For each such condition the predicate

$$\forall i \in \mathbb{N}_1 \bullet \text{enabled}(\tau_k) \wedge \uparrow(\tau_k, i) = \clubsuit((\text{state} = \text{sourcestate}) := \text{true}, i) + \text{time\_expression}$$

will be appended to the HISTORY clause of the Z++ class meaning that the operation is initiated after time_expression units of time from the moment the state is entered, provided the transition is enabled. This predicate need be checked by the human specifier, since no verification of the validity of the time expression is performed by the AFSD.

The translation of transitions continues until all trigger events present in the state diagram are processed, each trigger event leading to the creation of one or more transit operations. Then, all the transit operations created in the translation process will be used to generate mutex and self-mutex predicates, permission predicates, effect of transition predicates, and

reachability predicates, all included in the HISTORY clause of the Z++ class as indicated in Subsection 6.4.2.2. For the first category, it is assumed that transitions in UML state diagrams are both mutually exclusive and mutually self exclusive (see definition of these properties in Chapter 5), therefore the names of all transit operations will be included in both the mutex and self_mutex expressions appended to the HISTORY clause.

6.4.3 Algorithm for Formalising State Diagrams (AFSD)

In the same way the AFCD was described in Section 6.3, the AFSD is presented in this section through the structure of its input and output and through the pseudocode description of its executable contents. For separation of concerns purposes it is assumed that AFSD is invoked after AFCD, although they can be merged in an implementation, as discussed in Section 6.6. With this assumption, the Z++ class structure corresponding to the one developed in the UML space is already available, thus the AFSD only appends information to Z++ classes and is not concerned with the creation of classes.

6.4.3.1 AFSD Input

The input for the AFSD is provided by the Z++ specification resulted from the execution of the AFCD, specification given in the format presented in Subsection 6.3.3.2, and by a finite state diagram SD that consists of the tuple (S, T), where S is a set of states and T a set of transitions between states, $T : S \longleftrightarrow S$. In terms of the structure, the following are considered:

$$\begin{aligned} S &= \{S_0, .., S_{N-1}\}, N \geq 0 \\ T &= \{T_0, .., T_{M-1}\}, M \geq 0 \end{aligned} \tag{6.62}$$

Each state S in S has the following format:

$$S = (\text{name, kind, entry\_action, activity, exit\_action}) \tag{6.63}$$

where name is a string identifier (null if the state is not regular), and kind is one of the following: initial, regular, or final. The components entry_action and exit_action can be null, if not provided, or actions given in the form:

action = (name, params) (6.64)

while activity is either null (if not provided) or an action prefixed by the name of an object, which is a string identifier, possibly null:

activity = (objectname, action) (6.65)

In (6.64) params are given in the format indicated for operation parameters in (6.49) and (6.50).

Each transition T in *T* of (6.62) has the form:

T = (source, target, trigger, guard, time_range, actions) (6.66)

where source and target are states that belong to *S*, guard is a Boolean expression including the default value true, time_range is either null or given as an interval [lower .. upper] with lower and upper numerical values such that lower ≤ upper, and actions has the form:

$actions = \{action_0, \ldots, action_{Nact-1}\},\ Nact \geq 0$ (6.67)

with each action given in the format (6.65). The last component of a translation, the trigger event has the following form:

trigger = (kind, body) (6.68)

where kind is one of the following: none (used only for the transition from the initial state), call, change, or timing. If the kind of the trigger event is none, than its body is null, and if the kind of the trigger is call, then its body has the form:

body = (name, params) (6.69)

where name is a string identifier and params a list of parameters with the structure specified in (6.49) and (6.50). If the kind of the trigger is change, its body has the form:

body = (condition) (6.70)

where condition is a Boolean expression. If the kind of the trigger is timing, then its body has the form:

body = (duration) (6.71)

where duration is a timed-valued expression.

6.4.3.2 AFSD Output

The output of the AFSD is a Z++ specification having the structure described in Subsection 6.3.3.2. Under the assumption indicated at the beginning of Subsection 6.4.3, this output is generated by appending information to the Z++ specification provided as input to the AFSD.

6.4.3.3 AFSD Pseudocode

Using the convention (6.42) for the representation of procedures, the pseudocode description of AFCD is given in Figures 6.22 to 6.28. These figures show the higher level modules of the AFCD, designed according to the principles of translation outlined in Subsection 6.4.2. Since comments are included in procedures only some brief explanations are given below.

The SDTranslateProcedure of Fig. 6.22 coordinates the entire formalisation work. Its three major components are the TranslateStates, TranslateTransitions, and WriteHistoryPredicates procedures. The TranslateStates procedure shown in Fig 6.23 has two roles: the first of creating the enumerated type State and the attribute state of this type (with proper initialisation), and the second of coordinating the individual formalisation of states. Each state is processed individually by the TranslateState procedure (Fig. 6.24), which appends the name of the state to the members of the State type and formalises the entry action and the activity of the state, if available.

Transitions are processed based on their trigger event by the TranslateTransitions procedure (Fig. 6.25). Details on the formalisation of transitions triggered by call events are given in Fig. 6.26, which contains the pseudocode of the ProcessCallTrans procedure. Since call events are asynchronous method calls a simple transit operation is generated in any case for the event, based on the name and parameters of the call event. If there is a single transition signature for this event, it is assumed that the simple transit operation is the only such method needed by the developers of the state diagram, hence the additional work on the simple transit operation done by procedure CompleteUniqueTransitOperation (not detailed in the AFSD pseudocode). In fact, if there are no guards, time range, state exit action and transition actions in this single transition signature, the procedure does nothing else other than appending the simple transition operation created previously to the list of transit operations maintained by the state diagram. Since the processing of translations is driven by the trigger events present in the state diagram, it is necessary to mark as "processed" the transitions covered in each invocation of the ProcessCallTrans procedure.

If there are several transition signatures for the same call event, the GenerateTransitOperation is invoked for each such signature, as shown in Fig. 6.27. Formalisation work involving the processing of state exit action, of the guard condition, of the initialisation timing condition, and of the actions attached to transitions is performed here.

The last procedure shown for the AFSD, the WriteHistoryPredicates, appends to the `HISTORY` clause of the Z++ class a number of predicates, as indicated in Subsection 6.4.2.2.

```
-- UML to Z++ translation of a state diagram

procedure SDTranslate(SD:StateDiagram,zcls:String;ZPPS:ZPPSpec)
begin
   TranslateStates(SD,zcls;ZPPS);              -- process states
   TranslateTransitions(SD,zcls;ZPPC);         -- process transitions
   WriteHistoryPredicates(SD,zcls;ZPPC)        -- add predicates to the HISTORY clause
end SDTranslate;
```

Fig. 6.22 The SDTranslate Procedure

```
-- Translation of states

procedure TranslateStates(SD:StateDiagram,zcls:String;ZPPS:ZPPSpec)
   zppET: ZPPEnumType;                          -- enumerated type to be created
   state: ZPPAtt;                               -- and an attribute of this type
begin
   for i = 0 to N-1 do                          -- inspect all states in the state diagram
      TranslateState(SD,SD.S[i],zcls;ZPPS,zppET) -- translate each of them and
                                                -- create the enumerated State type
   end for;
   AddTypeToZPPClass(zppET,zcls;ZPPS);          -- add type to Z++ class
   Assign("state",zcls+"STATE";zatt);           -- create attribute state:ClassState
   AddZPPAttToClass(zatt,zcls;ZPPS);            -- add it to the class
   InitialiseStateAtt(SD,zcls;ZPPS);            -- and initialise the state attribute
end TranslateStates;
```

Fig. 6.23 The TranslateStates Procedure

```
-- Translation of an individual state

procedure TranslateState(SD:StateDiagram,S:State,zcls:String;
                         ZPPS:ZPPSpec,zppET:ZPPEnumType)
begin
   if (S.kind == final) then                    -- incrementally generate names of final
      AppendFinalState(;zppET,S.name)           -- states and append them to STATE type
   else if (S.kind = regular) then
      AppendState(;zppET);                      -- append name of reg. state to type
      if (S.entry_act /= null) then
         ProcessEntryAct(S,zcls;ZPPS);          -- formalise entry action
      if (S.activity /= null) then
         ProcessActivity(S,zcls;ZPPS);          -- formalise activity
   endif;
end TranslateState;
```

Fig. 6.24 The TranslateState Procedure

```
-- Translation of transitions

procedure TranslateTransitions(SD:StateDiagram,zcls:String;
                               ZPPS:ZPPSpec)
begin
   for i = 0 to M-1 do                                    -- inspect all transitions
      if (not Processed(T[i].trigger)) then               -- if not already processed
         if(T[i].kind == call) then                       -- process the transition
                                                          -- based on its trigger event:
            ProcessCallTrans(SD,T[i],zcls;ZPPS)           -- call event trigger,
         else if (T[i].kind == change) then
            ProcessChangeTrans(SD,T[i],zcls;ZPPS)         -- change event trigger, or
         else if (T[i].kind == timing) then
            ProcessTimingTrans(SD,T[i],zcls;ZPPS)         -- passage of time trigger
         end if;                                          -- (the transition from the
      end if;                                             -- initial state is not processed)
   end for;
end TranslateTransitions;
```

Fig. 6.25 The TranslateTransitions Procedure

```
-- Translation of transitions triggered by a call event

procedure ProcessCallTrans(SD:StateDiagram,T:Transition,
                           zcls:String;ZPPS:ZPPSpec)
   tsigns[]: TransSign;                                   -- holder for transition signatures
   postfixNo: int := 1;                                   -- number to be appended to op. names
begin
                                                          -- create simple operation  for this trigger
   GenerateSimpleTransitOperation(T.trigger,zcls;ZPPSpec);
   FormTransitionSignatures(SD,T.trigger;tsigns)          -- determine all trans. signatures
   if (tsigns.size == 1) then                             -- if one only, update simple op.
      CompleteUniqueTransitOperation(SD,tsigns[0],zcls;ZPPC)
   else                                                   -- otherwise generate a trans. op
      for i = 1 to tsigns.size do                         -- for each signature
         GenerateTransitOperation(SD,tsigns[i],zcls;ZPPC)
      end for;
   end if;
                                                          -- mark "processed" all transitions with this trigger
   MarkTransitionsProcessed (T.trigger;SD);
end ProcessCallTrans;
```

Fig. 6.26 The ProcessCallTrans Procedure

```
-- Creation of operations for a transition signature

procedure GenerateTransitOperation(SD:StateDiagram,tsign:TransSign,
                                   zcls:String;ZPPS:ZPPSpec)
   zop: ZPPOp;                                          -- transit op. to be created
begin
   if (isSimpleTransSignature(tsign)) then
      CompleteUniqueTransitOperation(SD,tsign,zcls;ZPPC)
   else
      SetName(T.trigger.name+getPostfix;zop);           -- assign postfix number
      ProcessExitAction(SD,tsign,zcls;zop,ZPPC);        -- process exit action
      ProcessGuard(SD,tsign,zcls,zop.name;ZPPC);        -- use guard for HISTORY
      ProcessTimeRange(SD,tsign,zcls,zop.name;ZPPC);-- use time range for HISTORY
      RelateStatesInOperation(SD,tsign;zop);            -- relate source and target in
                                                        -- operation body
      AppendActions(SD,tsign,zcls;zop,ZPPC);            -- create operations as needed
                                                        -- and append actions to op.
      AddOperation(zop,zcls;ZPPC);                      -- finally, attach op. to class
   end if;
end GenerateTransitOperation;
```

Fig. 6.27 The GenerateTransitOperation Procedure

```
-- UML to Z++ translation of a state diagram

procedure WriteHistoryPredicates(SD:StateDiagram,zcls:String;
                                 ZPPS:ZPPSpec)
begin
   WriteMutexSelfMutex(SD,zcls;ZPPS);   -- write mutex and self-mutex predicates,
   WritePermissions(SD,zcls;ZPPC);      -- permission predicates,
   WriteTransEffects(SD,zcls;ZPPC);     -- transition effects predicates,
   WriteReachability(SD,zcls;ZPPC)      -- and reachability predicates in HISTORY clause
end WriteHistoryPredicates;
```

Fig. 6.28 The WriteHistoryPredicates Procedure

6.4.4 Example of Formalising a State Diagram

In order to illustrate the proposed approach for formalising state diagrams the state diagram shown in Fig. 3.12 is reproduced here in a reduced form, stripped of annotations and with

shorter names for some of its states (Fig. 6.29). By applying the AFSD described in Subsection 6.4.3, the Z++ class presented in Fig. 6.30 is obtained.

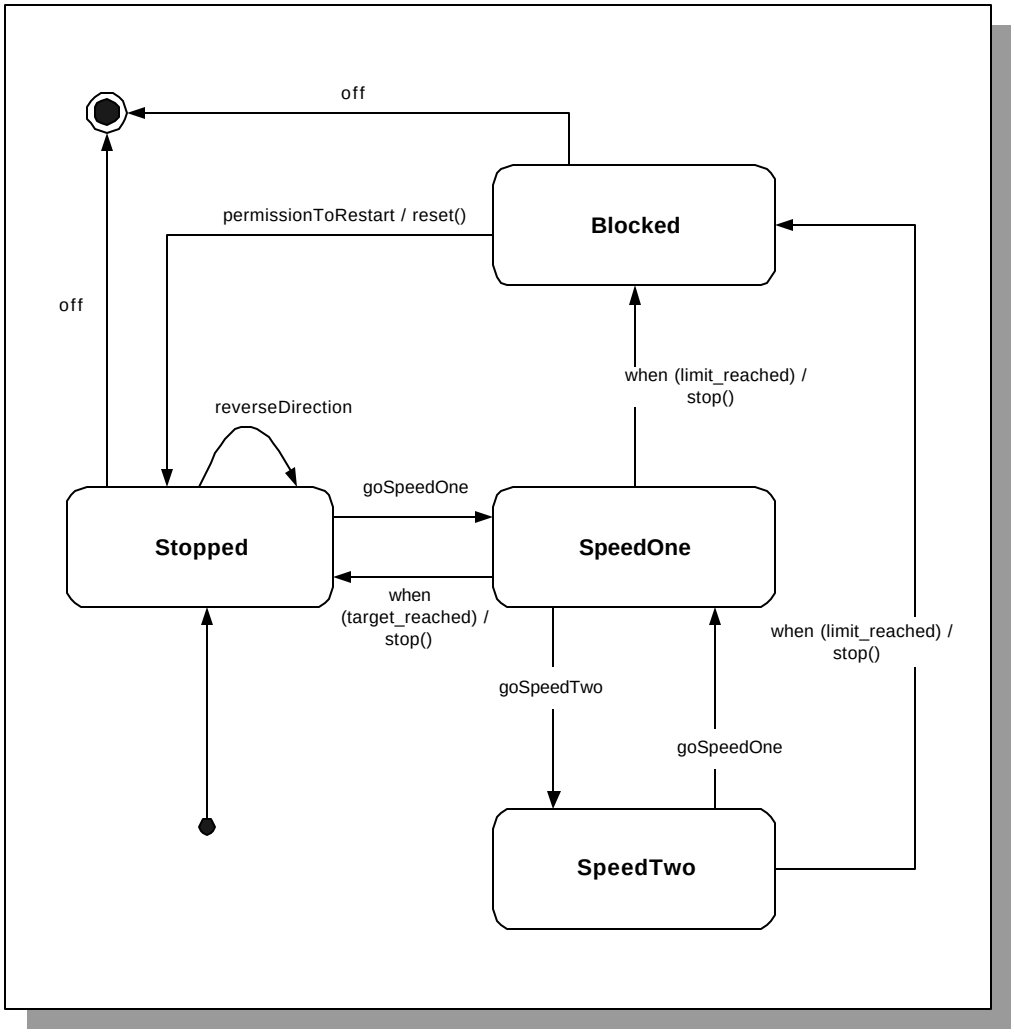

Fig. 6.29 DCMotor State Diagram from the ACTS

The notions of transition signature and transit operation can be easily related to the particular context of the DCMotor state diagram and of the DCMotor Z++ class obtained from it. To further describe the two notions, let us assume that another transition permissionToRestart, this time with two actions attached, stop() and reset(), is added to the state diagram, connecting the states SpeedTwo and Stopped (the latter being the target state of the transition).

```
CLASS  DCMotor EXTENDS Motor
PUBLICS

   permissionToRestart, reverseDirection, off, goSpeedOne, goSpeedTwo

TYPES

   DCMotorState ::= stopped | blocked | speedone | speedtwo | final

FUNCTIONS
OWNS

   state : DCMotorState

RETURNS
OPERATIONS

   permissionToRestart: → ;
   reverseDirection: → ;
   off: → ;
   goSpeedOne: → ;
   goSpeedTwo: → ;
  *t1: → ;
  *t2: → ;
   stop: → ;
   reset: →

INVARIANT
ACTIONS

   init ==> state' = stopped;
   permissionToRestart ==> reset;
                           state' = stopped;
   reverseDirection  ==>  state' = stopped;
   off ==> state' = final;
   goSpeedOne ==> state' = speedone;
   goSpeedTwo ==> state' = speedtwo;
  *t1 ==> stop;
          state' = blocked;
  *t2 ==> state' = stopped;
   stop ==> ;
   reset ==>

HISTORY

   // mutual exclusion properties

   mutex({permissionToRestart, reverseDirection, off, goSpeedOne,
          goSpeedTwo, t1, t2}) ∧
   self_mutex({permissionToRestart, reverseDirection, off, goSpeedOne,
          goSpeedTwo, t1, t2}) ∧
```

Fig. 6.30 Z++ Class DCMotor Generated by the AFSD (continued on next page)

```
    // permission predicates

        (permissionToRestart ⇒ state = blocked) ∧
        (reverseDirection ⇒ state = stopped) ∧
        (off ⇒ state = blocked ∨ state = stopped) ∧
        (goSpeedOne ⇒ state = stopped ∨ state = speedtwo) ∧
        (goSpeedTwo ⇒ state = speedone) ∧
        (t1 ⇒ state = speedone ∨ state = speedtwo) ∧
        (t2 ⇒ state = speedone) ∧

    // definition of transition effects

        (init ⇒ ○(state = stopped)) ∧
        (permissionToRestart ⇒ ○(state = stopped)) ∧
        (reverseDirection ⇒ ○(state = stopped)) ∧
        (off ⇒ ○(state = final)) ∧
        (goSpeedOne ⇒ ○(state = speedone)) ∧
        (goSpeedTwo ⇒ ○(state = speedtwo)) ∧
        (t1 ⇒ ○(state = blocked)) ∧
        (t2 ⇒ ○(state = stopped)) ∧

    // reachability properties

        (state = stopped ⇒ reverseDirection ∨ off ∨ goSpeedOne) ∧
        (state = blocked ⇒ permissionToRestart ∨ off) ∧
        (state = speed_one ⇒ goSpeedTwo ∨ t1 ∨ t2) ∧
        (state = speed_two ⇒ goSpeedOne ∨ t1) ∧

    // delay, duration, and other constraints

       (enabled(t1) ≡ (state = speedone ∨ state = speedtwo)
                       ∧ limit_reached) ∧

                    // >> check [limit_reached] is well-formed <<

       (enabled(t2) ≡ (state = speed_one) ∧ target_reached)

                    // >> check [target_reached] is well-formed <<

END CLASS
```

Fig. 6.30 Z++ Class DCMotor Generated by the AFSD (continued from the previous page)

In this situation two distinct transition signatures would exist for the transitions triggered by the call event permissionToRestart. In terms of operations, the permissionToRestart would still be generated as an operation (corresponding to a “simple” transition), but it would not be in fact a transit operation. Thus, it would no longer be included in HISTORY predicates, and its

body would be empty. The two distinct transition signatures would have associated two transit operations, permissionToRestart$_1$ and permissionToRestart$_2$, which would be used to describe state changes. In their bodies, an invocation to permissionToRestart would be included before the invocation of their specific actions. During the enhancement of the Z++ specification, the human formaliser could decide whether these three operations can be replaced by a single (but more complex) operation.

6.5 Deformalisation: From Z++ Specifications to UML Representations

As discussed in Section 6.2, the reverse mapping, from Z++ to UML, can be useful in certain situations. As in the case of formalisation, this "reverse" translation can be partially mechanised, but it should be noted that relevant information included in the Z++ specification can be lost (in particular, various types, constraints, and bodies of operations). In this section a number of guiding principles for deformalisation are suggested and the outline of an Algorithm for Deformalisation (ADF) is presented.

6.5.1 Principles of Deformalisation

In the following, it is considered that a Z++ specification with the structure given in Section 6.3.3.2 is available, based on which a class diagram together with a set of state diagrams associated to individual classes can be obtained. For the ADF the structure of the output class diagram is the one given in Subsection 6.3.3.1, while the state diagrams are represented as described in Subsection 6.4.3.1.

6.5.1.1 Assigning Types for UML Attributes, Parameters of Operations, and Operation Returns

Due to the specifics of Z++, not all attributes, parameters of operations, and returns of operations present in the Z++ specification will have their types translated to UML. Only

attributes specified as att:typespec in Z++, with typespec detailed as below, and only parameters of operations and returns of operations that correspond to input or output operation domains specified as typespec will have their types mapped to UML. The format of typespec that allows an automated translation of type to UML is one of the following:

(a) T ("scalar form"), where T is the name of a given set, or of an enumerated type, or of a regular Z++ class, or of a predefined Z type (ℕ, ℤ, or ℝ). If T is ℕ the corresponding UML type will be unsigned int, if T is ℤ the type in UML will be int, if T is ℝ the type in UML will be real, and in all other cases the type in UML will be T;

(b) seq(T), ℙT, or 𝔽T ("array form"), with T given as in (a) above. In this case, if T is ℕ the type used in UML will be unsigned int[ ], if T is ℤ the UML type will be int[ ], if T is ℝ the type will be real[ ], and in all other cases the corresponding UML type will be T [ ];

(c) T[params] ("generic form"), where T is the name of a generic class included in the Z++ specification and params a list of names denoting actual parameters whose types are assumed to be of form (a) (parameters of generic classes may not be arrays or instances of generic or binding classes). In this case, the translated type in UML will be T[params].

In practical terms, the above restrictions on typespec signify that more complex Z++ specifications of types (e.g., involving functions, relations, or Cartesian products) are not mapped automatically to UML.

6.5.1.2 Generating Attributes for UML Classes

The following apply for obtaining the attributes of a UML class C, whose correspondent Z++ class is C (for easier referencing the latter will be denoted ZC in the following):

- Each attribute att included in the OWNS clause of the ZC class will have a corresponding attribute att in the C class, provided that the type of the attribute is not a class type (attributes of class type will lead to the creation of associations and aggregations, as shown in Subsection 6.5.1.5). The property of this attribute will be changeable, the type of the attribute will be assigned according to the principles presented in Subsection 6.5.1.1

for the translation of types, and the visibility of the attribute will be public if att is included in the clause PUBLICS of class ZC, private if it is used in the hiding operation defining the Z++ class H_C, and protected otherwise. The initial value initval will be given to the attribute in the C class if an assignment statement att = initval exists in the init operation of class ZC;

- From the FUNCTIONS clause of ZC, each attribute att will be extracted and included in the UML class C if the definition att:typespec is present in a axiomatic definition included in the clause. The property of this attribute will be frozen, the type of the attribute will be assigned according to the principles for translating types presented in Subsection 6.5.1.1, and the visibility of the attribute will be private if the name of the attribute is used in the hiding operation defining the Z++ class H_C, and protected otherwise (attributes declared in the FUNCTIONS clause cannot be public). The initial value initval will be given to the attribute in the C class if a statement att = initval exists in the predicate part of the axiomatic definition of the FUNCTIONS clause.

6.5.1.3 Generating Operations for UML Classes

The following apply for obtaining the operations of a UML class C whose correspondent Z++ class is ZC:

- Internal operations of class ZC (operation prefixed by the symbol *) and the init operation of the class will not be translated to UML;
- All other operations of ZC will be treated as follows:
  - The name of the operation in ZC will be used as the name of the corresponding operation in C;
  - The visibility of the operation will be public if the name of the operation is included in the PUBLICS clause of ZC, private if the name appears in the hiding operation defining the class H_C, and protected otherwise;
  - The property of the operation will be query if the operation is declared in the RETURNS clause of ZC and none if it is declared in the OPERATIONS clause;

- The return type of the operation will be assigned according to the principles described in Subsection 6.5.1.1, based on the output domain of the operation specified in either the RETURNS or the OPERATIONS clause of the ZC class;
- The parameters of the operation in class C will receive the names used in the definition of the operation included in the ACTIONS clause of ZC. For each parameter, the direction of the parameter will be in if the name of the parameter is decorated with the symbol ?, out if it is decorated with the symbol !, and inout if the parameter appears in both the input and the output lists of the operation. The type of each operation parameter will be assigned as described in Subsection 6.5.1.1, based on the input and output domains of the operation, which are listed in either the RETURNS or the OPERATIONS clause of ZC;
- The precondition of the operation as well as the body of the operation will not be translated to UML. However, assignment statements included in the init operation will be used for assigning initial values to attributes in UML, and predicates involving the state attribute, if available, will be inspected when generating state diagrams.

6.5.1.4 Generating UML Classes

The following apply for obtaining UML classes from a Z++ specification:

- Each class C in Z++ that is not a descriptor of an association (associtaion descriptor classes were introduced in Subsection 6.3.2.5) will have a correspondent class C in UML. If the Z++ class C has an associated hiding class H_C in Z++, the list of hidden features used in the hiding operation that defines H_C will be employed to assign the visibility private to the corresponding features (attributes and operations) of the UML class C, as described in Subsections 6.5.1.2 and 6.5.1.3;
- Each generic class G in Z++ will be translated to generic class G in UML, the names of the formal class parameters of the Z++ class G being used as names for the formal class parameters of the UML class G;

- A binding UML class G[actual_params] will be created whenever a type G[actual_params] is encountered in the Z++ specification, with G matching the name of an existing generic class G in Z++ and the number of actual parameters actual_params equal to the number of the formal parameters of the Z++ class G (however, the names of the actual_params should not be the same with the names formal_params of the generic class). If not already present, a binding relationship between the binding class and the generic class will be drawn in the class diagram, with the names of the actual parameters used to differentiate the binding class from other possible classes that instantiate the same generic class (see also Subsection 6.5.1.5 on generating relationships);
- The attributes and the operations of each regular or parameterised UML class will be obtained as indicated in Subsections 6.5.1.2 and 6.5.1.3, based on the inspection of the corresponding Z++ class.

6.5.1.5 Generating Relationships

Relationships will be generated in UML class diagrams as follows:

- Generalisation relationships will be obtained based on the information included in the EXTENDS clause of Z++ classes. For each class P (parent) included in the EXTENDS clause of the Z++ class C (child) a generalisation relationship between P and C will be created in the class diagram. If the EXTENDS clause of C includes a hiding class H_P, the relationship in the class diagram will be nevertheless between P and C;
- Instantiation relationships will be obtained based on the attributes of generic type G[actual_params], where G is the name of a generic Z++ class. A binding class G[actual_params] will be created for each different set of actual parameters actual_params encountered for G, and a instantiation relationship between this class and the generic UML class G will be included in the class diagram;
- Associations will be obtained in two ways:
  (a) From association descriptor classes that exist in the Z++ specification (their description was given in Subsection 6.3.2.5). For each such descriptor class an

association relationship will be created in the class diagram between the classes A and B included in the definition of instancesOf attributes of the association descriptor class;

(b) From attributes of the type D, seq(D), ℙD, or 𝔽D where D is the name of a Z++ class. For each such attribute encountered in a Z++ class C an association relationship between UML classes C and D will be created in the class diagram. The attribute may indicate in fact an aggregation or a composition relationship, but the human formaliser will be required to change the type of the relationship if necessary;

- Aggregations and compositions will not be generated automatically by the ADF but, as mentioned above, some of the association relationships produced by the ADF may in fact be aggregations or compositions. It will be left to the human specifier to make the necessary changes.

6.5.1.6 Generating State Diagrams

State diagrams will be created by the ADF only for those Z++ classes C that have an enumerated CState (or State) type defined in their TYPE clause and an attribute state of this type declared in their OWNS clause. For each such Z++ class a state diagram "C's State Diagram" will be generated as follows:

- The names of the enumerated type State's members will be used as names of the states created in the state diagram (however, final states, which will be created as well, will not receive names);
- If an initialisation assignment state = entrystate exists in the init operation of the Z++ class, an initial state will be created and an anonymous, non guarded and actionless transition from the initial state to entrystate will be created;
- Based on the predicates included in the HISTORY clause of the Z++ class and on the predicates included in the transit operations of the class (specifically, predicates that relate source states with target states) transitions will be created in the state diagram. For each transition, the name of the transit operation that describes the transition in class C will be attached to the transition in the state diagram.

6.5.2 Outline of the Algorithm for Deformalisation (ADF)

Based on the principles proposed in Subsection 6.5.1 for the generation, starting from a Z++ specification, of a UML model consisting of a class diagram and of a set of state diagrams associated to classes, an outline for a deformalisation algorithm is presented in Fig. 6.31 to 6.33. This outline describes the ADF only in terms of its high level components, but it covers nevertheless all the significant aspects of the Z++ to UML translation process.

As a matter of general approach, the mapping of the Z++ specification to a UML model can be tackled in (at least) two ways. One alternative is to design the algorithm in a manner that allows the successive generation of the major modelling elements of the UML space, namely the classes, the relationships, and the state diagrams. This approach would require however a triple processing of the individual Z++ classes, the first for creating the UML class structure that mirrors the one present in the formal specification, the second for generating the relationships between classes, and the third for creating state diagrams for those classes in which state changes are explicitly described in Z++ via a *state* attribute. While this approach allows a better separation of concerns, an incremental development of the UML model in terms of major kinds of artefacts, and a less complex structure of the algorithm, it is however less efficient in terms of implementation.

Since this alternative involves a repeated treatment of each Z++ class and we envisage the possibility of applying the deformalisation process on an individual class or a group of selected classes, we have opted for a second approach, that of generating all types of UML elements –classes, relationships, and state diagrams– through a single inspection (processing loop) of the Z++ classes, each class being mapped to UML elements based on the information contained in its definition and on the information provided by the context of the Z++ specification. While this approach allows the complete treatment of an individual Z++ class in a single processing step, it has the disadvantage that the generation of some UML elements is "buried" in modules whose primary purpose is different, more precisely binding classes and association relationships are created, if necessary, during the processing of

attributes (this is nevertheless in agreement with the translation principles described in Subsection 6.5.1.5).

The approach we have taken is apparent in the top-level ADF procedure, presented in Fig. 6.31.

```
-- Z++ to UML translation

procedure ADF(ZPPS:ZPPSpec;CD:ClassDiagram,SDS:StateDiagrams)

begin
  for i = 0 to Nz-1 do                          -- process all Z++ classes
     TranslateZPPClass(ZPPS,ZPPS.ZC[i];CD,SDS);
  end for;
  PrintClassDiagram(CD);                        -- show/save results: class diagram
  PrintStateDiagrams(SDS);                      -- and state diagrams
end ADF;
```

Fig. 6.31 The ADF Procedure

The particular treatment of a Z++ class is handled by the TranslateZPPClass procedure, which coordinates the generation of the UML class, the processing of generalisations, and, if appropriate, the generation of the state diagram associated with the class (Fig. 6.32). The last procedure shown for the ADF, GenerateUMLClass, describes the work needed for the completion of the UML class (Fig. 6.33). It is here, in the procedures called by GenerateUMLClass, where the possible generation of associations and binding classes can take place, while dealing with the types of attributes (processing the types of parameters of operations and of operation returns may also prompt the creation of binding classes).

Nevertheless, as shown in Chapter 9, this organisation of the ADF suits better our modelling purposes. In fact, the closely related generation of the UML class and of the state diagram associated with the class in the TranslateZPPClass procedure forecasts the combined use of the regular UML class specification and of the state diagram associated with the class in the integrated modelling approach proposed in Chapter 7.

```
-- Translate individual Z++ class to UML

procedure TranslateZPPClass(ZPPS:ZPPSpec,ZC:ZPPClass;
                            CD:ClassDiagram,SDS:StateDiagrams)

begin
   if (isAssocDescriptor(ZC)) then            -- if the class describes an association
      GenerateAssociation(ZPPS,ZC;CD)         -- simply add association to class diagram;
   else                                       -- otherwise
      GenerateUMLClass(ZPPS,ZC;CD);           -- generate the corresponding UML class
                                              -- (in the process, create associations
                                              --  and binding classes, if detected)
      ProcessGeneralisations(ZPPS,ZC;CD)      -- process list of ancestors and
                                              -- update relationships in class diagram
      if (hasStateAtt(ZC)) then               -- if there is a 'state' attribute in the Z++
         GenerateStateDiagram(ZC;SDS)         -- create state diagram and add to
      end if;                                 -- the collection of state diagrams
   end if;
end TranslateZPPClass;
```

Fig. 6.32 The TranslateZPPClass Procedure

```
-- Generate UML Class from Z++ class ; in the process, generate associations and binding classes from type information
-- contained in the definition of attributes

procedure GenerateUMLClass(ZPPS:ZPPSpec,ZC:ZPPClass;
                           CD:ClassDiagram)

   C:UMLClass;                                -- UML class to be completed
begin
   SetNameAndType(ZC;C);                      -- name the class and establish its
                                              -- type (regular or parameterised)
   if (C.ctype == para) then
      SetClassParameters(ZC;C);               -- if generic, provide parameters
   end if;
   GenerateAttributes(ZPPS,ZC;C,CD);          -- attach attributes
   GenerateOperations(ZPPS,ZC;C,CD);          -- attach operations
   AppendClassToClassDiagram(C;CD);           -- then append class to the class diagram
end GenerateUMLClass;
```

Fig. 6.33 The GenerateUMLClass Procedure

6.6 Notes on the Application of Formalisation and Deformalisation Algorithms

At the conclusion of this chapter, several notes regarding the application of the three proposed algorithms for formalisation and deformalisation are necessary.

First of all, while the focus in this chapter was on those aspects of translations between UML and Z++ that can be automated, it is necessary to mention that the proposed algorithms are intended only to serve as aids during the modelling process, and in no way to substitute the human developer. In fact, we cannot stress enough the importance of the human factor in the process of formalisation (and, generally, in the development process), the quality of the software product depending essentially on the skills of its developers. Also, as shown in the next chapter, while we assign a prominent role in the modelling process to the activities of formalisation and deformalisation, the emphasis is not on automated translations between UML and Z++, but on the combined, efficient use of the two notations.

In practical terms, the three algorithms need be further refined in several aspects. In particular, in conjunction with the integrated specification environment described in Chapter 9, an environment whose design incorporates the mechanics of translation presented in this chapter, the following issues need be tackled (we suggest below solutions for each of them):

- While the AFCD applies to class diagrams, for practical purposes it is necessary to allow the formalisation of a single class or of a selected group of classes. The solution for this is to allow the AFCD to continue to operate within the context of the class diagram and to visually mark in the generated Z++ specification the references made from within the group of formalised classes to classes outside this group (e.g., by including a comment listing the names of referenced but not formalised classes). This would allow the developer to decide if additional classes need be formalised;

- Also regarding the AFCD, its application to two or more related class diagrams need be considered. This is not so much an issue of the algorithm itself as it is an issue of combining and representing the related class diagrams in the environment that uses the AFCD. The problem resides in classes included in one diagram that are in relationships with classes from another class diagram. The suggested solution is to attach a description to the class (similar to a property sheet) indicating the relationships in which the class is involved, irrespective of the class diagram;
- Although not a major issue, the combined use of the AFCD and of the AFSD can also be improved. At this point in time, AFCD is applied first, followed by the AFSD, the latter algorithm only appending information in a Z++ class created by the former. The AFSD can be extended without difficulty to create itself the target Z++ class and, more generally, the work of both algorithms can be integrated in a single formalisation algorithm. Since the same translation principles apply and the data structures used by the algorithms is already in place this integration should be straightforward;
- Regarding the AFSD, its extension to composite and concurrent states is a topic that deserves investigation. The first thing in such extension is to create an enumerated type for each composite state in the state diagram, with an attribute of this type describing the current local state. Then, more complex descriptions of transitions are necessary. Parallel executions can be expressed via the || operator available in RTL;
- Finally, the combined use of the three algorithms, the AFCD, the AFSD, and the ADF is to be considered in an integrated environment (see Chapter 9). The main issue is the "update problem," which arises when a model is switched back and forth between the two spaces, UML and Z++. The solution, similar to the one used in version control systems, is to let the developer decide on committing the changes. To help his or her decision, things to be added can be marked in a specific way (e.g., with indicators such as ">>>>>," meaning "in," or new information) and things to be removed in a different way (e.g., with "<<<<<," meaning "out," or information to be discarded).

6.7 Chapter Summary

In this chapter translations between structural and dynamic UML model elements and Z++ specifications have been discussed. The focus has been on the formalisation process, which has the role of generating formal specifications from UML class diagrams and state diagrams but the auxiliary reverse process, denoted deformalisation, has also been considered. Detailed principles and algorithms have been presented for the automated UML to Z++ translation and guidelines for the reverse translation have been proposed. In Chapter 7 the activities of formalisation and deformalisation are included in a larger procedural frame that is aimed at guiding the development of the integrated UML/Z++ model of TCS and in Chapter 8 the application of the formalisation algorithms are illustrated through an Elevator Controller case study.