Software Engineering Institute

Formal Development of ADA Programs Using Z and Anna: A Case Study

Patrick R. Place
William G. Wood

February 1991

TECHNICAL REPORT
CMU/SEI-91-TR-001

http://www.sei.cmu.edu

Carnegie Mellon

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER N-3097-AF	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Interdiction and Conventional Strategy: Prevailing Perceptions		5. TYPE OF REPORT & PERIOD COVERED interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Ian O. Lesser		8. CONTRACT OR GRANT NUMBER(s) F49620-86-C-0008
9. PERFORMING ORGANIZATION NAME AND ADDRESS The RAND Corporation 1700 Main Street Santa Monica, CA 90406		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Long Range Planning & Doctrine Div, (AF/XOXFP) Directorate of Plans, Ofc. DC/Plans & Operations Hq USAF Washington, DC		12. REPORT DATE June 1990
		13. NUMBER OF PAGES 53
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

No Restrictions

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Interdiction
Military Strategy
Conventional Warfare

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

see reverse side

DD FORM 1 JAN 73 1473

Perceptions about interdiction's role,
effects, and relationship to conventional
war continue to be shaped largely by images
drawn from the Allied experience in Europe
during World War II, but these are
increasingly remote from the current and
prospective environment. Destruction,
delay and disruption, diversion, and
demoralization do not have uniform
prospects for success. The effects of
interdiction are likely to be interactive,
divisible, and in some instances,
intangible. Broader strategic factors,
including war duration, intensity, and
phases, will shape the opportunities for
interdiction. A war of high intensity and
long duration will favor a strategy of
interdiction. An environment characterized
by smaller conventional forces on the one
hand and unconstrained surface-to-air
defenses on the other is likely to make the
interdiction mission at once more important
and more difficult.

Formal Development of Ada Programs Using Z and Anna:
A Case Study

Patrick R. H. Place
William G. Wood

Application of Formal Methods Project

David C. Luckham
Walter Mann
Sriram Sankar

Computing Systems Laboratory
Stanford University
Stanford, CA 94305

This technical report was prepared for the

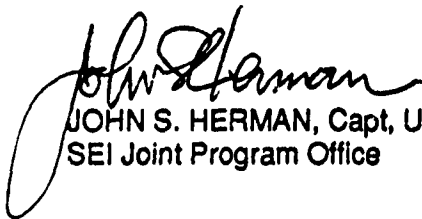SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official
DoD position. It is published in the interest of scientific and technical
information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

JOHN S. HERMAN, Capt, USAF
SEI Joint Program Office

Contents

Chapter 1

Introduction

Abstract: This report describes a method for the formal development of Ada programs from a formal specification written in Z. ANNotated Ada (Anna) is used as an intermediate language linking the more abstract Z specifications to the concrete Ada program. The method relies on the notion that successive small transformations of a specification are easier to verify than a few large transformations. Essentially the method uses three notations for the representation of the system: an implementation-independent notation for the specification of the system, an implementation-dependent notation for the representation of a lower level specification of the system, and the implementation language. Z and Anna are outlined briefly and examples of transformations are presented. A simple Z specification has been chosen and the transformations presented in the report are transformations of the Z specification into Anna. Conclusions are drawn about the development method presented.

This report describes recent work performed by the formal specifications project of the Software Engineering Institute in conjunction with members of the Anna project at Stanford University. Our work involved initial investigations into the development of software from a formal specification. This work resulted in a practical method for formal software development with some rules for transforming Z specifications into Anna specifications.

1.1 Motivation

Both safety-critical and security-critical systems require the greatest possible care in software development. In both cases, it is imperative that the developers exactly meet the intent of the system requirements. Thus, the system requirements must be stated as precisely and unambiguously as possible. The current best known approach to the precise, unambiguous statement of requirements is to use mathematics to create a formal specification of the requirements. The customers must then agree that the formal specification satisfies their requirements.

Having created a precise, unambiguous specification of the desired system, a design and implementation of that system must be developed. The traditional approach has been to develop the system with design and code reviews and then to enter a testing phase to ensure that the developed code exactly meets the specification. This traditional approach is very costly and

---

prone to errors being introduced at each phase of the development which are only discovered during the testing phase. The approach presented in this report should reduce the development expense by introducing less development errors than the traditional approach. Our approach is to transform the specification from an abstract statement of the requirements into the code by a sequence of transformation steps. The original specification needs to be complete, in that it describes all of the function of the system, since no new function should be added by the development process. Each step in the sequence may be shown (by formal proof if necessary) to meet the requirements stated in the previous step. Thus the developer is assured (again by proof if necessary, though testing may be preferable) that the resulting code meets the original specification and that no safety or security flaws have been introduced in the development process. Because each step in the sequence is a transformation of the previous step and because of the need to ensure that no safety or security flaws have been introduced, the original requirement must state not only what the system should do, but also what it must *not* do. This ensures that no additional functionality is introduced in the development process. Such requirements may be thought of as safety requirements for the system. Although our specifications do not include such safety requirements, due to our use of an existing example specification; however, the method is appropriate for these types of requirements, since they are expressed as predicates in the same way that the requirements on what the system should do are expressed.

This report describes a method for developing code from a formal specification that reduces (eliminates in the case of full formality) errors introduced in the process of developing code that conforms to the specification. The method is general and may be used for any appropriate specification and programming languages (although at present linguistic and machine support for our method is limited to the languages chosen). The general approach of the method is to transform the initial specification into a form that is programming language-specific. The programming language-specific specification is subsequently transformed into a program in the appropriate language. Because the resulting code meets the original specification, it is important that the original specification has been carefully analyzed for unexpected function (such as a security loophole) and that any undesired function has been eliminated before development takes place.

Z was selected as our language-independent notation because of its applicability to the specification of software systems. The structuring capabilities of the Z schema language were also considered advantageous in a specification language. Ada was chosen because it is a required language for Department of Defense contractors and because it has the language-specific specification language, Anna. The Anna language and associated tool set were chosen because the language is based on the same underlying mathematical model as Z and the tool set performs the necessary function of inserting checks into runtime Ada code based upon the Anna specifications.

1.2 Background

The notion of using a language-specific specification notation is not new. Larch [2] is based on the notion of using a language-independent notation for the specification of a system and a language-specific notation for lower level details not captured by the language-independent specification. Larch is based on equational logic while our approach is based on predicate logic and set theory.

Further, the Larch language dependent components have been developed specifically for Larch whereas our approach links Z with Ada, using Anna as an intermediate link.

The method presented in the report depends heavily on the notion of refinement of Z specifications from abstract specifications to more concrete specifications. This is, in some ways, similar to the work of Carroll Morgan [5] where a calculus of refinement is presented. Morgan's book addresses the use of fully formal refinements in code development, whereas our approach permits the use of a less formal approach if desired. Further, Morgan's book concentrates on refinements of specifications to code and performance issues. Our approach can use these refinements; however, we have tried to present a view of the process of code development using refinement, rather than to concentrate on the refinements themselves. Later work should investigate performance issues, since these are of importance to many systems: This report concentrates on the basic method, leaving performance as an issue to be addressed.

1.3 Structure of the Report

In the remainder of this report, Chapter 2 presents a brief overview of the Z language and Chapter 3 presents an equivalent overview of the Anna language. Neither of these chapters attempts to act as a language tutorial; they do, however, contain references to appropriate material. Chapter 4 describes the method through which we develop code from a specification; it makes brief comments with respect to tool support for the method. Chapter 5 lists the transformations between Z and Anna that have been developed for the purposes of the example used in the report. The example we use is taken directly from Mike Spivey's Z tutorial [6] and is the specification of a birthday book—a system to maintain a collection of names and associated birthdays, and having facilities to add new birthdays and query the birthday book for useful information. Chapter 6 presents our conclusions about the use of our method for developing code from a specification and also lists the future directions in which this work may go. The complete Z specification of the example may be found in Appendix A and the equivalent Anna specification is to be found in Appendix B.

Chapter 2

Z Overview

Z is a language for writing formal specifications using a mathematically based notation. The concepts of a system being specified are grouped into separate modules called schemas. In general, schemas consist of data definitions and constraints (expressed as predicates) on that data. These schemas may be combined into larger units using more of the Z notation.

The Z language has two components: the *mathematical language* which is used to write the contents of a schema and the *schema language* which is used to join schemas together to construct a system.

The mathematical language of Z uses the notation of predicate logic, sets, and functions to express the behavior of a system as a collection of abstract schemas. All the standard operations of sets such as set membership, intersection, union, powersets, are defined in Z. All mathematical concepts are built on the defined axioms of set theory. Among these is the concept of a function, and the mapping of a domain set to a range set. Total and partial functions may be described using the Z notation, as may other more general forms of mappings between domain sets and range sets.

A schema is defined as having two parts: a declarative part, and a predicative part (which may be empty). For example, a schema S is defined as:

$$S \mathrel{\hat{=}} [s_1 : T_1; \; s_2 : T_2; \; \ldots; \; s_n : T_n \mid f_1 \wedge f_2 \wedge \ldots \wedge f_m]$$

where s_i is a variable of type T_i, and f_j is a first-order predicate referring to variables defined in the schema. The declarative part defines a state of a schema; the predicative part constrains the values this state may have. If the predicative part is empty, then there are no constraints on the data (other than those imposed by the type declarations).

In the following example, it should be noted that the declaration is equivalent to schema declarations of the form already shown, the different syntax is convenient when a schema with many variables or predicates is to be declared. Consider the following schema:

```
__ BirthdayBook _____
  known : P NAME
  birthday : NAME ↦ DATE
_____
  known = dom birthday
```

The *BirthdayBook* schema declares two variables: *known*, whose type is the powerset of type NAME; and *birthday*, a partial function from NAME's to DATE's. The single predicate constrains the value of the two variables such that the domain of the *birthday* function must be the same as *known*.

A schema may include another schema in its declarative part; for example, in the schema:

```
__ InitBirthday _____
  BirthdayBook
_____
  known = ∅
```

The schema BirthdayBook is included in schema InitBirthday. When one schema is included in another, it is equivalent to declaring the variables of the included schema in the new schema, and adding the predicates to the new predicative part; thus InitBirthday contains variables *known* and *birthday*, the constraint of the original schema, in addition to the new constraint in its predicative part. Thus, we may conclude:

dom *birthday* = ∅

A schema may be included in another schema using Δ notation, for example, ΔBirthdayBook. This schema inclusion describes a state change in the original schema. The variables from the original schema are introduced, along with a new set of variables which define the state of the variables after the operation; the new variables are simply those of the schema decorated with a prime (*/*). Using the BirthdayBook example, *known* and *known'*, and *birthday* and *birthday'* are all variables of the new schema. An example of such inclusion is:

```
__ AddBirthday _____
  ΔBirthdayBook
  name? : NAME
  date? : DATE
_____
  name? ∉ known
  birthday' = birthday ∪ {name? ↦ date?}
```

In addition to the Δ notation described above, a schema may be included in another schema using the Ξ notation, for example, ΞBirthdayBook. This inclusion describes an operation which does not change the state of the original schema. One way to think of this is that, along with the new variables, the constraints *known'* = *known* and *birthday'* = *birthday* are added to the

predicative part. Further, the original constraints and the equivalent decorated constraints are added, so we would also add:

$$known = \text{dom } birthday$$

and

$$known' = \text{dom } birthday'$$

The schema language is used to join schemas (defined using the mathematical language) to form larger schemas that describe larger components of the system. This report covers only the use of the simple schema language connectives conjunction $\wedge$ and disjunction $\vee$ and schema inclusion which has already been described.

An example of the schema language is:

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$$

This definition states that the schema $RAddBirthday$ behaves either as the combined behaviors of $AddBirthday$ and $Success$ or it behaves like the schema $AlreadyKnown$. We have not shown the schemas $Success$ and $AlreadyKnown$ in this example because their behaviors are not of importance in discussing the use of the schema language. The full text of all examples may be found in Appendix A.

This chapter has been a very brief overview of the Z notation and concepts. A full description can be found in the Z reference manual [7].

Chapter 3

Anna Overview

ANNotated Ada (Anna) is a language extension to Ada [1] that includes facilities for formally specifying the intended behavior of Ada programs. Anna was designed to augment Ada with precise machine-processable annotations so that well established formal methods of specification and documentation could be applied to Ada programs. In this chapter, a brief outline of a few types of annotation is given. A full definition of Anna is provided in the Anna reference manual [4]. An introduction to Anna describing the concepts of the language and the way in which refinement may be performed using Anna may be found in Luckham's book [3].

Anna is based on first-order logic and its syntax is a straightforward extension of the Ada syntax. Anna constructs appear as *formal comments* within the Ada source text (as specialized Ada comments). Anna formal comments are introduced by special comment indicators in order to distinguish them from informal comments. The two kinds of Anna formal comments are *virtual Ada text*, each line of which begins with the indicator --:, and *annotations*, each line of which begins with the indicator --|.

Virtual Ada text is Ada text appearing in formal comments, but obeying all of the Ada language rules. Virtual text may refer to actual Ada code, but is not allowed to affect the computation of the actual program. Therefore, actual text cannot refer to virtual text. The purpose of virtual Ada text is to define *concepts* used in annotations. Often the formal specification of a program will refer to concepts that are not explicitly implemented as part of the program. For example, a function may be useful for the specification but may not be useful in the program. Such concepts can be defined as virtual Ada text. Virtual Ada text provides a computation model of the system. Since virtual Ada follows the rules of Ada code, it states how a function is to be performed, rather than stating only what function is to be computed. Generally, virtual Ada is used to provide function definitions required for specification purposes rather than to provide a specification of the system.

A concrete example of virtual Ada text is in the specification of a stack. If the stack is bounded, the specification will have to describe the bound by checking the current size of the stack against the maximum permitted size of the stack. Thus, a function to compute the size of the stack will be useful. Further, if we assume that the specification calls for the implementation to raise an exception if the bound should be exceeded, the interface available to users of the stack package

does not need to include the size of stack function. Thus, the size function is useful for the specification of the behavior of the package, but is not part of the interface of the package. In this case, the size function would be best written as virtual Ada text.

Annotations place constraints on the Ada program. They are specifications that apply within specific scopes of the Ada program. Annotations are constructed using boolean expressions.

An example of an annotation is:

```
MAX : INTEGER;
--| 0 ≤ MAX ≤ 100;
```

The annotation indicates that all values of the integer *MAX* must lie in the range of 0 to 100 inclusive.

The location of an annotation in the Ada program indicates the kind of constraints that the annotation imposes on the program. Anna provides different kinds of annotations, each associated with a particular Ada construct. The Anna expressions extend (are a superset of) the expressions in Ada. The annotations that appear in examples in this report include: *subtype annotations*, which are constraints on Ada types; *object annotations*, which constrain Ada variables to satisfy specified conditions; and *subprogram annotations*, which specify the behavior of an Ada subprogram, typically through input/output specifications. Annotations may be used to provide a declarative model of the system. They may be used to describe input and output conditions to functions as well as axioms relating functions within a package to each other.

A suite of tools for the transformation of Ada with annotations and the analysis of Anna specifications exists and is freely available from Stanford University.

Chapter 4

Software Development by Refinement

This report describes a particular version of a general approach to software development using notions of refinement, specifically using: a language-independent notation for initial representation and design of a system, a language-dependent notation for further design, and a programming language for implementation. For most programming languages, a suitable language dependent notation does not exist.

Deciding to use Z and Anna to aid the development of Ada programs is not sufficient; it is also necessary to have a method through which the notations are to be used. This chapter outlines a method for formal software development using Z and Anna and discusses the decisions a developer may make with respect to the relative use of Z and Anna.

Broadly, our approach is to capture the system requirements in the Z notation, refine the Z notation both to add detail to the specification and to bring the specification into a form suitable for transformation into Anna. The Anna specification may be further refined and then is extended with Ada until the specification has been satisfied.

Since Anna is a specification language, we could use it from the start of the development process; however, there are three reasons why we prefer to use Z initially and subsequently transform the Z to Anna:

1. Incremental development of the system specification is an important development technique for large systems. While this is possible using Anna, the Z schema language not only allows for incremental development, but also preserves the development history—an important feature when tracking down specification flaws.

2. Z has a notational convenience over Anna which leads to a simpler way of dealing with problems of incompleteness and ambiguity within the specifications.

3. Z has many more built-in facilities for expressing abstractions. Each of these built-in facilities has been developed with a notation and rules for use. Although these could be

---

provided for Anna, they would need to be added to the existing language. This has not been done to date.

❧

The first stage of this approach, the capture of the system requirements in the Z notation, is not covered by this paper. However, we do consider it important to confirm that the Z specification is an accurate description of the desired system. If the tools existed, we would analyze the Z specification until we were satisfied that it accurately described the desired system. However, since a tool that operates directly on Z does not exist, we transform the Z specification into an equivalent Anna specification (at present this is done manually, though the process can be automated).We may then use appropriate Anna tools to analyze the transformed specification to help to confirm the accuracy of the Anna description of the system with respect to the requirements. If the transformations preserve meaning and have been performed without error, then we can conclude that the Z specification is an accurate representation of the desired system and is, therefore, suitable as a basis for the design process (as is the Anna specification).

Typically, for any less-than-trivial system, the specification will be abstract and there will be no obvious correspondence between the specification and the implementation. The job of the developer is to refine the Z specification, introducing design choices that lead the specification toward an implementation. After completing each refinement, the developer can prove that the newly created Z specification is correct with respect to the previous specification. The proof obligation can be determined at each step by using the Z rules of refinement.

As the Z specification is refined with more detail added, it will become obvious that a corresponding Anna specification can be created using the transformations provided by the method. The corresponding Anna specification will then be used for further development. It should be noted that the transformation rules applied at this point may be different from those used when analyzing the specification. The reason for this difference is that developers will read and refine this Anna specification, but the transformation for purposes of analysis will be manipulated by Anna tools rather than humans.

The final stage of the approach is to implement the Anna specification by developing appropriate Ada code. This stage involves further refinements of the specification; these refinements will either be more modifications to the Anna specification or addition of Ada declarations and executable statements. As development proceeds, the executable Ada program will become complete and will be a correct implementation of the Anna specification. Compilation of the code will insert runtime checks ensuring that the code conforms to the Anna specification.

The development approach outlined above offers the developer a number of choices over the notations to be used. The outline suggests that initially the system will be represented by a description written in Z which will be refined in Z for some number of refinements; then it will be transformed into Anna and will be further refined; finally, code will be added. The developer chooses *when* to transform the Z specification to an Anna specification. The method allows a complete range of possibilities: from an immediate transformation from Z to Anna with all subsequent refinements performed using Anna refinement steps as described by Luckham [3], to a complete refinement in Z with a transformation into Anna just prior to the addition of executable Ada code. We expect that most developments will fall somewhere between these two

extremes, and that the tastes, skills and background of the developers will dictate when the Z to Anna transformation occurs.

Figure 4.1 illustrates the method described in this chapter. The figure has been annotated with names of appropriate Anna tools, as well as with descriptions of particular parts of the process (for example, design). Essentially, the method uses three types of transformation: from high-level to low-level Z, from Z to Anna, and from Anna to Ada. These transformations are addressed by Morgan et al., this report, and Luckham et al., respectively.

Requirements

Validated
specification

```
┌──────────────┐          ┌──────────────┐
│    Z spec    │─────────▶│   ANNA spec  │
└──────────────┘          └──────────────┘
```

Spec analyzer

Design

```
┌──────────────┐          ┌──────────────┐
│  Low-level   │─────────▶│  Low-level   │
│    Z spec    │          │  ANNA spec   │
└──────────────┘          └──────────────┘
```

Ada
code

Semantic analyzer

```
        ┌──────────────────┐
        │   Ada code with  │
        │    annotations   │
        └──────────────────┘
```

Annotation transformer

```
    ┌──────────────────────────┐
    │   Executable code with   │
    │     runtime checks       │
    └──────────────────────────┘
```
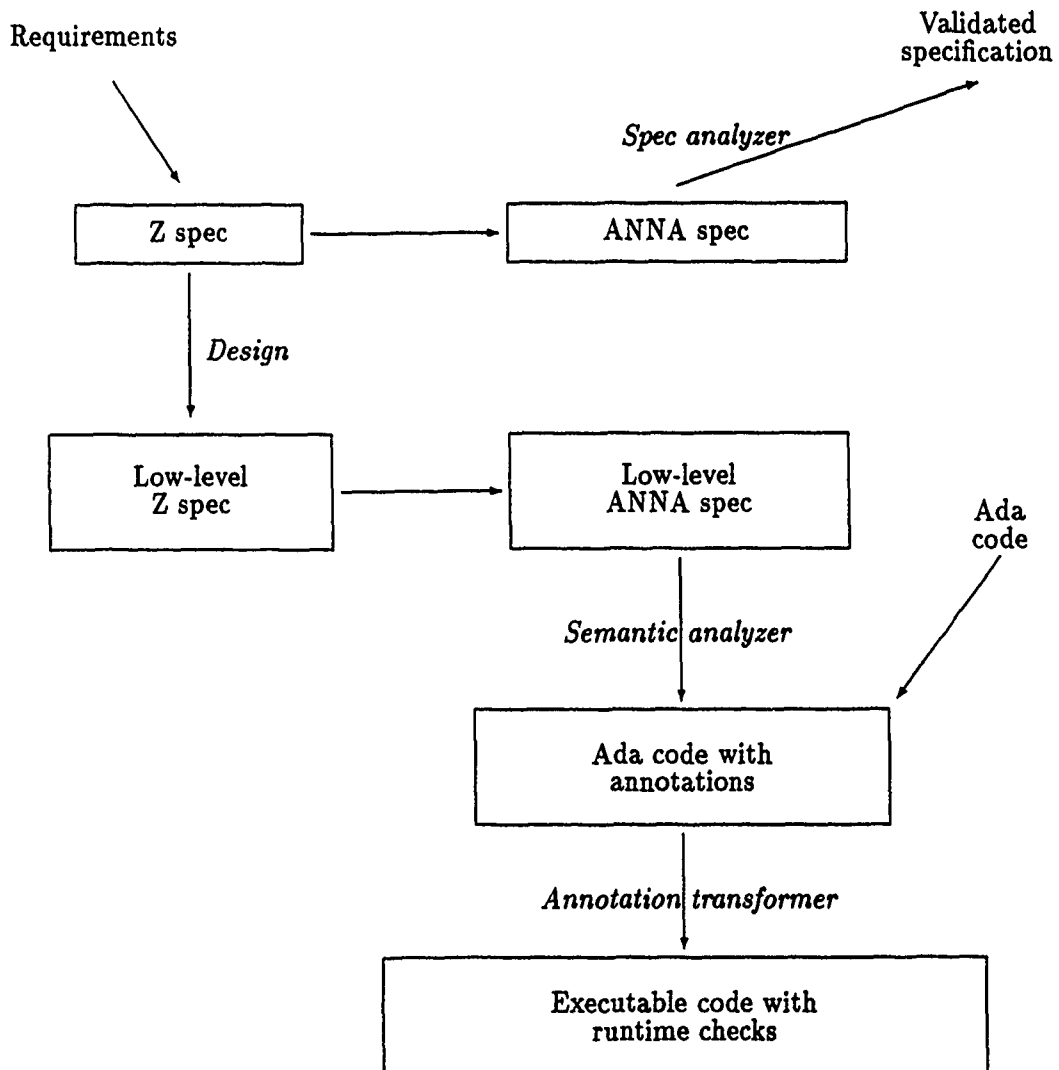
Figure 4.1: Development Method

Chapter 5

Z to Anna Transformations

This chapter describes the transformations from Z to Anna that have been identified through the development of the birthday book example. For each transformation, we describe the characteristics of the Z schema that is to be transformed, the Anna specifications that will be derived, and we give an example from the birthday book. We list the transformations according to the structure of the Z schema being transformed.

There are many ways in which a piece of Anna specification may be derived from a Z schema, and the particular transformation will depend upon the context of other transformations. For example, a variable in a Z schema may be transformed to either a function (acting as an outside observer on the state of a package) or to a package body variable depending on whether the transformation is being performed for the purposes of testing the Z schema or for further software development.

5.1   Simple Data Schema

A *simple data schema* is a Z schema that includes no other schemas and has no input or output variables. Thus, it consists of a name, some variables and their types, and a number (possibly zero) of constraints in the predicative part of the schema. The general structure of the schema is:

$$
\begin{array}{|l}
\hline
\text{SimpleData} \\
\hline
var1 : TYPE1 \\
\quad \vdots \\
varn : TYPEn \\
\hline
pred1 \\
\quad \vdots \\
predn \\
\hline
\end{array}
$$
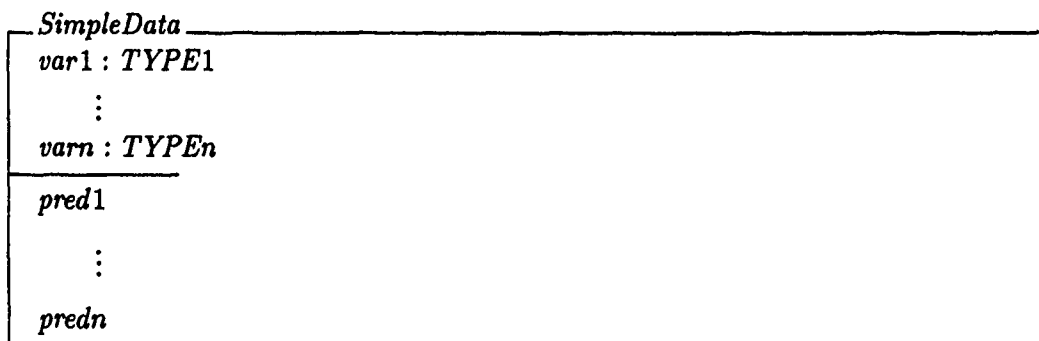
This schema declares Z state variables and places some constraints on those state variables.

The name of the schema provides the name of the package into which the state variables and constraints will be placed. This is a simplification for the purposes of this initial investigation; further investigation needs to take place with respect to inclusion of multiple schemas which will affect the decisions for package naming conventions. The choice of transformation, to functions or to variables, affects the rest of the transformations relating to the package. If the variables are transformed into functions, then the function specifications and all other procedures and functions defined in the package specification may refer to these variables. If the variables are transformed into package body variables, then the procedure and function specifications that refer to these variables will also be placed in the package body. In a full software development, the fact that the Anna specification is incomplete is not a disadvantage since specifications (in the Z form) for the Ada procedures and functions will exist.

5.1.1 Transformation to Functions

If the Z state variables are thought of as observers on the state of the schema, then an appropriate transformation of the state variables would be to Ada functions that act as observers on the state of a package. The Anna way of creating observers on the state of a package is by the use of functions that return appropriate values. This transformation is best when analyzing the Z specification using the Anna specification analyzer. The constraints then become axioms of the package specification.

Given appropriate type declarations, the simple data schema becomes:

package SimpleData **is**

> **function** var1 **return** TYPE1;
>
> $\vdots$
>
> **function** varn **return** TYPEn;
>
> --| **axiom**
> --| pred1,
>
> --| $\vdots$
> --| predn;

end SimpleData;

5.1.2 Transformation to Variables

Another way to think of the Z state variables is that they are variables that may be manipulated by the package developer, but not manipulated by users of the package. The obvious transformation is to create package body variables. It would be nice if the Z constraints could

then be transformed into invariants in the package body; however, such a transformation is incorrect. An Anna invariant must be true everywhere in the body of the package, but the Z constraints need only be true in the pre- and post- conditions of the schema. The Z constraints may be transformed into virtual Ada functions which would then be invoked by every function and procedure that is a transformation of a Z schema.

The transformation of schemas such as *SimpleData* into package variables needs further work, particularly with respect to the transformation of the constraints.

5.1.3 Transformation of BirthdayBook to Functions

There are two ways to transform the variables of the schema *BirthdayBook*: into functions or into variables. Since the choice affects the rest of the transformations in our complete example, and we have chosen to maintain all of the components of the birthday book specification in the Anna package specification, we have chosen to transform the variables into functions. The only reason for maintaining all the components of the specification in the package specification is so that the Anna package specification is complete in its own right. Otherwise, we would have to rely on the Z specification to give semantics to the procedure and function headings of the Ada package specification, with Anna being used to describe the semantics of the Ada package body.

If we transform the schema *BirthdayBook*

$$
\begin{array}{|l}
\hline
\_\,BirthdayBook \underline{\hspace{5cm}} \\
\quad known : \mathrm{P}\ NAME \\
\quad birthday : NAME \rightarrowtail DATE \\
\hline
\quad known = \mathrm{dom}\ birthday \\
\hline
\end{array}
$$

into functions, then we derive the following Anna specification. Throughout the Anna specifications, we will assume that generic packages *SET_CONCEPTS* and *MAP_CONCEPTS* are provided as part of the Anna library used for Z to Anna development. We have not listed these packages in this document; however, their specifications are such that the facilities provided as part of Z are available to Anna users.

The *SET_CONCEPTS* package will contain specifications of operations such as:

MEMBER — which, given a set and a value, returns **true** if the value is an element of the set.

UNION — which, given two sets, returns the set that is their set union.

ISEMPTY — which, given a set, returns **true** if there are no values in the set.

The *MAP_CONCEPTS* package will contain specifications of operations such as:

DOMAIN — which, given a map, will return the set of elements comprising the domain of the map.

MAPSTO — which, given an element of the domain and an element of the range of the map, will return a new map element.

There will be specifications of other operations in the *SET_CONCEPTS* and *MAP_CONCEPTS* packages. The operations listed above are the operations used in this report.

The types *NAME* and *DATE* are not described further in the Z document, so they will become **private** types in the Anna specification. If the Z specification were complete, we would see either *NAME* and *DATE* as parameters to the specification or they would appear through inclusion of a schema that models them appropriately.

```
generic

    type NAME is private;
    type DATE is private;

package BIRTHDAY_BOOK is

    package SET_OF_NAMES is
       new SET_CONCEPTS(ELEMENT_TYPE => NAME);
    use SET_OF_NAMES;

    function KNOWN return SET_OF_NAMES.SET;

    package SET_OF_DATES is
       new SET_CONCEPTS(ELEMENT_TYPE => DATE);
    use SET_OF_DATES;

    package NAME_DATE_MAPPING is
       new MAP_CONCEPTS(DOMAIN_TYPE => NAME;
                        RANGE_TYPE  => DATE;
                        DOMAIN_SET  => SET_OF_NAMES;
                        RANGE_SET   => SET_OF_DATES);
    use NAME_DATE_MAPPING;

    function BIRTHDAY return NAME_DATE_MAPPING.MAPPING;

    --| KNOWN = MAP_CONCEPTS.DOMAIN(BIRTHDAY);

end BIRTHDAY_BOOK;
```

5.2   Transformation of an Operation to a Procedure

A schema may include other schemas (of the *SimpleData* form) in order to operate on the included data in various ways. If the *SimpleData* schema is included with the Δ operator, then by convention, the meaning is that the state may change. Such a schema is translated into

an Anna procedure specification. In addition to the schema inclusion, additional Z variables may be declared. If these are decorated by a question mark (?), they are assumed to be input variables and if they are decorated with an exclamation mark (!), they are assumed to be output variables.

Then, Z schemas of the form

$$
\begin{array}{|l}
\_\,OpProc\,\rule{4cm}{0.4pt}\rule{0pt}{0pt}\\
\Delta SimpleData \\
in1? : TYPE1 \\
\quad\vdots \\
inn? : TYPEn \\
out1! : TYPEn + 1 \\
\quad\vdots \\
outm! : TYPEn + m \\
\hline
npred1 \\
\quad\vdots \\
npredp \\
\end{array}
$$
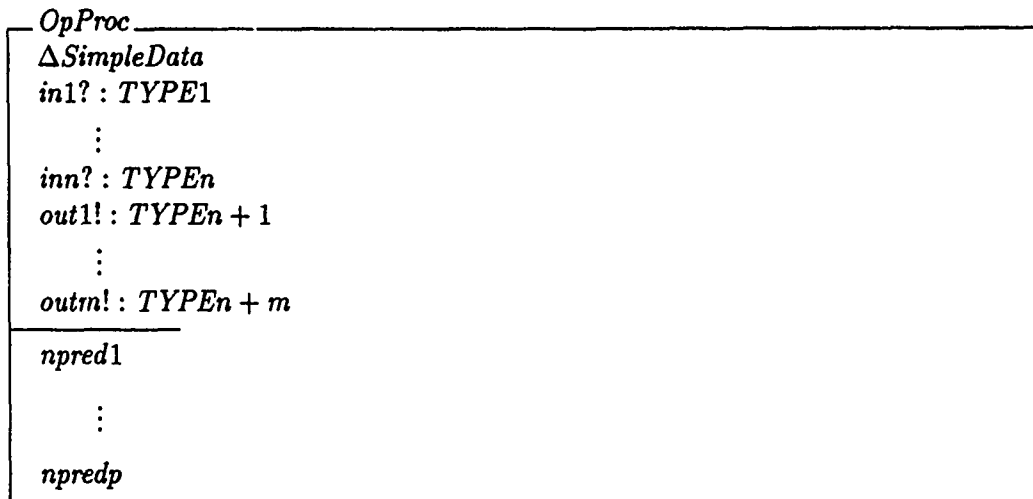
where the constraints *opred1 ... opredp* are constraints involving the input and output variables of this schema and the decorated (with a prime) and undecorated variables of the *SimpleData* schema. The constraints may be divided into two groups: input predicates and output predicates. Input predicates are those that involve only undecorated variables from the *SimpleData* schema and variables declared in the *OpProc* schema that are decorated with a question mark (the input variables); all other constraints are output predicates.

Thus, the input constraints for the translation will be a conjunction of the constraints of *SimpleData* and those constraints of *OpProc* that are input constraints. Similarly for the output constraints.

This translates into a procedure specification that is part of the package *SimpleData*.

```
procedure OpProc(in1 : in TYPE1; ... inn : in TYPEn;
                 out1 : out TYPEn+1; ... outm : out TYPEn+m);
--| where
--|     in (< Conjunction of input predicates>),
--|     out (< Conjunction of output predicates>);
```

5.2.1 Transformation of AddBirthday

Given the transformations of operations into procedures, we may now transform the schema *AddBirthday* into the appropriate Anna procedure specification:

```
┌─ AddBirthday ──────────────────────────────────────────
│ Δ BirthdayBook
│ name? : NAME
│ date? : DATE
├────────────────────────────────────────────────────────
│ name? ∉ known
│
│ birthday' = birthday ∪ {name? ↦ date?}
└────────────────────────────────────────────────────────
```

```
procedure ADD_BIRTHDAY(ANAME : in NAME; ADATE : in DATE);
--| where
--|     in (not SET_CONCEPTS.MEMBER(ANAME, KNOWN)),
--|     out (BIRTHDAY =
                SET_CONCEPTS.UNION(MAP_CONCEPTS.MAPSTO(ANAME, ADATE),
                    in BIRTHDAY));
```

5.3   Transformation of an Operation to a Function

Another way to include one schema in another is to use the Ξ annotation. This is similar to the
Δ form of inclusion just shown, but there is also a requirement that none of the state variables
will be changed by the operation. In all other respects, these schemas have the same form as
those shown in the previous section.

```
┌─ OpFun ────────────────────────────────────────────────
│ Ξ SimpleData
│ in1? : TYPE1
│     ⋮
│ inn? : TYPEn
│ out1! : TYPEn + 1
│     ⋮
│ outm! : TYPEn + m
├────────────────────────────────────────────────────────
│ opred1
│     ⋮
│ opredp
└────────────────────────────────────────────────────────
```

If there is only one output variable, this schema could be translated into an Anna function
specification. With more than one output variable, a procedure will probably be used. It is
possible, of course, to collect all output variables into an aggregate value and return this using
a function return. We do not recommend such an aggregation, since it makes the intent of the
specification less clear. Thus the general form of the transformation is:

```
procedure OpFun(in1 : in TYPE1; ... inn : in TYPEn;
                    out1 : out TYPEn+1; ... outm : out TYPEn+m);
--| where
--|     in (< Conjunction of input predicates>),
--|     out (< Conjunction of output predicates>);
```

The special case where there is only one output variable would lead to the following Anna specification:

```
function OpFun(in1 : in TYPE1; ... inn : in TYPEn)
              return TYPEn+1;
--| where
--|     in (< Conjunction of input predicates>),
--|     return out1: TYPEn+1 => (< Conjunction of output predicates>);
```

5.3.1 Transformation of FindBirthday

The Z schema *FindBirthday*

```
┌─ FindBirthday ──────────────────────────────────────────
│ ΞBirthdayBook
│ name? : NAME
│ date! : DATE
├──────────────────────────────────────────────────────
│ name? ∈ known
│
│ date! = birthday(name?)
└──────────────────────────────────────────────────────
```

is an example of schema inclusion with the Ξ decoration where there is only one output variable; thus we can transform this into the following Anna function specification.

```
function FIND_BIRTHDAY(ANAME : in NAME) return DATE;
--| where
--|     in (SET_CONCEPTS.MEMBER(ANAME, KNOWN)),
--|     return ADATE: DATE =>
--|         ADATE = MAP_CONCEPTS.MAPSTO(BIRTHDAY, ANAME);
```

All of the Z map concepts are collected together in the *MAP_CONCEPTS* generic package.

5.4 Transformation of a Predicate

As seen in preceding sections, Z schemas may include other schemas and add additional constraints. We have shown that a schema, when included with either a Δ or Ξ decoration, may be transformed into an Anna procedure or function specification.

There are occasions, however, where a simple data schema is included and additional constraints are added; such schemas have the following form:

```
┌─ ConsData ─────────────────────────────────────────
│ SimpleData
├────────────────────────────────────────────────────
│ apred1
│
│       ⋮
│
│ apredn
└────────────────────────────────────────────────────
```

These schemas are problematic and show a subtle difference between Z specifications and Anna specifications. Z is used to create declarative specifications with no implied execution model. The Z schemas are statements about facts of the system. Schemas of the form of *ConsData* are used to state additional facts about the state of the system. The predicates in these schemas do not need to be true for the entire life of the system, but only need to be true when the schema is included with other schemas. Anna is declarative, but does imply a model of execution, thus the purely declarative schemas of the *ConsData* form may not transform wholly automatically from Z to Anna. Context will be required to determine an appropriate transformation.

An example is the *InitBirthdayBook* schema

```
┌─ InitBirthdayBook ─────────────────────────────────
│ BirthdayBook
├────────────────────────────────────────────────────
│ known = ∅
└────────────────────────────────────────────────────
```

which is clearly a schema that describes the initial state of the *BirthdayBook* schema and would be translated into the following Anna
--| BIRTHDAYBOOK'INITIAL.ISEMPTY(BIRTHDAYBOOK'INITIAL.KNOWN);

In general such a simple transformation will not be possible. A more general approach will be required to transform such schemas into appropriate Anna specifications. The appropriate transformation may be an Ada procedure that raises an exception if the predicate is not satisfied. This is an issue for further investigation.

5.5 Transformation of Schema Conjunction and Disjunction

It is possible to connect schemas (by means of schema disjunction and conjunction) in ways that pieces of program cannot be connected. For example, a schema of the form

$$Schema1 \mathrel{\widehat{=}} (Schema2 \wedge Schema3) \vee Schema4$$

defines *Schema1* to have the behavior of either the combined behaviors of *Schema2* and *Schema3* or the behavior of *Schema4*.

The most appropriate way to deal with such a schema definition is to expand the definition using the Z rules of schema definition to create an equivalent schema definition that may be transformed using the rules presented in this chapter.

Let us assume that in the following example transformations, we have already defined the following schemas

```
┌─ Schema1 ─────────────────────────────────────────────────
│ Includes1
│ vars1 : TYPES1
├───────────────────────────────────────────────────────────
│ preds1
└───────────────────────────────────────────────────────────
```

```
┌─ Schema2 ─────────────────────────────────────────────────
│ Includes2
│ vars2 : TYPES2
├───────────────────────────────────────────────────────────
│ preds2
└───────────────────────────────────────────────────────────
```

where *Includes1* and *Includes2* are the schemas included (by simple naming, Δ or Ξ) into the schema; *vars1* and *vars2* are the lists of variables (and their types) declared in the schemas; and, *preds1* and *preds2* are the predicates constraining the variables in their respective schemas.

Then, schema conjunction,

$$Schema3 \mathrel{\widehat{=}} Schema1 \wedge Schema2$$

would be rewritten first as:

```
┌─ Schema3 ─────────────────────────────────────────────────
│ Includes1
│ Includes2
│ vars1 : TYPES1
│ vars2 : TYPES2
├───────────────────────────────────────────────────────────
│ preds1
│
│ preds2
└───────────────────────────────────────────────────────────
```

If there are common names in the lists *vars1* and *vars2*, or if *Includes1* and *Includes2* contain the same schema inclusion, then only one copy of the variable or schema will be included. This means that variables with the same name, included more than once (as can occur in examples such as this) must have the same type. In the case where *Includes1* contains $\Delta SimpleData$ and *Includes2* contains $\Xi SimpleData$, the combined schema will use $\Delta SimpleData$; for each variable *v* of *SimpleData*, a predicate

$$v' = v$$

will be added to the appropriate predicate part of the combined schema.

In a way similar to schema conjunction, schema disjunction

$$Schema3 \cong Schema1 \lor Schema2$$

will be rewritten as:

$$
\begin{array}{|l}
\underline{\;Schema3\;}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\\
Includes1\\
Includes2\\
vars1 : TYPES1\\
vars2 : TYPES2\\
\hline
(preds1) \lor (preds2)\\
\end{array}
$$

The resulting schemas may now be transformed according to our other transformation rules.

5.5.1 Example of Schema Conjunction and Disjunction Transformation

The birthday book example uses schema conjunction and disjunction in the creation of the stronger specification. We are interested in the transformation of

$$RAddBirthday \cong (AddBirthday \land Success) \lor AlreadyKnown$$

where

$$
\begin{array}{|l}
\underline{\;Success\;}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\\
result! : REPORT\\
\hline
result! = ok\\
\end{array}
$$

and

$$
\begin{array}{|l}
\underline{\;AlreadyKnown\;}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\\
\Xi BirthdayBook\\
name? : NAME\\
result! : REPORT\\
\hline
name? \in known\\
result! = already\_known\\
\end{array}
$$

According to our strategy, we first transform this into the following Z specification:

$$
\begin{array}{|l}
\hline
\_RAddBirthday _____ \\
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
result! : REPORT \\
\hline
(\quad name? \notin known \\
\quad birthday' = birthday \cup \{name? \mapsto date?\} \\
\quad result! = ok \\
)\ \vee \\
(\quad name? \in known \\
\quad result! = already\_known \\
) \\
\hline
\end{array}
$$

5.6 Transformation of a Schema Constant

Schema constants are schemas that define constant values within the Z specification. These take the form of including no other schemas and having only output variables (variables decorated with an exclamation mark). The general form is

$$
\begin{array}{|l}
\hline
\_ConstFun _____ \\
var1! : TYPE1 \\
\quad \vdots \\
varn! : TYPEn \\
\hline
var1 = CONST1 \\
\quad \vdots \\
varn = CONSTn \\
\hline
\end{array}
$$

and will be transformed in one of two ways, depending upon the developer. The schema will be transformed into either a collection of functions or constants. The transformation will create a new package, *ConstFun*. Thus, the general form of the transformation into functions will lead to

```
package ConstFun is

    function var1 return TYPE1;
    --|    where return CONST1;

    --|    :
    function varn return TYPEn;
    --|    where return CONSTn;

end ConstFun;
```

5.6.1 Transformation of Success

An example of a constant schema definition is the schema *Success*

```
 ┌─Success──────────────────────────────────────────────
 │  result! : REPORT
 ├──────────────────────────────────────────────────────
 │  result! = ok
 └──────────────────────────────────────────────────────
```

which simply defines a constant for use with other schemas (and will be used for schema conjunction and disjunction).

Assuming the appropriate definitions for REPORT, this would lead to the following Anna specification:

```
generic

type REPORT is private;

package Success is

    function result return REPORT;
    --|    where return REPORT.OK;

end Success;
```

It remains uncertain whether such transformations are required in practice. Although this type of transformation is acceptable, it may be that, because of our approach to the transformation of schema conjunctions and disjunctions, schema constants will not be present in the specification at transformation time. For the present, we describe this transformation until further investigation determines that it is unnecessary.

5.7 Possible Future Rules

It is clear that there are other rules still to be derived, particularly with respect to schemas that include multiple other schemas. Our approach to schema conjunction and disjunction may be too heavy-handed and we may find ways to connect Anna specification fragments in the same way that we connect Z schemas.

Also, further rules may need to be developed to consider the inclusion of a simple data schema within a "Δ" operation which is itself included in another operation with a Δ operator, in effect, creating $\Delta\Delta SimpleData$. Such an inclusion is of dubious value in a Z specification and perhaps it should be considered an error. However, if a genuine interpretation of $\Delta\Delta$ can be found, we should consider what it means in terms of an Anna specification. Z schemas may be connected.

Other aspects of Z specifications require further transformation rules. Z is able to specify the use of infinite sets, as has been shown by the birthday book example. Ada, as a programming language that executes on finite machines, cannot implement such sets. Generally, infinite sets are an abstraction used for specification purposes and the designers will develop a finite implementation of the sets. There may be an opportunity to develop an Anna specification of finite sets from the Z specification of infinite sets. However, since such a development will be system dependent, we cannot provide general transformations. An alternative is to eliminate the infinite sets in either the Z or the Anna notations. Investigations are required to determine the best development approach for infinite sets.

A further Z feature is the use of polymorphic types in the system specifications. It seems obvious that the specification of polymorphic types should transform to Anna specifications of generic packages. However, this transformation should be investigated to determine its appropriateness. Similarly, Z can hide variables declared in the schemas. It seems that hidden variables would transform to variables in the private parts of the package specifications. Such transformations need to be checked for appropriateness.

Z can specify non-deterministic behavior in the system. Anna can also be used to specify non-deterministic behavior. It remains to determine whether such behaviors should be eliminated before the transformation to Anna, or in the addition of Ada code. This issue is probably best left up to the developers of a specific system, since the appropriate time for elimination of non-deterministic behaviors (if they should be eliminated at all) is dependent on the particular system. The method should give guidelines on transformations of non-deterministic behaviors to deterministic behaviors in both the Z and Anna notations, leaving the choice of which transformations should be applied to the developers.

Chapter 6

Conclusions

This chapter presents the conclusions we have drawn from the development of the method based on our use of the birthday book example. We also describe planned future work that will provide more examples of the use of the method as well as enable us to further develop the method.

An important function of a specification is to provide the basis from which executable code may be developed. This report describes a method through which such development can take place. The method may be used with varying levels of formality ranging from the rigorous approach to a fully formal approach, the former giving the developer confidence in the developed system and the latter giving the developer certainty that the developed system meets the specification. Although we use heuristic guidelines to choose the transformations, we believe that the correctness of the transformations may be proven. So, regardless of the transformation chosen, the meaning of the specification will be preserved. Thus we claim a fully formal development process is achievable using this method.

Rather than take the more traditional verification approach of attempting to prove the final code against the original specification, we suggest an approach where small, incremental verifications may be performed. Each incremental proof step will be easier to perform than the entire step, thus enabling our method to be proven by hand or with less sophisticated tools than those used by the average developer.

We can take specifications written in Z and, using the Anna tool set, can convert these specifications using a sequence of transformations into runtime code. The runtime code developed using this method contains checks that ensure conformance with the Anna specification. The method ensures that the Anna specification is a correct derivative of the original Z specification; thus, we have confidence that the code, as long as it does not raise an Anna exception, conforms to the original Z specification.

If each step in the process is incrementally verified, the system (given input allowable by the original Z specification) will not raise Anna exceptions. In less formal developments, errors may arise and the code will raise Anna exceptions. We claim that it will be easier to debug the code, since the exception will be raised at the point at which the error occurs rather than at a later point. We have presented the transformations from Z to Anna based on the development of

---

the birthday book example. The transformations may be applied in a systematic manner that can be automated. We have confidence that the transformations may be applied in the same systematic manner on larger specifications.

The birthday book example is trivial to the extent that it is not clear why a developer would wish to make the refinement chosen rather than transform the specification directly into Anna. Indeed, a developer may write the specification in Anna immediately and feel confident that it is a specification of the desired system. We chose the birthday book as an example system in order to develop the approach described in the report so that it is easy for the reader to understand the specification and concentrate on the method rather than the problem. With more complex examples, the way in which Z schemas may be composed, using the schema language, may make it desirable to use Z initially rather than to use Anna immediately.

6.1 Future Work

There are a number of directions that must be explored to further the work presented in this paper.

Larger examples must be developed using the method presented in this report. This work may involve the development of further heuristics with respect to Z to Anna transformations, although if we discover that each new system developed using the method involves new transformations, we might conclude that the method is not appropriate for software development. There are still some forms of Z schema for which appropriate heuristics for deriving Anna have not yet been developed. An example is a schema that includes multiple "data" schemas.

The issue of tool support must be investigated further; the use of the Anna specification analyzer on the transformed Z specification is awkward since errors are presented in terms of Anna, but we are encouraging the developer to modify the Z specification. In the long run, it may make sense to convert the Anna specification analyzer to work directly on the Z specification. We have discussed a tool to automate the transformations between Z and Anna; it remains for this tool to be built.

The transformations between Z and Anna presented in this report must be shown to be semantic-preserving transformations. Although we believe this to be the case, we have not yet formally proven that the transformations are meaning-preserving.

We have stated throughout that this report describes a possible development path; it would be interesting to attempt such a development using other notations, for example, developing C code from Z. The problem with this extension is that the intermediate language we use for Ada (Anna) does not exist for C.

The system presented (and those foreseen as possible future examples) is sequential. Many systems being developed are inherently concurrent in nature. The method should be extended to cover concurrency. At a certain level of abstraction in non-distributed systems, a concurrent system may be viewed as being sequential (if only the hardware were fast enough, it could be implemented that way). Concurrency then becomes an implementation issue, so the Z notation should be investigated to see how it will handle such systems.

The work described above must be performed in order that the method presented in the report may be applied to real systems with full formality.

Bibliography

[1] American National Standards Institute. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD-1815A-1983, February 1983.

[2] J.V. Guttag, J.J. Horning, and J.M. Wing. *Larch in Five Easy Pieces*. Technical report, DEC Systems Research Center, 1985.

[3] D.C. Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, October 1990.

[4] D.C. Luckham, F.W. von Henke, B. Krieg-Bruckner, and O. Owe. *ANNA: A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[5] C. Morgan. *Programming From Specifications*. Prentice-Hall, 1990.

[6] J.M. Spivey. An Introduction to Z and Formal Specifications. *Software Engineering Journal*, January 1989.

[7] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

Appendix A

Full Z Specification

This appendix contains the complete Z specification of the birthday book example used throughout the document. The specification is taken directly from Spivey's paper introducing Z [6]. Only the specification has been presented; the interested reader is directed to the original paper for the rationale behind the specification and design.

The specification is presented in four sections: A specification of a basic birthday book system, a strengthened specification of the basic system to account for possible errors, the data refinement for the basic system with the appropriate abstraction functions, and the schema refinements of the basic system.

A.1   Basic System

$$
\begin{array}{|l}
\hline \_BirthdayBook _____ \\
known : \mathbf{P}\ NAME \\
birthday : NAME \nrightarrow DATE \\
\hline
known = \mathrm{dom}\ birthday \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \_AddBirthday _____ \\
\Delta BirthdayBook \\
name? : NAME \\
date? : DATE \\
\hline
name? \notin known \\
birthday' = birthday \cup \{name? \mapsto date?\} \\
\hline
\end{array}
$$

```
┌─ FindBirthday ─────────────────────────────────────────────
│ ΞBirthdayBook
│ name? : NAME
│ date! : DATE
├───────────────────────────────────────────────────────────
│ name? ∈ known
│ date! = birthday(name?)
└───────────────────────────────────────────────────────────
```

```
┌─ Remind ───────────────────────────────────────────────────
│ ΞBirthdayBook
│ today? : DATE
│ cards! : P NAME
├───────────────────────────────────────────────────────────
│ cards! = { n : known | birthday(n) = today? }
└───────────────────────────────────────────────────────────
```

```
┌─ InitBirthdayBook ─────────────────────────────────────────
│ BirthdayBook
├───────────────────────────────────────────────────────────
│ known = ∅
└───────────────────────────────────────────────────────────
```

A.2 Strengthened System

```
┌─ Success ──────────────────────────────────────────────────
│ result! : REPORT
├───────────────────────────────────────────────────────────
│ result! = ok
└───────────────────────────────────────────────────────────
```

```
┌─ AlreadyKnown ─────────────────────────────────────────────
│ ΞBirthdayBook
│ name? : NAME
│ result! : REPORT
├───────────────────────────────────────────────────────────
│ name? ∈ known
│ result! = already_known
└───────────────────────────────────────────────────────────
```

$RAddBirthday \hat{=} (AddBirthday \land Success) \lor AlreadyKnown.$

```
┌─ NotKnown ─────────────────────────────────────────────────
│ ΞBirthdayBook
│ name? : NAME
│ result! : REPORT
├───────────────────────────────────────────────────────────
│ name? ∉ known
│ result! = not_known
└───────────────────────────────────────────────────────────
```

$RFindBirthday \mathrel{\widehat{=}} (FindBirthday \wedge Success) \vee NotKnown.$

$RRemind \mathrel{\widehat{=}} Remind \wedge Success.$

A.3 Data Refinement for the Basic System

```
┌─ BirthdayBook1 ──────────────────────────────────
│ names : N₁ ⇸ NAME
│ dates : N₁ ⇸ DATE
│ hwm : N
├──────────────────────────────────────────────────
│ ∀ i, j : 1 .. hwm •
│     i ≠ j ⇒ names(i) ≠ names(j)
└──────────────────────────────────────────────────
```

$$\text{names} : \mathbb{N}_1 \nrightarrow NAME$$
$$\text{dates} : \mathbb{N}_1 \nrightarrow DATE$$
$$\text{hwm} : \mathbb{N}$$
$$\forall i, j : 1 .. hwm \bullet\ i \neq j \Rightarrow names(i) \neq names(j)$$

```
┌─ Abs ────────────────────────────────────────────
│ BirthdayBook
│ BirthdayBook1
├──────────────────────────────────────────────────
│ known = { i : 1 .. hwm • names(i) }
│ ∀ i : 1 .. hwm •
│     birthday(names(i)) = dates(i)
└──────────────────────────────────────────────────
```

$$known = \{\, i : 1 .. hwm \bullet names(i) \,\}$$
$$\forall i : 1 .. hwm \bullet\ birthday(names(i)) = dates(i)$$

```
┌─ AbsCards ───────────────────────────────────────
│ cards : P NAME
│ cardlist : N₁ ⇸ NAME
│ ncards : N
├──────────────────────────────────────────────────
│ cards = { i : 1 .. ncards • cardlist(i) }
└──────────────────────────────────────────────────
```

$$cards : \mathbb{P}\, NAME$$
$$cardlist : \mathbb{N}_1 \nrightarrow NAME$$
$$ncards : \mathbb{N}$$
$$cards = \{\, i : 1 .. ncards \bullet cardlist(i) \,\}$$

A.4 Schema Refinement for the Basic System

```
┌─ AddBirthday1 ───────────────────────────────────
│ Δ BirthdayBook1
│ name? : NAME
│ date? : DATE
├──────────────────────────────────────────────────
│ ∀ i : 1 .. hwm • name? ≠ names(i)
│ hwm' = hwm + 1
│ names' = names ⊕ {hwm' ↦ name?}
│ dates' = dates ⊕ {hwm' ↦ date?}
└──────────────────────────────────────────────────
```

$$\forall i : 1 .. hwm \bullet name? \neq names(i)$$
$$hwm' = hwm + 1$$
$$names' = names \oplus \{hwm' \mapsto name?\}$$
$$dates' = dates \oplus \{hwm' \mapsto date?\}$$

```
┌─ FindBirthday1 ──────────────────────────────────────────
│ ΞBirthdayBook1
│ name? : NAME
│ date! : DATE
├──────────────────────────────────────────────────────────
│ ∃ i : 1 .. hwm •
│     name? = names(i) ∧ date! = dates(i)
└──────────────────────────────────────────────────────────
```

```
┌─ Remind1 ────────────────────────────────────────────────
│ ΞBirthdayBook1
│ today? : DATE
│ cardlist! : N₁ → NAME
│ ncards! : N
├──────────────────────────────────────────────────────────
│ { i : 1 .. ncards! • cardlist!(i) }
│     = { j : 1 .. hwm | dates(j) = today? • names(j) }
└──────────────────────────────────────────────────────────
```

```
┌─ InitBirthdayBook1 ──────────────────────────────────────
│ BirthdayBook1
├──────────────────────────────────────────────────────────
│ hwm = 0
└──────────────────────────────────────────────────────────
```

Appendix B

Full Anna Specification

This appendix contains the complete Anna specification of the basic birthday book system. As in Appendix A, the specification is presented without explanatory text.

The specification presented in this appendix is typical of the output that should be generated by an automated tool transforming Z specifications into Anna.

We have not presented a specification of the refined basic system since the refinement has not been taken far enough for a precise corresponding Anna specification to be built. A further refinement must be made in the Z specification, making the infinite arrays into finite arrays (thus changing the meaning of the birthday book). It would be better to make this restriction in the top-level specification and then perform the refinement. We have chosen to follow the original example and would have to make the refinement at the lower-level Z specification, which we have not done. Such a refinement is feasible and the appropriate Anna specification could then be generated.

```
generic

    type NAME is private;
    type DATE is private;

package BIRTHDAY_BOOK is

    package SET_OF_NAMES is
       new SET_CONCEPTS(ELEMENT_TYPE => NAME);
    use SET_OF_NAMES;

    function KNOWN return SET_OF_NAMES.SET;

    package SET_OF_DATES is
       new SET_CONCEPTS(ELEMENT_TYPE => DATE);
    use SET_OF_DATES;
```

```
package NAME_DATE_MAPPING is
   new MAP_CONCEPTS(DOMAIN_TYPE => NAME;
                    RANGE_TYPE  => DATE;
                    DOMAIN_SET  => SET_OF_NAMES;
                    RANGE_SET   => SET_OF_DATES);
use NAME_DATE_MAPPING;

function BIRTHDAY return NAME_DATE_MAPPING.MAPPING;


--| KNOWN = MAP_CONCEPTS.DOMAIN(BIRTHDAY);


procedure ADD_BIRTHDAY(ANAME : in NAME; ADATE : in DATE);
--| where
--|      in (not SET_CONCEPTS.MEMBER(ANAME, KNOWN)),
--|      out (BIRTHDAY =
                SET_CONCEPTS.UNION(MAP_CONCEPTS.MAPSTO(ANAME, ADATE),
                    in BIRTHDAY));

function FIND_BIRTHDAY(ANAME : in NAME) return DATE;
--| where
--|      in (SET_CONCEPTS.MEMBER(ANAME, KNOWN)),
--|      return ADATE: DATE =>
--|            ADATE = MAP_CONCEPTS.MAPSTO(BIRTHDAY, ANAME);


procedure REMIND(TODAY : in DATE;
                 CARDS : out SET_OF_NAMES.SET);
--| where
--|      for all N : NAME =>
--|         SET_CONCEPTS.MEMBER(N, CARDS) <-> (MEMBER(N, KNOWN)
--|            and MAP_CONCEPTS.MAPSTO(BIRTHDAY, N) = TODAY);

procedure INIT_BIRTHDAY_BOOK;
--| where
--|      out (IS_EMPTY(KNOWN));

--| BIRTHDAY_BOOK'INITIAL.IS_EMPTY(BIRTHDAY_BOOK'INITIAL.KNOWN);

end BIRTHDAY_BOOK;
```

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-91-TR-1	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-91-1

6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213		7b. ADDRESS (City, State and ZIP Code) ESD/AVS HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/ AVS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962890C0003

8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS.			
	PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A	TASK NO. N/A	WORK UNIT NO. N/A

11. TITLE (Include Security Classification)
FORMAL DEVELOPMENT OF ADA PROGRAMS USING Z AND ANNA: A CASE STUDY

12. PERSONAL AUTHOR(S)
Patrick R. H. Place, William G. Wood, David C. Luckham, Walter Mann, Sriram Sankar

13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) February 1991	15. PAGE COUNT 40 pp.

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Formal Specification Development
			Ada Z
			Anna

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes a method for the formal development of Ada programs from a formal specification written in Z. ANNotated Ada (Anna) is used as an intermediate language linking the more abstract Z specifications fo the concrete Ada program. The method relies on the notion that successive small transformations of a specification are easier to verify than a few large transformations. Essentially the method uses three notations for the representation of the system: an implementation-independent notation for the specification of the system, an implementation-dependent notation for the representation of a lower level specification of the system, and the implementation language. Z and Anna are outlined briefly and examples of transformations are presented. A simple Z specification has been chosen and the transformations presented in the report are transformations of the Z specification into Anna. Conclusions are drawn about the development method presented.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION
22a. NAME OF RESPONSIBLE INDIVIDUAL JOHN S. HERMAN, Capt, USAF	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630
	22c. OFFICE SYMBOL ESD/AVS (SEI JPO)

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE   UNLIMITED, UNCLASSIFIED