From Z Specifications to Java Implementations and Back*

Alvaro Heiji Miyazawa[1], Paulo Salem da Silva[1], and Ana C. V. de Melo[1]

University of São Paulo
Department of Computer Science
São Paulo – Brazil
alvarohm@ime.usp.br salem@ime.usp.br acvm@ime.usp.br

Abstract. Formal specifications can be seen as abstract models of systems to be implemented. Typically, when implementing such specifications, the main worry concerns the correction of the implementation. In this way, it may be the case that the resulting code, while correct, cannot be easily related to the specification, since this is typically not a goal (e.g., for efficiency reasons). Therefore, events that take place during the execution of the program (e.g., errors) cannot be systematically traced back to their corresponding formal definitions. In this paper, we present a new approach to implementing formal specifications in a way that such tracing capabilities are possible. Given a formal specification written in a subset of the Z notation, our method generates the corresponding Java source code that implements a basic skeleton for the application, which programmers are then supposed to complement. To this end, we provide a Java framework that implements several elements from the Z notation, a set of rules to translate Z specifications into programs written with this framework, and a tool that implements such rules. This is a general approach for implementing Z specifications, but in this paper we focus on the tracing capabilities it allows. And while our technology is specific to Z and Java, one could easily build analogous mechanisms for similar specification and implementation languages.

1 Introduction

Formal specifications are abstract mathematical descriptions of systems one usually intends to implement. They are abstract because they do not describe the final implementation completely. Rather, they contain only what is relevant to assess the correctness of certain properties of interest. And they are mathematical because they can be manipulated and analyzed through mathematical methods. In the context of Model-Driven Engineering (MDE) [1], such specifications can be seen as models of software.

* The authors have received financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), which includes proc. 551038/2007-1, as well as from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for the production of the present work.

Eventually, they should also be implemented as working software. To this end, it is necessary to find ways to transform the abstract descriptions into actual programs. There are several approaches to this, each tackling the problem from a particular point of view or for a particular formal method. In general, however, these approaches are mostly based on formal refinements. That is, the manual step-by-step application of rules in order to put more detail in the specification without violating any of the properties defined in previous steps. Once sufficient detail has been inserted in this way, it becomes possible to map the specification into a programming language, thus generating correct code by construction (e.g., ZRC [2], the B-Method [3], a transformation to Miranda language [4]).

While highly desirable, such correctness frequently demands more thought from the person doing the refinements (i.e., to choose the appropriate rules to apply) than programming. Moreover, it assumes that one can develop the whole implementation using refinements, though sometimes this might not be the case (e.g., because the development team includes people not familiar with formal methods).

We have developed a different, more pragmatical, approach to implementing formal specifications. In this work we show how this new approach has the added benefit of allowing for easy tracing of events that take place on the implementation back to its specification. Our method consists in converting specifications made with a subset of the Z formal method [5] into actual Java [6] programs. We provide a software framework (JZed) that implements the larger blocks of Z specifications, as well as a translation tool (JZed-Gen) [7] to map particular specifications into elements of this framework. Hence, we depart from traditional formal methods to embrace the ideas of MDE.

In its purposes and methods, the JCSP framework [8] is the project that mostly resembles ours. JCSP provides a Java implementation of the CSP process algebra [9]. Using it, one can quickly create software from CSP specifications (e.g., as used in [10, 11]). But whereas CSP is a formalism to describe communication protocols, the Z method is intended to describe internal states and their transitions, hence presenting a different set of problems.

Our tool generates an application structure (i.e., concrete classes) which is then supposed to be complemented with custom code by programmers. The tracing of events, however, is built-in and requires no further intervention of the programmer. It depends only on the basic structure that is generated, which explicitly represents the elements of the specification. We believe, therefore, that the ideas behind our method provide an effective way of automating the implementation of tracing mechanisms.

The purpose of this paper is to present the general principles and technologies required for such a tracing. The text is organized as follows. In order to make it clear how our method works and what it is about, we begin by giving an example in Sect. 2. Then, in Sect. 3 we describe more systematically our overall method and tools. In in Sect. 4 we focus the discussion on the tracing mechanisms we provide. Sect. 5 reflects about the problems and potentials of our approach.

2 Example

To give straightaway a more concrete idea of what our method is about, let us begin by giving an example of how our approach can be used. Later in the article we shall refer to this example in order to describe more formally the elements of our approach.

Roughly, a Z specification is composed by *schemas* and *schema calculus expressions*. Schemas are used to define states and operations, and the calculus provides logical connectives to create more complex schemas out of simpler ones. In state schemas, one defines the data that is maintained by the state, as well as the axioms that rule such data. In operation schemas, in turn, one defines the inputs and outputs variables, as well as the axioms that relate one to the other.

In this example, we shall use the well-known birthday book specification, which is often used as a standard example in Z literature (e.g., used in [12, 13]; originally presented in [14]). It defines a database of birthdays, which can be manipulated in various ways by the users. Below we describe some parts of it, how our tools can be used to generate an overall application skeleton in Java from it and, finally, how this skeleton provides tracing capabilities.

The basic element defined in the specification is the *BirthdayBook* schema, which is responsible for storing names of people and their respective birthdays.

$$BirthdayBook == [known : \mathbb{P}\, NAME;\ birthday : NAME \pfun DATE \mid known = \mathrm{dom}\, birthday]$$

There are a number of operations that can be performed over this state schema. Let us consider only one, the *AddBirthday* operation, which adds a name and its birthday to a *BirthdayBook* schema.

$$AddBirthday == [\Delta BirthdayBook;\ name? : NAME;\ date? : DATE \mid name? \notin known \land birthday' = birthday \cup \{name? \mapsto date?\}]$$

This operation actually needs to be complemented using schema calculus in order to account for the case in which it fails (i.e., in case the name is already stored in the birthday book). To this end, there are other two operation schemas, namely, *Success* and *AlreadyKnown*. With them, we may compose the following *RAddBirthday* operation.

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

Using the method and tools described in this paper, we can automatically obtain an application skeleton for the birthday book. Our tool takes the specification as input and generates several classes as outputs (one for each schema). So, for instance, the tool generates classes named `BirthdayBook`, `AddBirthday`, `Success` and `AlreadyKnown` for the above schemas. As an example of generated code completed with custom code, let us consider the *BirthdayBook* state schema (Fig. 1) and the *RAddBirthday* operation (Fig. 2).

Now, once this code is generated, how the system developer can make use of the traceability features? Error detection and analysis, for example, can be

achieved in the following way. In the `BirthdayBook` class, it is the responsibility of the method `checkSpecificationInvariants()` to check whether the schema invariant holds or not. Every time the application state is updated, this method is called. If it fails, an exception is thrown showing exactly where the error occurred and which part of the specification is responsible for it. It could be the case, for example, that the invariant defined in the specification was actually wrong. In such a case, our traceability mechanism would be particularly valuable, since it would allow to easily detect and correct the problem.

The *RAddBirthday* operation is obtained by the schema calculus implementation provided by our framework. This implementation takes care to ensure the expected semantics from the logical connectives by calling appropriate methods in the connected schemas. This allows, for instance, one to record the failures and successes of the operation (i.e., by inserting logging procedures in the `Success` and `AlreadyKnown` classes that are also automatically generated).

Notice further that schemas can be reused in the same specification. For example, the schema *Success* could be reused in other operations to indicate successful execution. This means that all tracing mechanisms that are inserted in it are also reused. It is, thus, a compositional technique.

3 Generating Software Skeletons from Z Specification

Given a Z specification written in a subset of the Z notation, our method allows the generation of a basic application structure that must then be complemented by a programmer. To this end, we provide three main artifacts:

- *JZed*, a Java framework that implements a subset of the elements found in the Z notation (e.g., schemas);
- Rules to translate a Z specification into a program written following the conventions of JZed;
- *JZed-Gen*, a tool that actually implements these rules. It is capable of reading Z specification written in LaTeX and generating Java source code according to user defined parameters. It employs software from the CZT project [12] in order to read and represent such specifications. Apache's Velocity [15] is used for the code generation.

The method and its several elements are summarized in Figure 3.

The source code generated is essentially composed by extensions and instantiations of base classes found on the JZed framework. At the moment, they consists of three kinds, each associated with particular Z elements:

State schemas The abstract class `Schema` represents a Z schema in its simplest form. States are represented by such schemas. For each schema in a specification, there must be a corresponding concrete class that extends `Schema` in the implementation. In the example of Sect. 2, the `BirthdayBook` class is such an extension.

```
public class BirthdayBook extends Schema {

 @AVariableDeclaration
 protected HashSet<String> known;

 @AVariableDeclaration
 protected  HashMap<String, GregorianCalendar> birthday;

 public BirthdayBook(
          HashSet<String> known,
          HashMap<String, GregorianCalendar> birthday) {
   super("BirthdayBook");  this.known = known;
   this.birthday = birthday;
 }

 @Override
 protected void checkSpecificationInvariants()
         throws InvariantViolation {
   Assert.isTrue(birthday.keySet().equals(known));
}

@Override
 protected void execute() {
   System.out.println("Names␣currently␣stored␣=" +
                      known.toString());
  }  // ... other methods ...
}
```

Fig. 1. Part of the code automatically generated for the schema *BirthdayBook*. The methods are generated empty, and the programmer is responsible for implementing them. The framework takes care of calling the methods when appropriate.

```
Operation RAddBirthday = new SchemaCalcOR(
                  new SchemaCalcAND(aAddBirthday, aSuccess),
                  aAlreadyKnown);
```

Fig. 2. The transformation into Java code of the schema calculus expression that defines the *RAddBirthday* operation. The classes being instantiated are provided by the framework. They are responsible for calling the methods of the schemas given as parameters in order to provide the expected implementation for the calculus' connectives.

Operation schemas The abstract class `Operation` is an extension of `Schema` and it represents transitions between states. Again, for each operation schema in a specification, there must be a corresponding concrete class that extends `Operation` in the implementation. In the example of Sect. 2, the classes `AddBirthday`, `Success` and `AlreadyKnown` are all extensions of `Operation`.

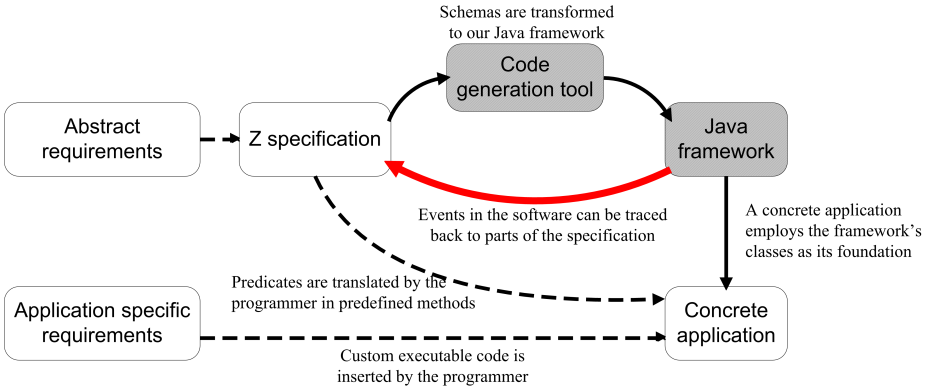

Fig. 3. The relationship between the several development artifacts. Shaded blocks show the ones which our method provides. Solid lines indicate steps that can be accomplished automatically, while dashed ones show what has to be done manually. The thicker, highlighted, arrow indicates the tracing mechanism, which takes places automatically at the framework level. In this paper, we focus on this tracing mechanism.

Schema calculus connectives Each schema calculus connective must have its own class, which should also be an extension of the base `Schema`. For example, the framework provides the `SchemaCalcAND` and `SchemaCalcOR` classes to model the connectives $\wedge$ and $\vee$, respectively. And contrary to states and operations, these classes are actually concrete and do not need to be extended. Rather, they are merely instantiated and their constructors take schemas as parameters.

The classes generated for a specific application (i.e., mostly `Schema` and `Operation` extensions) have a number of methods that are called by the framework during the execution of the program. As we shall see in Sect. 4, this inversion of control ensures that the execution follows certain patterns, which allows one to ensure to a certain extent that it conforms with the specification. However, the methods themselves are empty and are supposed to be complemented by a programmer. These methods are responsible for checking invariants, pre- and post-conditions, as well as for executing custom code defined by the programmer. For example, the method `checkSpecificationInvariants()` given in Fig. 1 is supposed to check the invariants defined in the specification, but the programmer has to fill it because our translation mechanism does not translate these inner invariants; it can only translate the overall schema structure.

The whole application comes together as an extension of the base abstract class `Specification`. In such an extension, the several schemas, operations and schema calculus expressions are grouped together. The base `Specification` provides methods to systematically run the whole system.

4 Tracing

The generated code preserves the structure found in the specification in order to relate events in the implementation to elements in the specification. This tracing can be used in different manners. Below we examine in some detail the mechanisms that allow such a tracing, and then we proceed to showing some applications for it.

4.1 Tracing Mechanisms

As explained above, the code generated by our method mimics the structure of its specification. In Z, the most important structural specification elements are the schemas and the connectives that allows one to compose complex schemas through a schema calculus. It is, thus, for these two kinds of specification elements that our framework defines corresponding implementation elements that mimic their behavior. In this way, the final implementation's execution is actually guided by the specification semantics, which allows one, at any time, to pinpoint exactly which specification element is being exercised. Consider, for example, the following schema calculus expression:

$$C == A \land B$$

It states that schema C is made of A and B (i.e., C is true if, and only if, A and B are true). Let us assume that A and B are simple schemas defined elsewhere. Then our method would translate the above expression into two new classes, for A and B, and a new object for C, which would actually be an instance of a base class found in the framework to represent the $\land$ connective.

The programmer has the ability to insert executable code in each of the new classes. Thus, for example, if an error occurs in the implementation of A, an exception would be thrown and it would be possible to know that A was responsible for it. Notice that this is not as simple as it seems, because one could implement C *without* keeping the distinction between A and B. By restricting the implementation to respect this distiction, we make such a precise detection possible.

The implemented system behaves in a discrete manner. That is to say, it acts according to a clock, and in each clock tick it updates all state schemas and state schema expressions that it is composed of. During these update procedures, events happen and tracing take place. When an operation is requested by the user, it also produces a state change and tracing happens as well. What such tracing will actually do, though, depends on its application, some of which we examine now.

4.2 Tracing Applications

The following are some of the most relevant applications for our tracing strategy.

Error Detection and Analysis If an error occurs during the execution of a program, our framework will allow one to see precisely which specification element is associated with the error. One will then be able to look in the specification for the definitions associated with the problem. It could be the case that the problem lies in the specification and, by pinpointing it there, the specification itself might be corrected. For instance, if the specification defines a schema that is actually composed by several other schemas, our translations preserves these schemas, creating a class for each one. If one of them violates an invariant, we can discover exactly which schema is related to the problem.

Logging and Profiling To understand the behavior of a system, one often records data about its behavior. With our approach, it is possible to insert such logging procedures on specific sections of the code in order to cover precisely the desired specification elements.

Besides merely recording the system's behavior, such logging can also be used to analyze the code's execution profile. It is possible, for instance, to discover which parts of the specifications are most used during actual execution of its implementation. This, in turn, may prompt the programmer to optimize not only the program, but the relevant specification parts as well (e.g., by finding a simpler formal definition).

In the example of Fig. 1, we could modify the `execute()` method to something like the following in order to log when it is used:

```
@Override
 protected void execute() {
    System.out.println("Names␣currently␣stored␣=" +
                    known.toString());

    Logger.log("Updated␣the␣BirthdayBook");

  }
```

Simulation If one defines a formal model to be simulated (e.g., for agent behavior [16]) and implements it, it is necessary to have a way to relate the simulation results back to the formal specification. Otherwise, it becomes harder to interpret such results, which renders them less useful. For instance, it might be useful to know precisely which schemas of the formal model have actually been used during a particular simulation experiment, in order to discover which assumptions are important for the observed result. Without an accountability mechanism, this sort of inquiry becomes impossible. And a standard mechanism, such as the one we provide, not only makes this possible, but also frees the programmers from having to think about how to implement it.

5 Conclusion

In this paper we have described a method and associated tools to generate Java programs from a subset of Z specifications. We focused our presentation on the tracing

capabilities that programs thus made have. We argued that the systematic preservation of the specification structure in the implementation allows a useful way to relate one to the other. Moreover, we noticed that this is not a simple feature, since typical refinement approaches do not bother to maintain such a mapping, and therefore make such tracing unfeasible. It is the conventions that we define which allow the tracing.

Concerning the flexibility of the generated code, it is expected that the programmer modify only the implementation of pre-defined methods in the generated classes. The interaction and composition among the generated classes (e.g., through schema calculus expressions) should not be modified, since this would compromise the correctness and the tracing capabilities of the system.

Currently, the framework and translation tool we developed only support state and operation schemas, and a few schema calculus connectives. This provides a minimal repertoire to work with, but clearly lacks important elements, such as axiomatic definitions. Many of these can be tackled in the same fashion, and we aim at implementing them soon. Others may prove to be harder, owing to limitations in the Java language (e.g., absence of higher order functions).

Furthermore, our method imposes an efficiency penalty. Because all specification schemas are represented explicitly, and the execution of the program involves calling their methods, it follows that there are several layers of indirection. Moreover, the implemented invariants are checked at runtime, which adds to the overhead. We believe, therefore, that the technique is most useful when efficiency is not a higher priority than the traceability requirements. This is the case, for instance, in prototypes, and also in applications in which safety and accountability requirements are strong (e.g., banking applications).

Our approach can also be specially useful when complex logical requirements are to be implemented. For example, in applications for complex accounting or legal procedures. In such domains, the ability to automatically translate logical expressions into code can make sure that no errors are introduced *in so far as* the logic is concerned. Moreover, one can then carefully develop basic schemas which are to be reused in future applications by schema calculus. And since schema calculus expressions are completely generated by our technology, it follows that the reuse of these basic schemas is very effective.

Finally, while our technology is specific to Z and Java, it should be clear that analogous mechanisms could be built for similar specification and implementation languages. Concerning the specification formalism, it suffices that it provides elements to describe states and operations. If it further provides a calculus to combine such elements, it should also be possible to implement it in a way similar to ours. And, of course, it is plain that any object-oriented language could easily substitute Java. Our method, therefore, is a general one, and our particular implementation can be taken as a proof-of-concept.

References

1. Schmidt, D.C.: Model-Driven Engineering. Computer (2006)
2. Cavalcanti, A.L.C., Woodcock, J.C.P.: ZRC—A Refinement Calculus for Z. Formal Aspects of Computing **10**(3) (1999) 267—289
3. Abrial, J.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)

4. Abdallah, A.E., Barros, A., Barros, J.B., Bowen, J.P.: Deriving correct prototypes from formal Z specifications. Technical Report SBU-CISM-00-27, South Bank University, SCISM, London, UK (2000) Available at http://citeseer.ist.psu.edu/abdallah00deriving.html.
5. ISO/IEC: Information technology – Z formal specification notation – Syntax, type system and semantics. Technical Report ISO/IEC 13568:2002(E), ISO/IEC (2002)
6. Sun Microsystems: Java technology (2007) http://java.sun.com/.
7. Miyazawa, A., da Silva, P.S., de Melo, A.C.V.: JZed-Gen: Towards Pragmatical Generation of Software from Z Specifications. In: Proceedings of the Brazilian Symposium on Formal Methods 2008 – Special Tracks. (2008)
8. Lea, D.: Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc. (1999) See also http://www.cs.kent.ac.uk/projects/ofa/jcsp/.
9. Hoare, C.A.R.: Communicating Sequential Processes. Commun. ACM **21**(8) (1978) 666–677
10. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs using *Circus*. PhD thesis, University of York (2005)
11. Freitas, A.F., Cavalcanti, A.L.C.: Automatic Translation from *Circus* to Java. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM 2006: Formal Methods. Volume 4085 of Lecture Notes in Computer Science., Springer-Verlag (2006) 115 – 130
12. Malik, P., Utting, M.: CZT: A Framework for Z Tools. In: ZB. (2005) 65–84
13. Wood, K.R.: A practical approach to software engineering using z and the refinement calculus. SIGSOFT Softw. Eng. Notes **18**(5) (1993) 79–88
14. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1992)
15. Apache Software Foundation: The apache velocity project (2009) http://velocity.apache.org/.
16. Salem, P., de Melo, A.C.V.: A Simulation-Oriented Formalization for a Psychological Theory. In Dwyer, M.B., Lopes, A., eds.: 10th International Conference, FASE 2007 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007 Proceedings. Volume 4422 of Lecture Notes in Computer Science., Springer-Verlag (2007) 42–56