JZed-Gen: Towards Pragmatical Generation of Software from Z Specifications

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

[1]University of São Paulo
Department of Computer Science
São Paulo – Brazil

{alvarohm, salem, acvm}@ime.usp.br

Abstract. *Formal specifications are useful to describe what systems are supposed to do without defining how they do it. And owing to their precise formulations, they can be used as a first system model and analyzed in a systematic way. Despite these benefits, formal specifications are not widely used in practice. One of the problems is related to time to market: transforming such specifications into actual implementations is usually laborious, involving lengthy manual transformations. To address this issue, this work presents both an automatic technique to partially transform Z formal specifications into Java programs and a tool – JZed-Gen – to support it. Our method focuses on the larger structures of the specification and generates an application skeleton which can be easily complemented by programmers, either manually or automatically. The implementation preserves the specification invariants and keeps the trace back to the actual specification parts, allowing runtime specification violations to be detected. In this paper we provide an overview of the technique and illustrate it with a simple example.*

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.

However, formal specifications should also, eventually, 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.

The authors have received financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), 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.

While highly desirable, such correctness frequently demands more thought from the person doing the refinements (i.e., to chose 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).

In this work, we take a different perspective. Instead of trying to answer how one can generate correct code by construction, we simply investigate how formal specifications can help developers gain more productivity, understanding and trust on their software. More precisely, we propose a pragmatical way to convert specifications made with a subset of the Z formal method [ISO/IEC 2002] into actual Java [Sun Microsystems 2007] programs. We provide JZed, a software framework that implements the larger blocks of Z specifications, as well as a code generation tool, JZed-Gen, that maps particular specifications into elements of this framework. Hence, we depart from traditional formal methods to embrace the ideas of Model-Driven Engineering (MDE) [Schmidt 2006].

JZed-Gen generates an application structure (i.e., concrete classes) which is then supposed to be complemented with custom code by programmers. Besides the productivity gains, this structure brings two main technical advantages. First, it enforces invariant checks to make sure that the custom code inserted by the programmer does not violate any invariants. Second, it is built in such a way that runtime events (e.g., errors) can be traced back to the particular specification elements (e.g., a particular schema) associated with it. As we shall see, these characteristics render the final software both more robust and more understandable. JZed-Gen, however, does not convert internal schema predicates to Java. This is supposed to be done either manually or by another tool.

Currently, we assume the specification to be composed only of state and operation schemas, and schema calculus expressions. Axiomatic paragraphs, for instance, are not currently supported, though we intend to add support for it. Moreover, the types used in a specification are supposed to be simple (e.g., given sets). Compound types support is currently limited to sets and partial functions.

In this paper we present the main elements of our approach, with a special focus on the code generation tool. The text is organized as follows. Section 2 contextualizes our work. Section 3 presents the foundations of our approach and explains the characteristics of the generated code. We assume that the reader is already familiar with Z and object-oriented programming. As a means of illustration, Section 4 gives a simple application example, which, in particular, shows part of the generated source code. Finally, Section 5 reflects about what has been achieved and what remains to be done.

2. Related Work

Both the Formal Methods and the Software Engineering communities have research relevant to the present work, which is placed in an intersection between them. Within Formal Methods, refinement approaches (e.g., ZRC [Cavalcanti and Woodcock 1999], the B-Method [Abrial 1996], a transformation to Miranda language [Abdallah et al. 2000]), specification languages for programs (e.g., JML [Burdy et al. 2003], Jass [Bartetzko et al. 2001]) and software for manipulating formal specifications (e.g., CZT [Malik and Utting 2005]) are specially related to our work. As to Software Engineering at large, our work has much similarity with MDE efforts

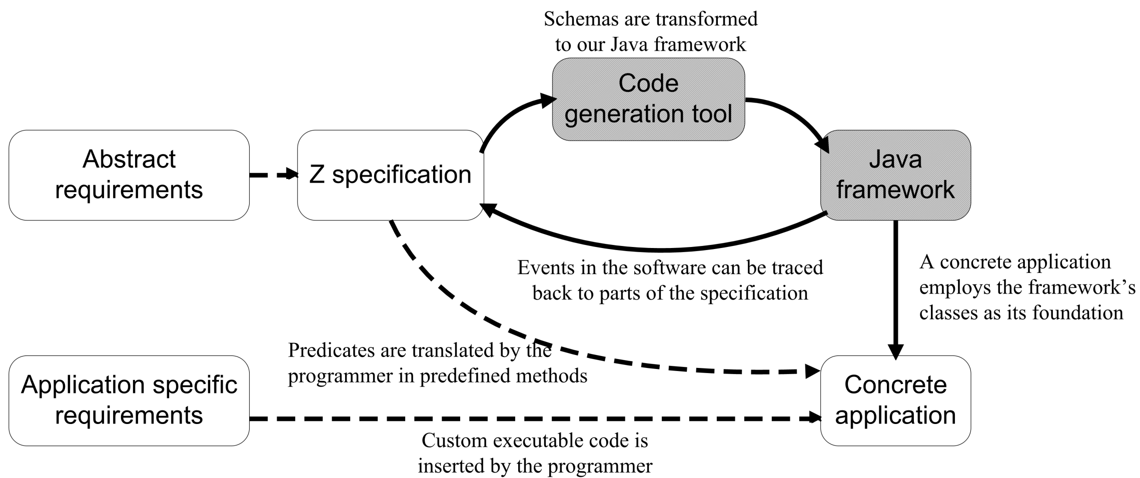

Figure 1. The relationship between the several development artifacts. Shaded blocks show the ones which our method provides. Solid lines indicate steps that can be accomplished automatically, while dashed ones show what has to be done manually.

[Schmidt 2006] , which aim at generating code from abstract models. The Model-Driven Architecture (MDA) [Miller and Mukerji 2003] is a popular concrete MDE methodology.

In its purposes and methods, the JCSP framework [Lea 1999] is the project that mostly resembles ours. JCSP provides a Java implementation of the CSP process algebra [Hoare 1978]. Using it, one can quickly create software from CSP specifications (e.g., as used in [Oliveira 2005, Freitas and Cavalcanti 2006]). Finally, we believe that the capability of relating the implementation back to its specification provided by our work can be specially useful when implementing tools that manipulate an underlying theory (e.g., our agent behavior specification [Salem and de Melo 2007]).

3. Approach Overview

Our approach focuses on quickly transforming a specification into executable code. Using JZed, a Java framework that captures the semantics of a subset of the Z notation, JZed-Gen creates a basic application structure from a specification, which then has to be complemented by the programmer with logical assertions and his/her own custom source code. The framework enforces several invariant checks, so that it becomes harder to introduce logical errors in the final program. Moreover, it also allows events (e.g., errors) in the program to be traced back to specification parts, which can be useful to analyze and improve both the code and the specification itself. Figure 1 depicts the relations among the several artifacts involved.

The JZed framework defines abstract classes for elements of Z. When transforming a specification into a concrete application, JZed-Gen examines each specification element and creates a new class that extends some base class found in the framework.[1] Owing to this inheritance relation, these new classes are endowed with appropriate semantics. The programmer, then, can insert his/her own code in predefined methods of the

[1] Currently, we support mainly state and operation schemas, and some schema calculus connectives. The same principles, however, apply to the other elements found in Z, and we aim at implementing them as well.

concrete classes.

The framework's design tries to balance adherence to the specification with the possibility of code customization. To this end, it defines three levels of implementation, which are present as abstract methods to be implemented by concrete subclasses:

- *Specification requirements.* In this level, there should be a direct implementation of what is found in the specification.
- *User requirements.* In this level, the user might create invariants that are not defined in the specification. This is provided in order to allow for the creation of invariants as needed during programming.
- *Custom code.* In this level, the user should provide any code he/she wishes. This code substitutes, to a large extent, the formal refinements that would be necessary in traditional methods. That is, we rely on the programmer's interpretation of the specification and allow him/her to implement it directly.

We shall now examine the structure of the main classes, explain how invariants, pre- and post-conditions are enforced, show how custom code can be inserted and define how the system as a whole is maintained in runtime.

3.1. Schemas and Schema Calculus

Schemas are useful to represent system states and operations over these states. The framework defines two classes to represent these concepts:

- `Schema`. Represents a Z state schema;
- `Operation`. Represents a Z operation. Actually, in Z these operations are just schemas that follow a special convention. However, since they are so important in a specification and have some special semantics associated with them, we chose to map them as a special subclass of the `Schema` class.

A state schema is transformed into a Java class that extends the `Schema` class with fields corresponding to the state variables declared in the schema. These fields must be annotated with the annotation *AVariableDeclaration* provided by the framework, and their types are given by a mapping function constructed from a basic mapping provided by the framework and complemented by both the tool and the user. The user must implement manually the constructor of the class, and the methods related to the verification of the state invariant.

An operation schema is transformed into a subclass of the `Operation` class with fields corresponding to the input, output and state variables. The annotation of these fields, as well as their types, are inserted in the same fashion as for a state schema. Usually, an operation imports one or more delta schemas. These can be translated in terms of schema inclusion, but we treat them as they are traditionally interpreted, namely, as state change. Moreover, besides the class constructor, methods related to the verification of pre- and post-conditions must be implemented.

It is important to be able to create more complex schemas from simpler ones, and this is achieved through schema import and connectives defined by the schema calculus. These complex schemas could be transformed trivially by unfolding them into simpler schemas, through the definition of schema inclusion and the connective of the

schema calculus. However, this approach would present a complication to our goal of tracing the code back to the specification. The transformation strategy adopted retains the structure of the complex schema by explicitly importing schemas and implementing the schema calculus connectives as classes that connect existing instances of state and operation schemas.

3.2. State Invariants, Pre-conditions and Post-conditions

The `Schema` class defines abstract methods to check invariants that must hold, and concrete subclasses must provide their implementation. The concrete subclasses of `Operation`, on the other hand, must implement pre- and post-conditions methods. Schema calculus formulae, in turn, don't require any further implementation, as they only call the relevant methods of the schemas being composed.

3.3. Custom Code Execution

For each concrete `Schema` subclass, the programmer may write custom code that shall be executed whenever the system state is updated and the schema's invariants hold. As to concrete `Operation` subclasses, the programmer may implement code that shall be executed whenever the operation is invoked an its pre-conditions are true.

3.4. System State Update

Clearly, the task of constantly checking invariants, pre- and post-conditions should be carried out by the framework, not the programmer. To achieve this, a special class called `Specification` holds all the states and operations in a specification. It then provides a method to check all state schemas under it. Operations, on the other hand, only need updates individually, as they are invoked.

4. Example

Let us explore a concrete application example now. We shall see how to use our technology to implement the birthday book specification, which is a canonical specification in Z literature (e.g., used in [Malik and Utting 2005, Wood 1993]; originally presented in [Spivey 1992]). However, owing to the the limited space, we will only show some parts explicitly.

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

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

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.

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

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.

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

There are, of course, several possible implementations for this specification. Let us now consider one of them.

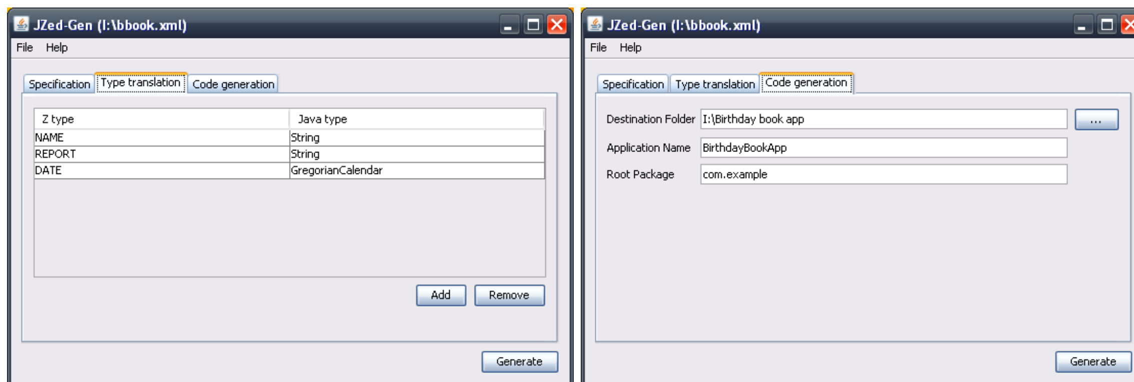

Figure 2. Two screenshots of JZed-Gen. The first shows the table where the user must define how to convert primitive types into Java classes. The second shows the application parameters, such as its name.

4.1. Generating Java Code

To perform the code generation we employ the JZed-Gen tool. In it, one may provide the specification files (written in LaTeX Z markup), the transformation for given types into Java classes and the details of the final application (e.g., its name and the folder to put its code). Figure 2 shows two screenshots of the tool.

Once the transformation data is configured properly, one may press the *generate* button to create the code. For schema *BirthdayBook*, a corresponding class named `BirthdayBook` is generated. Its code can be seen in Figure 3.

As we explained above, this class extends a standard base class that provides the fundamental mechanisms for a schema implementation and which defines abstract methods to be implemented in concrete extensions. For instance, consider the method `checkSpecificationInvariants()`. The generation tool creates it empty, but the programmer is supposed to code it. So, in this case, the programmer inserts an assertion, which corresponds to the invariant of the schema. The method `execute()` is also generated empty and might be filled in order to accomplish some custom task, which is not defined in the specification. In the code excerpt, for example, we see that this method has already been implemented by a programmer and it simply prints the current known names in the birthday book.

The operation *AddBirthday*, as well as all other schemas, are also transformed into classes. Then, the composition of the final *RAddBirthday* operation, which is a schema calculus expression, is given by the code shown in Figure 4.

4.2. Accountability and Safety

Since every schema in the specification has a corresponding class in the implementation, it is easy to trace problems to the exact part of the specification that caused them. For example, if the operation *AddBirthday* is implemented incorrectly, its post-conditions will be violated in its first execution, which shall stop the program with an exception showing that there is a problem in that particular class. Then, one may check the original specification, as well as the corresponding source code, in order to discover what caused the error. Thus, the code is accountable, since it can be related to its specification. Moreover, if somehow the `BirthdayBook` class is modified incorrectly, its invariants might be

```java
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 ...
}
```

Figure 3. Part of the code automatically generated for the schema *BirthdayBook*.

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

Figure 4. The transformation into Java code of the schema calculus expression that defines the *RAddBirthday* operation.

violated, and this would cause the program to stop as well. Hence, the code is also safe, since it can detect invariant violations and prevent execution in inconsistent states.

5. Conclusion

In this paper we presented a model-driven approach to build the overall structure of Java applications from a subset of Z formal specifications. We argued that our method brings productivity to the development process and understandability to the generated code, in the sense that it can be related to the underlying formal specification. We do not address more detailed concerns, such as refinements that could be applied to the specification prior to code generation. But we do recognize the value of such refinements and regard them as complementary to our work.

References

Abdallah, A. E., Barros, A., Barros, J. B., and Bowen, J. P. (2000). Deriving correct prototypes from formal Z specifications. Technical Report SBU-

CISM-00-27, South Bank University, SCISM, London, UK. Available at http://citeseer.ist.psu.edu/abdallah00deriving.html.

Abrial, J. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Bartetzko, D., Fischer, C., Moller, M., and Wehrheim, H. (2001). Jass - Java with assertions. In *In Workshop on Runtime Verification, 2001. held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.*

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. (2003). An overview of JML tools and applications. In Arts, T. and Fokkink, W., editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier.

Cavalcanti, A. L. C. and Woodcock, J. C. P. (1999). ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289.

Freitas, A. F. and Cavalcanti, A. L. C. (2006). Automatic Translation from *Circus* to Java. In Misra, J., Nipkow, T., and Sekerinski, E., editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 115 – 130. Springer-Verlag.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, 21(8):666–677.

ISO/IEC (2002). Information technology – Z formal specification notation – syntax, type system and semantics. Technical Report ISO/IEC 13568:2002(E), ISO/IEC.

Lea, D. (1999). *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc. See also http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

Malik, P. and Utting, M. (2005). CZT: A framework for Z tools. In *Proceedings of the 4th International Conference of B and Z Users (ZB2005)*, pages 65–84.

Miller, J. and Mukerji, J. (2003). MDA guide version 1.0.1. Technical Report omg/2003-06-01, The Object Management Group. See also http://www.omg.org/mda/.

Oliveira, M. V. M. (2005). *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, University of York.

Salem, P. and de Melo, A. C. V. (2007). A simulation-oriented formalization for a psychological theory. In Dwyer, M. B. and Lopes, A., editors, *FASE 2007, Held as Part of ETAPS 2007, Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag.

Schmidt, D. C. (2006). Model-driven engineering. *Computer*.

Spivey, J. M. (1992). *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK.

Sun Microsystems (2007). Java technology. http://java.sun.com/.

Wood, K. R. (1993). A practical approach to software engineering using Z and the refinement calculus. *SIGSOFT Softw. Eng. Notes*, 18(5):79–88.