

Mapping Formal Specifications to Java Contracts*

Michael Möller

Department of Computing Science, University of Oldenburg
26111 Oldenburg, Germany
michael.moeller@informatik.uni-oldenburg.de

September 19, 2005

1 Introduction

In no other technology sector faults are accepted to an extent comparable with software development. The reason seems to be that software systems are growing faster, and programming languages and development environments are developing faster than verification methods for software are evolving. The question of software correctness can only be answered if we know the specification, the description of what the software is supposed to do. This situation of missing software verification is constituted by two unsolved problems: how to show program correctness w.r.t. a given specification, and, possibly more important, how to write a specification that enables one to state the desired behaviour in a “readable” manner but also enables one to prove correctness.

Formal Methods are used to ensure the latter. They provide a specification language with a mathematical precise meaning, i.e., a semantics, such that given a precise semantics also for the program enables a mathematical proof of correctness. Even for programming languages, such as Java, specification languages are developed, e.g. the Java Modelling Language (JML) [10], Jass [1], etc. However these are very specific to the programming language and thus handle many low-level details.

A more abstract view is provided by high-level specification languages. Using these the specifier does not lose sight of the overall system structure. There is a tendency in research on high-level specification to combine well known specialised techniques that gain from the research done on the individual techniques and cover high complexity via the combination. Such a combined language is CSP-OZ [3] where the process algebra CSP (Communicating Sequential Processes, [4]) is complemented with the state base specification language Object-Z [2].

In this abstract we describe a way of how to derive a specification for a Java implementation (a *contract*) from such a high-level specification. The CSP-OZ specification can be seen as a way to hide the details of a concrete implementation when system properties have to be shown. Once the system specification matches all desired properties we generate contracts for the implementation. The challenge in implementing the system is then reduced to implement the contracts. A program

that is correct with respect to the contracts then will also satisfy the desired properties.

2 High-Level Specification

We will briefly introduce the specification language CSP-OZ. We use a *VendingMachine* example to keep this description simple, although we applied this approach to more complex case studies. CSP-OZ classes are the active components of a CSP-OZ specification and consist of three main parts. The first part is the interface describing the communication events that a class is participating in.

```
VendingMachine
-----
method coin : [ amount? : Coin ]
method drink : [ drink? : Drink ]
chan display : [ amount? : Amount ]
...
```

The example shows three operations of the class *VendingMachine* – two are *provided* by the class, indicated with the keyword `method`, and the operation *display* is *used* by the class, indicated with the keyword `chan`. The addressing parameters ($vm : \{\mathbf{self}\}; cus : CustomerRef;$) of the operations were skipped due to space limitations – these are used to identify the objects, i.e., instances of classes, that communicate with each other.

```
VendingMachine
-----
...
main =   coin → display → main
        □ drink → main
...
```

The second part of the class defines a CSP process `main` describing the behaviour and thus the way in which the participation in communication events takes place. In this example the process expresses, that the *coin* event is always followed by a *display* event while the *drink* event may happen as an alternative to the two events, only.

Finally, the Z part of a class defines the state space, application conditions for operations, and state changes triggered by operations.

*This research was partially supported by the DFG project ForMooS (grant OL 98/3-2).

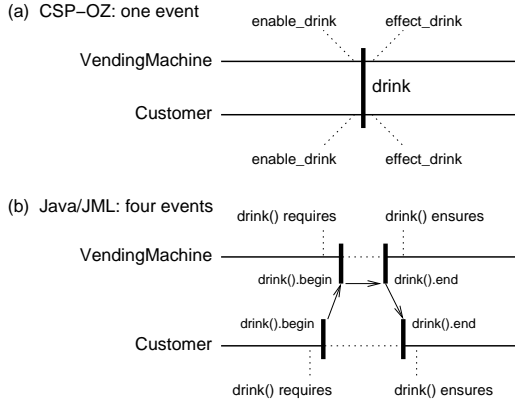


Figure 1: Events in CSP-OZ (a) and Java/JML (b)

effect schemas are translated to precondition (`requires` clause) and postcondition (`ensures` clause).

```
public interface VendingMachineSpec {
  //@ public model instance int money;

  /*@ public invariant money >= 0
   @
   && money <= 10;
   @*/
  ...
  /*@ public normal_behavior
   @ requires drink.price() <= money;
   @ assignable money;
   @ ensures money = \old(money)
   @ - drink.price();
   @*/
  public void drink (DrinkSpec drink);
  ...
}
```

To ensure correct ordering of events we use the `CSPjassda` contract (not shown here) – we generate one process per class to reflect the `main` process of that class and one process for every synchronisation of events.

Implementor’s Freedom: By translating the specification to contracts for Java³ only, there is much room left in choosing the “right” Java implementation. While some sorting predicate might be very easy to state the *best* implementation could depend – perhaps storing the data in a ordered collection is more efficient than applying some sorting algorithm or vice versa. The contracts do not force one or the other implementation strategy. But they ensure a correspondence of specification semantics and program behaviour. Our choice of techniques enables one to use e.g. runtime-verification [10, 6] as a light-weight method to establish this relationship.

5 Conclusion

In this abstract we sketched a way of how to represent high-level CSP-OZ specifications as contracts for Java programs. The CSP-OZ specification performs synchronised communication and simultaneous data manipulation. Our contribution shows how this concept can be mapped to the Java programming language. The single CSP-OZ event is mapped to four events of Java program execution.

³We also generate interfaces as containers for the contracts – so they are unavoidable in this approach.

We applied this translation to some case studies, e.g., “Holonic Manufacturing System” [13], “Automatic Teller Machine”, “Mail User Agent” (work in progress), and showed that even more complex specifications are covered by our translation rules. With providing implementations we filled our contracts with life, discovered limitations in the tools for runtime-checking and the language(s), but also developed strategies to work-around (some of) these limitations.

As future work we want to give a formal description of our translation rules. We also plan to develop further simplifications in the translation by using more specialised JML model classes to cover certain patterns in CSP-OZ specifications and thus improve readability of the Java contracts.

When applying our approach to case studies we also discovered that not embedding object-orientation deeply into the semantics of CSP-OZ is a disadvantage. It makes identifying objects at the modelling and translation level very hard. Therefore, we are developing a slightly modified dialect of CSP-OZ, but that will still preserve the main concepts presented here.

References

- [1] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In Klaus Havelund and Grigore Roşu, editors, *ENTCS*, volume 55. Elsevier, 2001.
- [2] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
- [3] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS ’97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
- [4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [5] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proc. OOPSLA 98*, volume 33 of *ACM SIGPLAN Notices*, pages 329–340, Oct. 1998.
- [6] The Jassda home page. <http://jassda.sourceforge.net/>.
- [7] The Java Modeling Language (JML) home page. <http://www.jmlspecs.org/>.
- [8] J. R. Kiniry and D. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. LNCS. Springer, 2004. accepted for the special proceedings of CASIS 2004.
- [9] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State Univ., Dept. of Computer Science, May 2003.
- [10] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. In *FMCO’02*, number 2852 in LNCS. Springer, 2003.
- [11] B. Meyer. *Object-Oriented Software Construction*. ISE, 2nd edition, 1997.
- [12] M. Möller. Specifying and Checking Java using CSP. In *Workshop on Formal Techniques for Java-like Programs – FTfJP’2002*. Computing Science Department, University of Nijmegen, June 2002. Technical Report NIII-R0204.
- [13] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Linking CSP-OZ with UML and Java: A Case Study. In *Integrated Formal Methods*, number 2999 in Lecture Notes in Computer Science, pages 267–286, March 2004.