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 loose 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

$$\begin{array}{l} \texttt{method}\ coin : \left[\ amount? : Coin\ \right] \\ \texttt{method}\ drink : \left[\ drink? : Drink\ \right] \\ \texttt{chan}\ display : \left[\ amount? : Amount\ \right] \\ \ldots \end{array}$$

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 ($vm : \{\texttt{self}\}$; $cus : CustomerRef$;) 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

$$\begin{array}{ll} \ldots & \\ \texttt{main} = & coin \rightarrow display \rightarrow \texttt{main} \\ & \Box\ drink \rightarrow \texttt{main} \\ \ldots & \end{array}$$

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).

VendingMachine

. . .

$money : Amount$

`Init`

$money = 0$

`enable`_$drink$

$drink? : Drink$

$drink?.price \leq money$

`effect`_$drink$

$\Delta(money)$

$drink? : Drink$

$money' = money - drink?.price$

. . .

As in Object-Z, the unnamed schema describes the state space for which the `Init` schema defines the allowed initial values. For every operation *op* the `enable`_*op* schema defines conditions that are required for the operation to work and the `effect`_*op* schema describes conditions that are established.

3 Java Contracts

3.1 JML: Modelling State Space

The Java Modeling Language (JML) [9] brings an extended Design-by-Contract concept to the programming language Java. Such specifications for programming languages formulate requirements and promises for the given piece of software comparable to a *contract*.

The Design-by-Contract (DbC, [11]) concept includes the conventional clauses for pre- and postconditions of methods as well as class invariants. Besides clauses for exceptions, JML's extensions include model and ghost variables which describe specification only data and thus allow modelling of abstract state space. Connection to concrete state space is established by "**represents**" clauses in the concrete classes and thus formulate a data refinement relation[1].

JML is widely accepted and supported by a range of tools covering the different levels of program verification from "real" interactive verification (LOOP project, [5]) to static checking (ESC/Java2, [8]) to runtime verification (Iowa State JML tools, via [7]). Typically JML extensions are encapsulated in specially formatted Java comments, so that any Java tool can still handle the source code.

3.2 Event Ordering: CSP$_{jassda}$

Speaking of an event in a Java program we refer to some point in program execution. To be more precisely we refer to the entry and exit points of methods. Thus for every method of a program we can observe two events – one triggered, when a method is entered and thus execution of the method body begins and the other one triggered when leaving the method body – either by normal termination or due to an exception.

For specifying the order of events with a standard DbC extension one very often ends up in a convoluted specification: model variables are used to specify the state of some automata (or processes) and with every method involved we check that these variables represent a state where the method is legal.

To overcome this limitation we developed CSP$_{jassda}$, a CSP-like specification language for events based on Java programs [12], and Jassda [6], a tool for runtime checking the conformance of a Java program run with a CSP$_{jassda}$ specification.

4 Mapping Specifications to Contracts

For mapping the CSP-OZ specification to the Java programming language we have to identify constructs in the Java program that match the building parts of CSP-OZ: types, expressions, events and CSP-OZ classes. In [13] we give a more detailed description of the translation. Here we will concentrate on the concepts.

Types: Types are directly mapped to Java types, i.e. Java interfaces providing the required methods. For certain integer subsets that fit in Java's `int` range we will use that, otherwise we map them also to interfaces that encapsulate JML's `\bigint`. Sets, relations, etc. can be mapped to JML's model classes, but we have to perform cast operations for retrieving the original types since JML does not yet support Java's generics.

Expressions: Expressions are rephrased as Java expressions. Since CSP-OZ uses a value semantics for the data we will always perform value comparisons (via `equals`) instead of testing for object identity. Predicates are simply mapped to Boolean expressions. Even quantifiers are supported by JML[2].

CSP-OZ classes and events: In section 3.2 we already mentioned that CSP-OZ events become Java methods. Thus even the active classes become "simple" Java interfaces. For a simulation of a CSP-OZ event we identify four events in the Java program: the class that participates in the event at the channel side enters its method, then the `method` class enters the method, and finally the methods are left in reverse order. Before an event may happen the current state of both classes satisfy the enable schemas of the method/channel operation and afterwards the effect schemas are valid. So we get the correspondence of enable schemas and preconditions and of effect schemas and postconditions. In figure 1 we illustrate this relationship.

To give an impression of the result of our translation we present here part of the generated interface for class *VendingMachine*. The `State` schema was translated to a model variable. The definition of type *Amount* as an integer subset results in a class invariant. Enable and

[1]For most cases it is advisable to give a functional relation between abstract and concrete state space, e.g. for runtime checking.

[2]But for runtime verification it is required that the range of quantification is defined by a given finite set.

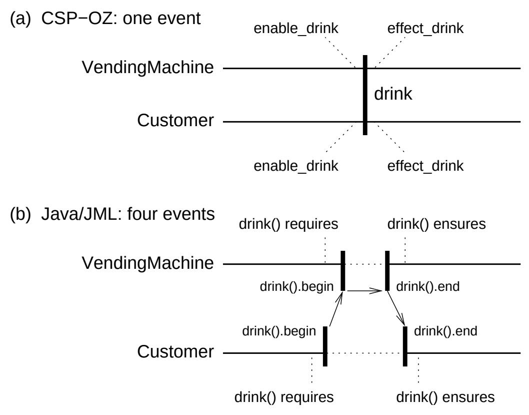

Figure 1: Events in CSP-OZ (a) and Java/JML (b)

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 CSP$_{jassda}$ 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 Java[3] 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.

[3]We 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 CASSIS 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 verifikation. 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.