Specifying and Checking Java using CSP[*]

Michael Möller

Universität Oldenburg, Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
`Michael.Moeller@informatik.uni-oldenburg.de`

Abstract. Currently several approaches are done in applying formal techniques to the Java programming language. A new trend is to take dynamic behaviour into account when designing such techniques. To bring formal techniques to practical applications one often has to reduce the goal coming down from full verification to runtime checking.
jassda [5,4] is a framework for performing such runtime checks at the byte-code level of Java. The Trace-Checker module of jassda allows one to test the dynamic behaviour of multiple Java virtual machines by monitoring whether the trace of all relevant events is a member of the trace semantics of a given CSP process or not.
In this paper we present the CSP dialect that is used to specify a set of allowed traces for Java programs. The underlying semantics allow partial specification of distributed Java programs and to recombine them while preserving properties.

1 Introduction

Applying formal techniques in the design of large distributed software systems is a major issue to obtain correctness. Correctness concerns the question whether the design or implementation of a system meets the requirements (or specifications) set out in earlier phases of the development. *Modelchecking* [6] of finite state systems is successfully applied in many areas to check that systems satisfy their specifications. However, modelchecking often suffers from the so-called state explosion problem which forbids verification because of complexity. Applying classical *program verification* techniques [11,1] to object-oriented languages (and its support by theorem provers) is a topic of current research (see for instance [13,14,20]). But these techniques often restrict the language (e.g. forbid the use of threads or exceptions), and/or require manual interaction, that is only viable by experts.

Very early specifications in Java programs were supported by the `javadoc` tool, that allows one to state the intended behaviour of methods, classes and packages as source-code comments in a structured way. However, these documentation comments are not formal in the sense of a mathematics. Many approaches to bring mathematical strength to this area concentrate on *Design by*

[*] This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1.

Contract, as proposed by Meyer for the object-oriented language Eiffel [19]. Examples of these attempts are the Java Modelling Language (JML [17]), Jass [3] or other assertion languages supported by their special tools, e.g. jContractor [15], iContract [16], etc. In most cases these assertion languages come with tools for inserting checking code for runtime checks of these assertions to test them during *program execution*.

Stating requirements for the *dynamic behaviour* in contrast to a state based view is almost new in these assertion languages. While standard assertions are the Design by Contract counterpart of state-based formal specifications, the counterpart of behaviour-oriented specifications may be expressed by process algebras or temporal logic, e.g. the JML mailing list discusses the introduction of some temporal logics. Our solution in Jass was the use of a CSP process in the class invariant that describes allowed traces and is therefore called trace assertion. But the expressiveness of Jass trace assertions was limited to a special subset of Java classes that are obtained when translating CSP-OZ [8,7] specifications to Java skeletons. Trace assertions were always bound to a single object instance so that static methods were not covered and other instances were only partially supported. A new trace assertion facility was implemented as a prototype in the jassda[1] framework[2] with the intention to overcome these limitations. We do not see this Trace-Checker as a replacement for Design by Contact but as an extension. State transformations are perfectly expressed by invariants and pre- and postconditions. But expressing the order of method calls within these assertions make them hard to read.

A similar approach to specify the behaviour of complex systems are abstract state machines (ASM). AsmL [10] is a modelling and programming language based on ASMs. Since AsmL is very close to a programming language it combines state transformations and the dynamic behaviour, i.e. the order of these state transformations. So using AsmL instead of trace assertions would also mean to replace Design by Contract assertions by this language. In addition, the process algebra view on the system could help not to mix up specification and implementation.

The straightforward way of specifying dynamic behaviour is to have an event based view on the Java program, or the distributed system of communicating Java programs: the trace assertion approach of Jass already identifies entry and exit points of method invocations as events of the specification. In jassda we improve this event view by taking more properties of an event into account, e.g. exceptions, object instances, threads, virtual machines, ..., and by letting Java classes classify those events and therefore getting more flexibility. We also abstracted from single events by speaking in terms of event sets. By choosing the same interleaving semantic model as CSP we will have to deal with exactly one event at every point in time.

[1] **Jass D**ebugger **A**rchitecture

[2] Work on the jassda framework will be continued as an OpenSource project (see `http://jassda.sourceforge.net/`)

As in the Jass attempt, which is a pre-compiler written in Java, we are only interested in runtime checks of specification violation. It might be possible to derive input for a modelchecker from the system under test (source-code or byte-code) and thus being able to prove a refinement relationship between program and specification. But our hope is to be able to check more complex systems than by static checks. The runtime checks are carried out at the byte-code level, so that a binary distribution with some API description for identifying the events is enough to write a specification. The jassda framework uses the Java Debug Interface to receive events, i.e. monitor the Java program on its own virtual machine, and thus no modification is performed on the system under test.

Jass was implemented as part of three master's theses [18,2,21] and is available free of charge[3]. The general ideas have been developed in [8], which also gives a formal semantics to Jass. The jassda framework and the Trace-Checker were implemented as part of a fourth master's thesis [4], which also gives some operational semantics to the CSP dialect for deriving the implementation.

2 Syntax

The design of our specification language was inspired by CSP, Communicating Sequential Processes [12], which is a well known process algebra. The textual presentation is close to CSP_M, the input language of the CSP-modelchecker FDR [9]. But we performed some modifications to make CSP more comfortable to state the requirements of a Java program. Since we use our CSP dialect in the context of the jassda framework we called it CSP_{jassda}.

2.1 Events and Event Sets

The events of a Java program that we want to observe are method invocations, or more precisely the entry points, normal termination and exceptional termination points of methods. This definition of events fits to the state view used in Design by Contract approaches for Java, like JML [17], where the entry point maps to a state where the precondition holds, the normal termination maps to a state where the postcondition holds and the exceptional termination represents a state where the signals clause holds.

Such an event of a system of Java programs has a number of properties that all may be relevant to distinguish it from other events: the method name, the signature, the class or interface (the type), the thread that executes the method, the virtual machine that executes the thread, ... Fixing all these properties in the specification is not very handy and often not what we want to specify. Therefore, while the prefix operator of CSP takes a single event as first argument we allow a set of events. This can be seen as syntactic sugar to abbreviate the choice over all processes starting with a prefix operator with one event of the set.

[3] see `http://semantik.Informatik.Uni-Oldenburg.DE/~jass/`

Event sets are defined by specifying the properties[4] of an event or by binding variables. The matching between concrete events and definitions or bindings is done by a `Java` handler class that may be specified by the reserved "`handler`" property key. So if the default handlers do not fit to the needs of the specification one is free to define new classes, that will be responsible to parse the rest of the properties. But in most cases the default classes will be sufficient.

$\langle$*EventSetDef*$\rangle$ ::= "`eventset`" $\langle$*Identifier*$\rangle$ "{" $\langle$*PropertyDefs*$\rangle$ "}"

To give an example we define all events that are emitted by any object instance of class "`jassda.Example`" concerning any method "`print`":

```
eventset example { class="jassda.Example" method="print" }
```

Property parsing is done by the predefined default handler, because no "`handler`" property is given here. The properties "`class`" and "`method`" are fixed to constant values. To fix properties to values at runtime we use variable bindings (see below).

In the specification we also allow anonymous event set definition and finally intersection and union of such event sets. For writing convenience we use "`.`" for set intersection – not for communication via channels like in CSP, but since communication forces the value to be communicated intersection is very similar to communication via channels. For the same reason we use "`!`" for intersection. Set union is expressed by "`,`" or "`+`".

2.2 Processes and Operators

CSP_{jassda} is very close to CSP. We allow similar basic processes and the same operators to build further processes. But there are slight differences that we want to mention here.

$\langle$*Process*$\rangle$ ::= "`STOP`" | "`TERM`" | "`ANY`"

Basic processes are like in CSP where "`TERM`" abbreviates the termination event of a virtual machine followed by the "`STOP`" process, that will not accept any further event. "`ANY`" corresponds to the *CHAOS* process of CSP.

A first difference to CSP was already pointed out in the previous section: the prefix operator takes an event set instead of a single event. So we get the syntactic rule

$\langle$*Process*$\rangle$ ::= $\langle$*EventSet*$\rangle$ "`->`" $\langle$*Process*$\rangle$

We also allow choice and parallel composition of processes:

$\langle$*Process*$\rangle$ ::= $\langle$*Process*$\rangle$ "`[]`" $\langle$*Process*$\rangle$ | $\langle$*Process*$\rangle$ "`||`" $\langle$*Process*$\rangle$

but we only allow synchronisation over the intersection of the complete alphabets of both processes. There are also "quantified" variants of both operators, that are similar to variable binding.

$\langle$*Process*$\rangle$ ::= "`||`" $\langle$*VarBinding*$\rangle$ "`@`" $\langle$*Process*$\rangle$
| "`[]`" $\langle$*VarBinding*$\rangle$ "`@`" $\langle$*Process*$\rangle$

$\langle$*VarBinding*$\rangle$ ::= $\langle$*Identifier*$\rangle$ "`:`" "`[`" $\langle$*PropertyDefs*$\rangle$ "`]`"

[4] Properties are specified by key value pairs.

2.3 Variables and Parameterised Processes

To propagate the events that occur in quantified operators we allow parameterised processes and (guarded) recursive process calls by referencing them.

$\langle \textit{Process} \rangle$ ::= $\langle \textit{ProcessIdentifier} \rangle$ "`(`" $\langle \textit{ProcParams} \rangle$? "`)`"
$\langle \textit{ProcParams} \rangle$ ::= $\langle \textit{EventSet} \rangle$ ("`,`" $\langle \textit{EventSet} \rangle$)*

As type for variables we allow nothing else but event sets. Since we express everything observable of programs by events this should be sufficient. One way to bind event sets to variables was already mentioned: the quantified operators. Another way to do this is in combination with prefixing. We may add variable bindings to event sets separated by "`?`". This is in analogy to communication in CSP: binding event properties to variables (and thus defining new event sets) is similar to receiving a value over a channel.

Bindings are used to fix properties of new event sets by accessing property values of the current event. This allows us for example to force that the return value of a method will become the value of an argument of a following method invocation. In the current implementation of the default handler class the bindings just allow to map properties without calculation (stating this property is fixed to the value of that property of the current event), but future implementations or user defined handler classes could provide more complex binding operations.

For example let us have an event set "`calculator`" describing all events concerning a calculator class, a set "`getAcc`" representing the return from any method named "`getAccumulator`" and a set "`calc`" representing the method calls to "`calculate`" of any class. Then the CSP_{jassda} process

```
calculator.getAcc?acc:[arg0=result] -> calculator.calc!acc -> STOP
```

will accept the normal termination of method "`getAccumulator`" of the calculator class and then a call to method "`calculate`", where the first argument is the return value of "`getAccumulator`". With the first event included in the set "`calculator.getAcc`" the current return value of that event (value of property "`return`") is bound to the property "`arg0`" (first argument of a method call) of the "`acc`" event set.

3 Semantics

Semantics of the specification language is not complicated, since we just monitor a Java program[5] and check for each element we receive whether it is consistent with the specification. Using the Java Debug Interface (JDI) the events of the system under test will be serialised so that the jassda tool will receive an interleaved view on concurrent events.

To check this sequence of events against the specification we will have to decide if the run is correct for each event we receive. For this reason we have

[5] or a system of Java programs

to deal with deterministic processes[6], only. This is because a nondeterministic choice will produce the same sequence of events that a deterministic choice does.

3.1 Alphabets and Trace Inclusion

Our goal is to check the traces of the system of Java programs against our specification given in terms of CSP_{jassda}. So the events that are of interest for us are only those, that are part of the specification. This projected view on the events of the system under test allows us to test partial specifications and finally to combine them, internally by parallel composition of the parts of the specification, to get the full test.

To retrieve this "events of interest" we have to calculate the alphabet of the specification. This alphabet can be given explicitly or can be calculated, e.g. in a way that it is minimal in fulfilling the conditions to alphabets given in [12]. This calculation can be done by choosing the union of all event sets that occur in prefix operators as the alphabet of a process. Here we do not take any variables into account since they are not bound prior to the run of the system. They are bound through prior already accepted events and thus do not restrict the alphabet. This alphabet is used to instruct the virtual machines to emit those events.

So if t is the trace of events of the system under test and $\alpha(\mathit{Spec})$ is the alphabet of the specification, i.e. the union of all alphabets of the processes, then we test if $(t \upharpoonright \alpha(\mathit{Spec})) \in \mathit{traces}(\mathit{Spec})$, i.e. the trace restricted to events from the alphabet is member of the set of traces defined by the specification.

3.2 Operational Semantics

While running the test of the system we use the operational semantics for efficiently tracking the state of the test. The semantics is based on a labelled transition system, where states are represented by processes. Each step is triggered by an event from the system under test. To process the event we first check if the process representing the current state accepts it. In case it is accepted we create the new state, i.e. the specification process, that represents the traces that will be accepted in the future.

The operational semantics is an interleaving semantics. This fits perfectly to the sequence of events delivered by the JDI, since the serialisation will interleave the events. Therefore for every point in time we have a single event to handle and a trace representation is adequate even for a system of programs running in parallel.

To be able to check the specification at runtime we define the operational semantics such that it is deterministic. All events have to be visible events and the choice is delayed in case the event is accepted by more than one subprocess.

[6] Although we allow processes that accept the same prefix of events for the choice operator this is still deterministic since we can rewrite the process avoiding this common prefix. This can be seen as syntactic sugar.

4 Example

To give an impression on how to specify properties and how expressive our CSP dialect is we will give a simple example. We will specify that every instance of an applet will first be initialised via its `init` method and after that `start` and `stop` may alternate before the applet is shut down by calling the `destroy` method.

As the first step we will define some event sets for use in the specification. These definitions do not define the alphabet of the process but are used as abbreviations in the process definitions.[7]

```
eventset applet  { instanceof="java.applet.Applet" }
eventset init    { method="init" }
eventset start   { method="start" }
eventset stop    { method="stop" }
eventset destroy { method="destroy" }
```

Then we specify the allowed behaviour and this way specify the alphabet of the specification. Since the behaviour has to be observed for each instance of an applet we define parallel processes, one process for each instance.

```
applets() {
  ||i:[instance] @ appletbehaviour(i)
}

appletbehavior(inst) {
  applet.inst.init.begin -> applet.inst.init.end
   -> appletrun(inst)
}
appletrun(i) {
  ( applet.i.start.begin -> applet.i.start.end
      -> applet.i.stop.begin  -> applet.i.stop.end -> aplletrun(i)
  )  [] appletdestroy(i)
}
appletdestroy(inst) {
  applet.inst.destroy.begin -> applet.inst.destroy.end -> STOP
}
```

We can calculate the alphabet by collecting the events of the specification. The first process "`applets`" will tell us, that the alphabet is the union of the alphabets of the "`appletbehaviour`" processes. The alphabet of an instance of process "`appletbehaviour`" contains the begin and normal termination events of method "`init`" of any subclass of "`java.applet.Applet`" and the alphabet of "`appletrun`". The event set variable "`inst`" (the parameter of the process) will restrict the events when instantiating processes during the runtime check.

[7] We also have predefined event sets for the different event types ("`begin`", "`end`" and "`exception`").

The above specification just states the order of allowed events. If we also want to specify that no exception may be thrown by one of the mentioned methods we have to add those events to the alphabet of the process.

```
alphabet applet.(init,start,stop,destroy)
```

will make no restriction on the type of events and therefore add the exceptional cases of the above methods to the alphabet. Because these events are never accepted by the process we guarantee that they will not occur in a correct run.

5 Conclusion

In this paper we presented CSP_{jassda}, a CSP dialect that we use in the jassda framework to specify and check properties of Java programs concerning the order of events. The specification is based on the well known process algebra CSP and thus profits from research results in that area. Our CSP dialect is specialised to the Java programming language, but it still has the character of a process algebra and not that of a programming language. The approach is applicable to every program that is executed in a Java virtual machine supporting the JDI. Definition of events is not restricted to special application areas, so that e.g. even servlets are manageable. But our approach should always be seen as an addition to state based assertions, not as a replacement.

As future work we envisage the *automatic generation* of trace assertions from formal specifications, in our case from specifications written in CSP-OZ [7], a formal method combining CSP with Object-Z. Our CSP dialect is closer to Java than the trace assertions of Jass, but further away from the CSP part of an CSP-OZ specification, so that the translation is a little less straightforward.

Concerning the jassda framework, we plan to improve the Trace-Checker and the underlying framework. There are still a number of operators missing in CSP_{jassda} that make specification of properties easier and more comfortable and that of course must be supported by the tool.

Acknowledgements. Many thanks to Mark Brörkens for constructive discussions concerning CSP_{jassda} and for implementing jassda as part of his master's thesis.

References

1. K.-R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 2nd edition, 1997.
2. D. Bartetzko. Parallelität und Vererbung beim "Programmieren mit Vertrag". Master's thesis, Universität Oldenburg, 1999. in German.
3. Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with Assertions. In Klaus Havelund and Grigore Roşu, editors, *Proceedings of the First Workshop on Runtime Verification (RV'01), Paris, France, July 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science, 2001.
4. Mark Brörkens. Trace- und Zeit-Zusicherungen beim Programmieren mit Vertrag. Master's thesis, Universität Oldenburg, 2002. in German.

5. Mark Brörkens and Michael Möller. jassda Trace Assertions. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, *Trends in Testing Communicating Systems*, International Confernece on Testing Communicating Systems (TestCom), pages 39–48, Berlin, Germany, March 2002.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM, 1983.
7. 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)*, IFIP, pages 423–438. Chapman & Hall, 1997.
8. C. Fischer. *Combination and Implementation of Processes and Data: From CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.
9. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR 2*, Dec. 1995. Manuscript.
10. Foundation of Software Engineering (FSE), Microsoft Reserach, Redmond, WA 98052, USA. *An Introduction to ASML 1.5*, March 2002. `http://research.microsoft.com/fse/asml/doc/asml15intro.pdf`.
11. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12:576–580, 1969.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
13. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 284 – 303. Springer-Verlag, 2000.
14. K. Huzing, R. Kuuiper, and SOOP. Verification of object-oriented programs using class invariants. In T. Maibaum, editor, *FASE 2000: Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 208 – 221. Springer-Verlag, 2000.
15. Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A Reflective Java Library to Support Design By Contract. Technical report, Department of Computer Science, University of California, Santa Barbara, 1998. `http://www.cs.ucsb.edu/~murat/jContractor.PDF`.
16. R. Kramer. iContract - the Java Design by Contract tool. Technical report, Reliable Systems, 1998. `http://www.reliable-systems.com`.
17. G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical report, Department of Computer Science, Iowa State University, 1998, revised 2001.
18. D. Meemken. Programmieren mit Vertrag in Java. Master's thesis, Universität Oldenburg, 1997. in German.
19. B. Meyer. *Object-Oriented Software Construction*. ISE, 2nd edition, 1997.
20. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
21. M. Plath. Trace Zusicherungen in Jass - Erweiterung des Konzepts "Programmieren mit Vertrag" . Master's thesis, Universität Oldenburg, 2000. in German.