Verification of JCSP Programs

Vladimir Klebanov, [a] Philipp Rümmer, [b,1] Steffen Schlager [c] and Peter H. Schmitt [c]

[a] *University of Koblenz-Landau, Institute for Computer Science,*
D-56070 Koblenz, Germany
[b] *Chalmers University of Technology, Dept. of Computer Science and Engineering,*
SE-41296 Gothenburg, Sweden
[c] *Universität Karlsruhe, Institute for Theoretical Computer Science,*
D-76128 Karlsruhe, Germany

Abstract. We describe the first proof system for concurrent programs based on Communicating Sequential Processes for Java (JCSP). The system extends a complete calculus for the JavaCard Dynamic Logic with support for JCSP, which is modeled in terms of the CSP process algebra. Together with a novel efficient calculus for CSP, a rule system is obtained that enables JCSP programs to be executed symbolically and to be checked against temporal properties. The proof system has been implemented within the KeY tool and is publicly available.

Keywords. Program verification, concurrency, Java, CSP, JCSP

1. Introduction

Hoare's CSP (Communicating Sequential Processes) [10,16,18] is a language for modeling and verifying concurrent systems. CSP has a precise and compositional semantics. On the other hand, the semantics of concurrency in Java [8] (threads) is only given in natural language. Synchronization is based on monitors and data transfer is primarily performed through shared memory; it has turned out that engineering complex programs using these concepts directly is very difficult and error-prone. In addition, verification of such programs is extremely difficult and existing approaches do not scale up well. The JCSP approach [13,20] tries to overcome the difficulties inherent to Java threads. It defines a Java library that offers functions corresponding to the operators of CSP. Using solely JCSP library functions for concurrency and communication (i.e., no explicit creation of threads and no communication via shared memory) allows to verify the (concurrent) behavior of the Java program on the CSP level instead of dealing with monitors on Java level. Since the use of JCSP only makes sense with a strict discipline not to resort directly to Java concurrency features, this should not be a severe restriction.

The paper is organized as follows. In Sect. 2 we give an overview of the architecture of our verification calculus which is presented in detail in Sect. 4–6. In Sect. 3 we present a JCSP implementation which evaluates polynomials and serves as a running example. The verification of some properties of the running example is described in Sect. 7. Finally, in Sect. 8 we relate our verification system to existing approaches and draw conclusions in Sect. 9.

---

[1]Correspondence to: Philipp Rümmer, Dept. of Computer Science and Engineering, Chalmers University of Technology, 412-96 Gothenburg, Sweden. Tel.: +46 (0)31 772 1028; Fax: +46 (0)31 165655; E-mail: philipp@cs.chalmers.se.

JavaCard calculus (1)	CSP model of JCSP (2)
CSP calculus (3)	
Calculus for modal logic correctness assertions (4)	

Figure 1. Architecture of the verification calculus

2. Architecture of Verification Calculus

Our calculus allows to derive truth of temporal correctness assertions of the kind $S : \phi$, where S is a process term and ϕ a formula of some modal logic. The intended semantics is that the process described by S has the property ϕ or, in more technical terms, S describes the Kripke structure ϕ is evaluated in. Our approach is not limited to a particular modal logic. E.g., in the implementation we use an extended version of HML enriched with a least-fixed point operator, which allows to express the liveness-property we proved for the running example presented in Sect. 3. However, in order to explain our approach in this paper we restrict ourselves to plain Hennessy-Milner-Logic (HML) [9] because of its simplicity.

An important part of our proof system is the calculus for the program logic JavaCard Dynamic Logic (JavaCardDL) that is developed in the KeY project [1]. JavaCard [19,5] roughly corresponds to the Java programming language omitting threads and is mainly used for programming smartcards.[1] The KeY tool is a system for deductive verification of JavaCard programs, respectively of Java programs without threads.

Fig. 1 shows the architecture of the verification system, which consists of four components. These correspond to the four stages of the main verification loop:

1. The first stage symbolically executes JavaCard statements until a JCSP library call is reached. This is performed by the standard KeY calculus [1].
   Due to our assumptions that allow only explicit inter-process communication, there is no interference between sequential process code. The sequential calculus from the KeY tool can thus be taken without modification. From a CSP point of view pieces of sequential Java code can be seen as processes that produce only internal events.
2. The second part—operating in parallel with (1)—replaces the JCSP library calls within the program by their CSP models (see Sect. 4).
3. Stage 3 is a rewriting system, which transforms the process term into a normal form that allows to easily deduce the first steps of the process (see Sect. 5).
4. Finally, in stage 4, temporal correctness assertions are evaluated with respect to the possible initial behaviors of the process term (see Sect. 6).

As an important aspect concerning interactive proving, a translation of the considered JCSP program *as a whole* to a different formalism does never take place. Instead, each of the components works as "lazy" as possible, and all layers play together in an interleaved manner.

3. Verification Example

In order to illustrate the programs that can be handled by our verification system we start with describing a simple application, an implementation of Horner's rule [12] in the JCSP framework. The program only makes use of some of the basic JCSP classes; other functionality like processing of integer streams, which is also provided by JCSP, is re-implemented to obtain a self-contained system.

---

[1]JavaCard lacks some more features of Java, e.g. floating point numbers and support for graphical user-interfaces, but also offers support for transactions, which is not available in Java.

```java
import jcsp.lang.*;


abstract class BinGate
    implements CSProcess {
  protected ChannelInputInt input0, input1;
  protected ChannelOutputInt output;
  public BinGate
      ( ChannelInputInt  input0,
        ChannelInputInt  input1,
        ChannelOutputInt output ) {
    this.input0 = input0;
    this.input1 = input1;
    this.output = output;
  }
}


class Adder extends BinGate {
  public Adder
      ( ChannelInputInt  input0,
        ChannelInputInt  input1,
        ChannelOutputInt output ) {
    super ( input0, input1, output );
  }
  public void run () {
    while ( true )
      output.write ( input0.read () +
                     input1.read () );
  }
}


class Multiplier extends BinGate {
  public Multiplier
      ( ChannelInputInt  input0,
        ChannelInputInt  input1,
        ChannelOutputInt output ) {
    super ( input0, input1, output );
  }
  public void run () {
    while ( true )
      output.write ( input0.read () *
                     input1.read () );
  }
}

class Prefix implements CSProcess {
  private int              value, num;
  private ChannelInputInt  input;
  private ChannelOutputInt output;
  public Prefix
      ( int               value,
        int               num,
        ChannelInputInt  input,
        ChannelOutputInt output ) {
    this.value  = value;
    this.num    = num;
    this.input  = input;
    this.output = output;
  }
  public void run () {
    while ( num-- != 0 )
      output.write ( value );
    while ( true )
      output.write ( input.read () );
  }
}
```

```java
class Propagator implements CSProcess {
  private int              delay, num;
  private ChannelInputInt  input;
  private ChannelOutputInt output;
  public Propagator
      ( int               delay,
        int               num,
        ChannelInputInt  input,
        ChannelOutputInt output ) {
    this.delay  = delay;
    this.num    = num;
    this.input  = input;
    this.output = output;
  }
  public void run () {
    while ( delay-- != 0 )
      output.write ( input.read () );
    while ( num-- != 0 )
      CSProcessRaiseEventInt(input.read());
  }
}


class Repeat implements CSProcess {
  private int[]            values;
  private ChannelOutputInt output;
  public Repeat
      ( int[]             values,
        ChannelOutputInt output ) {
    this.values = values;
    this.output = output;
  }
  public void run () {
    int i = 0;
    while ( true ) {
      output.write ( values[i] );
      i = ( i + 1 ) % values.length;
    }
  }
}

public class PolyEval
    implements CSProcess {
  private int[]            values;
  private int              degree, num;
  private ChannelInputInt coeff;
  public PolyEval
      ( int[]             values,
        int               degree,
        int               num,
        ChannelInputInt coeff ) {
    this.values = values;
    this.num    = num;
    this.degree = degree;
    this.coeff  = coeff;
  }
  public void run () {
    One2OneChannelInt[] c =
      One2OneChannelInt.create ( 5 );
    new Parallel (new CSProcess[]
      { new Repeat    (values, c[0]),
        new Prefix    (0, num, c[4], c[1]),
        new Adder     (c[1], coeff, c[2]),
        new Propagator(degree*num, num,
                       c[2], c[3]),
        new Multiplier(c[0], c[3], c[4]) })
    .run ();
  }
}
```

Figure 2. The source code of the verified system for evaluating polynomials (JCSP library classes, interfaces, and method calls are in bold). Apart from the special call CSProcessRaiseEventInt, all classes can directly be compiled using the JCSP library [20] and a recent version of Java. The statement CSProcessRaiseEventInt(v) makes the symbolic JavaCard interpreter implemented in KeY raise an observable CSP event *jcspIntEvent*(v), but does not have any further effects. For actually executing the network, one can for instance replace the statement with System.out.println(v).

The evaluation of polynomials is carried out by a network of 5 gates performing basic operations on streams of integers, which are connected using synchronous JCSP channels. The code of the complete system is given in Fig. 2 and introduces the following classes: `Adder,Multiplier`: Processes that compute point-wise sums and products of integer streams. In contrast to similar classes that are provided by JCSP, pairs of input values are read sequentially and not in parallel, which makes the code a lot shorter and does not affect the functionality of the network in the present setting. `Prefix`: A process that first outputs a fixed integer value `num` times, and afterwards copies its input stream to the output. `Propagator`: A process that copies the first `delay` input values to its output, and that for the subsequent `num` input values v_i raises an observable event $jcspIntEvent(v_i)$. We use such "logging" events to make the result of the computation visible to the formula ϕ of a correctness assertion $S : \phi$. `Repeat`: A process that creates a periodical stream of integers by repeatedly writing the contents of an array to its output. `PolyEval`: The complete network that evaluates a number of polynomials in parallel. The computation result is made observable by an instance of `Propagator`.
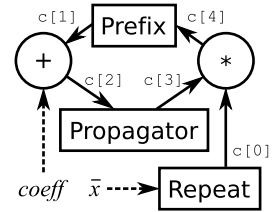
In principle, the cyclic network can be used to evaluate an arbitrary number of polynomials $p_i(x) = c_{i,n}x^n + \cdots + c_{i,0}$ (for $i = 1, \ldots, k$) of the same degree n in parallel. Therefore, the input vector $\bar{x}$ lists the positions $(x_1, \ldots, x_k)$ that are examined, and the network is fed the coefficients of the polynomials through the stream $coeff = (c_{1,n}, c_{2,n}, \ldots, c_{1,n-1}, c_{2,n-1}, \ldots)$. The gates `Prefix` and `Propagator` have to be set up with the correct number k and degree n of the polynomials.

For the purpose of this paper, however, we restrict the capacity of the network by choosing its channels to be zero-buffered. As each of the nodes is only able to store one intermediate result at a time, set up like this the system is bound to lock up as soon as more than three polynomials are evaluated at the same time. This can be observed both by actually executing the Java program and by symbolically simulating the network using our system. Symbolic execution with up to three polynomials is described in Sect. 7.

3.1. Verified Property of the System

When evaluating polynomials $(p_1, \ldots, p_k)$ at points $(x_1, \ldots, x_k)$, the network is expected to produce, after a finite number of (hidden) execution steps, a sequence of distinguished events $jcspIntEvent(p_1(x_1)), \ldots, jcspIntEvent(p_k(x_k))$. In terms of temporal logic, this is captured by the requirement that on every computation path eventually this sequence occurs and is only preceded or interleaved with unobservable steps. The temporal formula describing this behavior is subsequently denoted with $eventually(p_1(x_1), \ldots, p_k(x_k))$ and can for instance be expressed in the modal μ-calculus [4].[2] Verification of this particular kind of properties is for a fixed number of polynomials of fixed degree possible without inductive proof arguments; for handling polynomials of unbounded degree, which lead to an unbounded runtime of the network, induction would be necessary. Since we have not yet investigated the usage of induction techniques (as in [6]) in combination with our verification system, we stick to the simpler scenario and only consider quadratic polynomials in this document.

To set up the verification problem, the coefficients of the polynomials are stored in a buffered JCSP channel, and the network is created with the correct parameters. The resulting program is judged by the temporal formula, which for evaluation of two polynomials in parallel leads to the following proof obligation:

---

[2]At this point HML is not expressive enough, because the number of computation steps is unknown. Here we have enriched HML with a least fixed-point operator borrowed from modal μ-calculus. This extension does not require induction in the calculus.

```
T( jcsp.lang.One2OneChannelInt coeff =
        new jcsp.lang.One2OneChannelInt ( new jcsp.util.ints.BufferInt ( 10 ) );
   coeff.write(c12); coeff.write(c22);
   coeff.write(c11); coeff.write(c21);
   coeff.write(c10); coeff.write(c20);
   new PolyEval ( new int[] { x1, x2 }, 2, 2, coeff ).run ();)
```

$$: \quad \textit{eventually}(\texttt{c12} \cdot \texttt{x1}^2 + \texttt{c11} \cdot \texttt{x1} + \texttt{c10}, \;\; \texttt{c22} \cdot \texttt{x2}^2 + \texttt{c21} \cdot \texttt{x2} + \texttt{c20}) \tag{1}$$

4. CSP Model of JCSP

Process algebras like CSP allow processes to be assembled using algebraic connectives, for instance using interleaving composition ⦀ (we assume familiarity with the CSP notation). JCSP follows this concept roughly, but offers communication means (particularly channels) that only remotely correspond to the operators of CSP. For investigating the behavior of JCSP programs we need a more accurate modeling of JCSP semantics, which we achieve by a (non-trivial) translation of JCSP primitives into CSP. This approach follows ideas from [13], though we are not aiming towards a complete replication of multi-threaded Java but concentrate on JCSP.

The usage of its own interaction features is not strictly enforced by JCSP—for practical reasons—and programs can be written in an "unclean" manner and circumvent JCSP by using shared memory or similar native Java functionality. Since we believe that such programs are not in line with the principles of JCSP, we regard them as ill-shaped. The following models of JCSP operations are simplified insofar as they do not predict the correct behavior of JCSP and Java for ill-shaped programs. Using such a simplified semantics for verification is beneficial because it shortens proofs, but in practice it has to be complemented with checks that prohibit the treatment of ill-shaped programs right from the start. Though we have not yet investigated how to realize such tests, it seems possible to reach a sufficient precision by employing static analysis or type systems to this end (in a completely automated manner).

Our principal idea for modeling JCSP programs is to construct a CSP process term in which sequential Java code can turn up as subterms (wrapped in an operator $T(\cdot)$). JCSP components (such as channels) used to set up the network determine the way in which the sequential Java parts are connected. To illustrate this, the process term representing the scenario of two sequential JCSP processes (implemented as Java programs α, β) that communicate through a JCSP channel is:

$$\Big((id_c : \textit{CHAN}) \, [\![\, id_c.\Sigma \,]\!] \, \big(T(\alpha) \, ⦀ \, T(\beta)\big) \Big) \setminus id_c.\Sigma \tag{2}$$

CHAN is a process modeling the JCSP channel that interfaces with the Java processes $T(\alpha)$, $T(\beta)$ through messages of the alphabet Σ. To distinguish different channels, messages are tagged with an identifier id_c.

4.1. JCSP Processes with Disjoint Memory and their Interfaces

The basis for assembling JCSP systems is to give terms $T(\alpha)$ that wrap Java programs semantics as processes. Therefore, we assume that such a process can only interact with its environment through the use of JCSP operations; this immediately rules out shared-memory communication, or any kind of communication that is not modeled explicitly through observable events raised by $T(\alpha)$.

For defining the behavior of $T(\alpha)$, we equip Java with an operational semantics in which each execution step can 1. transform α into a continuation α', 2. change the memory state of the process $T(\alpha)$, or 3. make $T(\alpha)$ engage in an event a that is observable by the rest of the system (the three possible outcomes do not exclude each other). Designing transition rules for symbolically executing Java code based on this semantics, we were able to start with

the operational semantics of sequential Java that is implemented in the KeY system, which essentially means that we only had to add rules for item 3. Concerning 1 and 2, the behavior of a program follows [19,8].

In JavaCardDL, memory contents are represented during the symbolic execution of a program using so-called *updates*, which are lists of assignments to variables, attributes and arrays. Terms and formulas can be preceded with updates in order to construct the memory contents that are in effect. With updates, for instance, the transition rule for side-effect free assignments is

$$T(\{\texttt{x=e;} \ \ldots\}) \ \leadsto \ \{\texttt{x} := \texttt{e}\}T(\{\ldots\})$$

The KeY system covers the complete JavaCard language and large parts of Java in terms of such transition rules.

Observable events are raised by a process $T(\alpha)$ only when JCSP operations (like channel accesses `c.write(...)`) are executed. The protocol that is followed for communication through a channel is described in Sect. 4.3; a simpler operation is the logging command that is used in Sect. 3 to make results visible. Such operations are handled with additional rules that insert CSP connectives as necessary:

$$T(\{\texttt{CSPProcessRaiseEventInt(v);} \ \ldots\}) \ \leadsto \ jcspIntEvent(\texttt{v}) \to T(\{\ldots\})$$

4.2. Class Parallel

The most basic way of assembling processes in JCSP is the class `Parallel` for parallel composition. Modeling this feature in CSP is rather simple—assuming disjoint memory for processes—and boils down to inserting the interleaving operator $\vertiii{}$. The magic operation that has to be trapped is `Parallel.run`, because this is the place where new processes are actually spawned. For an object `parallel` that is set up with children processes `p1`, ..., `pn`, the effect of the `run`-method can be modeled in CSP as follows:

$$T(\{\texttt{parallel.run();} \ \ldots\}) \ \leadsto$$
$$\big(T(\{\texttt{p1.run();}\}) \vertiii{} \cdots \vertiii{} T(\{\texttt{pn.run();}\})\big); \ T(\{\ldots\})$$

Sequential composition ; is used to make the parent process continue its execution after termination of the children. Because memory contents are stored in updates in front of terms $T(\alpha)$, each of the processes that are created will inherit the memory of the parent process, but will consecutively operate on a copy of that memory: write access of the programs `pi` are not visible to other processes.

4.3. Channels

We model the different kinds of channels that are provided by the JCSP library—which differ in the way data is buffered and have different access arbitration—following ideas from [13]. As already shown in Eq. (2), the behavior of a channel is simulated by an explicit routing process *CHAN* that is attached to a Java process as a slave. As a starting point, we adopted the CSP model from [13] of a zero-buffered and synchronous channel (Fig. 3):

$$LEFT \ = \ write\,?\,msg \to transmit\,!\,msg \to ack \to LEFT$$
$$RIGHT \ = \ ready \to transmit\,?\,msg \to read\,!\,msg \to RIGHT$$
$$ONE2ONECHANNEL \ = \ \big(LEFT \,\llbracket\, transmit.\Sigma \,\rrbracket\, RIGHT\big) \setminus transmit.\Sigma \qquad (3)$$

Our implementation contains further channel models, for instance an extended version of the model shown here that also supports the JCSP alternation operator. Channels with bounded buffering (as used in the example Fig. 2) can be handled by the system as well. However, a complete set of CSP characterizations for the JCSP channels, together with a systematic verification that the models faithfully represent the actual JCSP library is still to be developed.

Figure 3. Model of a zero-buffered channel

The JCSP operations for creating and accessing channels are again realized by translating them to CSP connectives. Channels are created by allocating a new channel identifier id_c (which in our implementation is just the reference to the created object of class `One2OneChannel`) and by spawning the appropriate routing process:

$$T(\{\texttt{c=new One2OneChannel(); ...}\}) \rightsquigarrow$$

$$\left(\left(id_c : ONE2ONECHANNEL \right) \| id_c.\Sigma \| \left(\{\texttt{c} := id_c\} T(\{\ldots\}) \right) \right) \setminus id_c.\Sigma$$

The Java process can then interact with the channel according to a certain protocol, which for the zero-buffered channel looks as follows.

$$T(\{\texttt{c.write(o);...}\}) \rightsquigarrow id_c.write\,!\,msg_o \rightarrow id_c.ack \rightarrow T(\{\ldots\})$$

$$T(\{\texttt{o=c.read();...}\}) \rightsquigarrow id_c.ready \rightarrow id_c.read\,?\,msg_o \rightarrow \{\texttt{o} := \ldots\} T(\{\ldots\})$$

Because of the disjoint-memory assumption it is necessary to encode the complete information that messages contain as some term msg_o, which we have so far implemented for integers (in combination with the JCSP channels for integers that for instance are used in Fig. 2). Treating arbitrary objects is possible through manipulations of updates and will be added to the proof system in a later version.

5. CSP calculus

The gist of evaluating HML-assertions for processes is that certain events can or have to be fired in a given state. It is thus crucial to obtain, for the process term at hand, the summary of events that it can fire in the next step and the corresponding process continuations. This goal is usually achieved by rewriting the process term into a certain normal form, from which this information can be syntactically gleaned.

When working with a naive total-order semantics, a typical exploration (rewriting) of a process term (here the interleaving of two processes) looks like this:

$$a \rightarrow P \,\|\| \, b \rightarrow Q \quad \rightsquigarrow \quad a \rightarrow (P \,\|\| \, b \rightarrow Q) \,\Box\, b \rightarrow (a \rightarrow P \,\|\| \, Q)$$

The subterms P and Q are duplicated, and in general the term size increases exponentially.

On the other hand, Petri nets have been used in the past to give processes a partial-order semantics (also called step semantics) [3]. The net approach avoids a total ordering of independent events, which helps containing the state explosion. The representation of a transition system as a net graph is also usually more compact than a tree. Following this tradition, we combine Petri nets and conventional process terms into one formalism (we call it netCSP), which allows succinct reasoning. We model CSP events as net transitions, and the evolution of the net marking corresponds to the derivation of adjacent processes that are reached when a process performs activated execution steps
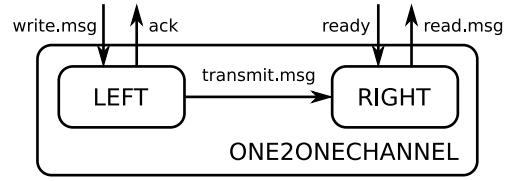
netCSP terms are built-up incrementally from the conventional CSP process terms by the rewriting system outlined in the following. The incremental, or "lazy", manner of exploration allows to have Java programs inside processes, since finite nets are not Turing-complete. It is the first (to our knowledge) rewriting system for efficiently creating combined process representations from conventional ones, and for exploring their behavior.

5.1. Monotonic Petri nets

Petri nets (see [15] for an introduction) are a formal and graphically appealing model long used for modeling non-sequential processes. To model CSP process behavior in a faithful and efficient way we introduce a slightly modified version of Petri nets, which

Figure 4. Life cycle of a place marking

we call *monotonic Petri nets*. Every place in such a net is in one of the three following states: empty (E), marked (M), or dead (D). A transition t of a monotonic Petri net is called *enabled* for a marking M (a mapping from places to states), if all its input places *in* are marked and all its output places *out* are empty:

$$M(in(t)) \subseteq \{M\} \ \wedge \ M(out(t)) \subseteq \{E\}$$

An enabled transition t can *fire* leading to a new marking, which for a place p is

$$M_{\text{new}}(p) \quad := \quad \begin{cases} D & \text{if } p \in in(t) \\ M & \text{if } p \in out(t) \\ M(p) & \text{otherwise} \end{cases}$$

Thus, a marking of a place can only evolve in monotonic progression as depicted in Figure 4. This allows far-reaching estimations on the behavior of the net (e.g. places depending on dead places are blocked forever). Another immediate and favorable consequence of the above net semantics is the fact that every non-isolated transition can fire at most once, just as any particular CSP event can only be raised once. Finally, since monotonic nets are easily translated to standard 1-safe Petri nets, all common analysis techniques are still available.

5.2. netCSP: Combining Nets and Process Terms

The combination of conventional process terms and Petri nets is described algebraically by enriching the set of usual CSP operators with the following four:

^{i}P Token consumption: this term attaches a CSP process P to the place i of the net. The execution of P is now causally dependent on i. If i is marked with E then P is currently blocked. P is not blocked if i is marked with M. Then execution of P consumes the token in i. If i is marked with D then P is blocked forever (and can be removed). In lieu of a single place i a set of places can turn up. In this case a token is consumed from every place.

$^{i}a^{o}$ The transition operator expresses that a CSP event a is raised by the term, whilst a causal dependency token is consumed from place i and placed in place o. Again, sets of places can play the role of i and o.

$p[v] : P$ The causal state operator sets the marking of the place p in P to value v (which is one of E, M, or D).

$P \parallel L[X]R \parallel Q$ This consruct is a "bookkeeping" version of the standard parallelism operator $P \parallel X \parallel R$, see Section 5.3.4.

The new operators are initially introduced by the rewriting system, which transforms conventional CSP terms into the combined representation. This rewriting system is described in the following section.

5.3. Rewriting System For Exploring Process Behavior

5.3.1. The Desired Normal Form

The rewriting system presented in this section transforms a CSP or a netCSP term into the following normal form (together with an implied marking M):

$$ {}^{i_1}a_1{}^{o_1} \;|||\; \cdots \;|||\; {}^{i_n}a_n{}^{o_n} \;|||\; R \tag{NF} $$

where ${}^{i_k}a_k{}^{o_k}$ are enabled transitions, and the remainder R is blocked w.r.t. M, i.e., cannot raise an event at the current stage. The latter condition can be checked by a simple syntactic criterion on M due to the benign properties of monotonic nets described above.

    The rewriting system achieves the normal form (NF) by pulling transitions out of the scope of the leading operator and moving them towards the root of the term. Since terms are finite, this procedure is guaranteed to terminate.

Example 1 Rewriting the channel routing process *ONE2ONECHANNEL* that is defined in Sect. 4.3, Eq. (3) to normal form yields the following term (p and q are initially empty):

$$ C \;=\; ready^{\{p\}} \;|||\; write\,?\,msg^{\{q\}} \;|||\; $$
$$ \underbrace{\left({}^{\{p\}}(transmit\,!\,msg \to \cdots) \,[\![\, transmit.\Sigma \,]\!]\, {}^{\{q\}}(transmit\,?\,msg \to \cdots) \right) \setminus transmit.\Sigma}_{R \text{ (currently blocked)}} $$

In graphical representation:
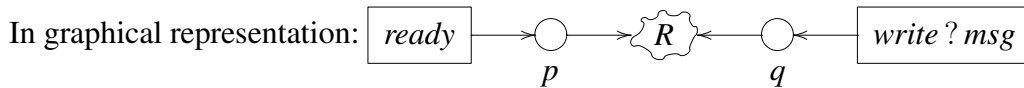
The first steps of the process C are thus either *ready* or *write ? msg*.

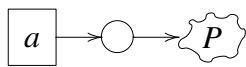### 5.3.2. Translating Events (Prefix Operator)

Figure 5. CSP events as net transitions

Events are modeled as transitions of the Petri net. Firing of a transition corresponds naturally to the process' engagement in an event. This transformation is captured by the following rule:

$$ a \to P \quad \rightsquigarrow \quad p[\mathrm{E}] : (a^{\{p\}} \;|||\; {}^{\{p\}}P), \; p \text{ new in } P $$

In practice, the rewriting strategy would, sensibly, start applying this rule at the leftmost possible position in a term.
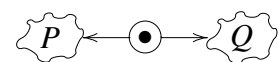
5.3.3. Translating the Choice Operator

Figure 6. Nondeterministic choice

The choice operator also lends itself to a natural representation in the Petri net process framework. This is achieved by the following rule:

$$ P \,\square\, Q \quad \rightsquigarrow \quad p[\mathrm{M}] : ({}^{\{p\}}P \;|||\; {}^{\{p\}}Q), \; p \text{ new in } P \text{ and } Q $$

5.3.4. Translating the Parallelism Operator

The behavior of the parallelism operator $P \, [\![X]\!] \, R$ varies with the synchronization set X from total synchronization of two processes ($P \parallel Q$) to interleaving ($P \parallel\!\parallel Q$). Interleaving has a special place within this scale as it introduces no dependencies between its operands. It is treated separately in the next section.

Here, in contrast, we assume that the synchronization set X is not empty. For events included in X we identify "matching" transitions in both operands and "merge" them outside of the scope of the parallelism operator. Since removing transitions out of the scope loses vital information, it is necessary to do some additional bookkeeping. This is achieved with two lists of already worked-off transitions ("buffers") L and R, which are part of the extended operator $[\![L[X]R]\!]$. In the beginning, our rewriting system replaces the parallelism operator by this variant with the buffers initially empty:

$$ P \, [\![X]\!] \, Q \quad \rightsquigarrow \quad P \, [\![\emptyset[X]\emptyset]\!] \, Q $$

The main rewriting step then records every (synchronized) worked-off transition from an operand in the corresponding buffer:

$$ P \, [\![L[X]R]\!] \, \left({}^i a^o \parallel\!\parallel Q \right) \quad \rightsquigarrow \quad \begin{cases} {}^i a^o \parallel\!\parallel \left(P \, [\![L[X]R]\!] \, Q \right) & \text{if } a \notin X \\ U \parallel\!\parallel \left(P \, [\![L[X]R']\!] \, Q \right) & \text{if } a \in X \end{cases} $$

where $R' := R \uplus \{ {}^i a^o \}$ and U is an interleaving of transitions, which arises from merging ${}^i a^o$ with all transitions of the same name in buffer L:

$$ U \quad := \quad \underset{{}^{i_l} a^{o_l} \in L}{\parallel\!\parallel} \; {}^{i \cup i_l} a^{o \cup o_l} $$

The *stop* process can stand in for an empty Q, and a symmetrical rule can be given for the left operand.

Example 2 We continue Example 1 and complement term C with a process $ready \rightarrow Q$ that accesses the channel for reading. By repeatedly applying the rule for handling parallelism, pending events are added to the buffers of the parallelism operator, and it is deduced that the whole system can engage in event *ready* as its first step. The buffer contents are underlined.

$$ C \, [\![\Sigma]\!] \, \left(ready \rightarrow Q \right) \quad \rightsquigarrow \quad \cdots \quad \rightsquigarrow \quad r[\mathrm{E}] : \left(C \, [\![\Sigma]\!] \, \left(ready^{\{r\}} \parallel\!\parallel {}^{\{r\}}Q \right) \right) $$

$$ \rightsquigarrow \quad \cdots \quad \rightsquigarrow \quad r[\mathrm{E}] : \left(ready^{\{p,r\}} \parallel\!\parallel \left(R \, [\![\, \underline{ready^{\{p\}}, write \, ? \, msg^{\{q\}}} \, [\Sigma] \underline{ready^{\{r\}}} \,]\!] \, {}^{\{r\}}Q \right) \right) $$

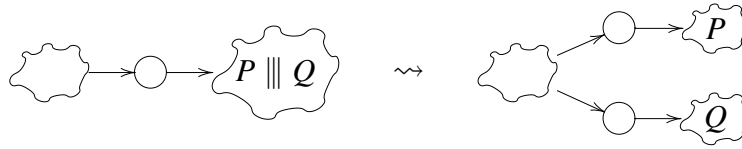In the following net diagram, buffered transitions are denoted with dashed boxes:
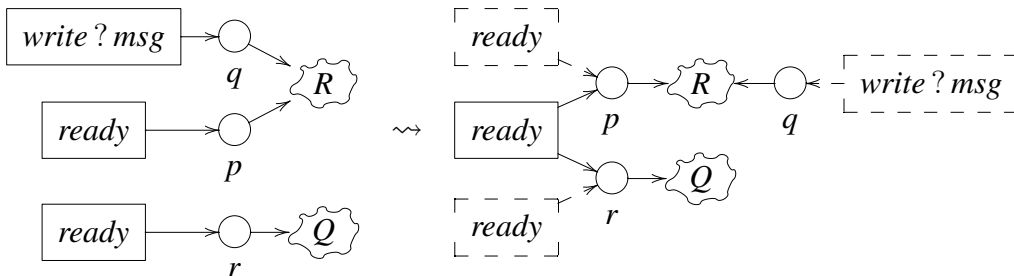
Figure 7. Interleaving of processes is easy

5.3.5. Translating Interleaving

The interleaving composition of two processes ($A \parallel\!\parallel B$) builds a "base case" of the rewriting system. It has a very natural Petri net representation, due to the concurrency inherent to Petri nets. This way $A \parallel\!\parallel B$ can be translated with the nets for A and B simply written side by side. Care should be taken though while connecting to other processes. In this case the interface places have to be duplicated, as well as the connecting transitions. This is described with the rule shown in Fig. 7. Due to lack of space we refrain from formally stating the rule and refer to [17] where a straightforward but lengthy formulation is given.

5.3.6. Further CSP Operators

The CSP operators for labeling, hiding, and message passing (e.g., $a\,?\,x \rightarrow P$) are also treated by the system, but omitted here for space reasons.

5.3.7. Correctness of the Rewriting System

We have shown the correctness of our rewriting system, by first developing a coalgebra-based denotational semantics of the process algebra at hand (based on Roscoe's SOS [16]). Then we have proved that our rewriting system preserves the meaning of process terms relative to this semantics. This result is documented in [17].

6. Evaluation of Temporal Correctness Assertions

In this section we consider generalized correctness assertions of the form $S : M : \phi$ where S is a netCSP-term, M its initial marking, and ϕ is a formula of some modal logic. Here we use HML for simplicity reasons, but more expressive logics like temporal logic or μ-calculus can be handled as well.

   The syntax of HML is defined by the grammar

$$For_{\text{HML}} ::= \mathit{true} \mid \neg\, For_{\text{HML}} \mid For_{\text{HML}} \wedge For_{\text{HML}} \mid \langle Event \rangle For_{\text{HML}}$$

where *Event* ranges over a set of events. The meaning of the Boolean connectives is as usual; formula $\langle a \rangle \phi$ holds *iff* the concerned process, by engaging in an event a, reaches a state, in which ϕ holds.

   Tab. 1 shows some HML correctness assertions and their truth values. Two of the correctness assertions evaluate to ff. The reason is that in both cases place o is already marked and, as a consequence, event a cannot be fired (since firing a requires place o to be empty).

6.1. Evaluation of netCSP Terms in Normal Form

The rules of the calculus presented in Sect. 5 transform a netCSP term into the normal form (NF) and a corresponding marking M (implied):

$$^{i_1}a_1{}^{o_1} \parallel\!\parallel \cdots \parallel\!\parallel {}^{i_n}a_n{}^{o_n} \parallel\!\parallel R, \qquad \text{and } R \text{ is blocked w.r.t. } M$$

that is an efficient syntactical representation of the possible first events the process may fire. Now calculus rules for evaluating HML correctness assertions can be applied. We use a

Gentzen-style sequent calculus. Sequents are of the form $\Gamma \vdash \Delta$ where Γ and Δ are multi-sets of correctness assertions. The semantics of a sequent is that the conjunction of the correctness assertions on the left of the sequent symbol $\vdash$ implies the disjunction of the assertions on the right.

The semantics of a sequent calculus rule is that if the premises (i.e., the sequents above the horizontal line) can be derived in the calculus then the conclusion (i.e., the sequent below the line) can be derived as well. Note, that in practice sequent rules are applied from bottom to top. The following rule allows to evaluate HML correctness assertions. Applied from bottom to top, it produces a number of new correctness assertions about the continuations of the process that have to be examined subsequently.

$$\frac{\Gamma \vdash \bigvee_{\substack{k=1,\ldots,n \\ (i_k,o_k)\in\text{En}(M)}} a_k \doteq b \wedge \left({}^{i_1}a_1{}^{o_1} \parallel\!\parallel \cdots \parallel\!\parallel {}^{i_n}a_n{}^{o_n} \parallel\!\mid R \right) : \left(M + (i_k,o_k) \right) : \Phi, \ \Delta}{\Gamma \vdash \left({}^{i_1}a_1{}^{o_1} \parallel\!\parallel \cdots \parallel\!\parallel {}^{i_n}a_n{}^{o_n} \parallel\!\mid R \right) : M : \langle b \rangle \Phi, \ \Delta} \ (\parallel\!\mid R)$$

The rule considers all transitions a_k which are enabled, i.e., input places are marked and output places are empty ($(i_k, o_k) \in \text{En}(M)$). The expression $M + (i_k, o_k)$ denotes the new marking after transition a_k has fired.

As an example we derive the HML correctness assertion

$$ {}^{\{i_1\}}a \ \parallel\!\mid \ {}^{\{i_2\}}a \ : (M, M) : \langle a \rangle \langle a \rangle \textit{true}$$

expressing that there is a possibility for the process ${}^{\{i_1\}}a \parallel\!\mid {}^{\{i_2\}}a$ with initial marking (M, M) to fire two consecutive events a. Markings M are here represented as pairs $(M(i_1), M(i_2))$ since the process term only contains the places i_1 and i_2 (we assume $i_1 \neq i_2$). A proof using rule $(\parallel\!\mid R)$ contains redundancy since the only difference between the newly generated correctness assertions is their marking. Both, process term and HML-formula stay the same. Thus, an obvious improvement is to consider correctness assertions with sets of markings. Then the example from above can be derived more efficiently:

$$\frac{\dfrac{*}{\vdash {}^{\{i_1\}}a \parallel\!\mid {}^{\{i_2\}}a : \{(D, D)\} : \textit{true}} \ (\textit{true } R)}{\dfrac{\vdash {}^{\{i_1\}}a \parallel\!\mid {}^{\{i_2\}}a : \{(D, M), (M, D)\} : \langle a \rangle \textit{true}}{\vdash {}^{\{i_1\}}a \parallel\!\mid {}^{\{i_2\}}a : (M, M) : \langle a \rangle \langle a \rangle \textit{true}} \ (\parallel\!\mid R)} \ (\parallel\!\mid R)$$

7. Verifying the Example

After loading proof goal (1) into the KeY prover its verification proceeds without further user interaction. Automated application of rules is in KeY controlled by so-called *strategies*,

Table 1. Examples of HML correctness assertions

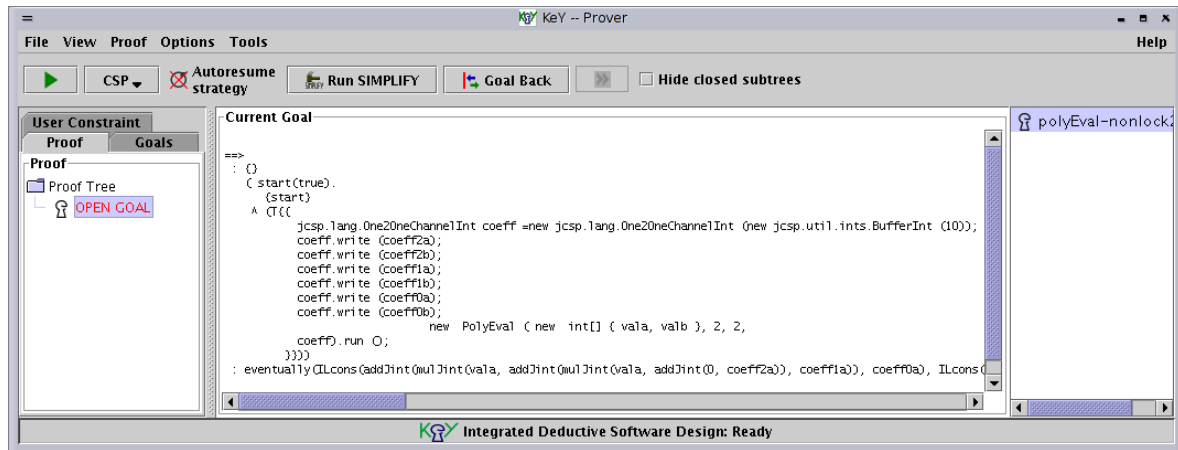| netCSP term S	initial marking $M(o)$	HML formula ϕ	truth value
$a^{\{o\}}$	E	$\langle a \rangle \textit{true}$	tt
	M	$\langle a \rangle \textit{true}$	ff
$a^{\{o\}} \parallel\!\mid {}^{\{o\}}b$	E	$\langle a \rangle \langle b \rangle \textit{true}$	tt
	M	$\langle a \rangle \langle b \rangle \textit{true}$	ff
	M	$\langle b \rangle \textit{true}$	tt

Figure 8. The KeY prover after loading the verification example

Table 2. Number of rule applications and invocations of JCSP primitives for evaluation of polynomials

# Polynomials:	1	2	3
Rule applications in total	23551	40647	57047
`One2OneChannelInt.read`	19	34	49
`One2OneChannelInt.write`	17	32	47
`new ZeroBufferInt`	5	5	5
`new BufferInt`	1	1	1
`Parallel.run`	1	1	1

which in each proof situation select a particular rule that is supposed to be applied next. For the example we are using a strategy that is implemented as described in Sect. 6, which eventually reduces (1) to the tautology *true*, proving that the stated property holds.

7.1. Shape of the Proof

During execution of the polynomial evaluation program essentially two phases can be identified: In a first part, the network is set up, i.e., JCSP processes are spawned and channels are created. The symbolic execution thereof needs about 7000 applications of rules and results in a CSP process term that contains 6 JCSP processes—the gates that make up the network as well as the network itself—and 6 further subterms modeling the JCSP channels according to the concept from Sect. 4. On the JavaCard level, this corresponds to 22 objects being created, of which 2 are arrays and the remaining 20 mostly belong to (the internal implementation) of channels.

The second phase covers the execution of the initialized network; the number of rule applications necessary in this part depends on how many polynomials are evaluated in parallel (see Tab. 2). Further processes are not spawned in this part of the proof, which means that the shape of the CSP term is mostly preserved. Consequently, the proof gives a good presentation of the step-wise execution of the network—similarly to what can be achieved with a debugger—that is moreover completely symbolic. The second phase ends with a sequence of events $jcspIntEvent(p_1(x_1))$, ..., $jcspIntEvent(p_k(x_k))$ raised by an instance of class `Propagator` and this completes the whole proof.

Tab. 2 gives an overview about the JCSP primitives that are invoked during the progression of the network. The `write` primitive is called less often than `read`, as some of the gates are already waiting for their next input (in vain) when the proof is closed.

The verification for one polynomial takes about 30min on a common desktop computer (Pentium4, 2.6GHz), and is mostly determined by the currently limited performance of KeY

when dealing with very large terms like the netCSP process term during symbolic execution. More generally, the required time depends on each of the four components of the verification system of Sect. 2. For mostly deterministic programs, symbolic execution (parts (1), (2), (3)) will be the dominating factor, which scales essentially linear in the code length, whereas for indeterministic programs the exploration of the state space (part (4)) becomes more costly. We currently only have a naive implementation of the techniques described in Sect. 6, which makes the verification time climb to about 5h when treating two or three polynomials simultaneously in our example.

8. Related Work

To our knowledge, this paper describes the first verification system for Java programs in combination with the JCSP library.

An approach that has already been investigated, in contrast, is the automatic generation of JCSP programs from verified "pure" CSP implementations, as for instance [14]. For JCSP systems that happen to be created this way it can be expected that verification is much simpler and can be handled more efficiently, as interpretation of Java code is avoided. We have not compared performance empirically as we consider the two problems too different.

A further direction is the modeling of native Java concurrency features in CSP as a basis for verification, which is performed in [13]. Again, this idea differs significantly from the concept underlying our system.

The EVT system [2] provides a verification environment for Erlang programs based on the first-order μ-calculus. Similar to our method is the usage of temporal correctness assertions in EVT, and we expect that many results derived in the EVT project—particularly concerning induction for the μ-calculus and compositional verification—can also be useful for verifying JCSP programs.

A combination of Petri nets and process algebra is investigated in [3], and the algebra netCSP is designed following this idea to a considerable degree. Apart from that, the comparison of process algebra and Petri nets has a long tradition, see for instance [7]. A translation of CSP process terms to Petri nets comparable to our calculus for netCSP is outlined in [11] (but without integrating the two formalism into one language and giving a rewriting system), where the Petri net representation is used for analysis purposes.

9. Conclusion

We have presented a complete verification approach for concurrent Java or JavaCard programs written using the JCSP library. The method has been implemented on top of the KeY system for deductive verification of Java programs and can be applied for ensuring properties of real-world programs, with the restriction that concurrency in the programs must be implemented purely using JCSP functionality instead of the corresponding native Java features (like shared memory).

Our verification system consists of four different layers that are mostly orthogonal to each other, and that can all be realized or developed further independently. The basis is a calculus for the symbolic execution of sequential Java programs, which in our implementation is the already existing (complete) symbolic interpreter of the KeY prover. This interpreter is lifted to the concurrent case by embedding sequential Java programs in CSP terms. In order to make the execution of JCSP primitives possible, we add CSP models of JCSP classes and methods: currently a selection of different JCSP channels, alternation, and the most important JCSP process combinator (parallelism) are supported.

These first two components enable an incremental translation of JCSP programs to CSP terms. The behavior of such terms (resp. the represented processes) is explored stepwise by a calculus for CSP, for which we have chosen a rewriting system that operates on an extension of CSP (called netCSP) integrating process algebra with Petri nets. The usage of Petri nets at this point avoids an early total ordering of execution steps and has in our implementation found to be by far more efficient than rewriting systems establishing tree-shaped normal forms of CSP terms.

In a last phase, the behavior of the CSP process is checked against a temporal specification. That issue is discussed for the particularly simple logic HML in this paper, which can be regarded as basis for practically more relevant temporal logics like the μ-calculus.

Apart from the interpreter for sequential Java, we consider each of the components of the verification system as target of future work: 1. Complement the set of supported JCSP features and verify that the CSP models are faithful, 2. improve the netCSP calculus by integrating Petri net reachability analysis, which can be used to simplify process terms, 3. add complete support for more powerful temporal logics and induction, 4. investigate how our method can be combined with compositional verification techniques as for instance described in [6].

Acknowledgement

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] T. Arts, G. Chugunov, M. Dam, L. å. Fredlund, D. Gurov, and T. Noll. A tool for verifying software written in erlang. *Int. Journal of Software Tools for Technology Transfer*, 4(4):405–420, August 2003.

[3] J.C.M. Baeten and T. Basten. Partial-order process algebra (and its relation to Petri nets). In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, North-Holland, 2001.

[4] Julian Bradfield and Colin Stirling. Modal logics and mu-calculi: an introduction. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*. Elsevier, North-Holland, 2001.

[5] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, 2000.

[6] M. Dam and D. Gurov. Mu-calculus with explicit points and approximations. *Journal of Logic and Computation*, 12(2):255–269, April 2002. Abstract in Proc. FICS'00.

[7] U. Goltz. *On Representing CCS Programs by Finite Petri Nets*. Number 290 in Arbeitspapiere der GMD. Gesellschaft für Mathematik und Datenverarbeitung mbH, Sankt Augustin, 1987.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.

[9] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309. Springer-Verlag, 1980.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985. & 0-13-153289-8.

[11] Krishna M. Kavi, Frederick T. Sheldon, and Sherman Reed. Specification and analysis of real-time systems using CSP and Petri nets. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):229–248, 1996.

[12] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*. Addison-Wesley, 1997. Third edition.

[13] P.H.Welch and J.M.R.Martin. A CSP Model for Java Multithreading. In P.Nixon and I.Ritchie, editors, *Software Engineering for Parallel and Distributed Systems*, pages 114–122. ICSE 2000, IEEE Computer Society Press, June 2000.

[14] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 63–81, 2003.

[15] Wolfgang Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., 1985.

[16] A. W. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1998.

[17] Philipp Rümmer. Interactive verification of JCSP programs. Technical Report 2005–01, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2005. Available at: `http://www.cs.chalmers.se/~philipp/publications/jcsp-tr.ps.gz`.

[18] Steve Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. John Wiley & Sons Ltd., 2000.

[19] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Platform Specification*, September 2002.

[20] P.H. Welch and P.D. Austin. *Java Communicating Sequential Processes home page*. `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.