Teaching Academic Concurrency to Amazing Students

Sebastian Biewer[1,2], Felix Freiberger[1,2], Pascal Leo Held[1], and Holger Hermanns[1(✉)]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
hermanns@cs.uni-saarland.de

[2] Saarbrücken Graduate School of Computer Science, Saarbrücken, Germany

Abstract. Milner's CCS is a cornerstone of concurrency theory. This paper presents CCS as a cornerstone of concurrency practice. CCS is the semantic footing of pseuCo, an academic programming language designed to teach concurrent programming. The language features a heavily simplified Java-like look and feel. It supports shared-memory as well as message-passing concurrent programming primitives, the latter being inspired by the Go programming language. The behaviour of pseuCo programs is described by a formal translational semantics mapping on value-passing CCS and made executable using compilation to Java. pseuCo is not only a language but an interactive experience: pseuCo.com provides access to a web application designed for first hands-on experiences with CCS and with concurrent programming patterns, supported by a rich and growing toolset. It provides an environment for students to experiment with and understand the mechanics of the fundamental building blocks of concurrency theory and concurrent programming based on a complete model of the program behaviour. Altogether this implements the TACAS (Teaching Academic Concurrency to Amazing Students) vision.

1 Introduction

In our times, *concurrency* is a topic that affects computing more than ever before. The *Calculus of Communicating Systems*, CCS, is a foundational pillar of concurrency theory, developed by Robin Milner in Edinburgh [21,22] in the 80ies of the last century. In this period, Kim Larsen was working towards his PhD thesis under the guidance of Milner [17]. And Rance Cleaveland, Joachim Parrow, and Bernhard Steffen were working on a verification tool for CCS, the Concurrency Workbench [6,7].

The Concurrency Workbench is an automated tool to cater for the analysis of networks of finite-state processes expressed in Milner's Calculus of Communicating Systems. It was part of a first wave of initiatives providing tool support for process algebraic principles. Other initiatives at that time included the AUTO/AUTOGRAPH project by Robert de Simone and Didier Vergamini [3] in Sophia Antipolis, the tool VENUS [26] by Amelia Soriano, as well as

L. Aceto et al. (Eds.): Larsen Festschrift, LNCS 10460, pp. 170–195, 2017.
DOI: 10.1007/978-3-319-63121-9_9

the Caesar/Aldebaran tools developed by Hubert Garavel and coworkers in Grenoble [10]. The latter focussed on LOTOS, an ISO standard developed by a committee around Ed Brinksma [2,16] from Twente. In turn, AUTOGRAPH pioneered graph visualisation and animation using early versions of tcl/tk. All three initiatives provided inspirations, in one way or another, to Kim Larsen in his early efforts to pioneer tool support for real-time system verification [18], the topic for which he would later become world famous [4]. And Bernhard, Ed, Kim, and Rance later joined forces to found TACAS, a scientific conference that is considered the flagship of European verification research by many.

In subsequent years, further tools emerged, such as the FDR toolset [24] supporting Hoare's CSP approach to concurrency theory, or the PAC tool [5] that aimed at providing a front-end for different process algebras via instantiation of the individual operational semantics. Also, the principles behind the Concurrency Workbench were further developed by Rance Cleaveland in North Carolina [8]. Lately, an Aalborg edition of the workbench has been announced [1] for the purpose of teaching concurrency theory with CCS [19].

At the turn of the millennium, Jeff Kramer and Jeff Magee proposed a new level of tool support for process calculi as part of their textbook on "Concurrency – state models and Java programs" [20]. This book came with an easy-to-install and ready-to-use tool LTSA supporting their language FSP, a CSP-inspired process calculus. The (to our opinion) most remarkable aspect of this toolset was the deep integration of process-algebraic thinking into a lecture concept introducing concurrency practice: To develop a thorough understanding of concurrent programming principles and pitfalls, informal descriptions and concrete Java examples were paired with abstract FSP models, readily supported by the LTSA tool.

The present paper follows this very line of work by presenting an even deeper integration of concurrency theory into concurrency practice for the purpose of teaching concurrent programming. It revolves around a programming language called pseuCo. The language features a heavily simplified Java-like look and feel. It supports shared-memory as well as message-passing concurrent programming primitives, the latter being inspired by the Go programming language. The behaviour of pseuCo programs is described by a formal translational semantics mapping on value-passing CCS and made executable using compilation to Java. pseuCo is not only a language, it is an interactive experience: pseuCo.com provides access to a web application designed for first hands-on experiences with CCS and with concurrent programming patterns, supported by a rich and growing toolset. It provides an environment for students to experiment with and understand the mechanics of the fundamental building blocks of concurrency theory and concurrent programming based on a complete model of the program behaviour. This platform provides access to the tools targeting pseuCo, most notably: the pseuCo-to-Java compiler, the translation of pseuCo programs to CCS, the CCS semantics in terms of LTS, and more.

2 The Concepts Behind pseuCo

This section describes the context, features, and semantic embedding of pseuCo.

2.1 Context

A profound understanding of concurrency has to be part of the basic repertoire of every computer scientist. Concurrency phenomena are omnipresent in databases, communication networks, and operating systems, in multi-core computing, and massively parallel graphics systems, as well as in emerging fields such as computational biology. Nowadays, software developers are confronted with concurrency problems on a daily basis, problems which are notoriously difficult to handle. Therefore, competence in this field is a must for every computer scientist. Unlike sequential systems, even non-critical applications can no longer be adequately tested for functional correctness. Therefore, it is indispensable that formal verification procedures are known, at least conceptually, to every undergraduate computer science student we educate. A solid theoretical underpinning of the matter and its interrelation with the practice of concurrent programming is a necessary prerequisite.

A Lecture on Concurrent Programming. For this purpose, the lecture "Concurrent Programming" at Saarland University develops these competences starting off with a solid explanation of concurrency theory and then lifts and intertwines them with practical aspects of concurrent programming. The lecture is a mandatory module worth 6 ECTS points in the Bachelor education of computer science and related fields and is currently in its tenth edition. It is scheduled at the end of the second year but we encourage talented students to already enrol into it at the end of their first year. It received the 2013 award for innovations in teaching from the *Fakultätentag Informatik*, the association of German computer science faculties.

After an extensive motivation which stresses the relevance of the matter, the students embark into the basics of CCS including syntax, labelled transition systems, operational semantics, trace equivalence, strong bisimulation, and observational congruence, and finally value-passing CCS. At various places along the lecture, the understanding is supported by the CCS view of pseuCo.com shown in Fig. 3.

At this point, the main innovation of our approach gradually enters the stage: pseuCo. Contrary to CCS, pseuCo is a real programming language. It supports both the shared-memory and message-passing programming paradigms. In order to connect concurrency theory and concurrency practice, pseuCo has a translational semantics mapping on value-passing CCS.

Listing 1.1: Distortion-tolerant transmission protocol in CCS

```
1  range Dig := 0..9 // defining a finite range of values
2
3  Sender := put?x. send!x. Sending[x]      // take the value,
   ↳ send it out, remember it
4  Sending[x] := ack?. Sender               // forget value if ok
5              + nAck?. send!x. Sending[x] // resend if not ok
6
7  Medium[snd] := snd?x:Dig. (receive!x.1 + i. garbled!.1);
   ↳ Medium[snd]
8
9  Receiver[n] := receive?x:0..9. get!x. ack!. Receiver[n+1]
10              + garbled?. nAck!. Receiver[n]
11              + when (n==4) println!"succ"^"ess".0
12
13 Protocol := (Sender | (Receiver[0]) | Medium[send]) \ {send,
   ↳ receive,ack,nAck,garbled}
14
15 (Protocol | put!2. put!4. put!2. put!8) \ {put}
```

CCS with Value-Passing. We will base the discussion of our pragmatic extension of CCS on the example shown in Listing 1.1. It describes a simple distortion-tolerant transmission protocol between a sender and a receiver. While the data is transmitted over a medium that may distort messages, for simplicity, acknowledgements are assumed to travel directly from receiver to sender.

The first line defines a finite range of values to be used later. In line 3, we see the defining equation of the `Sender` process. It receives a value `x` and continues by sending out that value with the action `send`. It then turns into the process `Sending` defined in line 4 which is parametric in the value `x` so as to make it possible to remember the value in case retransmissions are needed. This demonstrates that process definitions can be parametrised by data values. Our CCS dialect allows Booleans, integers or strings as process parameters. The `Sending` process waits for either a positive acknowledgement and then returns to being a `Sender` or for a negative acknowledgement which triggers a retransmission of the value.

Line 7 defines the `Medium` over which `Sender` and `Receiver` communicate data. First, the `Medium` receives a value that was sent with the action `snd!`. After having received a value in the range defined before, the `Medium` decides nondeterministically to either pass it on or to instead transmit a (distorted) `garbled!` message. In both cases, this part of the process ends as the special process `1` indicating successful termination. This allows the sequence operator `;` to continue to a fresh `Medium` which waits for the next transmission. Sequencing is usually not part of CCS but since it increases specification convenience and is semantically well understood [2,22], it is included in our CCS dialect. The operator is present in this line of our example for the sole purpose of demonstrating its use, and the same holds true for the action `snd` appearing as a process parameter of the

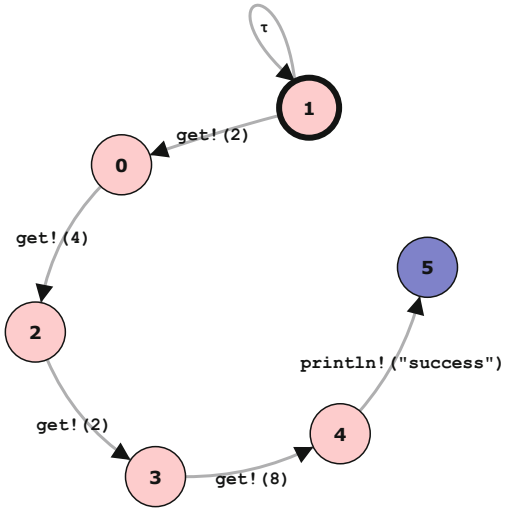


Fig. 1. Behaviour of the process shown in Listing 1.1

`Medium` process. Using actions as process parameters enables emulating restricted forms of action relabelling. Other than this, no explicit relabelling operator is supported in our dialect of CCS.

Line 9 contains the definition of the `Receiver` process. It offers up to three behaviours: (a) If a value `x` is received that lies in the integer range from 0 to 9, that value is passed on with the `get!` action and acknowledged; (b) if a `garbled?` message is received, a negative acknowledgement is sent; or (c) if 4 digits have been received, `"success"` (formed via string concatenation) is printed and the receiver stops. For the latter, the process uses a data parameter and simple arithmetic to count the digits that have been received. A `when` guard disables the `"success"` message until this counter reaches 4. Such guards form the expectable link between data values and behaviour.

Line 13 composes `Sender`, `Receiver` and `Medium` (appropriately parametrised) running parallel to form a single `Protocol` process. The restriction operator \ enforces synchronisation between the processes where appropriate. Finally, line 15 defines the overall process consisting of the `Protocol` and a user requesting the values 2, 4, 2 and 8 to be sent. The resulting behaviour (more precisely the quotient transition system under observational congruence) of this example is depicted in Fig. 1.

2.2 The Language pseuCo

Nowadays, mainstream programming is carried out in imperative programming languages. pseuCo is an imperative language featuring a heavily simplified Java-like look and feel paired with language concepts inspired by the Go programming language. It also has similarities with Holzmann's Promela language [15]. A first, very simplistic pseuCo example is depicted in Listing 1.2.

Listing 1.2. Shared memory concurrent counting in PSEUCO

```
int n = 10;
lock guard_n;

void countdown() {
    for (int i = 5; i >= 1; i--) {
        lock(guard_n);
        n--;
        unlock(guard_n);
    }
}

mainAgent {
    agent a = start(countdown());
    countdown();
    join(a);
    println("The value is "+ n);
}
```

This program implements concurrent counting. A shared integer, `n`, is initialised to `10`. The procedure `countdown()` decrements this counter five times. The `mainAgent`, which is run when the program is started, starts a second agent that runs `countdown()` before calling `countdown()` itself. After both agents have executed this procedure, the `mainAgent` prints the final value of `n`. To ensure mutually exclusive access to the shared variable, a globally defined `lock` named `guard_n` is used within the `countdown()` procedure.

An alternative (and usually recommended) way to perform safe computations in the presence of concurrency and shared-memory is to encapsulate critical sections within a monitor [12,14]. This concept is indeed supported in PSEUCO. For the example above, this would mean to wrap the shared variable in a monitor `AtomicInteger`, offering procedures to read and to modify its value without interference by others.

We demonstrate the PSEUCO support for monitors by an implementation of a semaphore [9]. Semaphores provide means of controlling access to common resources, similar to locks, but are more general: they manage a pool of an initially provided, limited number of resources (allocated by `init(v)`). Any agent can request one of the resources by calling `down()`. When the agent does not need the resource anymore, it can hand it back to the semaphore by calling `up()`. Listing 1.3 presents the implementation of a semaphore providing the three procedures `init`, `down`, and `up` as a `monitor` in PSEUCO. An instance `sem` of such a monitor is obtained by declaring `Semaphore sem`. For these instances, no explicit locking is necessary because the data structure is declared as a `monitor` as opposed to a simple `struct`. This means that each instance has an implicit, built-in lock. This lock is automatically locked at the entrance of any procedure declared in the monitor and unlocked on its exit.

Listing 1.3. A monitor in PSEUCO implementing a semaphore

```
1  monitor Semaphore {
2      int value ;
3      condition valueNonZero with (!( value ==0));
4
5      void init (int v) {
6          value = v;
7      }
8      void down () {
9          waitForCondition(valueNonZero);
10         value-- ;
11     }
12     void up () {
13         value++ ;
14         signalAll (valueNonZero);
15     }
16 }
```

As in the example, monitors can be equipped with conditions and condition synchronisation in a way that very closely follows the original proposal [13]. For the semaphore example, an agent has to wait in case it is requesting a resource but finds the pool of resources to be empty. This non-emptiness `condition` (a predicate on variables guarded by the monitor) is declared in line 3. It is checked in line 9. The semantics of `waitForCondition` is exactly as described above: If the condition is satisfied, the procedure continues. If it is not, the implicit lock of the monitor is released and the agent needs to wait. In order to wake up the agent and let it re-check the condition, the waiting agents are `signal`ed in line 14 after a resource has been handed back (which makes the condition satisfied). While `signalAll` wakes up all agents waiting for the specific condition, `signal` would nondeterministically pick a single waiting agent to wake up.

Message-passing concurrency is arguably less difficult to handle than shared-memory concurrency. PSEUCO provides native support for message-passing concurrency, and indeed, this is explained to the students before discussing the latter. An example is presented in Listing 1.4. An agent running the procedure `concat` interacts via three different channels with the `mainAgent`. In a nutshell, `concat` builds up a string `s` by prefixing it with parts received from channel `arg`. Empty parts make it report an error on channel `err` while otherwise the updated value of `s` is reported on channel `res`. This channel is declared globally in line 1, the others are parameters of `concat` and declared in lines 16 and 17. Two of them are channels that can hold strings (`res` and `parts`), one can hold Booleans (`err`). Channel `parts` is a FIFO buffer which can hold up to 2 elements, the others are unbuffered, meaning that they induce a handshake between the agents sending to (via `<!`) and receiving from (via `<?`) them. After starting the agent, the `mainAgent` feeds three strings into the channel `parts` (one of them empty)

Listing 1.4. Message-passing concurrency in PseuCo

```
stringchan res;

void concat (stringchan arg, boolchan emp) {
    string s = "";
    while (true) {
        string pref = <? arg;
        if (pref == "") emp <! true;
        else {
            s = pref + s;
            res <! s;
        }
    }
}

mainAgent {
    stringchan2 parts;
    boolchan err;

    start(concat(parts, err));
    parts <! "strand";
    parts <! "";
    parts <! "Guld";

    string r;
    while (true) {
        select {
            case r = <? res: {
                println(r);
            }
            case <? err: {
                println("Empty string reported!");
            }
        }
    }
}
```

and then waits for results sent back to him. These may arrive on two different channels (`err` and `res`) and therefore a `select-case` statement is used to specify dedicated reactions. In case an `error` is reported, this is reported to the user in a `println`. Otherwise, the results received on channel `res` are printed out. PseuCo has borrowed the `select-case` concept from Go [11]. A `select` statements consist of several `case`s. Except for default cases, each case has a guard and a statement. The guard contains exactly one send (`<!`) or receive operation (`<?`). At runtime, a case can be selected only if the message-passing operation of the guard is possible, i.e. if the channel can be read or be written to, respectively. One of those

cases is selected nondeterministically and its guard and statement are processed. A default case can always be selected. If there are multiple cases that can be selected, one of them is selected non-deterministically.

These examples give an impression of the features provided by PSEUCO, all of which are given semantics by translation to CCS.

2.3 Translational Semantics

Several of the peculiarities of imperative programming languages do not have a direct counterpart in process calculi like CCS. The most challenging ones include jumping and branching, buffers, memory and dynamic object referencing, buffered channels, reentrant locks, monitors, and condition synchronisation. In the following sections, we give an intuition of how PSEUCO programs are given semantics in terms of CCS. We cover each part of the PSEUCO language and explain the non-trivial parts in particular detail.

Program Structure. In Listings 1.2 and 1.4, we have seen that concurrency is supported in PSEUCO by the possibility of wrapping procedures into *agents* which run concurrently to the remainder of the program. Similar to the Go language [11], any procedure can be started as an agent by using the `start` primitive in front of the procedure call (lines 13 and 19). On the CCS level, each agent corresponds to a process that runs in parallel to the others. In addition, further parallel processes implement the necessary bookkeeping and coordination.

The translational semantics of PSEUCO to CCS is of compositional nature and best explained by looking at the abstract syntax tree of a PSEUCO program. Roughly speaking, the closer nodes are to the root of the tree, the more they determine the top-level structure by influencing whether processes are composed in parallel or sequentially. Nodes that are lowest in the syntax tree map PSEUCO terms to CCS terms. When passed to their parent nodes, these are usually composed sequentially or as nondeterministic alternatives such that the control flow of the PSEUCO program is respected. The topmost nodes compose global variables, locks, conditions, arrays, and channels (and the bookkeeping needed to support those) as processes running in parallel to the execution of the main agent. Moreover, for each procedure that is wrapped as an agent, there is a process responsible for starting it, also running in parallel to the other processes. On the outermost level, we hide all actions of the resulting model that are not `println!` or `exception!`.

Whereas composing everything in parallel is very intuitive, finding appropriate CCS terms and coordinating them in a control-flow-preserving way requires some interesting concepts. In the remainder of this section, we provide a description of the main ideas and concepts structured along the different language features of PSEUCO.

Expressions and Assignments. PSEUCO supports arithmetic and Boolean expressions, constants, and variables. For now, we only consider variables local to an agent. Global variables are discussed later. Local variables in PSEUCO are mapped

to process parameters on the CCS level. However, assignments to variables are not supported there. Instead, an assignment triggers that on the CCS level, all occurrences of that variable are substituted by the expression on the right side of the assignment. For each arithmetic and Boolean operator in PseuCo there is a counterpart operator in CCS. For instance, the following program is compiled into the single CCS action `println!3+(2*3)+(2*3)`:

```
int x = 3;
int y = 2 * x;
x = x + y;
println(x + y);
```

Memory. While local variables in a PseuCo program can be cast into process parametrisation on the CCS level, we need to proceed differently for non-local PseuCo variables. This applies especially to shared variables in shared-variable concurrency. In order to represent such variables on the CCS level, we need to represent memory in CCS. A common abstraction of memory is a set of memory cells. A cell is independent of the representation of values (e.g. bits), its only purpose is to store a particular value. In CCS, a memory `Cell` can be modelled as a parallel process that provides actions for getting the currently stored value `cur` and for setting it to a `new` value:

```
Cell_x[cur] := get_x!cur. Cell_x[cur]
             + set_x?new. Cell_x[new]
```

When a process reads or writes the value of a cell, it needs to perform a handshake synchronisation with the action provided by the cell as in the following CCS snippet:

```
(get_x?x. println!x. 0  |  Cell_x[-3]) \{get_x}
```

Object References. Imperative programming languages use references in order to access objects in memory. In PseuCo, structs and monitors are accessed by references, but also locks, arrays and channels are accessed this way. Consequently, there can be arbitrarily many objects. Hence, memory cells of an object must store the reference to the object they represent. Therefore, the cell definitions for a member of a structure or monitor `A` need an additional argument `i` for the reference as in `Env_class_A[i, x]`. For example, the process for the monitor `Semaphore` in Listing 1.3 has the name `Env_class_Semaphore[i, g, value]` because it holds a reference to its implicit lock `g` and it stores the `value` for its variable.

Anyhow, the actions for accessing variable `x`, `get_x` and `set_x`, would still be shared across all memory cells for `x` of the struct (or the `monitor`), so using `get_x` (for example) would cause an `x`-value from any `struct` instance to be read. In

order to make access to `struct` and `monitor` instances unique, each cell needs unique actions for accessing it. Since there may be arbitrarily many instances, making actions unique is not trivial. With the intention of keeping the resulting CCS code readable and intuitive, we have added the possibility of parametrising action names to our CCS dialect. This parametrisation effectively extends the expressiveness of our dialect to that of the π-calculus [23] since the parameters can be passed around as values. However, we restrict this mechanism to integer parameters and allow integer arithmetic on those. The PSEUCO semantics uses this for integer references to memory cells and other objects. The definition of such cells is to be adjusted as follows, where the `(i)` is the parametrisation occurring in the action name:

```
Env_class_A[i, x] :=
  env_class_A_get_x(i)!x. Env_class_A[i, x]  +
  env_class_A_set_x(i)?v. Env_class_A[i, v]
```

Another example of objects that are accessed by reference are arrays. Arrays are modelled as simple CCS processes that have as many process parameters as there are elements in the array (plus one reference `i` for identifying the object, as explained above). In order to access a specific array element, the user must first communicate the element index and can then choose between reading or writing. For each array capacity occurring in the PSEUCO program, there must be a dedicated CCS process. Below we present one for capacity 3:

```
Array3[i, v0, v1, v2] := array_access(i)?idx. (
  when (idx == 0) (
    array_get(i)!v0. Array3[i, v0, v1, v2]  +
    array_set(i)?v. Array3[i, v, v1, v2]
  ) + when (idx == 1) (
   ...
  ) + when (idx == 2) (
   ...
  )
```

As it is the case for real memory, it must be possible to *allocate* a memory cell. For example, in Listing 1.3, there is an allocation of a lock in line 2. Hence, each type of cell has a constructor that manages the references and adds a new memory process in parallel to the program processes whenever necessary. The following listing shows the constructor for a struct `A` containing a single variable `x`. The process `Env_class_A` is as shown above.

```
Env_class_A_cons[i] := class_A_create!(i).
  (Env_class_A[i, 0] | Env_class_A_cons[i+1])
```

Some types of objects are supported by several constructors. Arrays, for example, have a distinct constructor for every array size. Still, all arrays must share the

same reference space. There is one process that manages the references (using integer arithmetic on references) for arrays and each array constructor requests a free reference for the next instance it is supposed to initialise.

```
ArrayManager[i] := array_new!(i). ArrayManager[i+1]
```

Below is how the constructor of arrays of size 3 uses the array manager. The constructor itself acts independent of the type of the elements so it is necessary for initialisation to provide the constructor with a default value for the cells (i.e. `Array3_cons` can be used for bool, int, or string arrays so the default value is false, zero, or the empty string).

```
Array3_cons := array_new?i. array3_create!(i).
  array_setDefault(i)?d. (Array3_cons | Array3[i, d, d, d])
```

The necessary communication with the array manager introduces some superfluous interleavings because the next array reference can be requested at any time. This can be avoided at the price of a more involved encoding.

Procedures and Jumping. In CCS, the base "execution" order is linear, i.e. its semantics executes one prefix after the other. There is no built-in history that enables going back to a previous point in execution. In PseuCo (and other languages), however, it is possible to call procedures. After the procedure has been executed, the execution of the program jumps back to the point where the procedure was called. Hence, procedure calls can not be defined within CCS directly, but they can be tackled by means of sequencing (;). With sequencing, it is indeed straightforward to embed procedures in CCS processes. When a procedure is called, the process' name appears in front of the sequence operator.

For non-void procedures, it is necessary to return a value to the caller, but a direct handshake is not possible (since they run sequentially, not in parallel). In our encoding, we use a dedicated parallel process which collects and delivers the values to return. With that, a procedure can return a value by sending it to the dedicated process, and the caller can receive it from there as part of a sequence operator.

Control Flow. Most imperative programs need conditional branching. For example, the `for` loop in Listing 1.2 must jump from the end of its body either to the beginning of the loop or behind the loop, and the `if-then-else` in Listing 1.4 determines the control flow according to its condition. In CCS, jumping and branching is not supported directly. However, processes can be given names (appearing on the left hand side of defining equations). Similarly to what has been discussed for procedures, we accommodate conditional branching by splitting into several named sub-processes. Each sub-process name is the equivalent of a jump label, and branching to a sub-process boils down to using the name of the sub-process. The following listing shows a simple loop:

```
while(a > 0) {
  println("loop");
  a = a-1;
}
println("a␣is␣zero");
```

Its semantics is the CCS code below:

```
P[a] := when (a <= 0) Q[a]
      + when (a > 0) println!"loop". P[a-1]
Q[a] := println!"a␣is␣zero". 0
```

The presented pattern can be used for the other branching statements as well.

Mutual Exclusion and Locks. PSEUCO supports locks, monitors, and conditions. In its basic form, a lock is encoded as a process occupying one out of two states, locked and unlocked. Only one parallel process can perform the locking at a time:

```
Lock[i] := lock(i)?. unlock(i)!. Lock[i]
```

As we have seen previously, the parameter `i` is a reference to allow uniquely identifying a specific lock. The actions `lock` and `unlock` can then be used to lock and unlock the lock, respectively, as demonstrated in the example `Lock[i] | lock(i)!. println!"CriticalSection". unlock(i)?. 0`.

An advanced variant of locks are reentrant locks where a single agent, the lock owner, is allowed to take the lock multiple times. The lock is released to other potential owners only if the owner has unlocked it the same number of times it has been locked. Modelling this in CCS is more intricate than single-entrant locks. The lock process needs two additional arguments: one that holds the agent identity owning the lock and one that keeps a count of the number of `lock`s still to be `unlock`ed. The process allows anyone to become owner by acquiring the lock provided nobody else already owns it. If an agent `a` owns the lock, then the process ensures that further locks are only made possible for `a`. It throws an exception if an `unlock` is requested by a non-owner:

```
Lock[i, c, a] :=
  when (c==0) lock(i)?a. Lock[i, 1, a] +
  when (c>0) (
    lock(i)?(a). Lock[i, c+1, a] +
    unlock(i)?a2. (
      when (a==a2) Lock[i, c-1, a] +
      when (a!=a2) exception!("Exception").0))
```

In this fragment, there is a peculiar difference between the `lock` actions in lines 2 and 4. In line 4, there are additional parentheses around `a`. This forces the current value of the expression `a` to be evaluated (instead of overriding it) and therefore ensures that `lock(i)!e` (effectuated by some agent referenced as `e` who may not be the owner) and `lock(i)?(a)` can handshake if and only if `e` and `a` evaluate to the same value (a concept called value matching in some calculi). This is how we ensure that re-entrances are only granted to the agent that already owns the lock.

Agents. For each procedure that is used as an agent, there is an agent process in parallel to existing agents. It is responsible for starting the agent, similar to a constructor. The following listing shows the agent process `Agent_f` that is responsible for starting a procedure `f` with one argument `x`:

```
1 Agent_f := get_free_ref?a. start_f!a. set_arg_x?v.
2                                 (Agent_f | Proc_f[a, v]; 0)
```

As we have seen already for arrays, agents share a common reference space, so `Agent_f` first gets an unused reference from a management process. It then offers to send this reference with `start_f!a`. As soon as an agent wants to start `f` as a new agent, it begins by receiving this message `start_f?a` to get the reference for the new agent. If `f` needs arguments to be called, as in the example above, then the starting agent has to send values for each of the arguments. Afterwards, `Agent_f` calls the procedure `f` with the arguments it has received in parallel with a fresh copy of itself.

Other agents can choose to wait for this new agent's termination before they continue their own execution. In PseuCo, this is done by calling the primitive `join`. Hence, we augment the CCS representation so that after termination of the agent, the process continues with a process that offers an unlimited number of `join(a)!` actions. The translational semantics of the `join` primitive is the complementary action `join(a)?`. Due to the unlimited offers of individual `join(.)!`-transitions provided by each agent upon termination, any attempt to `join(.)?` an already terminated agent will not block.

```
1 AgentJoins[a] := join(a)!. AgentJoins[a]
2 Agent_f := get_free_ref?a. start_f!a. set_arg_x?v.
3                    (Agent_f | Proc_f[a, v] ; AgentJoins[a])
```

Message Passing via Channels. As shown in Listing 1.4, PseuCo supports message-passing communication via unbuffered and via (FIFO) buffered channels of fixed capacity (as in lines 17 and 16). CCS, on the other hand, provides unbuffered communication via handshaking of complementary actions with value-passing. Hence, PseuCo's unbuffered channels can be encoded directly. For buffered channels, we need the ability to store the buffer state which is comprised of the items to be buffered and their order. Basically, a buffered channel behaves

similarly to an array of memory cells, however it has restricted actions for access (namely only pushing and popping). A straightforward encoding is as follows:

```
Buffer_n[i, c, v_1, ..., v_n] :=
  when (c==0) put(i)?v_1. Buffer_n[i, c, v_1, ..., v_n] +
  ... +
  when (c==n-1) put(i)?v_n. Buffer_n[i, c, v_1, ..., v_n] +
  when (c>0) chan(i)!v_0. Buffer_n[i, c-1, v_2, ..., v_n,0]
```

Here, `n` is the capacity of the buffer, `c` is the number of items that are currently buffered and `v_j` are the cells of the buffer (`i` is the buffer reference as usual). The channel allows sending values to it provided $\texttt{c} < \texttt{n}$. Due to the dedicated `when`-statements guarding each `put`, the value received over action `put` is stored in the next free cell. Values can be received from the channel provided $\texttt{c} > 0$. The channel sends the value contained in `v_0` over action `chan`, decreases the item counter by one and shifts all buffered values to the left.

The type system of PSEUCO needs to accommodate for the fact that channels can be referred to without specifying whether they are buffered or unbuffered. For example, in Listing 1.4, the type of the parameter `arg` of `concat` is `stringchan` although the main agent passes a `stringchan2` to it. Once such a channel is used for sending, it is necessary to determine the buffer type dynamically because unbuffered channels are used with action `chan(i)!` and buffered channels are filled with action `put(i)!`. We overcome this problem by using negative numbers as references for unbuffered channels and positive numbers for buffered ones. The following listing shows the CCS code corresponding to sending a value `v` over a channel with reference `i` for which the buffer type is not known in advance.

```
when (i < 0) chan(i)!v. 1  +  when (i > 0) put(i)!v. 1
```

Select Statement. The `select` statement introduces nondeterministic choice to PSEUCO. For example, in Listing 1.4, the main agent can nondeterministically choose to either process a result or an error sent by `concat`. There is a direct counterpart in CCS, namely the nondeterministic choice operator `+`. In the encoding, it is important to assure that on the CCS level, the leftmost prefix of each resulting nondeterministic alternative corresponds to the channel appearing in the respective `case` to ensure that the selection is made based on the correct external stimulus.

Monitors and Conditions. As already shown in Listing 1.3, PSEUCO supports monitors in the form of a `struct` that has an implicit, built-in lock. We have seen that monitors can be enhanced with conditions so as to support condition synchronisation. The `Semaphore` in Listing 1.3 employs a condition to make agents wait until a resource becomes available. A waiting agent does not perform any work. In particular, it does not run in a loop that tries to enter and

exit the monitor over and over until the condition is found to be satisfied (a so-called *busy-wait*). Instead, the classical monitor concept comes with a notification mechanism where waiting agents wait until they are notified. PSEUCO supports this via the two primitives `signal` and `signalAll` that need to be used actively by some agents. The `Semaphore` example in Listing 1.3 uses `signalAll` in line 14 once a resource has been handed back.

We illustrate the behaviour of a condition as agents that rest inside a waiting room until they are notified that a change they are waiting for happened. This metaphor emphasizes that agents actually stop working and do not have to do anything actively until they receive a signal. In the semaphore example, an agent that finds the pool of resources empty goes to the waiting room and stays there until a resource is handed back and thus `signalAll` is called. PSEUCO's conditions are only available inside monitors so there is always the implicit monitor lock that the condition is related to.

In CCS, we adapt the waiting room metaphor and use two processes: one for the waiting room and one that broadcasts the signal to waiting agents.

```
1  WaitRoom[i, c] :=
2    signal(i)?.(
3      when (c==0) WaitRoom[i, c] +
4      when (c>0) wait(i)?.WaitRoom[i, c-1] )  +
5    add(i)?.WaitRoom[i, c+1]  +
6    signal_all(i)?.WaitDistributor[i, c] ; WaitRoom[i, 0]
7
8  WaitDistributor[i, c] :=
9    when (c<=0) 1 +
10   when (c>0) wait(i)?.WaitDistributor[i, c-1]
```

The waiting room counts the number of waiting agents and supports the following four operations:

- If an agent wants to use a condition, it must perform two steps. First, it must add itself to the waiting room (using `add`) while still holding the monitor lock.
- After an `unlock`, the agent synchronises over channel `wait`. However, the waiting room joins the synchronisation only when the agent is supposed to continue its work.
- Working agents use the action `signal` in order to notify one of the waiting agents that the condition may have changed. When a signal is received, the waiting room either ignores it if no agent is waiting or it offers a single `wait` for synchronisation to the waiting agents. Which of the waiting agents synchronises with the waiting room is non-deterministic.
- Similarly, `signal_all` is used to notify all waiting agents. This task is delegated to the process `WaitDistributor`. The wait distributor gets the number of waiting agents and then offers each of them a `wait` action for synchronisation.

The resulting CCS encoding of some program using conditions may look as follows:

```
WaitAgt[m, c] := lock(m)!. getB?b.
  when (b) println!"Condition␣holds". unlock(m)!. 1  +
  when (!b) add(c)!. unlock(m)!. wait(c)!. WaitAgt[m, c]
WorkAgt[m, c] := lock(m)!.
  setB!true. signal_all(c)!.
  unlock(m)!. 1

WaitAgt[m, c] | WorkAgt[m, c] | Lock[m] | WaitRoom[c, 0]
```

3 pseuCo.com – A Web Platform for Learning pseuCo

To propel the use of pseuCo in academic teaching, we have developed a web application available on https://pseuco.com/. It serves as an interactive platform for students learning CCS and pseuCo as part of our concurrent programming lecture and provides the user with access to the translational semantics of pseuCo described in Sect. 2.3. The following section provides a detailed description of pseuCo.com.

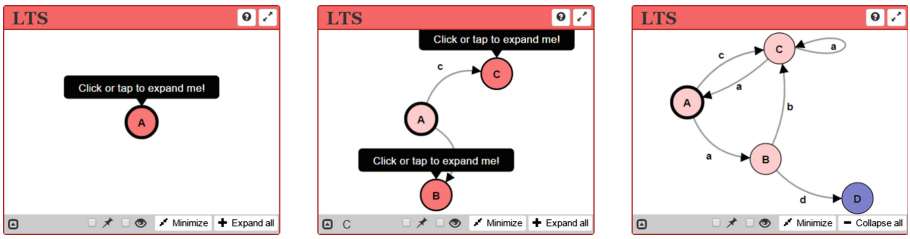
LTS Viewing. pseuCo.com users will often find themselves looking at an LTS – either one that was generated by the semantics of a CCS expression that they entered or that was generated by the pseuCo compiler, or one that was sent to them. In all cases, pseuCo.com uses the same approach to display them:

In the beginning, only the initial state of the LTS is shown. Clicking or, for touch-enabled devices, tapping it reveals the initial state's successors. The user can continue expanding states step by step, either by continuing to click states they are interested in, or by using the Expand all button which will reveal the successors of all *visible* states. Expanded states can also be collapsed by another click on them which hides all successor nodes that no longer have a path to the initial state. Figure 2 demonstrates this behaviour in a small LTS.

This behaviour and the absence of support for infinitely branching transition systems ensure that pseuCo.com never tries to display an infinite number of states. The LTS itself can be infinite. In that case, the users will never be able to fully expand the system. Still, he is free to explore any part of the graph he can reach from the initial state.

Because of this interactive approach, we have certain requirements for our graph layout algorithm:

1. Graph layout must be *continuous*: When the graph changes (for example because a node has been expanded), the existing nodes should not move far, and this move should be animated so users can easily keep track of the states.
2. Graph layout must be *interactive*: Users need to be able to move nodes as they see fit, and the graph layout should pay respect to the user-given node placement while still optimising overall readability.



(a) Initial view (b) After first click (c) Fully expanded LTS

Fig. 2. A labelled transition system as shown in PSEUCO.COM. The user has to click on a state to reveal its successors. This is explained to first-time users by a floating hint shown until two states have been expanded.

3. Graph layout must *feel natural* so users are not confused by the visual changes they cause by moving nodes.

To fulfil these requirements, we use a force-based approach to graph layout: The visible part of the LTS behaves like a physical simulation where nodes are inert masses that carry an electrical charge, connected by springs, with a gravitational pull towards the middle of the graph. The inertia ensures a smooth, continuous movement of all nodes. The electrical charge ensures that nodes repel each other, avoiding overlap and keeping a reasonable distance from other nodes. The springs can push or pull on nodes to achieve consistent spacing between nodes. Finally, the gravitational pull centres the whole system within the allocated space.

Because transitions are labelled by actions and multiple transitions may exist between the same pair of states, both states *and transitions* correspond to nodes in the simulation. The nodes for states directly correspond to the position where the state is rendered. For transitions, the corresponding nodes (which have a weaker simulated electrical charge than states) correspond to the position where the transition's label is drawn. Therefore, the electrical charge of these nodes automatically optimises for overlap-free rendering of the labels. The transition is rendered as a Bézier curve between the two states with the transition node's position serving as the control point. This ensures that multiple transitions between the same states are rendered correctly as demonstrated in Fig. 2c by the transitions $A \xrightarrow{c} C$ and $C \xrightarrow{a} A$. It also allows self-loops to automatically rotate away from other transitions or states.

If the user wants to layout parts of the graph himself, he can drag and drop nodes at any time using the mouse or touch. In the latter case, we also support simultaneous dragging of multiple nodes. The remaining nodes continue to be affected by the forces, and after the user stops dragging a node, it returns to normal behaviour. This allows the user to get the graph into another stable state that he prefers.

To gain even more control over the placement of nodes, the user can check Pin dragged nodes. If this setting is enabled, after the user lets go of a node,

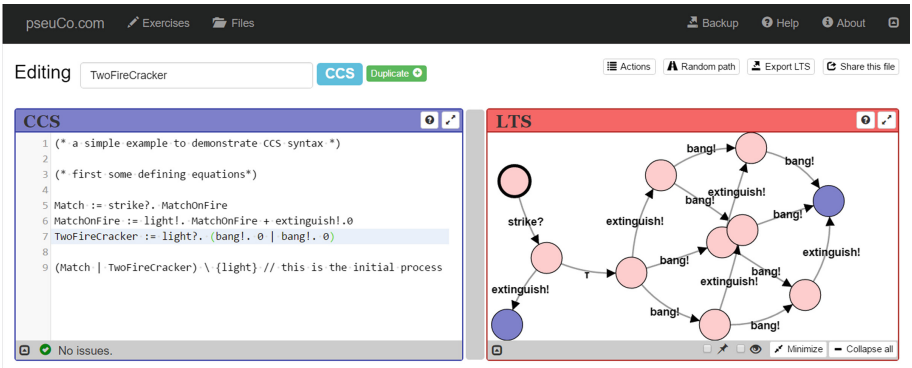


Fig. 3. A screenshot of the CCS editing interface

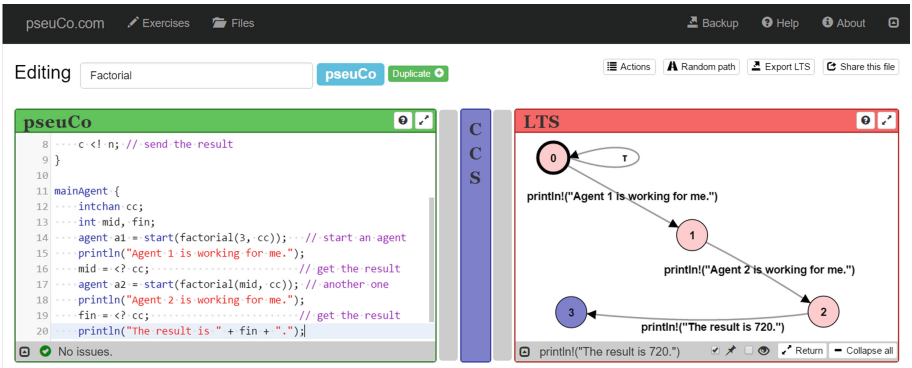


Fig. 4. A screenshot of the PSEUCO editing interface

it will seize all movement and will no longer be affected by the forces. This allows the user to layout any part of the graph manually without interference from the automatic layout while the remaining part of the graph will still be arranged automatically.

CCS Editing. While editing a CCS file, the user sees two windows as demonstrated in Fig. 3. The first one contains a text editor with his CCS code while the second one contains the LTS editor described in Sect. 3. The user can drag the grey separator bar to freely distribute the horizontal space between both windows or use a Maximize button in the title bar to allocate all horizontal space to that window. In that case, the other window will be reduced to a thin vertical strip.

During editing, the CCS expression is parsed continuously so the LTS shown to the right is never stale.

PSEUCO *Editing.* When editing a PSEUCO file, the user is shown the PSEUCO code, the CCS code produced by the PSEUCO compiler, and the corresponding LTS, as demonstrated in Fig. 4. As with CCS editing, the input is evaluated

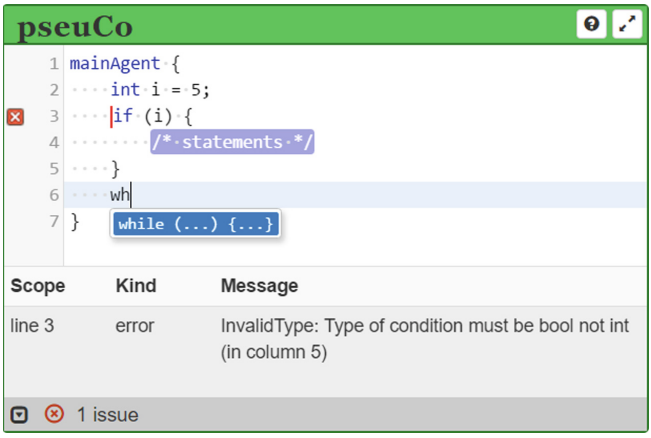


Fig. 5. A screenshot of the pseuCo text editor

automatically, and the user can freely divide the horizontal space between the windows. While the CCS code is read-only (as it is generated by the compiler), the user can experiment with it: A Fork button in the title bar of the CCS window opens the generated CCS code as a new file to allow the user to edit it. Of course, the user can always go Back to his original pseuCo file.

The pseuCo editor provides the support typical of a basic IDE including snippet completion and in-line highlighting of compiler error messages as demonstrated in Fig. 5.

File Management and Sharing. pseuCo.com allows users to store and manage files in a virtual file system stored locally by their web browser. Files are always saved automatically and kept until they are deleted assuming that the user has entered a file name. Even unnamed files are saved temporarily to allow recovering lost data but they expire after one week.

To encourage students to discuss their findings, pseuCo.com features a file sharing facility. At any time during editing, users can select Share this file to upload a copy of their current file to our server. In return, they receive a sharing link containing a random identifier for them to share, for example https://pseuco.com/#/edit/remote/50xshwwyuza86w4pjtke. Anyone opening this link in a modern browser will immediately be presented with a read-only view of this file.

pseuCo.com also allows users to export transition systems as `TRA`-, `AUT`-, and `DOT`-files (in addition to its own `JSON`-based format) for processing with external tools. It can also import `AUT`[1]-files.

Tracing Support. The application provides a way to compute random traces through an LTS which can by shown including or omitting τ-steps.

[1] http://cadp.inria.fr/man/aut.html.

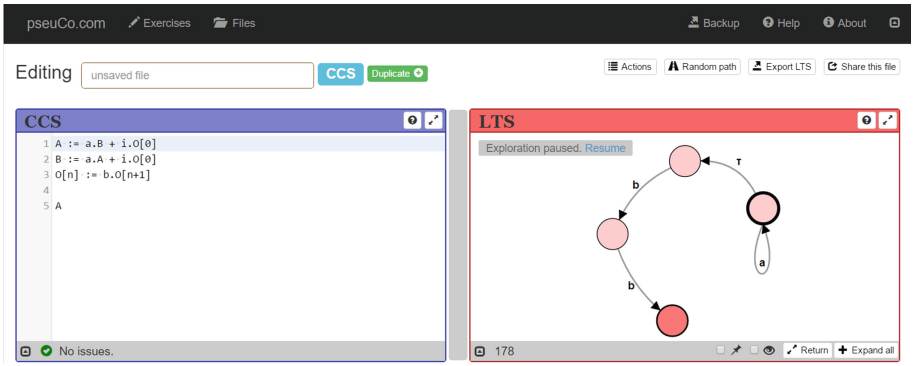


Fig. 6. A compressed, infinite LTS where states A and B have been merged

In addition to allowing random tracing, PSEUCO.COM also collects a list of (non-τ) actions found within an LTS (which can be shown by clicking Actions) and can compute traces leading to these actions by backchaining. For example, this can be used to check whether a PSEUCO program can produce an unexpected output and to synthesise an interleaving explaining this output.

Minimization. To aid in analysing large LTS, PSEUCO.COM implements minimization under observational congruence [22]. This feature can be invoked by the Minimize button in the LTS toolbar.

Whenever an LTS is displayed, a background thread is dispatched to precompute all its states. When minimization is invoked, it only considers states that have been found up to this point. If minimization is started before all states have been explored, the result is not guaranteed to be minimal – it is only minimal with respect to the already explored part of the system. While exploration is running, the minimization button is labelled Compress to emphasize this fact.

This behaviour allows compression of systems that are infinite or too large to be explored in a reasonable time frame. Figure 6 shows an example of this.

Offline Use. PSEUCO.COM is a pure, JavaScript-based web application. While this provides many benefits, most importantly the ability to run without any installation or bootstrapping, this opens up the question of whether the application can be used without an internet connection. Indeed, PSEUCO.COM provides full support for offline use using the HTML5 Application Cache APIs. Upon the first visit of PSEUCO.COM, all major modern web browsers automatically download the full web application and store it in a permanent cache. Afterwards, network connectivity is only needed to download application or template updates and to upload and download shared files.

All computation, including PSEUCO-to-Java compilation and the CCS semantics, is always performed directly in the user's browser.

Modes, Exercises, and Use in Teaching. While PseuCo.com can be used by anyone as a tool for exploring CCS and PseuCo semantics, it is specifically tailored to educational usage. The needs of students learning concurrent programming differ from those of an expert looking for a tool. By our experience, providing features like CCS semantics or LTS minimization to students that have not yet understood the corresponding concepts *impedes* learning because it takes away an incentive to explore and apply these concepts by hand.

To accommodate these different needs, PseuCo.com can be operated in two different modes. In *tool mode*, all features are available without restriction. In *teaching mode*, users are prevented from creating files initially, only allowing them to view (but not edit) files that were shared with them. Instead, they get access to a set of *exercises*.

These exercises are meant to accompany a lecture or book teaching concurrent programming and provide milestones that unlock PseuCo.com features. For example, users need to demonstrate their ability to infer transitions of CCS terms by the SOS rules and to write small CCS terms with predetermined characteristics to unlock the ability to create and edit CCS files including the automatic creation of the corresponding LTS.

Upon first use, PseuCo.com will normally ask the user to select a mode. When providing links to students as part of the lecture notes, we use special links that cause PseuCo.com to default to teaching mode.

PseuCo.com has been introduced in the *Concurrent Programming* lecture at Saarland University in the summer term 2014.

4 Analysis Support

In our concurrent programming lecture, we sensitise our students to problems related to concurrency. We explicate that data races (race conditions), deadlocks and such are considered program errors and must be avoided at all costs. To further aid our students, we want to have tool support to detect, investigate, and possibly fix such errors.

This requires sophisticated and often computationally intensive program analyses. This is where the PseuCo.com platform – and its JavaScript basis – reaches its limits. To circumvent this issue and to complement the capabilities of the PseuCo.com platform we have started exploring new ways of giving students access to enhanced tools and analyses.

The first tool developed under this premise is able to statically detect data races on a subset of the language facilities of PseuCo. It is implemented in C++.

Next, we describe the major ideas that underlie this analysis based on the example in Listing 1.2 from Sect. 2.2.

Static Data Race Detection. In PseuCo, a data race occurs if two or more agents can access the same global variable simultaneously and at least one of these accesses is writing. In this respect, the example presented in Listing 1.2 is free of data races. However, the statement in line 7 is an access to the global variable `n`.

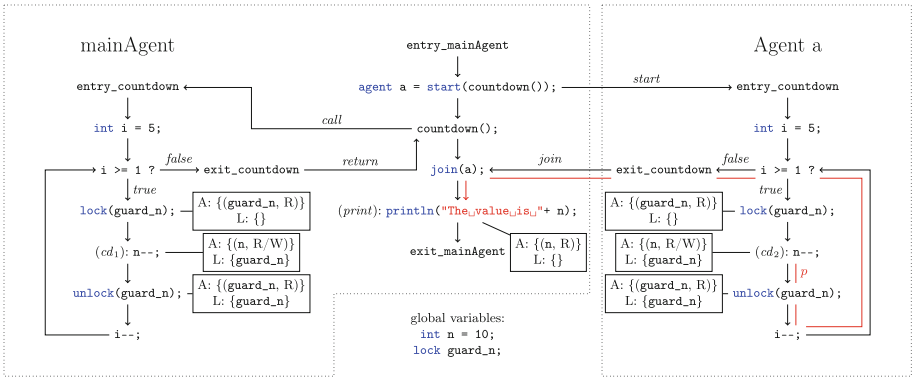


Fig. 7. Annotated control flow graph. A: accesses to global variables (R: read, W: write, R/W: read and write), L: locks guaranteed to be held. Nodes without annotation default to A: {} and L: {}

Without the `lock` and `unlock` statements in lines 6 and 8, respectively, this program would have a data race due to the concurrent, i.e. potentially simultaneous accesses to `n`. How can we infer this automatically without running the program?

To do so, we use a static program analysis that computes an overapproximation of all possible program behaviours and then tries to find unguarded concurrent memory accesses to global variables. This is done in three steps:

1. Compute a graph that holds all control flow information.
2. Annotate this graph with information relevant to identifying data races: accesses to global variables and which regions are guarded by locks.
3. Perform a graph search to find pairs of memory accesses.
4. Check each of these pairs for a potential data race.

Figure 7 shows the relevant part of the annotated graph for the example program. Deriving it is mostly straightforward[2].

The graph search is a basic reachability analysis and applied to pairs of memory accesses. First, all candidate memory accesses are collected (i.e. accesses to global variables) and grouped by the accessed variables. Then, for each such access group a list containing all the individual access pairs is constructed[3].

The characteristics differentiating potentially harmful from harmless access pairs are more interesting:

1. Do both accesses happen in the same agent?
2. Are both accesses read-only?

[2] There are some corner cases requiring careful analysis. We omit them due to scope and space constraints.

[3] Reflexive and symmetric pairs are omitted directly for precision (by construction a memory access cannot race with itself) and efficiency reasons.

3. Do these accesses share common locks, i.e. is there at least one lock that is held during both of the accesses?
4. Using the graph search, is it possible to construct a causality path between the two accesses?

If at least one of these indicators is satisfied, the affected pair is no longer considered to be a data race candidate. Otherwise, the access pair is considered a *potential* data race. This does not imply that an actual data race materialises but warns the user that the program may need more thought and reasoning. This imprecision is rooted in our decision to overapproximate program behaviour and to underapproximate locking information, if necessary. An argument showing the absence of a data race in a program that exhibits more behaviour and has weaker locking information than the real program (and its semantics) also extends to that original program while the converse does not.

Continuing with our example, there are three memory accesses to `n`: cd_1, cd_2 and *print*. This yields the following data race candidates: (cd_1, cd_2), $(cd_1, print)$, and $(cd_2, print)$. The first candidate is covered by the third indicator since `guard_n` is a common lock held during both accesses. Candidate two is not considered a data race as both accesses happen in the same agent and thus satisfy indicator one. As illustrated in Fig. 7, there is a causality path p (indicated in red) connecting cd_2 to *print*. Hence, the last candidate satisfies the fourth indicator and is no data race either.

Without lines 6 and 8, the pair (cd_1, cd_2) would not satisfy any of the indicators and would be returned by the analysis as a potential data race.

5 Conclusion

This paper has introduced the PseuCo approach to teaching concurrent programming. We have presented an overview of the language features and presented details of a translational semantics that maps any PseuCo program to the variant of CCS we presented. That semantics is implemented as part of an interactive web platform, pseuco.com. This platform provides access to the tools targeting PseuCo, most notably: the translation of PseuCo programs to CCS, the CCS semantics in terms of LTS, the PseuCo-to-Java compiler, and more. Due to space constraints, we have not covered the compilation of PseuCo to Java which makes it possible to execute any PseuCo program, and we have only sketched our efforts to enable deep semantic analyses for PseuCo, especially for flagging data race problems in shared-memory concurrency. We will continue to work on the TACAS (Teaching Academic Concurrency to Amazing Students) vision.

Acknowledgments. This work is supported by the ERC Advanced Investigators Grant 695614 (POWVER) and by the CDZ project CAP (GZ 1023).

References

1. Andersen, J.R., Andersen, N., Enevoldsen, S., Hansen, M.M., Larsen, K.G., Olesen, S.R., Srba, J., Wortmann, J.K.: CAAL: concurrency workbench, aalborg edition. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 573–582. Springer, Cham (2015). doi:10.1007/978-3-319-25150-9_33
2. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. **14**, 25–59 (1987)
3. Boudol, G., Roy, V., de Simone, R., Vergamini, D.: Process calculi, from theory to practice: verification tools. In: Sifakis [25], pp. 1–10
4. CAV award (2013). http://i-cav.org/cav-award
5. Cleaveland, R., Madelaine, E., Sims, S.: A front-end generator for verification tools. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 153–173. Springer, Heidelberg (1995). doi:10.1007/3-540-60630-0_8
6. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench. In: Sifakis [25], pp. 24–37
7. Cleaveland, R., Parrow, J., Steffen, B.: The concurrency workbench: a semantics-based tool for the verification of concurrent systems. ACM Trans. Program. Lang. Syst. **15**(1), 36–72 (1993)
8. Cleaveland, R., Sims, S.: The NCSU concurrency workbench. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 394–397. Springer, Heidelberg (1996). doi:10.1007/3-540-61474-5_87
9. Dijkstra, E.W.: Over seinpalen. Circulated privately, n.d
10. Garavel, H.: Compilation et vérification de programmes LOTOS. Ph.D. thesis, Joseph Fourier University, Grenoble, France (1989)
11. The Go programming language specification. http://golang.org/ref/spec
12. Hansen, P.B.: Shared classes. In: Operating System Principles. Prentice-Hall Series in Automatic Computation, pp. 226–232. Prentice-Hall (1973)
13. Hansen, P.B.: Monitors and concurrent pascal: a personal history. In: Lee, J.A.N., Sammet, J.E. (eds.) History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, 20–23 April 1993, pp. 1–35. ACM (1993)
14. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
15. Holzmann, G.: The Spin Model Checker - Primer and Reference Manual, 1st edn. Addison-Wesley Professional, Boston (2003)
16. ISO. Information processing systems - Open Systems Interconnection - LOTOS - a formal description technique based on the temporal ordering of observational behaviour. ISO ISO 8807:1989, International Organization for Standardization, Geneva, Switzerland (1989)
17. Larsen, K.: Context-dependent bisimulation between processes. Ph.D. thesis, University of Edinburgh, Mayfield Road, Edinburgh, Scotland (1986)
18. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT **1**(1–2), 134–152 (1997)
19. Aceto, L., Ingólfsdóttir, A., Larsen, K.G., Srba, J.: Reactive Systems: Modelling, Specification and Verification. Cambridge University Press, Cambridge (2007)
20. Magee, J., Kramer, J.: Concurrency - State Models and Java Programs. Wiley, Hoboken (1999)

21. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
22. Milner, R.: Communication and Concurrency. PHI Series in Computer Science. Prentice Hall, Upper Saddle River (1989)
23. Milner, R.: Communicating and Mobile Systems - The Pi-calculus. Cambridge University Press, Cambridge (1999)
24. Roscoe, A.W.: Modelling and verifying key-exchange protocols using CSP and FDR. In: The Eighth IEEE Computer Security Foundations Workshop (CSFW 1995), 13–15 March 1995, Kenmare, County Kerry, Ireland, pp. 98–107. IEEE Computer Society (1995)
25. Sifakis, J. (ed.): CAV 1989. LNCS, vol. 407. Springer, Heidelberg (1990)
26. Soriano, A.: Prototype de venus: un outil d'aide á la verification de systemes communicantes. In: Cori, R., Wirsing, M. (eds.) STACS 1988. LNCS, vol. 294, pp. 401–402. Springer, Heidelberg (1988). doi:10.1007/BFb0035867