Model Checking Concurrent Programs for Autograding in pseuCo Book

Felix Freiberger[0000−0001−9282−9665]

Computer Science – Saarland Informatics Campus
Saarland University, Saarbrücken, Germany
freiberger@depend.uni-saarland.de

Abstract. With concurrent systems being prevalent in our modern world, concurrent programming is now a cornerstone of most computer science curricula. A wealth of platforms and tools are available for assisting students in learning concepts of concurrency. Among these is pseuCo, a light-weight programming language featuring both message passing and shared memory concurrency. It is supported by pseuCo Book, an interactive textbook, focusing on the theoretical foundations of pseuCo, concurrency theory. In this paper, we extend pseuCo Book with a chapter on *Programming with pseuCo*. At its core is a custom verification system, based on pseuCo's Petri net semantics, enabling practical programming exercises to offer fast in-browser model checking that can validate the program's internal use of concurrency features and provide comprehensive debugging features if a fault is detected.

Keywords: Verification · Model checking · Autograding · Concurrency · Education · Colored Petri nets · Programming

1 Introduction

Since its infancy, computer science education has been a task of growing importance – and difficulty. The ever-growing use of concurrency, with multi-core CPUs now being prevalent even in embedded devices, has certainly furthered that trend. Today, instruction in practical concurrent programming, and the theoretical underpinning needed to fully understand it, is an essential component of a complete computer science curriculum.

Instruction in practical computer science often involves having students write programs. This has created interest in *autograding*, technologies to automatically check solutions for mistakes and provide feedback or assign a grade [10, 11]. While this may seem like an excellent use case for verification techniques, in practice, autograding is often based on testing. This approach is powerful in many cases, e.g. even allowing autograding of full Android applications [2], however, testing-based autograding is particularly challenging in concurrency-related exercises, as concurrent programs are usually nondeterministic and may have bugs that are hard (or even impossible) to detect in a test environment [3].

Moreover, independent of whether testing or stronger verification-based techniques are used in autograding, they are often used to verify a program's externally visible behavior. While this often is sufficient, in many cases, a deeper look into a program's internals is required to ensure students have solved an exercise in the intended way. This is a typical requirement in introductory-level concurrency exercises which are often exploitable with easy non-concurrent solutions that have the same externally visible behavior than the concurrent program students were intended to write.

In this paper, we present a verification-based approach for autograding introductory-level concurrent programming exercises. Our approach is built around pseuCo [1], an academic programming language designed to teach students the basics of message passing and shared memory concurrency. We use pseuCo's Petri-net-based semantics [5] to gain insight into the semantics of the pseuCo program under analysis, enabling verification of properties about the use of concurrency-related features. For example, this allows us to not only verify the output of a program, but confirm that it uses channel-based communication between agents in a predetermined way, or that it is free of data races.

Our verification system is deeply integrated into pseuCo Book [4], a web-based interactive textbook focused around teaching concurrency theory and practice. It backs the interactive exercises in a new *Programming* chapter of pseuCo Book that guides students through their first contact with message passing and shared memory features, ensuring that the exercises only accept programs using these constructs correctly and as intended. When a student's program fails verification, we use the pseuCo Debugger, a Petri-net-backed debugging tool for pseuCo programs [5] originally developed for the web IDE pseuCo.com [1], to display failure traces to students in a way that is easily understandable without any knowledge about verification technologies or Petri nets.

Structure of this paper. The remainder of this paper is structured as follows. Section 2 formalizes the properties that are to be analyzed. Section 3 documents the implementation of the model checker as part of pseuCo Book. Section 4 describes the new Programming chapter of pseuCo Book that is supported by this technology. Finally, Section 5 concludes this paper.

2 pseuCo Program Properties

2.1 Motivating Example: Message-Passing-Based Termination

In this section, we'll look at an example exercise from pseuCo Book.

PseuCo's message passing features allow the programmer to use synchronous and asynchronous channels to transfer primitive values between agents. An example of this – typically one of the first pseuCo program students see – is printed in Listing 1.1. This program computes the value of (3!)! using a `factorial` agent that computes the factorial of any number it receives on a synchronous (handshaking) channel, then sends it back on the same channel.

Listing 1.1. A simple message passing pseuCo program

```
void factorial(intchan c) {
    int z, j, n;
    while (true) {
        z = <? c; // receive input

        n = 1;
        for (j = z; j > 0; j--) {
            n = n*j;
        }

        c <! n; // send result
    };
}

mainAgent {
    intchan cc;
    agent a = start(factorial(cc));
    cc <! 3;
    int mid = <? cc;
    println("3!␣evaluates␣to␣" + mid + ".");
    cc <! mid;
    println("(3!)!␣evaluates␣to␣" + (<? cc) + ".");
}
```

While receiving messages from a specific channel is relatively straightforward, in some cases, a programmer may need to set up an agent to react to multiple possible message passing actions, e.g. incoming messages on two different channels. Doing so requires a dedicated language construct, which pseuCo borrowed from Go: the `select case` statement.

To teach students how to use this statement, pseuCo Book contains an exercise asking students to modify the program from Listing 1.1 such that the `factorial` agent terminates after it is no longer needed by the main agent.

There are two apparent methods to do so:

1. reserve a special value, like -1, that triggers the agent's termination, or
2. add a second, dedicated channel for termination requests.

The first method does introduce a corner case, so the exercise asks students to add termination cleanly, using the new `select case` statement to add a Boolean-typed control channel.

What does an autograder need to check when validating a solution to this exercise? Adding termination does *not* actually change the externally visible behavior of the program[1]. But even if the autograder was able to determine whether all agents have terminated at the end of execution, this would not actually test whether students have implemented the exercise in the intended way. For example, students could replace the `while (true)` loop with a hardcoded `for (int i = 0; i < 2; i++)` loop to cause the agent to terminate after two

[1] The pseuCo semantics does not allow externally distinguishing between termination and a deadlock.

iterations which would work in this specific example, but not in general (and does not demonstrate knowledge of how to use the `select case` statement).

While some of these pitfalls can be overcome by testing the students' submission in multiple, slightly different contexts, a more thorough solution is for the autograder to also inspect the program's internals, checking that every possible execution of the program completes these steps in order:

1. start an agent (agent 1) from the main agent (agent 0)
2. use synchronous communication to send the value 3 from agent 0 to agent 1
3. use synchronous communication to send the value 6 from agent 1 to agent 0
4. have agent 0 print `"3!␣evaluates␣to␣6."`
5. use synchronous communication to send the value 6 from agent 0 to agent 1, with agent 1 being in a `select case` statement with 2 cases for receiving values
6. use synchronous communication to send the value 720 from agent 1 to agent 0
7. and then, in any order
   - print `"(3!)!␣evaluates␣to␣720."` from agent 0
   - complete these steps in order:
     (a) use a synchronous channel to send a Boolean from agent 0 to agent 1, with agent 1 being in a `select case` statement with 2 cases for receiving values
     (b) terminate agent 1

Indeed, this is the approach we will follow. The following sections formalize this type of property.

2.2 pseuCo Verification Formalities

Let *pseuCo* be the set of pseuCo programs. We define colored Petri nets following Jensen [8], i.e. as a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ where Σ is the set of color sets, P the sets of places, T the set of transitions, A the set or arcs, N the node function, C the color function, G the guard function, E the arc expression function and I the initialization function. For clarity, we refer to the steps of a colored Petri net's execution as *firings*.

The pseuCo compiler [5] translates every valid pseuCo program $p \in pseuCo$ into a colored Petri net CPN and $labels : P \cup T \mapsto 2^{\mathcal{L}}$, a *pseuCo label function* that assigns sets of labels to both places and transitions. The set of labels $\mathcal{L}$, not described in full detail here, contains labels that describe the role of places and transitions in pseuCo terminology. For example, a place could be labeled (global-variable, "x") to indicate that it holds the value of a global variable named x, or a transition could be labeled (send-async) to indicate that it handles sending a message to an asynchronous channel (i.e. writing the value to its buffer).

To formalize our properties, we use LTL [9]. We assume a set AP of atomic propositions, deferring details to the next section. Skipping details for brevity, we assume a mapping from firings of the Petri net to subsets of atomic propositions.

The properties that are relevant for autograding pseuCo exercises can then be expressed as *LTL formulas*, i.e. terms φ with

$$\varphi ::= \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \bigcirc\varphi \mid \Box\varphi \mid \diamond\varphi \mid \varphi \,\mathcal{U}\, \varphi \mid ap \qquad (1)$$

and $ap \in AP$.

Atomic Propositions. Using the pseuCo label function (and knowledge of the internals of the pseuCo-to-CPN compiler), a Petri net firing can be analyzed to determine which pseuCo action the firing represents. This allows us to define and recognize atomic propositions that describe whether a firing

- prints a specific value,
- has specific agents participate in that firing (identified by their IDs),
- has a (single) participating agent with a specific expected recursion depth,
- represents a synchronous message passing transaction (handshaking), with a specified value,
- represents writing to or reading from an asynchronous (buffered) channel, with a specified value,
- originates from a `select case` statement (with a certain number of branches),
- starts or terminates an agent,
- represents a procedure call (with specific arguments),
- initializes, locks, or unlocks a lock,
- reads or writes a global variable (with a specific name and value), and
- reads or writes a structure field (with a specific name and value).

These atomic propositions can then be used by exercise designers to describe the intended behavior of a pseuCo program without insight into the specifics of the pseuCo-to-CPN compiler.

Step Checklists. Generally, the full flexibility of LTL is not needed to express the properties used for autograding pseuCo exercises. To simplify the process of specifying these properties – and allow representing the property, and its current state, more easily to a user – we introduce a simplified syntax for these properties, called *step checklists*. The set of step checklists $\mathcal{S}$ is defined as

$$s_1, \ldots, s_n \ni \mathcal{S} ::= \text{Step}\,(v_1) \mid \text{Sequence}\,(s_1, \ldots, s_n) \mid \text{Parallel}\,(s_1, \ldots, s_n) \quad (2)$$

$$v_1, \ldots, v_n \ni \mathcal{V} ::= v_1 \wedge v_2 \mid v_1 \vee v_2 \mid \neg v_1 \mid ap \quad (3)$$

with $ap \in AP$. Conceptually, a step checklist is a list of steps a pseuCo program has to complete – in a fixed order, in arbitrary order, or in arbitrarily nested fixed-order and free-order blocks.

Together with the atomic propositions described previously, step checklists allow a compact representation of properties. For example, the first message passing exercise in pseuCo Book uses the atomic propositions *startAgent* that holds when an agent is started, $agents(x)$ that holds when the set of agents participating in a step is exactly x, $handshake(v)$ that holds when value v is passed by handshaking, and $print(v)$ that holds when value v is printed:

$$\text{Sequence}\begin{pmatrix}\text{Step}\,(startAgent)\,,\\ \text{Step}\,(agents(\{0,1\}) \wedge handshake(\texttt{"World"}))\,,\\ \text{Step}\,(agents(\{1\}) \wedge print(\texttt{"Hello,␣World!"}))\end{pmatrix} \quad (4)$$

This step checklist ensures the main agent starts an agent, passes `"World"` to it, after which that agent prints a greeting.

A step checklist s can easily be converted into an LTL property $[\![s]\!]$:

$$ltl(\text{Step}\,(v)) \coloneqq v \tag{5}$$

$$ltl(\text{Sequence}\,(s_1, s_2, \ldots, s_n)) \coloneqq ltl(s_1) \wedge \bigcirc \diamond\,(ltl(s_2) \wedge \bigcirc \diamond\,(\ldots ltl(s_n))) \tag{6}$$

$$ltl(\text{Parallel}\,(s_1, \ldots, s_n)) \coloneqq (\diamond ltl(s_1)) \wedge \cdots \wedge (\diamond ltl(s_n)) \tag{7}$$

$$[\![s]\!] \coloneqq \diamond ltl(s) \tag{8}$$

3 Verification

LTL formulas can be model checked efficiently by conversion to a Büchi automaton [6, 7]. Here, we follow the same approach, with some optimizations and extensions specific to our use case.

3.1 Implementation & Integration into pseuCo Book

For use in pseuCo Book's programming exercises, the verification system has been implemented in JavaScript, based on the pseuCo-to-CPN compiler and the `colored-petri-nets` JavaScript library [5] for pseuCo Semantics. The step checklist created by the exercise designer is converted directly to an automaton, skipping the intermediate LTL step for efficiency. An exhaustive search of the cross product of this automaton and the reachability graph of the Petri net is then performed. For efficiency, the atomic propositions are not precomputed, but dynamically evaluated during search. Verification starts only on demand, when the user explicitly "submits" their program. All computation is done locally in the user's browser, allowing offline use, without relying on a centralized service. Using the Web Worker API, all computation is performed in a background thread, ensuring the UI stays reactive and verification can be cancelled if needed.

This verification technology backs the interactive exercise in pseuCo Book's new *Programming with pseuCo* chapter. It is not otherwise accessible to the user – notably, users cannot input new specifications.

When verification succeeds, the corresponding exercise is marked as solved. (Students can refine or re-do their solution and run verification again, but the exercise will continue to be marked as solved.)

When verification fails, a failure trace is generated – a sequence of Petri net firings. Students are then presented with an error message stating that an execution of their program failed to meed the specification, as demonstrated in Fig. 1. To generate this error message, in the implementation, atomic propositions are associated with a human-readable description of the behavior they are looking for, which are then used to assemble the final error message, for example:

> The program has terminated or deadlocked. It was expected to send `"Hello"` on an asynchronous channel from the main agent.

In addition, users are given the option to inspect the failure trace. This trace by itself is not suitable to show to users of pseuCo Book as they are not expected to know Petri nets semantics nor the details of the pseuCo-to-CPN translation.

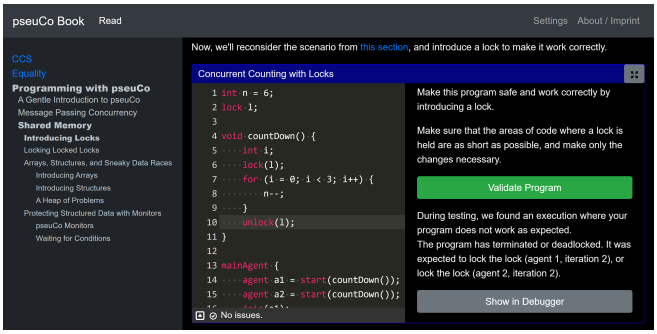

Fig. 1. Screenshot of a programming exercise in pseuCo Book containing a fault

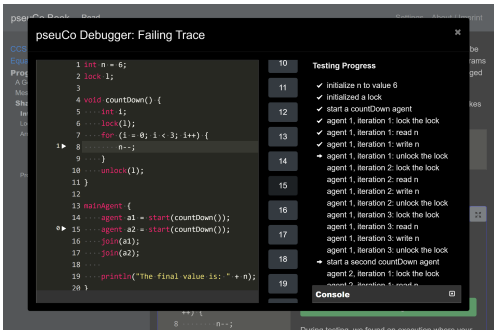

Fig. 2. Screenshot of a failing trace as shown in the debugger view

Fortunately, pseuCo.com, the web IDE for pseuCo, contains the *pseuCo Debugger* [5], a debugger-like interface that allows "executing" a pseuCo program like in a traditional IDE while maintaining full control over all possible executions allowed by the language specification. PseuCo Debugger is also built on top of the pseuCo-to-CPN toolchain – technically, it is a tool to explore the reachability graph of a colored Petri net, but it uses the labelling function *labels*[2], plus knowledge of the internals of the pseuCo-to-CPN compilation, to convert the marking into pseuCo terminology, fully hiding the Petri net and instead showing a debugger-style interface.

This allows us to use the existing UI of pseuCo Debugger to visualize failing traces. When verification fails, users are given an option to invoke the debugger. This launches a slightly modified version of pseuCo Debugger, shown in Fig. 2, with the following differences from the original version:

– The trace is fixed: The debugger opens with the full trace already pre-selected, and users have no option to view or change nondeterministic choices – they can only navigate forwards or backwards in time.

[2] The labelling function used by pseuCo Debugger is identical to the one used in the verification system.

– For every step of the trace, the debugger also shows the *verification state*: The step checklist is converted into a flat list, presented within the debugger interface like a todo list. (This does not allow users to see which steps are sequential and which are in parallel order, but a checkmark and an arrow will highlight items that are completed or currently due, respectively).

This allows the user to not only see the full sequence of actions their program took, presented in the style of a traditional high-level debugger, but also shows them how their program progresses (or fails to progress) through the step checklist, helping them discover where their program deviated from the specification.

3.2 Advanced Validation Features

While pseuCo Book's exercises generally use step checklists as described above, some exercises need additional expressivity for their properties, or additional features that do not fit into the theoretical framework described previously.

The following sections give an overview of these extensions as implemented in pseuCo Book.

Fail Fast & Cycle Detection. As described previously, step checklists always correspond to LTL formulas starting with $\diamond$, and thus can only be violated by pseuCo programs terminating (or diverging) without having satisfied the requirements. While this is sufficient to express most relevant properties for the exercises in pseuCo Book, such properties often yield unhelpful error messages.

For example, consider a problem statement asking students to compute 3!, then printing it. Assume a student solution contains a mistake that causes the program to compute and print an incorrect solution. Then, the error message generated from the underlying LTL formula will complain that the program terminated without printing 6, which mosts students will find less helpful then being informed that their program printed an incorrect value.

To improve this, in the implementation, step checklists can also *ban* groups of atomic proposition. If such an atomic proposition is encountered while the corresponding step is active, verification is immediately terminated, and a custom error message is returned as the verification result. This can then be used by the exercise designer to explicitly ban "near-miss" behavior.

Similarly, pseuCo programs allowing cycles are typically incorrect and will fail verification (because they permit an execution that spins without making progress, therefore never completing the step checklist). To speed up execution and provide better error messaging, unless otherwise configured, the verifier will identify cycles and abort verification with an error should one be found.

Syntactic Checking. In the Programming chapter of pseuCo Book, most exercises will want to control exactly how students use concurrency features. Part of this control is to ensure that students do not use shared memory in exercises about message passing, and vice versa.

This can, of course, be handled semantically by failing verification when an unauthorized concurrency feature is used. However, this is unnecessarily slow and complicated. A simpler approach, implemented in pseuCo Book, is to run an additional suite of syntactic checks when requested by the exercise, forbidding message passing or shared memory constructs as needed.

These checks run even before the user requests validation, after parsing and type-checking, allowing violations to be shown live in the editor UI, just like other types of syntactical errors. This also helps create the impression that the message passing and shared memory parts of pseuCo constitute different dialects of the language that cannot be freely mixed.

Banning message passing is done by simply banning any declaration of channel variables. Detecting and preventing shared memory is slightly more involved:

- Global declarations are banned, unless they are channels, and global channel variables cannot be assigned to.
- Locks and monitors cannot be declared, and `join()` statements are forbidden, all of which are considered shared-memory features.
- Procedures that take structures and arrays cannot be started. This prevents usage of shared memory by sharing pointers to heap-stored data structures across thread boundaries.
- Methods of structs cannot be started, and `start()` calls cannot be within a struct. This prevents sharing the implicit reference to the structure ("`this`-reference") between threads.

Together, these rules ensure that the only data that can be shared between agents directly is read-only global channel declarations.

Firing Set Validation & Data Race Detection. Most exercises in the shared memory section want to disallow *data races*. A data race occurs when a program can access a variable by two agents in parallel, with at least one access being a write access.

In the Petri net semantics of pseuCo, a data race is a marking that permits firings that are *conflicting*, i.e. (a) they represent actions taken by different agents, (b) they encode a global variable access to the same variable, (c) at least one access is writing, and (d) they access different *paths*.[3]

Formally, this can be made accessible from LTL by introducing a new atomic proposition that is applied to firings originating from markings that permit conflicting firings.

In the implementation, this is handled by extending the verification algorithm: In addition to the step checklist, a function *firingSetAllowed* can be passed to the verifier. For each marking of the Petri net, after evaluating the set of enabled firings, this function can inspect this set and may reject the combination of firings. To prevent data races, if enabled by the exercise, pseuCo Book applies a *firingSetAllowed* function that identifies conflicting firings and, if one is found, triggers a verification failure with an error message explaining the data race.

[3] This allows e.g. concurrent access to different indices of arrays, or different fields of a struct, despite these being stored within the same variable internally.

Restricted Actions. In some exercises, the exercise designer wants to strictly control the use of certain language features, e.g. message passing. To allow expressing this easily, without needing the full power of LTL, groups of atomic propositions can be declared as *restricted* in an exercise. Firings of the Petri net that use any restricted action are permitted only if they are required by the step checklist, i.e. advance the program's progress through the checklist.

Testing Mode. In practice, a small percentage of exercises in pseuCo Book permit solutions that exceed verification times that we consider acceptable in a teaching context (about 10 seconds on a reasonably modern computer and browser). For these exercises, at the expense of soundness, the framework can be configured to use *testing mode*. This replaces the exhaustive search with a fixed number of random walks, providing a compromise between a reasonably safe assurance a student's solution is correct and verification times.

4 Programming in pseuCo Book

Using the verification technology discussed in the previous section, we have expanded pseuCo Book to include a new chapter on Programming with pseuCo. Targeted at students that are already familiar with single-threaded programming in languages like Java or C, this chapter does not introduce any programming basics, but assumes it is a student's first contact with practical *concurrent* programming. It provides programming exercises that guide students through their first uses of all relevant concurrency features and concepts covered in the chapter, leaving more complex exercises to be covered in traditional exercise sheets of an accompanying lecture.

4.1 Structure

The chapter is structured into three main sections.

A Gentle Introduction to pseuCo. This section serves to introduce students to the pseuCo syntax and the basics of concurrency. It starts with a discussion of a simple, sequential pseuCo program, before asking students to write a *Hello World* program (using a procedure call). Next, another example introduces the `start()` statement, teaching students how to use the most basic form of concurrency (without any communication between the agents). To prevent students from accidentally using shared memory, global variables are banned for these examples.

Message passing is generally considered to be easier to use *correctly*, but to a naïve student, simply allowing shared global variables may appear easier. To motivate the need for more controlled communication mechanisms between threads, the introduction section closes with an "intermezzo" section explaining *The Dangers of Uncoordinated Access to Shared Memory*. First, this subsection asks students to write a variant of a *Hello World* program that is multi-threaded,

using a shared string variable to "send" a greeting message to a worker thread that reads and prints it. To analyze whether such a program is correct, pseuCo Book then discusses a traditional *concurrent counting* example, using two agents to repeatedly decrement a variable using a postfix expression (`n--`). Students are encouraged to analyze this program in pseuCo.com and asked to determine the possible outputs. Because the postfix decrement operation is not atomic, this example can produce surprising results. PseuCo Book discusses this, and the general risks associated with data races, concluding this section.

Message Passing Concurrency. This section introduces channels, beginning with synchronous (handshaking) channels. After a brief explanation of their syntax and use, students are asked to write a first minimal example, using a string channel to assemble and print a greeting in a worker agent.

Then, pseuCo Book introduces asynchronous (buffered) channels. To demonstrate their ability to store messages, students are asked to write a similar program than before, but this time writing a message to a channel *before* starting the agent that retrieves it.

Then, this section finishes introducing the essential message passing features by explaining the use of the `select case` statement as discussed in Section 2.1.

To provide students some guidance on how programs using message passing concurrency can be structured, this section concludes by introducing two standard concepts:

Producer & Consumer: To introduce the producer-consumer-pattern, pseuCo Book focuses on an example program that computes the series of factorials of prime numbers, i.e. 2!, 3!, 5!, 7!, and so on. Students are given a sequential implementation and are then asked to parallelize this by splitting prime finding and factorization into two agents, with an asynchronous channel in between. This constitutes a producer-consumer pattern with a single producer and consumer each. The section discusses this, as well as the implications of adding more producers and consumers.

Pipelining: To explain pipelining, this subsection focuses on prime generation. Assuming all primes up to $\sqrt{n}$ are already known, primes up to n can be found by testing whether any of the smaller primes is a factor of the number in question. This lends itself well to a pipelining approach where each candidate number is passed from agent to agent, with each agent eliminating multiples of one specific prime. The students are asked to implement this, writing a program that uses prime sieving agents for 2, 3, and 5 to identify all primes between 6 and 25.

Shared Memory Concurrency. This section lifts the restrictions on global variables and sharing references to heap-stored data and gives students the tools and knowledge to control the problems this causes.

After a brief reminder of the dangers of uncoordinated shared memory already explained in the intermezzo section before the introduction of message passing,

this section begins by introducing locks. Starting with a discussion of the features and correct usage of locks, the section revisits the concurrent counting example seen previously. In an exercise, students are asked to modify the program by adding locks to make it safe.

Next, pseuCo Book discusses reentrancy, i.e. the feature of locks allowing safely calling methods while already holding a lock needed by the callee, by allowing locks to be *re-entered* by the same thread as often as needed.

Then, pseuCo Book introduces arrays and structs. While these data structures, by themselves, are not related to concurrency, they introduce a potential source of hidden data races: In pseuCo, these data structures are stored on the heap. This means that a data race can be created without unsafely sharing a pseuCo variable directly, by copying a reference to an array or heap and then using both copies to access the same field on the heap concurrently. To illustrate this point, an exercise asks students to deliberately write a pseuCo program that has a data race without using any global variables.

Finally, pseuCo Book introduces monitors, i.e. data structures that handle protecting their data internally and can thus be used concurrently without additional protection from the outside.

In many languages, monitors are an implicit construct (e.g. achieved in Java by writing a class where all fields are `private` and all `public` methods are `synchronized`). PseuCo Book begins by introducing this concept abstractly, then asks students to apply it manually to a data structure called `MessageBox` implementing a simple, shared storage for a single integer. The corresponding exercise asks students to manually add a lock to a given template implementation of `MessageBox`, then write a sample program that uses two agents where one writes a message to the box and the other uses polling to retrieve the message as soon as possible.

In pseuCo, a `monitor` is an explicit language feature, automating the process of protecting all methods of a structure. A pseuCo `monitor` is similar to a `struct`, with the following differences:

- When a monitor is instantiated, an implicit, managed lock is initialized.
- The lock is acquired and returned automatically on all entry and exit paths of every method of the monitor.
- Monitors allow declaring and using `condition`s, allowing condition synchronization, i.e. waiting and signalling similar to e.g. Java's `wait()` and `notify()` mechanism.

PseuCo `struct`s already do not permit direct access to fields, so no change is needed in this regard.

After a brief explanation of pseuCo's `monitor` features, pseuCo Book discusses the problems associated with busy waiting (as used by the students in the previous exercise) and attempting to wait for another agent to make a change to a data structure while holding the lock to it. To finish this section, students are then asked to implement a `monitor`-based `MessageBox` (by adding condition synchronization to an otherwise complete template).

4.2 Exercises & Verification

Overall, the *Programming* chapter of pseuCo Book contains 11 programming exercises, spread throughout the chapter and integrated into the narrative.

10 of these exercise use true verification, with verification runtimes not exceeding 10 seconds on reasonably modern systems for the intended solutions. Some incorrect solutions can create longer runtimes or cause the verifier to diverge, e.g. when containing an infinite loop that changes the program state. If verification takes longer than a pre-determined warning threshold, students are informed that the process "is taking longer than usual" and are asked to check for infinite loops and try to simplify their program. This is merely a warning – students can wait for verification to finish, or cancel at any time. (As verification is run on the student's machine, there is no need to enforce a timeout.)

A single exercise, the prime sieve, is set to testing mode (see Section 3.2) to combat high verification runtimes. It uses 100 random walks to provide reasonable assurance that a submission is correct. This is also indicated in the exercise's UI.

Most exercises disallow infinite loops as a violation of the specification. A single exercise, the student's first attempt to manually create a monitor-like structure called `MessageBox`, allows – and in fact requires – the presence of an infinite loop. This does not pose any technical difficulty for verification as this loop does not significantly increase the size of the state space.

All programming exercises share the same user interface and logic. Therefore, adding a new exercise to pseuCo Book does not require custom code, only

- an exercise configuration file for the frontend, describing
  - which pseuCo dialects (message passing / shared memory) are allowed,
  - the description of the exercise as shown to students, and
  - the template given to students when they begin (if any); and
- a verification background worker, i.e. a verification configuration containing
  - the property to analyze, i.e. the step checklist, composing pre-made groups of atomic propositions,
  - any additional checks (e.g. syntax checks or firing set validators), and
  - whether to use testing mode and if so, the number of traces to check.

This is demonstrated in Listings 1.2 and 1.3, showing the internal definition of the "Hello Message Passing World!" exercise.

5 Conclusion

In this paper, we have developed an extension of pseuCo Book: A chapter on concurrent programming, designed as the first contact point of students with concurrency in practical programming. It uses pseuCo, a programming language specifically designed for teaching, to introduce both message passing and shared memory concurrency features. Integrated programming exercises help give students their first experiences with concurrency in a controlled environment. An integrated autograder, running directly inside the web app on students'

Listing 1.2. Exercise configuration for the "Hello Message Passing World!" exercise

```
const config: PseuCoProgrammingConfiguration = {
    allowedDialects: { // configure text editor
        mp: true,      // message passing = OK
        sm: false      // shared memory = syntax error
    },
    exerciseDescription: <div>
        <p>Write a pseuCo-MP program that prints <code>"Hello, World!"</code> by
         ↳ sending <code>"World"</code> on a synchronous channel to an agent that
         ↳ assembles and prints the greeting.</p>
    </div>,
    getWorker: () => new Worker(new URL('./worker.ts', import.meta.url))
                     // call this worker for verification
};
```

Listing 1.3. Definition of the background worker handling verification in the "Hello Message Passing World!" exercise (see also Eq. (4))

```
const validatorConfiguration: FlowReachabilityGraphValidatorConfiguration = {
    steps: {
        order: "sequential",
        steps: [{
            moveCompletesStep: moveValidatorStartAgent(),
            description: `start an agent`
        }, {
            moveCompletesStep: moveValidatorAnd(moveValidatorParticipatingAgents
             ↳ ([0, 1]), moveValidatorHandshaking((v) => /^World!?$/i.test(v.
             ↳ toString()), false)),
            description: `send "World" from the main agent to the first started
             ↳ agent on a synchronous channel`
        }, {
            moveCompletesStep: moveValidatorAnd(moveValidatorParticipatingAgents
             ↳ ([1]), moveValidatorPrintLn(/^Hello,? World!?$/i, false)),
            description: `print "Hello, World!" (from the first started agent)`
        }]
    },
    restrictedMoves: [{ // no message passing except as required above
        detector: moveValidatorMessagePassing(),
        description: "performed a message-passing action"
    }]
};
registerValidationWorkerCallback(flowReachabilityGraphValidator(
 ↳ validatorConfiguration), { mp: true, sm: false });
```

devices, implements LTL-based model checking, backed by pseuCo's Petri net semantics. It relies on a set of rich, compiler-generated labels on the Petri net. This enables more than verifying the externally visible behavior of the program: It grants a deep insight into the internal workings of the program, ensuring the programming exercises are solved using the concurrency control mechanisms specified by the exercise designer. The new chapter of pseuCo Book is freely accessible at `https://book.pseuco.com/#/read/pseuco/`.

Acknowledgements. This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 389792660 – TRR 248 – CPEC, see `https://perspicuous-computing.science`. The author would like to thank Holger Hermanns for his contributions.

References

1. Biewer, S., Freiberger, F., Held, P.L., Hermanns, H.: Teaching Academic Concurrency to Amazing Students. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. LNCS, vol. 10460, pp. 170–195. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-319-63121-9_9
2. Bruzual, D., Freire, M.L.M., Francesco, M.D.: Automated Assessment of Android Exercises with Cloud-native Technologies. In: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020, pp. 40–46. ACM (2020). https://doi.org/10.1145/3341525.3387430
3. Carbunescu, R., Devarakonda, A., Demmel, J., Gordon, S.I., Alameda, J., Mehringer, S.: Architecting an autograder for parallel code. In: Annual Conference of the Extreme Science and Engineering Discovery Environment, XSEDE '14, Atlanta, GA, USA - July 13 - 18, 2014, 68:1–68:8. ACM (2014). https://doi.org/10.1145/2616498.2616571
4. Freiberger, F.: pseuCo Book: An Interactive Learning Experience. In: ITiCSE 2022: Innovation and Technology in Computer Science Education, Dublin, Ireland, July 8 - 13, 2022, Volume 1, pp. 414–420. ACM (2022). https://doi.org/10.1145/3502718.3524801
5. Freiberger, F., Hermanns, H.: Concurrent Programming from pseuCo to Petri. In: Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings. LNCS, vol. 11522, pp. 279–297. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-21571-2_16
6. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_6
7. Gerth, R., Peled, D.A., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995. IFIP Conference Proceedings, pp. 3–18. Chapman & Hall (1995)
8. Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use – Volume 1. Springer (1992)
9. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pp. 46–57. IEEE Computer Society (1977). https://doi.org/10.1109/SFCS.1977.32
10. Stahlbauer, A., Kreis, M., Fraser, G.: Testing scratch programs automatically. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pp. 165–175. ACM (2019). https://doi.org/10.1145/3338906.3338910
11. Wang, W., Zhang, C., Stahlbauer, A., Fraser, G., Price, T.W.: SnapCheck: Automated Testing for Snap*!* Programs. In: ITiCSE '21: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education

V.1, Virtual Event, Germany, June 26 - July 1, 2021, pp. 227–233. ACM (2021). https://doi.org/10.1145/3430665.3456367

Copyright Notice