PrDK: Protocol Programming with Automata

Sung-Shik T.Q. Jongmans[1,2(✉)] and Farhad Arbab[3]

[1] Open University, Heerlen, The Netherlands
ssj@ou.nl
[2] Radboud University Nijmegen, Nijmegen, The Netherlands
[3] Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands

Abstract. We present PrDK: a development kit for programming protocols. PrDK is based on syntactic separation of process code, presumably written in an existing general-purpose language, and protocol code, written in a domain-specific language with explicit, high-level elements of syntax for programming protocols. PrDK supports two complementary syntaxes (one graphical, one textual) with a common automata-theoretic semantics. As a tool for construction of systems, PrDK consists of syntax editors, a translator, a parser, an interpreter, and a compiler into Java. Performance in the NAS Parallel Benchmarks is promising.

1 Introduction

In the early 2000s, hardware manufacturers shifted their attention from manufacturing faster—yet purely sequential—unicore processors to manufacturing slower—yet increasingly parallel—multicore processors. In the wake of this shift, *concurrent programming* became essential for writing scalable programs on commodity hardware. Conceptually, concurrent programs consist of *processes*, which implement primary modules of sequential computation, and *protocols*, which implement the rules of concurrent interaction that processes must abide by.

As programmers have been writing sequential code for decades, implementing processes poses no new fundamental challenges. What *is* new—and notoriously difficult—is programming protocols. One contributing factor to the complexity of this activity is today's popular programming languages not providing programmers explicit, high-level elements of syntax for programming protocols. Instead, programmers need to use rather low-level reads/writes to shared memory protected by mutual exclusion—locks, semaphores, monitors, and the like.

In a long-term project at CWI, we study an alternative approach to concurrent programming, based on syntactic separation of processes from protocols. In this approach, programmers write their (sequential) processes in a *general-purpose language* (GPL), while they write their (concurrency) protocols in a *domain-specific language* (DSL). Paraphrasing the definition of DSLs by Van Deursen et al. [3], a DSL for protocols "is a programming language that offers, through appropriate notations and abstractions, expressive power focused on, and [..] restricted to, [programming protocols]." The semantics of our DSL is based on automata; on top of it, we have both a graphical and a textual syntax.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 547–552, 2016.
DOI: 10.1007/978-3-662-49674-9_33

```
public interface OutputPort extends Port {
  public void put(Object obj)
    throws InterruptedException;
  public void putUninterruptibly(Object o);
  public void resume();
}

public interface InputPort extends Port {
  public void get(Object obj)
    throws InterruptedException;
  public void getUninterruptibly(Object o);
  public void resume();
}
```

```
public class Processes {
  public static void Producer(
      OutputPort p, int id) {
    String message = id + ": Hello, World!";
    while (true)
      p.putUninterruptibly(message);
  }

  public static void Consumer(InputPort p) {
    while (true)
      System.out.println(
        p.getUninterruptibly());
} }
```

Fig. 1. API for ports (left) and example hand-written processes (right) in Java

In this paper, we present a development kit for our DSL for protocols. In Sect. 2, we briefly present the DSL. In Sect. 3, we present our development kit, available at http://www.open.ou.nl/ssj/prdk. Section 4 concludes this paper with some performance numbers and future work. We invite the reader to consult the first author's PhD thesis for details and examples [7].

2 The DSL

Processes, implemented in a GPL, primarily perform sequential computations. To interact with each other, in our programming model, every process also owns a set of *ports*. Ports mark the interface between processes: *output ports* let processes *offer* data to other processes, while *input ports* let processes *accept* data from other processes. Processes can perform two blocking operations on ports: `put` and `get`. When a process performs a `put` (`get`) on an output port (input port), this operation becomes *pending* on that port and the process itself becomes suspended. When a `put` (`get`) completes, its previously suspended process resumes and offers (accepts) a datum. Whenever a process offers (accepts) a datum in this way, it does not know whereto (wherefrom) this datum goes (comes); only protocols, programmed as syntactically separate modules from processes through explicit, high-level elements of syntax in a DSL, control when `put`/`get` operations may complete on which ports and how data *flow* between ports. As such, protocols effectuate only *admissible interactions* among (the ports of) the processes in a program. We stipulate that `put`/`get` have *value passing* semantics (although programmers are free to pass and interpret references to shared data as values). Figure 1 shows an API for ports and two processes in Java, defined as two static methods (*not* directly as Java threads, which programmers do not need to manually manage, or even know about, in our programming model). The actual API also has versions of `put`/`get` with timeouts (omitted here to save space).

By effectuating only admissible interactions, protocols essentially constrain the completion of `put`/`get` operations. Formally, we can represent such constraints with *automata* [7], whose every transition models a data-flow between ports with a pending `put`/`get` operation. Figure 2 shows an example. The automaton in this figure models a producers/consumer protocol involving two output ports A and B (each owned by a different producer, presumably) and an input

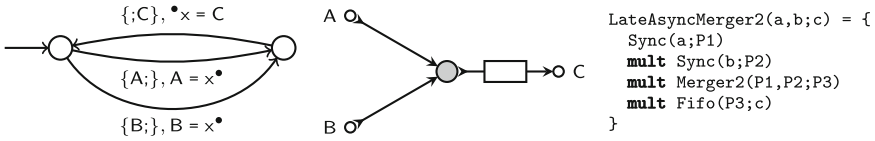

Fig. 2. Example automaton for a producers/consumer protocol (left), its graphical syntax (middle), and its textual syntax (right)

port C (owned by the consumer). Initially, a `put` by the producer owning A can complete, causing that producer to offer a datum into internal buffer x (modeled by expression A = x•). Alternatively, a `put` by the producer owning B can similarly complete. Subsequently, only a `get` by the consumer owning C can complete, causing the consumer to accept the datum previously stored in x (modeled by expression •x = C). This protocol, thus, admits asynchronous, unordered, reliable, transactional communication from two producers to a consumer.

Providing programmers syntax for writing protocols directly as automata has at least one major issue: automata quickly grow prohibitively large. A more scalable approach for defining automata is one based on their (parallel) *composition*: programmers should construct complex protocols out of simpler ones, by composing (*multiplying*) smaller automata into larger ones, starting from a predefined "core set" of primitive automata. We consider two declarative syntaxes for representing such multiplication expressions: *Reo* [1] and *Pr* [7]. Given such a core set, in Reo, programmers *draw* multiplication expressions as data-flow graphs; in Pr, programmers *write* multiplication expressions as automata signatures. Figure 2 exemplifies both Reo and Pr (for the same protocol). In the graph, every node/vertex denotes a primitive automaton in the core set; in the text, the same applies to every signature (and their multiplication is, in turn, denoted by a new signature `LateAsyncMerger2`).

3 The Development Kit

Our development kit, called PrDK, consists of tools (Eclipse plugins) for protocol programming with automata (without ever exposing programmers to automata directly): editors for Reo and Pr, an animation engine for Reo, a parser/interpreter for Pr, a Reo-to-Pr translator, and a Pr-to-Java compiler. The Reo editor and its animation engine have previously been developed as part of the ECT (http://reo.project.cwi.nl), a collection of Eclipse plugins for Reo.

In PrDK's basic workflow, programmers start by drawing a protocol as a Reo graph for a small number of processes, using the drag/drop interface of the Reo editor. The animation engine enables programmers to visualize the admissible data-flows through the graph, which is an instructive and helpful aid in protocol debugging. Subsequently, programmers can import processes, by drag/dropping Java files onto the same canvas (which appear as boxes alongside the graph, with distinct markers for their ports), and *link* (the ports of) those processes to (the nodes in the graph of) the protocol as desired. The resulting diagram

```
public class Protocol extends Thread {
  private Port A;
  private Port B;
  private Port C;

  public Protocol(Port A, Port B, Port C) {
    ...
  }

  // Event-driven code to simulate an
  // an automaton by firing its transitions:
  ...
}
```

```
public class Program {
  public static void main(String[] args) {
    OutputPort A = new OutputPortImpl();
    OutputPort B = new OutputPortImpl();
    InputPort C = new InputPortImpl();
    (new Protocol(A,B,C)).start();
    (new Thread() { public void run() {
        Process.producer(A,18); } } ).start();
    (new Thread() { public void run() {
        Process.producer(B,06); } } ).start();
    (new Thread() { public void run() {
        Process.consumer(C); } } ).start();
} }
```

Fig. 3. Example compiler-generated protocol (partial) and `main` in Java

comprehensively implements a full program. By invoking the Reo-to-Pr translator, the Pr parser/interpreter, and the Pr-to-Java compiler on this diagram, PrDK generates Java code for the protocol and merges this compiler-generated code with hand-written code for processes according to their links in the diagram (detailed below). A Java compiler, then, can translate everything into an executable binary. In the basic workflow, the Pr syntax is completely hidden from programmers (i.e., the Reo-to-Pr translator, the Pr parser/interpreter, and the Pr-to-Java compiler are transparently chained, giving the programmer the illusion of a Reo-to-Java compiler).

Often, programmers need different versions of a program with different numbers of processes (e.g., depending on the number of cores of the target hardware). The Reo syntax does not conveniently support this. For instance, Reo requires programmers to draw a *specific* diagram for a protocol among two processes, another *specific* diagram for the same protocol among three processes, etc.; Reo does not support drawing a *generic* diagram for k producers and one consumer. Pr, in contrast, does support such parametrization. The basic workflow can, thus, be extended with an extra step in which programmers explicitly use the Reo-to-Pr translator to translate their Reo diagram into a Pr text, which they subsequently can modify by parametrizing the protocol in its number of ports.

From a theoretical prespective, the most interesting tools in PrDK are the Pr parser/interpreter and the Pr-to-Java compiler. The parser consumes a Pr program as input and produces a syntax tree as output (if the input unambiguously satisfies Pr's concrete syntax); we implemented the parser using the Antlr parser generator. The interpreter consumes a syntax tree (produced by the parser) as input and produces a list of automata, which represents a multiplication expression of automata, as output (if the input is well-typed). Finally, the compiler consumes a list of automata for a protocol (produced by the interpreter) and a list of method signatures for processes (in the syntax tree produced by the parser) as input and produces Java code as output.

Roughly, the compiler and its generated code work as follows. First, the compiler computes the product of the automata in its input list. Second, the compiler translates the resulting product automaton (which comprehensively models a protocol) into a singleton Java class (which effectively encapsulates a

state machine for simulating that automaton). The constructor of such a class has a number of formal port parameters, to bind its single instance to actual ports at "construction-time". After construction-time, then, a thread monitors these bound ports for new `put/get` operations performed by processes. Whenever a `put/get` occurs, this thread checks if that operation—together with the already pending `put/get` operations—enables the firing of a transition out of the current state. If so, the thread makes that transition and completes the `put/get` operations involved. As the constructor of a compiler-generated "protocol class" (e.g., Fig. 3), hand-written "process methods" (e.g., Fig. 1) have formal port parameters, to bind thread-wrapped calls of those methods to actual ports at construction-time. The task of constructing ports and passing them *both* to the constructor of a protocol class *and* to process methods is performed in the `main` method. This `main` method is, as the protocol class, generated by the compiler (based on linkage information either in a Reo diagram or in its Pr equivalent).

We significantly simplified our description of the workings of the compiler and its generated code. For instance, we tacitly assumed that a program consists of only one protocol, but PrDK supports also programs with multiple protocols. Also, notably, while computing the product of automata, the compiler applies a number of provably correct (i.e., bisimulation-preserving) optimizations and automata transformations to improve the performance and scalability of its generated code. We presented these optimizations in previous work [8–11]; a comprehensive overview, including formal definitions and proofs of their correctness, appears elsewhere [7]. Also, although PrDK currently supports only Java as the target GPL, we do not use any Java-specific features; our choice for Java is, in that sense, arbitrary. Our only requirement for a target GPL is that it supports some form of multithreading. For instance, extending the compiler with support for C+Pthreads is straightforward, as already worked on by a MSc student [12].

4 Conclusion

To evaluate the performance of the code generated by the compiler in PrDK, we compared the Java reference implementation of the NAS Parallel Benchmarks [4]—a popular benchmark suite for parallel performance—against an implementation developed with PrDK, on a machine with 24 cores using the workflow described in Sect. 3. In seven benchmarks, we considered six numbers of processes (2, 4, 8, 16, 32, 64) for various problem sizes, yielding a total of 126 tests. Figure 4 summarizes our results, where every bar represents the percentage of times the PrDK-based implementation achieved a certain speedup relative to the reference implementation. In 37 % of cases (gray bars), the PrDK-based implementation is *at most* only 10 % slower than the reference implementation; in 38 % percent of cases (black bars), the PrDK-based implementation is faster. Given the high level of abstraction supported by Reo/Pr, and the

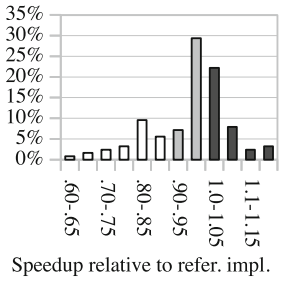

Fig. 4. Benchmark results

consequent burden carried by the compiler—instead of the programmer—to produce efficient code, these are promising first results. Details appear elsewhere [7].

Another recent initiative based on syntactic separation of processes from protocols is *Scribble* [5,13]. In Scribble, protocols are expressed through multiparty session types [6]. One fundamental difference between Scribble and our approach is that in Scribble, all interaction is asynchronous, order-preserving, and reliable, whereas our automata allow for mixing synchrony and asynchrony (in the same protocol) and support nondeterminism (both of orderings and reliability).

Our present version of PrDK does not include previous verification tools for Reo, notably model checking [2]. We are currently investigating how to best integrate those existing tools for a seamless implementation/verification experience.

References

1. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011)
2. Baier, C., Blechmann, T., Klein, J., Klüppelholz, S., Leister, W.: Design and verification of systems with exogenous coordination using Vereofy. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 97–111. Springer, Heidelberg (2010)
3. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Not. **35**(6), 26–36 (2000)
4. Frumkin, M., Schultz, M., Jin, H., Yan, J.: Performance and scalability of the NAS parallel benchmarks in Java. In: Proceedings of IPDPS 2003, p. 139 (2003)
5. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011)
6. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: ACM SIGPLAN Notices, Proceedings of POPL 2008, vol. 43, no. 1, pp. 273–284 (2008)
7. Jongmans, S.S.: Automata-Theoretic Protocol Programming. Ph.D. thesis, Universiteit Leiden (2016)
8. Jongmans, S.-S.T.Q., Arbab, F.: Take command of your constraints!. In: Holvoet, T., Viroli, M. (eds.) Coordination Models and Languages. LNCS, vol. 9037, pp. 117–132. Springer, Heidelberg (2015)
9. Jongmans, S.S., Arbab, F.: Global consensus through local synchronization: a formal basis for partially-distributed coordination. Sci. Comput. Program. **115–116**, 199–224 (2016)
10. Jongmans, S.-S.T.Q., Halle, S., Arbab, F.: Automata-based optimization of interaction protocols for scalable multicore platforms. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 65–82. Springer, Heidelberg (2014)
11. Jongmans, S.S., Santini, F., Arbab, F.: Partially-distributed coordination with reo and constraint automata. Serv. Oriented Comput. Appl. **9**(3), 311–339 (2015)
12. van de Nes, M.: Developing Efficient Concurrent C Application Programs Using Reo. Master's thesis, Universiteit Leiden (2015)
13. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The Scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Heidelberg (2014)