R-SHIM: Deterministic Concurrency with Recursion and Shared Variables

Olivier Tardieu and Stephen A. Edwards*
Department of Computer Science, Columbia University, New York

Abstract

Concurrent programming languages are good for embedded systems because they match the parallelism of their environments, but most concurrent languages are nondeterministic, making coding in them unwieldy.

We present R-SHIM, the core of a language with concurrent recursive procedure calls and disciplined shared variables that remains deterministic—the behavior of a program is scheduling-independent.

1. Introduction

We extend our SHIM model [1], based on CSP-style rendezvous [2] with Kahn-like [3] determinism, with concurrent, recursive procedure calls and disciplined shared variables while guaranteeing its overall behavior remains independent of scheduling decisions. R-SHIM provides familiar by-value and by-reference mechanisms for passing information to called procedures and augments it with multiway rendezvous for communication among concurrently-running procedures. The result is a Java-like language that provides deterministic concurrency without requiring careful attention to the use of semaphores or monitors found in many concurrent programming languages.

2. The R-SHIM Language

R-SHIM consists of an imperative C-like language that includes local variable declarations and procedure calls, but not pointers, augmented with concurrent procedure calls (the *par* keyword) and rendezvous-style inter-process communication through the *next* statement.

$e ::= L \mid V \mid op_1\ e \mid e\ op_2\ e \mid$ `(` e `)`	**expressions**
$c ::= P$`(` $V(,V)^*$ `)`	**procedure calls**
$s ::= V$ `=` e`;` $\mid$ `next` V`;` $\mid c$ (`par` $c)^*$`;`	**statements**
$\mid$ `{` s^* `}` $\mid$ `if (` e `)` s `else` $s \mid$ `while (` e `)` $s \mid T\ V$`;`	
$d ::= T\ V \mid T$ `&`V	**parameter declarations**
$p ::=$ `void` P`(` $d(,d)^*$ `) {` s^* `}`	**procedure declarations**
$p ::= p^*$ `void main() {` s^* `}`	**programs**

L are literals, T types, V variables, and P procedures.

Parameters are normally passed by value; parameters prefixed by an & are passed by reference. Each procedure receives its own copy of a pass-by-value variable that it can modify independently. To avoid aliasing problems, each variable may be passed by reference at most once per group of concurrent procedures.

```
void f(int  a) { a = a + 1; } // a passed by value
void g(int &a) { a = a + 5; } // a passed by reference
void main() {
  int a; a = 5;
  f(a) par g(a);  // a is 10 after the call
}
```

*{tardieu,sedwards}@cs.columbia.edu Edwards and his group are supported by the NSF award, gifts from Intel and Altera, an award from the SRC, and by New York State's NYSTAR program.

Passing a variable to multiple concurrent procedures allows it to be used to communicate among the procedures. Concurrently-running procedures normally execute asynchronously, but when a procedure executes *next v* on a variable *v*, it waits for all other procedures that "know about" *v* to synchronize before copying the master value of *v* to all local copies. This is one-to-many rendezvous communication that can deadlock but remains deterministic.

```
void f(int a)  { a = 3; next a; /* receive 5 */ }
void g(int &b) { b = 5; next b; /* send 5 */ }
void main() {
  int c; c = 0; f(c) par g(c); /* c = 5 afterward */
}
```

Recursion is a key contribution of R-SHIM over the earlier SHIM formalism. Concurrent, recursive procedure calls can be used to spawn an arbitrary number of threads under algorithmic control. The example below puts this to practical use by creating an *n*-place buffer by repeatedly calling the *buffer* procedure. Arguments to *fifo* are the input channel, output channel, and the desired buffer size.

```
void buffer(
 int i, int &o) {
  while (1) {
    next i;
    o = i;
    next o;
  }
}
```

```
void fifo(int i, int &o, int n) {
  int c; int m = n - 1;
  if (m) {
    buffer(i, c) par fifo(c, o, m);
  } else {
    buffer(i, o);
  }
}
```

3. Semantics and Implementation

We have formalized R-SHIM's semantics in the form of SOS rules to express transitions between program states. A state is a pair consisting of a *residue*—the code remaining to be executed—and a *store* with variable values. The residue is a tree of code fragments whose leaves run concurrently. Calling concurrent procedures adds children, which disappear on termination. Each tree node maintains a *view* that binds variable identifiers to locations in the store. Procedure parameters have a pair of locations—one for its current value and one for the "shared" variable, i.e., copied by *next*.

We also have an R-SHIM compiler that generates single-threaded C code. One challenge is that activation records must be stored on a heap instead of a stack because of the concurrency. The activation records contain links that are used to determine when and where to communicate.

References

[1] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, 2006. To appear.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[3] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.