Analyzing Java Classloader Deadlocks Using CSP and FDR

Jonathan Lawrence
IBM United Kingdom Limited
jlawrence@uk.ibm.com

Abstract

This paper describes a recent project within the IBM Java Technology Centre at Hursley, to use CSP and the FDR model-checking tool to analyse the cause of certain deadlocks within the Java class loader. Techniques for the CSP modelling of several procedural programming patterns such as recursion, multi-threading and locking are presented, together with their application to the specific case of the Java class loader.

Keywords: CSP, FDR, Java, model-checking, procedural programming

1. INTRODUCTION

The work described in this paper was motivated by the observation of deadlocks within the Java class loader under certain conditions involving multiple loaders. CSP (Communicating Sequential Processes process algebra, Hoare 1985) and FDR (Failures-Divergences Refinement model-checking tool, Formal Systems 2007) make an ideal combination for investigating such behaviour because the notation is suited to modelling concurrent program structures and the tool automates the checking of an implementation model against required behaviour expressed as a CSP specification. The emphasis of the paper is more on the techniques for modelling and verifying multi-threaded procedural software, rather than the details of the case-study, which has been greatly simplified.

2. MODELLING TECHNIQUES

It is useful to have a repertoire of techniques and standard patterns for procedural software modelling in CSP, as this makes the modelling work fast and repeatable, and results in more easily understood models. Some of the main patterns used in the class loader model are summarized here:

2.1 Procedural software

The software stack is modelled as an assembly of interacting processes which may be broadly categorised into two types: procedure or data. A procedure process typically has input and output channels representing call and return from methods. A data process may have several channels representing atomic operations on a datatype.

FIGURE 1: Typical structure of a CSP model of procedural software

In practice the distinction between procedure and data processes may be somewhat blurred. There does not necessarily have to be a 1-1 relation between methods and CSP processes, however this can help to make the model clearer.

2.2 Recursion

A recursive procedure requires a copy of the corresponding process for each level of recursion included in the model. Primed channel tags are a useful convention to represent invocation of the next level. The CSP_M (machine-readable dialect of CSP) replicated linked parallel construct provides a useful mechanism for assembling several levels of recursion.

FIGURE 2: Typical structure of a CSP model of a recursive procedure

The occurrence of a primed channel event indicates the equivalent of stack overflow and hence either unbounded recursion or that more levels (larger stack) are required in the implementation model.

2.3 Multi-threading

Each thread is represented by a separate instance of the procedure stack (including recursively invoked procedures if applicable) where external events are labelled with a unique thread ID. These thread-labelled instances are interleaved to represent the independent parallel execution of each thread, and then composed in parallel with singleton instances of the processes representing shared data or control components such as datatypes, monitors and locks.

2.4 Locking (synchronization)

CSP has no built in concept of a lock, so an explicit model of locking is required. In its initial state a lock may be obtained by any thread. Once locked, a lock may be locked or unlocked by the owning thread. Unlocking by any other thread is invalid, and no other thread may obtain the lock. The depth of nested locking must be bounded, or the lock process will be infinite and FDR will not be able to process it.

There are several ways to incorporate the locks into the model:

1. Add explicit lock/unlock events to the implementations of synchronized methods to invoke the lock processes directly.
2. Allow the locks to observe (& hence control) the entry/exit events for synchronized methods.
3. Other techniques, including hybrids of the above.

Option (2) is the least invasive and most flexible approach for the present case, since it allows the locking model to be modified while leaving the definitions of the methods unchanged: only the assembly of the system need be changed to alter the synchronization pattern. This will work as long as the relevant events are not hidden by the assembly constructs, e.g. linked parallel may not be used for channels which are to be synchronized.

2.5 Arbitrary structures

One of the strengths of FDR is its ability to search a state space for behaviour in conflict with a specification, including all possible outcomes of a non-deterministic choice. We can therefore use a non-deterministic model to search arbitrary data-structures. Non-determinism can be introduced explicitly through the CSP ND-choice operator or implicitly when a deterministic choice is hidden.

3. CASE STUDY

3.1 Specification

From the point of view of a single thread, the Loaders will load any class successfully via any loader, and then be ready to load the same or any other class again.

FIGURE 3: Single-threaded CSP specification of Java class loaders

When multiple threads are present, each thread should have the potential to behave as an independent, interleaved copy of `Loaders`.

```
MTLoaders = ||| t: ThreadId @ Loaders[[load_i <- t_load_i.t, load_o <- t_load_o.t]]
```

3.2 Implementation

The model presented here is an abstraction of the actual implementation, intended to model only relevant aspects of the design for the purpose of investigating erroneous classloader behaviour. The mapping between Java class loader methods and the corresponding CSP processes and associated input & output channels is given below:

Java method	CSP process(es)	Input channel	Output channel	Notes:
<code>loadClass()</code>	<code>LoadClass</code>	<code>loadc_i</code>	<code>loadc_o</code>	procedure
<code>findClass()</code>	<code>FindClass</code>	<code>findc_i</code>	<code>findc_o</code>	procedure
<code>resolveClass()</code>	<code>ResolveClass</code>	—	—	procedure
—	<code>ClassLoader</code>	<code>find</code> , <code>define</code>		data process

TABLE 1: Mapping from Java methods to CSP processes in the implementation model

`Wiring` implements an arbitrary but consistent mapping of `ClassId` to `LoaderId`: the initial choice of which loader to use for a given class is made on the first use of `Wiring` for that class. This choice becomes non-deterministic when the `wire` events are hidden during the assembly of the system. With this definition, FDR checks automatically include all possible distributions of classes between loaders.

```

channel wire: ClassId.LoaderId
Wire(c) = wire.c?l -> Wired(c,l)
Wired(c,l) = wire.c.l -> Wired(c,l)

```

```

Wiring = ||| c:ClassId @ Wire(c)

```

wire: ClassId.LoaderId

Wiring

FIGURE 4: CSP definition of the Wiring function from ClassId to LoaderId

`ClassLoader` is a simple datatype process and hence does not allow detection of possible interleaving of `find` and `define` methods on a class loader, however we can detect an attempt to find or define a class in the wrong loader, by diverging after an invalid access attempt or if `define` is invoked with the relevant class already loaded.

FIGURE 5: Schematic diagram of the ClassLoader data process

The main part of the implementation model comprises two procedure processes, `LoadClass` and `FindClass`, which itself has a sub-process `ResolveClass`. There is a recursive invocation of `LoadClass` which may originate from either process. The following diagram illustrates the structure of a single level of the stack:

FIGURE 6: Structure of a single level of recursion of the implementation processes

At present, recursion is limited to invocation of `loadClass()`, either by direct delegation to another class loader, or during resolution from `findClass()`. If too few levels are used, this results in an invalid refinement due to a primed event in the trace.

The use of replicated linked parallel is possible for assembling the recursive stack because `loadClass()` is not synchronized, so the `loadc_` events may be hidden at this stage of the assembly. If synchronization might ever be required on `loadClass()` then `loadc_` events would need to remain visible in order to communicate with the locks.

For the multi-threaded implementation model we simply replicate the code stack for each thread, labelling all events with the originating thread where relevant (i.e. all except interactions with `ClassLoaders`, `Wiring` and `Classes` which are thread agnostic).

The threads are assembled with `ClassLoaders` and `Wiring`, which are shared by all threads, hiding `ClassLoader` channels but leaving `findClass()` entry and exit channels visible for potential synchronization.

In this example we need a lock for each loader, and assembling the implementation requires a slightly complicated linkage because of the mismatch in channel types between the locks (`LoaderId.ThreadId`) and the synchronization events `{|t_findc_i,t_findc_o|}`.

3.3 FDR refinement model-checking:

The CSP_M script includes three refinement assertions:

1. The single-threaded specification is refined by the single-threaded implementation.
2. The multi-threaded specification is refined by the multi-threaded implementation without locking.
3. The multi-threaded specification is refined by the multi-threaded implementation with synchronization of the `findClass()` method.

In all cases the most general semantic model of CSP (failures-divergences) is used for the refinement check. The first of the above checks succeeds, while the last two do not. Use of the FDR debug tool reveals that the implementation without locking diverges due to duplicate `define()` invocations caused by a race between threads; while the synchronized version deadlocks as two threads may attempt to obtain the locks on the two class loaders in reverse order. In the latter case the debug tool also provides an example of a wiring and class hierarchy in which the deadlock arises.

4. CONCLUSION

This paper has illustrated some techniques for modelling procedural multi-threaded software in CSP with reference to a simple example derived from the analysis of deadlocks in the Java class loader, and shown how the FDR tool may be used to investigate possible behaviours of the system.

REFERENCES:

- [1] Hoare, C.A.R. (1985) *Communicating Sequential Processes*, Prentice-Hall International.
- [2] Roscoe, A.W. (1998) *Theory and Practice of Concurrency*, Prentice-Hall International.
- [3] Failures-Divergences Refinement, FDR2 User Manual (1998) Formal Systems (Europe) Ltd., Oxford.

TRADEMARKS:

IBM is a trademark of International Business Machines Corporation in the United States, or other countries, or both. Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.


```
[find <-> find,define <-> define] ClassLoaders)
[wire<->wire] Wiring
```

```
assert Loaders [FD= LoaderImplSingleThread
```

-- Add multi-threading

```
datatype ThreadId = t1 | t2
```

```
channel t_load_i,t_find_i,t_load_i': ThreadId.LoaderId.ClassId
channel t_load_o,t_find_o,t_load_o': ThreadId.LoaderId
```

```
MTLoaders = ||| t: ThreadId @ Loaders
    [[load_i <- t_load_i.t,load_o <- t_load_o.t]]
```

```
MTLoaderImplStack = ||| t: ThreadId @ LoaderImplStack
    [[load_i <- t_load_i.t,load_o <- t_load_o.t,load_i' <- t_load_i'.t,
    load_o' <- t_load_o'.t,find_i <- t_find_i.t,find_o <- t_find_o.t]]
```

```
LoaderImplMultiThread = ((MTLoaderImplStack[parent<->parent]Classes)
    [find <-> find,define <-> define] ClassLoaders)
    [wire<->wire]Wiring
```

-- Without locking, hide t_find_i/o events so they occur unhindered

```
LoaderImplNosync = LoaderImplMultiThread \ {|t_find_i,t_find_o|}
assert MTLoaders [FD= LoaderImplNosync
```

-- Locks (synchronization) with nested locking capability

```
channel lock,unlock : ThreadId
maxdepth = 3
```

```
Lock = lock?t -> Locked(t,1)    -- Locked once by any thread t
    [] unlock?_ -> DIV          -- Usage error, unlock not allowed
```

```
Locked(thread,depth) = lock.thread ->
    (if depth == maxdepth then DIV else Locked(thread,depth+1))
    [] unlock?t -> if t!=thread then DIV
        else (if depth==1 then Lock else Locked(thread,depth-1))
```

-- Labelled copy of Lock for each class loader.

```
channel llock,lunlock: LoaderId.ThreadId
Locks = ||| l:LoaderId @ Lock[[lock <- llock.l,unlock <- lunlock.l]]
```

-- Implementation with findClass() synchronization - link to Locks

```
LoaderImplLockFind = LoaderImplMultiThread
[t_find_i.t.l.c <-> llock.l.t , t_find_o.t.l <-> lunlock.l.t |
l <- LoaderId,t <- ThreadId,c <- ClassId] Locks
```

```
assert MTLoaders [FD= LoaderImplLockFind
```