An Asynchronous Java Interface to MATLAB

Andreas Naderlinger Josef Templ Stefan Resmerita Wolfgang Pree
C. Doppler Laboratory Embedded Software Systems
University of Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

ABSTRACT

MATLAB, an interactive environment for numerical and symbolic computation, supports a number of interfaces to foreign programming languages including Java. However, there is no appropriate support for calling back MATLAB functions from within the Java Virtual Machine integrated with MATLAB. This paper presents such an interface which is based exclusively on documented and portable mechanisms supplied by Java and MATLAB. Our approach is based on asynchronous communication between Java threads and MATLAB and follows the producer/consumer pattern. We also present performance measurements and discuss the impact of an optimization for calling MATLAB functions that return a result value back to Java.

Categories and Subject Descriptors

D.2.12 [**Interoperability**]

General Terms

Algorithms, Languages

Keywords

MATLAB, Simulink, Java, asynchronous, interface

1. INTRODUCTION

MATLAB from *The MathWorks* [9] is an extensible and interactive computing environment based on a high-level programming language called *M*. M is a compact, dynamically typed, interpreted language designed for scientific computations. While it is well suited for performing matrix operations, plotting graphs, and so forth, it is not a general-purpose programming language such as C or Java. For high performance computations or for providing full fledged graphical user interface components, for example, one has to combine MATLAB with a more efficient and flexible general purpose programming language.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

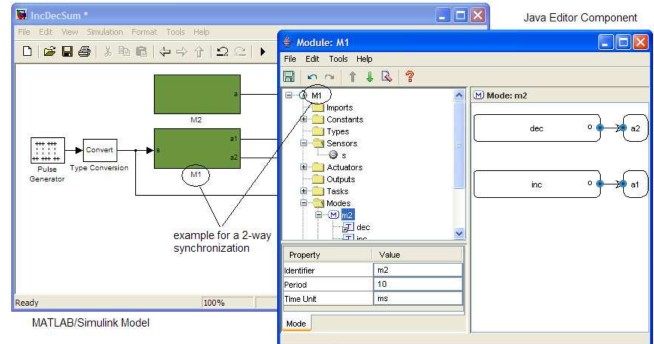

Figure 1: Two-way synchronization between a MATLAB/Simulink model and a Java component.

During the course of implementing a tool chain for deterministic and portable real-time systems, we extended MATLAB/Simulink with a graphical editor for describing the real-time behavior of a system [5]. This editor (Figure 1) is written in Java and supposed to be integrated seamlessly with Simulink. Thus, changes within a Simulink model must be reflected by the editor and vice versa. For the first direction, Simulink allows the installation of so-called *callback functions*, which are executed on certain changes in the Simulink model. It is possible to call a Java method from a callback function. For the opposite direction, the Java editor needs to initiate the evaluation of MATLAB code. Note that although Simulink is usually operated interactively, most tasks can also be performed via an API represented as a set of M functions. In addition to this two-way synchronization, the editor also needs to transform the Simulink model such that the specified properties are considered during a simulation run. This transformation consists of hundreds or even thousands of elementary operations. Since the core of our tool chain was programmed in Java anyway and MATLAB supports the integration of Java, it was a natural choice to implement this MATLAB/Simulink extension in Java.

In the subsequent sections we outline MATLAB's capabilities to integrate foreign programming languages in general and Java in particular. We shall see that one important scenario is currently not supported. There is no appropriate support for calling back M functions from within the Java VM integrated with MATLAB. Section 3 then presents a solution for this case. Finally, we evaluate and compare the performance of the presented approach and discuss the effect of an optimization for calls that return a value.

2. MATLAB PROGRAMMING MODEL AND FOREIGN LANGUAGES

MATLAB is a single-threaded system. Although recent releases ($\geq$ 2007a) support multi-threaded execution for a number of built-in functions (linear algebra functions for example), the overall environment is still single threaded. Also, there is no supported implementation of calling a single session of MATLAB in separate threads [9].

MATLAB has a long tradition in providing interfaces for calling functions written in C, C++, or Fortran and to call back MATLAB functions from those external programs. For that purpose, MATLAB provides a compiler which produces so-called MEX files (MEX=Matlab EXecutable). A MEX file is represented by a shared library (e.g., a `.dll` under Windows or a `.so` under Linux) that is dynamically loaded into the MATLAB process. The functions provided by a MEX file are executed by MATLAB on its single thread and can be used in an M script just like other M-functions. Calling back to MATLAB is well supported by means of the *MEX API* as long as the call originates from the single MATLAB thread and not from another thread (see below).

Integration with a language such as Java (or C#) that requires its own virtual machine is more difficult and is not handled via the MEX mechanism. The basic complication comes from the fact that Java provides its own threads of control, for example the so-called *AWT* thread, which handles the graphical user interface and user interactions. Whenever a graphical Java component is added to MATLAB, there are at least two active threads.

Stand-alone applications written in C or Fortran may use the *MATLAB Engine* interface to communicate with a remote MATLAB process for calling mathematical routines, for example, or for using MATLAB as back-end for an application specific user interface.

2.1 MATLAB and Java

Since version 5.3 (R11) MATLAB includes a Java Virtual Machine (JVM) which is started together with MATLAB. MATLAB's graphical user interface is (at least partially) based on Java and there exists a configuration file for defining the Java class path. This class path can be extended in order to include application specific Java classes. In addition, Java support has been integrated in M and it is also possible to integrate MATLAB in a stand alone JVM as explained in more detail below.

2.1.1 Calling Java from MATLAB

M provides built-in support for execution of Java methods, instantiation of Java objects, access to fields, catching exceptions, and more. Data type conversion of method parameters passed from M to Java and of primitive types passed from Java to M is done implicitly. Conversion of arrays and strings passed from Java to M must be done explicitly by means of conversion functions.

2.1.2 Calling MATLAB from Java

A subset of MATLAB's functionality may be integrated into a stand-alone JVM by means of tools provided with MATLAB (MATLAB Compiler and MATLAB Builder JA). In this case, a full MATLAB installation and license is not required; only Java and an additional set of shared libraries is used. Also the open-source framework *JStatCom* [3] supports this approach by the *JMatlab/Link* plug-in.

Alternatively, the Java Native Interface (JNI) [8] could be used for accessing the C API of the MATLAB Engine from a stand-alone JVM. Java applications may thereby start up a new MATLAB engine, execute MATLAB commands, or exchange data [2]. The open-source project JMatLink [4] provides an appropriate Java API.

It should be noted that the approaches described in this subsection serve only to integrate MATLAB or a subset thereof into a stand-alone Java application. It is not possible to use them for the JVM integrated with MATLAB.

2.1.3 Calling back MATLAB from Java

It turned out that there is no documented and supported interface available for calling back an M function from the JVM integrated with MATLAB. There are, however, some undocumented ways, which are discussed in section 5. Since none of them worked for us, we decided to create our own interface, which is described in the subsequent section.

3. AN ASYNCHRONOUS JAVA INTERFACE TO MATLAB

This section presents an approach that is purely based on Java and documented MATLAB functionality. The core idea is that we don't try to initiate the execution of M functions from Java directly. Instead, we build on MATLAB's capability to call Java and introduce a periodic asynchronous MATLAB service that executes an M function on behalf of Java. For the interaction between Java and MATLAB, we apply the *producer/consumer pattern*. One or more Java threads produce requests that are put into a queue from where they are consumed and executed by the periodic MATLAB service. Conceptually, this resembles the principle used by Java's Abstract Window Toolkit (AWT) for executing every request in the *event dispatch thread* since it also relies on thread confinement [1]. In our case, requests are not handled by a Java thread but by a periodically executed M function within the single-threaded MATLAB environment. Figure 2 depicts the communication mechanism between Java and MATLAB. The approach presented below is platform independent and requires surprisingly little code. Therefore, we are able to present the complete Java and M source code in this paper instead of indicating the functionality with pseudo code only.

Our approach has been developed for and used in the TDL tool chain [6] and was tested for MATLAB R14-SP3, 2006a, 2008a, and 2009b on a Windows platform, and for MATLAB 2008a on a Linux platform. As we rely only on documented MATLAB features and pure Java, we expect it to work also under MacOS and under future MATLAB versions.

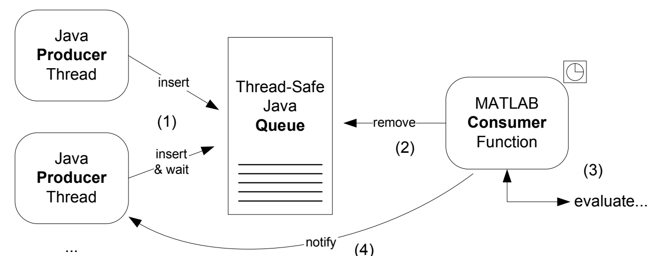

Figure 2: Schematic overview of the Java/MATLAB communication

3.1 The Core of the Interface (Java Class)

Listing 1 shows the core of the Java/MATLAB interface, namely the class `Matlab`. For the Java side of the interface, this class provides two methods for executing MATLAB commands:

- a *non-blocking* method without a return value: `feval`
- a *blocking* method with a return value: `blockingFeval`

Both methods take the name of the M function to be evaluated as the first parameter followed by an arbitrary number of arguments. They may be used for evaluating an expression such as:

```
Object x =
  asyncjmi.Matlab.blockingFeval("evalc", "1+1");
```

or to call a Simulink API function such as:

```
asyncjmi.Matlab.feval("add_block",
                      "built-in/Constant",
                      "untitled/c1");
```

Each request is encapsulated in a `MatlabCall` object `mc` (see subsection 3.2) and added to the FIFO queue `mcQueue`. While `feval` returns immediately, `blockingFeval` causes the producer thread to wait on the object `mc` until the result is available. We use a thread-safe queue implementation because (1) the dequeue operation is invoked by a separate (consumer) thread and (2) this also allows multiple Java threads to add queue entries concurrently without any further synchronization requirements.

Listing 1: Java: class Matlab

```java
package asyncjmi;

import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;

public class Matlab {

  private static final Queue<MatlabCall> mcQueue
    = new ConcurrentLinkedQueue<MatlabCall>();

  public static void feval(String cmd,
                           Object... args) {
    mcQueue.add(new MatlabCall(cmd, args, false));
  }

  public static Object blockingFeval(String cmd,
      Object... args) throws InterruptedException {
    MatlabCall mc =
      new MatlabCall(cmd, args, true);
    synchronized (mc) {
      mcQueue.add(mc);
      mc.wait();
    }
    if (mc.errorMsg != null) {
      throw new RuntimeException(mc.errorMsg);
    }
    return mc.result;
  }

  public static MatlabCall getNextCall() {
    return mcQueue.poll();
  }

  //put inner class MatlabCall here
}
```

The MATLAB side of the interface is the dequeue operation `getNextCall()`, which is also provided by the class `Matlab`. It removes and returns the next `MatlabCall` object or `null` if there is none.

It should be noted that the line `mcQueue.add(mc);` must be within the `synchronized(mc)` statement and not in front of it. Otherwise the following situation might occur. The MATLAB service accesses the `mc` object before the producer thread is waiting on `mc` for the result. The result is passed back before the producer thread is waiting for it and no thread is notified. The subsequent `wait();` would then be infinite because the notification has occurred before. If the line `mcQueue.add(mc);` is within the `synchronized(mc)` statement, passing back the result is delayed until the producer thread waits for it because passing back the result is also a synchronized operation on `mc`. For the details of passing back the result and the required synchronization see subsection 3.2.

3.2 Representing a MATLAB call (Java Class)

The class `MatlabCall` is shown in Listing 2. We expressed it as a nested class because this allowed us to keep its internals private. An instance of this class is used for

- encapsulating a request, which consists of a function name (`cmd`), its arguments (`args`), and a flag (`hasResult`) indicating if it is a blocking call or not,
- encapsulating the corresponding result, which consists of an object (`result`) (if it represents a blocking call) and a possible error message (`error`),
- synchronization between the Java producer threads and the MATLAB consumer thread.

Listing 2: Java: inner class MatlabCall

```java
public static class MatlabCall {

  public final String cmd;
  public final Object[] args;
  public final boolean hasResult;
  private Object result;
  private String errorMsg;

  private MatlabCall(String cmd, Object[] args,
                     boolean hasResult) {
    this.cmd = cmd;
    this.args = args;
    this.hasResult = hasResult;
  }

  public synchronized void
  handleResult(Object result) {
    this.result = result;
    this.notify();
  }

  public synchronized void
  handleError(String errorMsg) {
    if (hasResult) {
      this.errorMsg = errorMsg;
      this.notify();
    } else {
      System.err.println(errorMsg);
    }
  }
}
```

Whenever a return value of a blocking call is available in MATLAB, `handleResult()` passes the return value back to Java and notifies the waiting producer thread.

In case of an error message returned by the executed MATLAB function, `handleError()` either passes the error message back to the waiting producer thread (if it was a blocking call) or outputs the error message to the MATLAB console by means of writing to the Java error stream, which is displayed on the MATLAB console window. As a trivial change, one might consider to introduce a special exception class instead of using a Java `RuntimeException`.

Both `handleResult()` and `handleError()` are only called from the MATLAB service and serve only to pass back the results from MATLAB to Java.

3.3 Consumer (M Function)

Listing 3 shows the corresponding consumer part, which is implemented as an M function called `handleMatlabCalls`. It is registered as a callback function of a periodic timer. Such a callback function must match a certain signature, which explains the two unused function parameters `obj` and `event`. Timers have been introduced in MATLAB release 13 (version 6.5) released in the year 2002.

The function works off `MatlabCall` objects in the queue and executes the requests by invoking the built-in `feval` function of MATLAB with the function name and the optional arguments as parameters. For accessing the `mcQueue` and for dealing with the queue's elements, this function uses the Java support included with MATLAB's M scripts. The first line, for example, gets the next queue element and the second line checks if it is `null`. For this test, it is required to use a special built-in function (`isempty()`) because M does not support Java's notion of `null` directly.

For blocking calls, in other words calls that need a return value, the result of the evaluation is passed back to the `MatlabCall` object by calling `handleResult()`. The `handleResult` method (see Listing 2) also notifies the thread which invoked `blockingFeval` and which is therefore waiting for the result. Both, the arguments of the `MatlabCall` object and the return value are subject to the data type conversion between MATLAB and Java.

Listing 3: MATLAB: function handleMatlabCalls

```
function handleMatlabCalls(obj, event)
  mc = asyncjmi.Matlab.getNextCall();
  while (~isempty(mc))
    try
      cmd = char(mc.cmd);
      args = cell(mc.args);
      if (mc.hasResult)
        result = feval(cmd, args{:});
        mc.handleResult(result);
      else
        feval(cmd, args{:});
      end
    catch
      try
        mc.handleError(lasterr);
      catch
        display(lasterr);
      end
    end
    mc = asyncjmi.Matlab.getNextCall();
  end
end
```

Listing 4: Timer initialization

```
t = timer('TimerFcn', @handleMatlabCalls,
          'ExecutionMode', 'fixedSpacing',
          'Period', 0.1);
start(t);
```

Exceptions that occur during the execution are caught and the error message (`lasterr`) is passed back by calling `handleError()`. The implementation of the error handling strategy is encapsulated in the class `MatlabCall` (see subsection 3.2).

3.4 Consumer Initialization (M Function)

The initialization code of the timer is listed in Listing 4. The timer is set up to execute the function `handleMatlabCalls` (parameter pair 'TimerFcn', @handleMatlabCalls). The timer period is set to 100 milliseconds (parameter pair 'Period', 0.1), which was a reasonable value for our requirements. Of course, this interval could be made smaller, but this would unnecessarily increase the CPU load even in the common case that there is nothing to process. The `fixedSpacing` option (parameter pair 'ExecutionMode', 'fixedSpacing') ensures that this is the minimum time between finishing one execution and starting the next one.

As MATLAB is a single threaded system, the timer function is executed from the main MATLAB thread. This implies that blocking calls must not be initiated from the MATLAB thread. Otherwise a *deadlock* would occur. This is because the blocking call will execute a `wait` for the result value, and this wait will block the MATLAB thread itself. Consequently, the timer function will not be called to handle the request and to notify the waiting thread. This fundamental restriction can also be found in alternative approaches such as JMI (see section 5).

In order to ensure MATLAB's reactivity in case a Java component produces massive amounts of calls, a limit for the *while* loop in Listing 3 may be introduced and/or the timer interval may need to be adapted.

The timer may be initialized on MATLAB start as part of the file *startup.m*, for example, or as soon as some customized Simulink library is loaded, which implies that the library's `PreLoadFcn` callback function is executed. The `LoadFcn` callback of any block can also be used to initialize the timer, e.g. when loading an existing Simulink model.

3.5 Optimization for Blocking Calls

Without further provisions, executing a sequence of blocking calls from a single Java thread may suffer from poor performance. More precisely, the performance depends on the priority of the Java thread that issues the calls. After the timer function has evaluated the blocking call, it invokes the `handleResult` method of the `MatlabCall` object, which notifies the waiting Java thread. Depending on the priority of the notified thread and Java's scheduling behavior, several scenarios are possible.

If the priority of the notified thread is lower than the priority of MATLAB's main thread (in current releases `Thread.NORM_PRIORITY`), the notified thread will in many cases not resume immediately, but the MATLAB thread will continue and poll the queue before the producer thread is able to add the next entry. This would result in terminating the timer loop right after executing the first call and consequently approximates the worst case behavior of executing only one call per timer invocation.

Listing 5: Optimization in class Matlab

```
//(1)
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
...
  private static final
  BlockingQueue<MatlabCall> mcQueue
    = new LinkedBlockingQueue<MatlabCall>();
...
//(2)
  public static MatlabCall
  getNextCallWait () throws InterruptedException {
    return mcQueue.poll(1, TimeUnit.MILLISECONDS);
  }
}
```

Listing 6: Optimization in handleMatlabCalls

```
...
%(3)
      mc = asyncjmi.Matlab.getNextCallWait();
    end
end
```

If the priority of the notified thread is higher than the priority of MATLAB's main thread, the notified thread may resume execution immediately and will add the next entry to the queue before the timer polls it. This would result in approximating the best case behavior of executing all calls within one timer invocation.

If the priority of the notified thread is the same as the priority of MATLAB's main thread, the behavior will be somewhere in between the above cases.

In order to be independent of the thread priority we use a bounded wait within the timer's *while* loop and notify the waiting MATLAB thread upon adding an entry to the queue. The MATLAB thread needs to wait only for a short amount of time (one millisecond, for example) for this optimization to work smoothly.

The bounded wait mechanism may be implemented by polling the queue and, if the queue is empty, by calling the appropriate `wait(timeout)` method provided by the Java class `Object` and polling the queue again after the wait. In addition the waiting thread must be notified upon adding an element to the queue. Fortunately, this functionality is already provided by the Java interface `BlockingQueue` and its thread-safe implementation in the class `LinkedBlockingQueue`. So we (1) change `mcQueue` to `BlockingQueue`, (2) as a small performance optimization we add a new polling method `getNextCallWait` which avoids the passing of parameters in the M function, and (3) in `handleMatlabCalls` inside the while loop we replace the call to `getNextCall()` by `getNextCallWait()`. For completeness, the modified and additional code is shown in Listing 5 and Listing 6.

4. PERFORMANCE EVALUATION

Table 1 and Table 2 show a performance comparison between the undocumented JMI (see section 5) and the presented asyncjmi approach together with the results for plain M code executed from the MATLAB console window as a baseline. The JMI variant has been used via a simplified interface named *MatlabControl (MC)* (see section 5). It should be noted that by means of some experimentation we were able to use the interface for carrying out some of the benchmarks. We were, however, not able to use it as a general purpose interface because it showed unexpected behavior in a number of situations (deadlocks and null pointer exceptions, for example) and it even failed for some of the benchmark tests. We have also tried to access JMI directly, but the problems have been the same.

We differentiate between a call to MATLAB (`evalc('1+1')`), which evaluates an expression represented as a string, and a call to Simulink (`add_block`), which adds a graphical rectangular block to a simulation model. In each case, the MATLAB call is executed 1, 10, 100, 1000, and 10,000 times respectively. All tests were run 10 times and the presented values represent an average time consumption in milliseconds. For MatlabControl we use the blocking `blockingFeval` and the non-blocking `feval`.

All tests were run in MATLAB 2006a from Java's AWT thread on a Windows XP computer featuring an Intel Core 2 Duo processor with 2 GHz and 2 GB of RAM memory.

As can be seen from the `evalc` column, which does not involve any complex internal data structures, both approaches scale linearly with the number of invocations. Thus, the overhead per call is constant. For the `add_block` command, the non-linear behavior is caused by Simulink for creating and managing 10,000 blocks within a simulation model.

In the case of non-blocking calls, the MatlabControl interface is about three times faster than our approach. For blocking calls the difference is about a factor of two, but there are tests were the MatlabControl approach fails.

Note that starting and ending a time measurement was also executed as MATLAB calls (`tic` and `toc`). Thus, the measurements do not include the variable time span up to the timer invocation which executes the first MATLAB call. In other words, we measured only the overhead regarding the throughput (or CPU load), not the latency. This is justified because the latency of 100 ms is hardly perceptible by the user. The limiting factor is the throughput, not the latency.

For a large sequence of blocking calls the latency introduced by the periodic timer is of course perceptible by the user unless the optimization described above is used. Table 3 compares the optimized and the un-optimized case for 1000 blocking calls of `evalc('1+1')`. It also shows the number of timer invocations required to process these calls.

	non-blocking calls [ms]					
	add_block			evalc		
#	asyncjmi	MC	M	asyncjmi	MC	M
1	0.38	0.18	0.19	0.46	0.17	0.07
10	2	0.62	1	3	1	0.63
100	22	6	11	26	8	6
1,000	332	161	197	257	76	56
10,000	16,353	13,879	14,058	2,587	750	569

Table 1: Performance comparison: non-blocking

	blocking calls [ms]					
	add_block			evalc		
#	asyncjmi	MC	M	asyncjmi	MC	M
1	0.28	-	0.19	0.78	0.44	0.07
10	3	-	1	4	3	0.63
100	28	-	11	35	18	6
1,000	387	-	197	316	147	56
10,000	17,150	-	14,058	3,233	-	569

Table 2: Performance comparison: blocking

	1000 asyncjmi blocking calls				
	un-optimized			optimized	
thread priority	3	5	7	3	7
timer invocations	95	17	13	1	1
time [ms]	10,361	2,014	1,613	316	316

Table 3: Performance comparison between the un-optimized and optimized blocking calls

5. RELATED WORK

This section outlines the existing approaches as far as they are known to us and discusses their shortcomings.

Java MATLAB Interface (JMI). The `jmi.jar` library that comes with MATLAB is a collection of Java interfaces and classes. It supports the usage of MATLAB from Java and also allows MATLAB commands to be executed from Java. It has been included in MATLAB since the bundling of Java with MATLAB in release 5.3 about ten years ago. However, the JMI API is still undocumented, not supported by The MathWorks and likely to change in future releases. We have done a lot of experimentation with JMI before we decided to create our own interface. Without a proper documentation we were not able to use it successfully. Since JMI is based on native code, it is not possible to infer the missing documentation from a look at the decompiled source code. We encountered many deadlocks and null pointer exceptions in particular for calls that return a result value. Nevertheless, various projects, such as MatlabControl [11] and the MATLAB integration of TinyOS [10] build on JMI. As can be seen from the performance evaluations, JMI provides an efficient interface if it works. JMI shares with our approach that it is not possible to call back MATLAB from Java if the call originates from the main MATLAB thread. In JMI this limitation also applies for calls that do not return a result value.

MatlabControl. MatlabControl [11] provides a thin wrapper over JMI (see above) with the intention to make JMI easier to use and for providing an indirection that allows for compensating changes in JMI without invalidating client code. MatlabControl thereby inherits all the problems of JMI and we were not able to use it successfully, in particular for calls that return a result value. The runtime overhead of the wrapper classes introduced by MatlabControl are negligible and the performance figures of JMI and MatlabControl can be regard as equivalent.

C-based MEX interface. An approach that immediately comes to one's mind is to use the C-based MEX interface in combination with the Java Native Interface (JNI). However, it turned out that this mechanism is highly unstable and continuously causes MATLAB to crash and shut down non-deterministically. This mechanism only works well if the initiator of the call is the main MATLAB thread itself. We are not aware of any explicit synchronization mechanism provided by MATLAB's C-based MEX interface. Thus, calling back to MATLAB from Java's AWT thread, for example, is not possible with this approach.

MATLAB Event Handler. In a private communication, the author of [7] pointed us to yet another form of interaction between MATLAB and Java. By means of employing the undocumented *callbackproperties* of MATLAB *handles*, it is possible to register an event handler that calls a MATLAB function on the main MATLAB thread when a specific Java method is executed, for example, on the AWT thread. This approach, like ours, is based on asynchronous communication but does not need a periodic timer. Since the underlying mechanism is not officially supported by The MathWorks and only communicated via experts from the service hotline after a number of requests, we did not want to base our production code on it. Those experts from The MathWorks support team also issued a warning that it will be very tricky to pass back the result of a MATLAB function to Java. Also, the JMI interface does not rely on this mechanism, which increases our doubts in its usefulness.

6. SUMMARY AND CONCLUSIONS

We have presented a portable, compact, and sufficiently efficient approach for calling back MATLAB functions from within the Java Virtual Machine integrated with MATLAB. This closes an important gap for integrating fully fledged Java components into MATLAB.

Our approach is based on asynchronous communication following the producer/consumer pattern and relies exclusively on pure Java and documented and portable mechanisms provided by MATLAB. It has been tested and used successfully on a variety of MATLAB versions and it can be expected that it works independent of the underlying operating system for all MATLAB releases ≥ 13 (version 6.5) introduced in the year 2002. A simple optimization significantly improves the efficiency of blocking MATLAB calls even if the priority of the producer thread is smaller or equal to the priority of the main MATLAB thread.

Acknowledgments

We would like to thank Gerald Stieglbauer for sharing his experience regarding MATLAB and Java with us. His work provided a valuable starting point for our approach because he has ruled out various dead-end approaches. He also identified some core problems with the C based MEX interface and the threading issues involved.

7. REFERENCES

[1] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

[2] A. Klimke. How to access MATLAB from Java. Technical report, University of Stuttgart, 2003.

[3] M. Krätzig. JStatCom. `www.jstatcom.com`, 2008.

[4] S. Müller. JMatLink connect MATLAB and Java. `http://jmatlink.sourceforge.net`, 2005.

[5] A. Naderlinger, W. Pree, and J. Templ. Visual modeling of real-time behavior. Symposium on Automotive/Avionics Systems Engineering, 2009.

[6] preeTEC. `www.preeTEC.com`, 2010.

[7] G. Stieglbauer. *Model-based Development of Embedded Control Software with TDL and Simulink*. PhD thesis, University of Salzburg, 2007.

[8] Sun Microsystems. Java Native Interface 6.0 specification. `http://java.sun.com/javase/6/docs/technotes/guides/jni`, 2006.

[9] The MathWorks. `www.mathworks.com`, 2009.

[10] TinyOS Alliance. TinyOS. `www.tinyos.net`, 2009.

[11] K. Whitehouse. MatlabControl. `www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html`, 2009.