

TDL Tutorial

Josef Templ

August 30, 2005

Abstract

This tutorial provides the information needed to successfully write, compile and execute programs based on the Timing Definition Language (*TDL*) developed as part of project MoDECS at University of Salzburg[1]. It assumes that the reader is familiar with the programming language Java and uses a Java-based E-machine for execution of real time applications.

Contents

1	Introduction and Overview	1
2	Installation Steps	2
2.1	Preliminaries	2
2.2	Installing and Using the <i>TDL</i> Compiler	2
2.3	Installing and Using the Java-based E-machine	3
3	Single Module Examples	3
3.1	Single Mode/Single Rate Example	3
3.2	Single Mode/Multi Rate Example	4
3.3	Multi Mode/Multi Rate Example	6
4	Multi Module Examples	7
4.1	Independent Modules	7
4.2	Statically Known Sets of Modules	7
4.3	Service and Client Modules	7

1 Introduction and Overview

This tutorial is intended to support beginners in learning how to use *TDL*, which is specified in more detail in [1]. It also serves to get familiar with the tools *TDL* provides. No particular knowledge in real time control applications is assumed and, in fact, real world control applications are not even used as examples. This simplifies the examples considerably, because there is no dependency on any external hardware.

The tutorial uses Java command line tools only and resists the temptation of integrating the tools into an integrated development environment, which would make some things easier, of course, but at the same time it might limit the learning effect and the understanding of the basic mechanisms. For advanced Java users it is of course possible to automate the required steps using IDE features, make files or ANT scripts.

The tutorial starts with the installation steps. The remaining is organized as a sequence of examples with increasing complexity. The first set of examples consists of a single module only, the second set consists of multiple modules. Please follow the steps in this order to get used to *TDL* as fast and as convenient as possible without missing any important basic knowledge required for more complex scenarios.

2 Installation Steps

2.1 Preliminaries

The *TDL* tools required for this tutorial consist of pure Java programs only and therefore should be executable on any platform that supports a compliant Java Virtual Machine (JVM). We have developed the tools using Java 2 Version 1.4 and we have tested the tutorial with this very same version. It can be expected, however, that the tools and the tutorial examples also work under Version 1.3 or under 1.5. Please let us know if there are any problems related to different Java versions.

In the following we assume that Java tools (e.g. `java`, `javac`) are available from the command line and we use commands appropriate for a Microsoft Windows Command Prompt or Microsoft Windows `.bat` files. For every step in the tool chain we show the basic command line interface first and show and use possible convenience layers on top of that later. The scripts could easily be adapted to Unix shells.

2.2 Installing and Using the *TDL* Compiler

The *TDL* compiler's main function is located in class `emcore.tools.tdlc.Compiler`. Thus, this class and all imported classes must be available for execution by command `java`. For your convenience, the executable Java archive `tdlc.jar` contains all the class files required for the compiler. To test if the compiler can be executed, simply execute one of the following commands, where `tdlc.jar` may need an additional path name depending on the place where it has been installed on your system.

```
> java -jar tdlc.jar
or
> java -cp tdlc.jar emcore.tools.tdlc.Compiler
```

The compiler outputs a usage note in this case where braces surround iterative parts, brackets surround optional parts, and vertical bars separate alternatives. Angle brackets denote information supplied by the user.

```
usage: java emcore.tools.tdlc.Compiler {option | <inputPath>}
  option = -d <destDir> | -ep <ECODEPATH> | -java | -ansic | -cpp
          | -platform <className>]
```

The task of the compiler is to read a *TDL* input file (recommended file name extension `.tdl`) and transform it into a binary form, the so called *ecode* file (file name extension is `.ecode`) suitable for execution with an E-machine. In addition, the compiler may produce platform specific additional files, where *platform* refers to the target E-machine. If the target is the Java-based E-machine, there will be an auxiliary Java source file being created, which supports efficient interaction between the E-machine and the Java-based functionality code. This will be explained later in more detail.

The meaning of the arguments used in this tutorial is defined as:

- d the following argument specifies the destination directory the compiler uses to write its output files to.
- ep the following argument specifies the ECODEPATH to be used by the compiler in order to look for imported modules. This is similar to the CLASSPATH variable in Java but used for files of type `.ecode` rather than `.class`. An alternative way of specifying this option is to define the Java system property `tdl.ecode.path` using the `-Dtdl.ecode.path=<ECODEPATH>` JVM paramter.
- java this option specifies that in addition to a `.ecode` file the compiler should produce platform specific output for the Java platform, i.e. for the Java-based E-machine.
- <inputPath> this argument specifies the path and filename of the *TDL* input file to be compiled by the compiler. There may be an arbitrary number of files specified.

In order to simplify the activation of the *TDL* compiler, a simple script file could be used as shown below as file `tdlc.bat`. This command will be used in the subsequent examples.

```
java -jar tdlc.jar -java %*
```

2.3 Installing and Using the Java-based E-machine

The Java-based E-machine's main function is located in class `emcore.tools.emachine.Interpreter`. Thus, this class and all imported classes must be available for execution by command `java`. For your convenience, the Java archive `emachine.jar` contains all the class files required for the E-machine. To test if the E-machine can be executed simply execute the following command. Please note that `emachine.jar` is not an executable jar file, because it depends on additional class files not known statically.

```
> java -cp emachine.jar emcore.tools.emachine.Interpreter
```

Again, this outputs a usage notice with the same meta characters as for the *TDL* compiler. The argument to the E-machine activation is a sequence of *TDL* module names to be loaded and executed.

In the following examples we will start the Java-based E-machine by using the short hand command `emachine` with the modules to be executed as command line arguments. We assume the required Java classes to be added to the `-cp` argument of command `java`, e.g. `java -cp .;emachine.jar`. A script file `emachine.bat` is given below.

```
java -cp .;emachine.jar emcore.tools.emachine.Interpreter %*
```

Please note that unexpected WCET (worst case execution time) violations may be reported by the E-machine due to the way most JVMs load and run Java classes. Class loading is delayed until the first usage of the class and after that methods that require a lot of CPU cycles will be compiled to native code by an integrated jit (just in time) compiler. Both, class loading and jit compilation adds to the execution time of a task and may result in a WCET violation until the system has 'warmed up'.

3 Single Module Examples

3.1 Single Mode/Single Rate Example

A simple example for a real time application is the implementation of a ticker, whose sole task is to provide a tick event in a periodic way and to perform some activity when a tick happens. Of course, such an example could be easily programmed in Java directly, but we will see that using *TDL* provides some advantage when it comes to more complex scenarios.

For the ticker application we start with a *TDL* specification assumed to be stored in file `Ticker1.tdl`. In this very first example, we use timing attributes with explicit names (`wcet`, `period`, `freq`). In later examples we use the short form without the names.

```
module Ticker1 {  
  
    task tick[wcet=100ms] {  
        uses doTick(); //uses external functionality code  
    }  
  
    start mode main [period=1000ms] {  
        task[freq=1] tick(); //1 Hz; FLET = 1 sec  
    }  
}
```

In order to generate the ecode file and Java platform specific output we execute command

```
> tdlc Ticker1.tdl
```

The following files will be created in the current working directory. Use option `-d` to specify a different location.

Ticker1.ecode contains the binary representation of module `Ticker1` and must be specified as input to the E-machine.

Ticker1\$.java auxiliary file created for the Java-based E-machine due to option `-java`

In order to be able to compile the Java class *Ticker1\$* we must provide the functionality code referenced in module *Ticker1* in form of Java code. In case of module *Ticker1* the only external method is `doTick`. The Java language binding rules for *TDL* define that the functionality code has to be provided in a Java class named after the module name. So, we simply have to provide a Java source file `Ticker1.java` which provides corresponding static methods for every external reference.

Note: We deliberately avoid direct output to `System.out` but use an auxiliary class (`Out`) in order to write log output to the console. This avoids potential deadlock problems and other anomalies with the globally shared resource `System.out` in case of preemption of a task that holds the lock on `System.out`.

```
import emcore.tools.emachine.Out;

class Ticker1 {
    static void doTick() {
        //must be executed within 100 ms
        Out.println("Ticker1.doTick");
    }
}
```

Now we can compile both Java classes (`Ticker1$`, `Ticker1`) and invoke the E-machine with the following commands. Note that for compiling `Ticker1$` we need to specify `emachine.jar` as `-classpath` parameter since this class implements and thus imports an interface from the E-machine.

```
> javac -classpath emachine.jar Ticker1$.java Ticker1.java
> emachine Ticker1
```

The output on the console will be

```
Ticker1.doTick
Ticker1.doTick
Ticker1.doTick
... a line every second
```

The simplicity of this example results from the fact that it has only one mode and within this mode it executes tasks with only one frequency. This case is usually called a single-mode/single-rate system. A more general case is the introduction of a multi-rate system in the following example.

3.2 Single Mode/Multi Rate Example

For the moment we stay with a single mode but we introduce a multi-rate system by specifying an additional task, which is executed with a different frequency. In order to show the communication between tasks in a multi-rate system, we introduce an invocation counter for task `tick1` which is used as input for task `tick2`.

```
module Ticker2 {

    task tick1 [100ms] {
        output int cnt; //counts the number of invocations
        uses doTick1(cnt); //uses external functionality code
    }

    task tick2 [100ms] {
        input int i;
        uses doTick2(i); //uses external functionality code
    }
}
```

```

start mode main [1000ms] {
  //defines a task invocation with frequency 1, i.e. FLET = 1000 ms
  task [1] tick1();
  //defines a task invocation with frequency 2, i.e. FLET = 500 ms
  task [2] tick2(tick1.cnt);
}
}

```

We provide a functionality class named `Ticker2`, which now includes two methods.

```

import emcore.tools.emachine.types.ref_int;
import emcore.tools.emachine.Out;

class Ticker2 {
  static void doTick1(ref_int cnt) {
    cnt.val++;
    Out.println("Ticker2.doTick1.cnt.val=" + cnt.val);
  }
  static void doTick2(int i) {
    Out.println("Ticker2.doTick2.i=" + i);
  }
}

```

After issuing the corresponding commands

```

> tdlc Ticker2.tdl
> javac -classpath emachine.jar Ticker2$.java Ticker2.java
> emachine Ticker2

```

the output on the console will look like the following:

```

Ticker2.doTick2 i = 0
Ticker2.doTick1 cnt.val = 1
Ticker2.doTick2 i = 0
Ticker2.doTick2 i = 1
Ticker2.doTick1 cnt.val = 2
Ticker2.doTick2 i = 1
Ticker2.doTick2 i = 2
Ticker2.doTick1 cnt.val = 3
Ticker2.doTick2 i = 2
Ticker2.doTick2 i = 3
Ticker2.doTick1 cnt.val = 4
Ticker2.doTick2 i = 3
Ticker2.doTick2 i = 4
Ticker2.doTick1 cnt.val = 5
Ticker2.doTick2 i = 4
Ticker2.doTick2 i = 5
Ticker2.doTick1 cnt.val = 6
Ticker2.doTick2 i = 5
Ticker2.doTick2 i = 6
... a doTick1 line once per 1000 ms
... a doTick2 line once per 500 ms

```

Please note that the actual timing and order of output lines is not specified completely in *TDL* but a result from the schedule being generated internally for executing the tasks in a way that all deadlines are guaranteed. What we know for sure is that everything is repeated with the mode period and that *tick1*

is activated once per period and *tick2* is activated twice per period and that *tick2* is executed the first time within the first half of the mode period and the second time within the second half. This follows from the fixed logical execution time of *tick2*, which is 500ms.

What is also interesting in this example is the value of input parameter *i* of task `tick2`. In the third output line the value is still zero, which might be surprising to some of the readers. The reason is that the input parameter of task `tick2` is read at the beginning of the tasks's period or in other words at the begin of its fixed logical execution time (FLET). At that time the value of `tick1.cnt` is still zero. In general, the output values are only available for other tasks after the FLET of the producing task has elapsed. This is the basic communication mechanism, which is sometimes referred to as 'FLET semantics'.

3.3 Multi Mode/Multi Rate Example

As the next generalization step we introduce a second mode of operation. This mode is called *freeze* because it does not contain any periodic activities at all and there is no way to return to the normal mode of operation. The normal mode of operation changes to freeze whenever the boolean function *finished* returns true. In general, any mode can define any number of periodic activities, of course.

```

module Ticker3 {

  task tick1[100ms] {
    uses doTick1(); //uses external functionality code
  }

  task tick2[100ms] {
    uses doTick2(); //uses external functionality code
  }

  start mode main [1000ms] {
    task [1] tick1();
    task [2] tick2();
    //defines a conditional mode switch with frequency 1
    mode [1] if finished() then freeze;
  }

  mode freeze [1000ms] {}
}

```

The implementation of method *finished* uses a random number generator to return true with probability 1/5.

```

import java.util.Random;
import emcore.tools.emachine.Out;

class Ticker3 {
  private static Random rnd = new Random();
  static void doTick1() {
    Out.println("Ticker3.doTick1");
  }
  static void doTick2() {
    Out.println("Ticker3.doTick2");
  }
  static boolean finished() {
    return rnd.nextInt(5) == 0;
  }
}

```

When we run this example, we will observe that the output will be unchanged after some time, which means that the E-machine has entered mode *freeze*. We do not want to go into any discussion whether this is a suitable behavior for a real time control application or not. It simply serves as an example to show the idea of multi mode systems. In general there may be an arbitrary automaton with various transitions, possibly back and forth, between the individual states. Most mode switches will be conditional, but unconditional switches are permitted as well.

As a further generalization of *TDL* we will look at systems that consist of multiple parallel automatons.

4 Multi Module Examples

4.1 Independent Modules

The previous examples can already be used to construct a system which consists of multiple parallel automatons. All we need to do is to start the E-machine with multiple modules as arguments. If we want to start `Ticker1` and `Ticker2` in parallel, we issue the command:

```
> emachine Ticker1 Ticker2
```

The modules will be started synchronously and they will be kept synchronized for the complete execution time.

4.2 Statically Known Sets of Modules

If we want to express that a statically known set of modules is to be loaded into the E-machine, we can introduce a new top-level module, which uses the `import`-clause to specify which modules are members of this group. The following module is an example of a module, by the way, which does not declare any tasks or modes and in particular, it does not define a start mode. Therefore, this particular example module is used only for expressing the static grouping of a set of modules. In general, it would of course be possible to declare additional entities and to provide a start mode.

```
module Tickers {  
  import Ticker1;  
  import Ticker2;  
  import Ticker3;  
}
```

Loading module `Tickers` into the E-machine is done using the command below. Note that the imported modules will be loaded implicitly by loading a client module.

```
> emachine Tickers
```

4.3 Service and Client Modules

The module construct provided by *TDL* also allows us to specify that one module uses another module in some way. A module `M2`, for example, could import module `M1` and use some of the ports exported by module `M1`. `M1` is called a service module in this respect and `M2` is called a client module. The following example introduces a service module which exports (using the attribute `public`) a set of sensors together with some statistical data. Since in general, calculating statistics may take some time, we introduce a task, which does this work.

```
module Sensors {  
  public sensor int s1 uses getS1;  
  public sensor int s2 uses getS2;  
  
  public task stat [10ms] {  
    input int s1; int s2;
```

```

output int minS1 := 10000; int maxS1 := 0;
    int minS2 := 10000; int maxS2 := 0;
uses doStat(s1, s2, minS1, maxS1, minS2, maxS2);
}

start mode main [1000ms] {
    task[1] stat(s1, s2);
}
}

```

In a real world example the external functions `getS1` and `getS2` would read the sensor values from the physical environment. In our tutorial example we simply use random values. In order to be able to pass output parameters, the E-machine provides wrapper classes such as `ref_int`, `ref_char` etc. According to the Java language binding rules of *TDL*, these types must be used for state and output parameters of primitive *TDL* types.

```

import java.util.Random;
import emcore.tools.emachine.types.ref_int;

class Sensors {
    private static Random rnd = new Random();
    static int getS1() {
        return rnd.nextInt(40);
    }
    static int getS2() {
        return rnd.nextInt(10);
    }
    static void doStat(int s1, int s2, ref_int minS1, ref_int maxS1,
        ref_int minS2, ref_int maxS2) {
        if (s1 < minS1.val) minS1.val = s1;
        if (s1 > maxS1.val) maxS1.val = s1;
        if (s2 < minS2.val) minS2.val = s2;
        if (s2 > maxS2.val) maxS2.val = s2;
    }
}
}

```

A client module may import module `Sensors` and use the exported ports as shown below.

```

module Client {
    import Sensors as S;

    public task something [100ms] {
        input int i1; int i2; int maxI1; int maxI2;
        uses doSomething(i1, i2, maxI1, maxI2);
    }

    start mode main [1000ms] {
        task[1] something(S.s1, S.s2, S.stat.maxS1, S.stat.maxS2);
    }
}
}

```

Note that the FLET (fixed logical execution time) semantics for task activation defined in *TDL* specifies which values are used at which time. Whenever task `something` is activated, the `maxS1` value is actually one cycle older than the current `s1` value. Thus, it must be interpreted as the maximum previous

value of `s1`. This follows from the fact that both modules run at 1 Hz, both are kept in sync and the output of task `Sensors.stat` is only available after its FLET has elapsed.

```
import emcore.tools.emachine.Out;
class Client {
    static void doSomething(int i1, int i2, int maxI1, int maxI2) {
        Out.println("i1=" + i1 + ",i2=" + i2
            + ",maxI1=" + maxI1 + ",maxI2=" + maxI2);
    }
}
```

References

- [1] Templ, J.: TDL Specification and Report. <http://www.softwareresearch.net/site/publications/C055.pdf>.