TDL Specification and Report

Josef Templ

Department of Computer Science
University of Salzburg
Austria

Technical Report T001
November 2003
http://cs.uni-salzburg.at/pubs/reports/T001.pdf

Abstract

This report defines the syntax and semantics of the software description language *TDL* (timing definition language), which has been developed as part of project MoDECS at the Paris Lodron University of Salzburg (Austria). *TDL* allows to specify the timing behavior of a hard real time control application in a descriptive way and separates the timing aspect of such applications from the functionality, which must be provided separately using an imperative programming language such as Java, C or C++. *TDL* is conceptually based on *Giotto*, but provides more convenient syntax for many applications and an improved set of programming tools.

1 Introduction

This document defines the syntax and semantics of the software description language *TDL*, which was developed as part of project MoDECS at the Paris Lodron University of Salzburg (Austria). This report is not an introduction into the emerging field of time triggered control systems and model based development.

We deliberately avoid the term programming language for *TDL*, but use the more general notion of *software description language*, which was suggested by Prof. N. Wirth at EmSys Summer School 2003 at Salzburg. *TDL* allows to *describe* the timing properties of a hard real time control application and thereby separates the timing aspect of such applications from the functionality. *TDL* programs are purely declarative, all imperative parts of a control application must be provided separately using an imperative programming language such as Java, C or C++. This separation allows to write platform independent *TDL* timing models, which may be implemented on an open set of target platforms.

The following sections describe the lexical structure, the syntactical structure and the semantics of *TDL* step by step. A complete definition of all lexical and syntactical rules as well as a complete example is presented in the Appendix.

1.1 Relation to Giotto

TDL is conceptually based on the time triggered modelling language *Giotto*[2], but provides more convenient syntax and an improved set of programming tools. The *TDL* compiler and E-machine resulted from a cleanroom implementation without access to the Giotto compiler or E-machine sources. We tried to preserve the spirit of Giotto as far as possible and made only changes and extensions which we believe are absolutely necessary for applying this technology in an industrial environment as opposed to the research lab usage of Giotto. Please see Section 5 for a list of differences.

1.2 Acknowledgement

I would like to thank Christoph M. Kirsch, the author of the original Giotto compiler, for many hints regarding subtle points of the Giotto specification and his willingness to discuss possible modifications of Giotto finally leading to *TDL*. I also want to thank Wolfgang Pree and the members of the MoDECS team for their contributions. Finally I want to thank Hanspeter Mössenböck for providing the excellent compiler generator Coco/J free of charge and for the changes he made in response to my needs.

2 Lexical Structure

An *TDL* module is represented as an ASCII text. Sequences of characters form words, also called tokens, and the sequence of tokens forms the text. White space between tokens is ignored, as well as comments are ignored. Tokens may be keywords, operators, identifiers and literals. Keywords are reserved and must not be used as identifiers.

2.1 White Space and Line Separators

Blank, line feed (LF), carriage return (CR) and tabulator (TAB) characters are ignored and commonly referred to as *white space*. They serve to separate tokens but have no further meaning except that line feed and carriage return characters are used to count line numbers in order to emit precise error messages. *TDL* supports three common forms of line separators: CR, LF and CR+LF.

2.2 Comments

TDL allows comments as in the programming language Java, i.e. line comments start with `//` and end with the end of line, and block comments are enclosed within `/*` and `*/`. Block comments may not be nested, however, block comments may contain line comments.

2.3 Identifiers

An identifier starts with an ASCII-letter (A-Z,a-z) followed by an arbitrary sequence of letters and digits (0-9). Identifiers must not contain white space and must be different from keywords.

2.4 Keywords and Operators

The following set of keywords is defined in *TDL*:

```
actuator const false if input mode module output package schedule sensor
start state task then true type uses
```

The following set of operators and special symbols is used in *TDL*:

```
{ } [ ] ( ) ; = . := ,
```

2.5 Literals

TDL supports numeric and string literals. A numeric literal is a sequence of digits, a string is a sequence of arbitrary characters enclosed in single or double quotes. The enclosing character must not occur inside the string.

```
Examples: 0, 123, 'abc', "xyz", "a man's world"
```

3 Syntactical Structure

The syntax of *TDL* is defined using *Extended Backus-Naur Form* (EBNF) rules. Keywords, operators and special symbols are enclosed in double quotes. The following EBNF meta symbols are used to define the grammar.

=	separates the non terminal symbol (left hand side) of a production from the right hand side.
.	terminates a production.
|	separates alternatives.
[ ]	encloses optional parts (zero or one).
{ }	encloses iterated parts (zero or more).
( )	overides binding rules.

The overall goal of the chosen syntax is that *TDL* programs should be easily readable by humans. Since many of the readers are expected to be used to work with Java or C programs, some aspects are similar to those languages. In addition some constructs have been borrowed from Pascal style languages and of course from Giotto.

In the following subsections, we proceed in a top-down fashion and start with the definition of a compilation unit, which in *TDL* is called a *module*.

3.1 Module

Note: In the final version, the *TDL* tools will support modular decomposition of a control application. Therefore, an *TDL* compilation unit is called a *module*. The current version supports only single module programs. The semantics of imports has to be defined.

A module may be associated with a package, which acts as a namespace for modules. Packages serve to distinguish TDL components provided by independent vendors very much like in the Java programming language.

A module has a name (after keyword "module") and provides a namespace for definition of constants, types, sensors, actuators, tasks and modes. The namespace is enclosed within curly brackets. Names declared within the module are visible from the point of declaration up to the end of the module. There may only be a single module per input text, which means that EOF (end of file) must follow the module.

Please refere to the appendix for an example of a complete module.

```
emcoreModule = [packageSpec] "module" ident "{"
  {"const" {constDecl ";"}}
  {"type" {typeDecl ";"}}
  {"sensor" {sensorDecl ";"}}
  {"actuator" {actuatorDecl ";"}}
  {"task" taskDecl}
  {modeDecl}
  "}"
  EOF.

packageSpec = "package" extIdent ";".
```

3.2 Constant Declaration

A constant declaration associates a name with a constant value. The constant value may be denoted as a literal or as the name of another constant. Currently there are no operators allowed within constant expressions. This may be added in a later version. Constants may be used for initialization of state and output variables.

```
constDecl = ident "=" constExpr.
constExpr = ["-"] number | constExprBoolean | string | constDesignator.
constExprBoolean = "true" | "false".
```

3.3 Type Declaration

A type declaration introduces a new type or provides an alias for an existing type. A new type consists only of the type's name and is opaque for *TDL*. In order to execute a control application, the type must be provided in a form accepted by the E-machine being used. For a Java-based E-machine, for example, a class with the name of the type must be provided. This is, however, outside the scope of the *TDL* language definition.

TDL provides a set of basic types, which matches those found in the programming language Java. The basic types are predeclared in a universal scope outside the module and named `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`, and `string`.

```
typeDecl = ident ["=" typeDesignator].
```

3.4 Sensor Declaration

A sensor declaration defines a sensor, which is the input mechanism of an *TDL* program. Sensors are typed variables which may be connected with the environment using a so-called *getter* function. The getter is an external parameterless function which returns a value compatible with the sensor's type. It must be implemented according to the E-machine and environment the program is executed in.

```
sensorDecl = typeDesignator ident ["uses" extIdent].
extIdent = ident {"." ident}.
```

3.5 Actuator Declaration

An actuator declaration defines an actuator, which is the output mechanism of an *TDL* program. Actuators are typed variables which may be connected with the environment using a so-called *setter* function. The setter is an external function with a single parameter compatible with the actuator's type. It must be implemented according to the E-machine and environment the program is executed in.

```
actuatorDecl = typeDesignator ident ["uses" extIdent].
```

3.6 Task Declaration

A task declaration defines a task, which encapsulates a computation to be carried out by a control application. Tasks provide a namespace for declaration of input, output and state ports. In addition, a task has an associated external procedure (including arguments), which performs the computation. The arguments of the external procedure call

are taken exclusively from the task's ports and must be treated by the external procedure as value or reference parameters accordingly.

Tasks may be connected via their input and output ports to other program entities. State ports, however, are always private to the task and serve only to preserve state between repeated invocations. The details of connecting tasks will be defined in mode declarations further below.

Output and state ports must be initialized either with a constant value or with an external function, called an *initializer*. Initializers are, like getters, parameterless functions, which must return a value compatible with the port type.

```
taskDecl = ident "{"
  {"input" {inPortDecl}}
  {"output" {portDecl}}
  {"state" {portDecl}}
  [taskWcetAnnotation]
  "schedule" call ";"
  "}".
inPortDecl = typeDesignator ident ";".
portDecl = typeDesignator ident ("uses" extIdent | ":=" constExpr) ";".
call = extIdent "(" [portDesignator {"," portDesignator }] ")".
```

A task may have a worst case execution time (wcet) annotation, which specifies the maximum time the computation needs. The property name is `WCET`, the duration is defined in units, which may be `ms` (milliseconds) or `us` (microseconds). An example wcet annotation looks like `[WCET=42ms]`.

```
taskWcetAnnotation = "[" ident "=" number unit "]".
```

3.7 Mode Declaration

A mode declaration defines a mode, which is a particular state of operation of a control application. In general, control applications may consist of multiple modes[1], one of them will be the *start* mode. Starting an *TDL* program means to switch the E-machine into the distinguished start mode.

An *TDL* mode consists of a set of activities executed periodically. The period of a mode is defined in units, which may be `ms` (milliseconds) or `us` (microseconds). Activities carried out in a mode include task invocations, actuator updates and mode switches.

```
modeDecl = ["start"] "mode" ident "[" period unit "]" "{"
  {"task" {taskInvocation}}
  {"actuator" {actuatorUpdate}}
  {"mode" {modeSwitch}}
  "}" .
```

Every activity is performed with a particular frequency per period. The mode period must be divisible by this frequency without remainder.

Every activity may be guarded by an external function, called a *guard*. A guard takes sensors or task output ports as arguments and returns a boolean result. The activity will only be carried out if the guard evaluates to *true*.

```
taskInvocation = frequency guard taskDesignator assignList.
frequency = "[" number "]" .
guard = ["if" call "then"] .
assignList = "{" {ident ":=" portDesignator ";"} "}"
  | ["(" [portDesignator {"," portDesignator}] ")" ] ";" .
```

A *task invocation* means that the task's input ports are updated according to the assignment list and the task's computation is scheduled for execution. The assignment list may be specified either by a set of assignment statements or by providing an argument list, where each port is assigned to an input port in declaration order. The source ports must either be sensors or task output ports.

[1] A Helicopter control system, for example, may consist of a hover mode and a cruise mode. In hover mode the system tries to maintain a fixed position, in cruise mode it will try to reach a previously defined position. The control tasks will be different for both modes, although there may also be common functionality.

Execution of the computation may be done in parallel with other activities and constitutes an asynchronous operation. The output values, however, will only be available after the fixed logical execution time of the task (flet) has elapsed. The *flet* is defined as (mode period / actuator frequency). In case of using the output ports of a task by other activities before the task's flet has elapsed, the previous values of the output ports are used. The intermediate values of output ports are never visible to other program entities.

Note that the sum of the worst case execution time (wcet) of all task invocations must not exceed the mode period.

```
actuatorUpdate = frequency guard ident ":=" portDesignator ";" .
```

An *actuator update* means that the value of an actuator is set according to the specified assignment. In addition, the setter of the actuator will be called. An actuator update is a synchronous operation taking place in logical zero time with an *update period* defined as (mode period / actuator update frequency). Actuator updates start after the update period has elapsed, i.e. they are not carried out at time zero or at the time of a mode switch, but with a delay of one update period.

```
modeSwitch = frequency guard modeDesignator assignList.
```

A *mode switch* means that the control application switches to the specified target mode after the specified assignments have been executed. A mode switch is a synchronous operation taking place in logical zero time with a *switch period* defined as (mode period / mode switch frequency). Mode switches in the target mode are never executed at the time of the mode switch but with a delay of one switch period. This prevents mode switch cycles without any time passing.

4 Language Bindings

Functionality required by an *TDL* program is provided as static (global) functions in a particular programming language. In principle, there is an open set of languages, which may be used by an E-machine. The following subsections define the recommended conventiones for three popular programming languages.

4.1 Java

For every external function (sensor getter, actuator setter, port initializer, task implementation, guard) there must be a corresponding public static Java function with appropriate parameters and return types. The external function may be qualified in the *TDL* program by a dot-separated list of identifiers in front of the function's name or it my be unqualified. The following naming conventions apply.

4.1.1 Naming conventions

The Java name for an unqualified function `f` is `packageName.ModuleName.f`. Thus, it must be defined in a class named after the module. The value of `packageName` is taken from the module's package specification. The anonymous Java package is used if no package specification exists.

Qualified external functions must be provided in a class and package as specified by the qualification.

4.1.2 Type mapping

The basic *TDL* types are mapped 1:1 to primitive Java types. For opaque *TDL* types, a public class named after the type must be provided. In addition this class must have a public no-arg constructor and it must implement interface `emcore.tools.emachine.types.Opaque` in order to provide the ability to copy itself.

TODO qualification of type names??

For output and state ports of a primitive type, an auxiliary *reference* class has to be used[2]. These classes are contained in package `emcore.tools.emachine.types` for all primitive types. The naming convention is that for a primitive type `T` there exists a corresponding reference class named `ref_T`.

For output and state ports of an opaque type, there is no need to provide auxiliary reference classes since objects are passed by reference in Java anyway. Opaque types are treated like `struct` in C or `RECORD` in Pascal and are copied by the E-machine when assigned to a port.

[2]Note that Java does not provide reference parameters. Therefore we have to emulate them by using auxiliary classes.

4.2 C

TODO

4.3 C++

TODO

5 Differences to Giotto

The most visible syntactical differences between *TDL* and Giotto are:

- the introduction of a top level language construct (module) and the reorganization of mode declarations, where 'start' is a modifier of a mode declaration in *TDL*.
- the elimination of global output ports, which are replaced by task output ports in *TDL*,
- the elimination of explicit task and mode drivers, which are merged into mode declarations in *TDL*,
- the addition of constants, which may also be used to initialize ports in *TDL*,
- the introduction of units for timing values in *TDL*.

The following list explains differences to the Giotto semantics.

program start an *TDL* program is started by switching to the start mode. This means that at time zero, there are neither actuator updates nor mode switches. In Giotto, the actuator updates and mode switches of the start mode take place at time zero. There are, however, no further actuator updates or mode switches of the target mode at time zero.

mode switch Giotto allows to switch a mode even if there are running tasks as long as those tasks exist with the same task period in the target mode. However, there may be delays involved when switching to the target mode. Furthermore, the task will deliver output values to the target mode, which do not correspond to inputs specified there. *TDL* does not allow this kind of mode switch and probably never will. We are thinking about alternative ways of performing even faster mode switches without the need to continue running tasks in the target mode, with simpler semantics and, last but not least, without any delays.

actuator update A guarded actuator update in Giotto means that the actuator setter is called independently of the guard's result. In *TDL*, actuator update *and* actuator setter are both guarded and performed only if the guard returns true.

The following list describes tool related differences between *TDL* and Giotto.

E-code *TDL* defines a binary, platform independent E-code file format and uses statically typed APIs for connecting programs with external functionality code. The structure and semantics of E-code instructions has not been changed be *TDL*.

Time Resolution *TDL* uses microseconds internally for all timing values, whereas Giotto is based on milliseconds. This means, that *TDL* programs may use mode periods below 1 millisecond, given that the underlying E-machine supports fast enough scheduling.

Java based E-machine is designed as a JavaBean, which means that it is possible to register any number of listeners. This may be used to visualize execution of *TDL* programs, for example, without including visualization in the basic E-machine directly.

A Appendix

A.1 *TDL* EBNF Grammar

The lexical and syntactical structure of *TDL* is defined using the compiler generator *Coco*. The complete grammar without attributes and semantic actions is shown in the following. CHARACTERS defines the character sets for the lexical tokens, IGNORE defines the characters being ignored in addition to blank characters, TOKENS defines the lexical token classes, COMMENTS defines the structure of comments and PRODUCTIONS defines the syntax of *TDL*.

```
COMPILER emcorec;

CHARACTERS
  letter   = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz_".
  digit    = "0123456789".
  tab      = "\t".
  lf       = "\n".
  cr       = "\r".
  noQuote1 = ANY - "'" - cr - lf.
  noQuote2 = ANY - '"' - cr - lf.

IGNORE cr + lf + tab

TOKENS
  ident     = letter {letter | digit}.
  string    = "'" {noQuote1} "'" | '"' {noQuote2} '"'.
  number    = digit {digit}.

COMMENTS FROM "/*" TO "*/"
COMMENTS FROM "//" TO cr
COMMENTS FROM "//" TO lf


PRODUCTIONS

emcorec = emcoreModule EOF.

emcoreModule = [packageSpec] "module" ident "{"
  [hardwareAnnotation]
  {"const" {constDecl ";"}}
  {"type" {typeDecl ";"}}
  {"sensor" {sensorDecl ";"}}
  {"actuator" {actuatorDecl ";"}}
  {"task" taskDecl}
  {modeDecl}
  "}".

packageSpec = "package" extIdent<^Sem.pkgName> ";".

constDecl = ident "=" constExpr.

constExpr = ["-"] number | constExprBoolean | string | constDesignator.

constExprBoolean = "true" | "false".

typeDecl = ident ["=" typeDesignator].

sensorDecl = typeDesignator ident ["uses" extIdent] [portAnnotation].
```

```
extIdent = ident {"." ident}.

actuatorDecl = typeDesignator ident ["uses" extIdent] [portAnnotation].

taskDecl = ident "{"
  {"input" {inPortDecl}}
  {"output" {portDecl}}
  {"state" {portDecl}}
  [taskWcetAnnotation]
  "schedule" call ";"
  [taskTimingAnnotation]
  "}".

inPortDecl = typeDesignator ident ";".

portDecl = typeDesignator ident ("uses" extIdent | ":=" constExpr) ";".

call = extIdent "(" [portDesignator {"," portDesignator }] ")".

modeDecl = ["start"] "mode" ident "[" period unit "]" "{"
  {"task" {taskInvocation}}
  {"actuator" {actuatorUpdate}}
  {"mode" {modeSwitch}}
  "}" .

taskInvocation = frequency guard taskDesignator assignList.

frequency = "[" number "]" .

guard = ["if" call "then"].

assignList = "{" {ident ":=" portDesignator ";"} "}"
  | ["(" [portDesignator {"," portDesignator}] ")" ] ";" .

actuatorUpdate = frequency guard ident ":=" portDesignator ";" .

modeSwitch = frequency guard modeDesignator assignList.

designator = ident {"." ident}.

/* renamed productions */
unit = ident.
period = number.
constDesignator = designator.
typeDesignator = designator.
taskDesignator = designator.
portDesignator = designator.
modeDesignator = designator.

/* annotation currently used */
taskWcetAnnotation = "[" ident "=" number unit "]".

/* annotations currently ignored */
annotation = "[" {ANY} "]".
hardwareAnnotation = annotation.
portAnnotation = annotation.
```

```
taskTimingAnnotation = annotation.
modeAnnotation = annotation.
modeConnectionAnnotation = annotation.
modeSwitchAnnotation = annotation.
taskAnnotation = annotation.

END emcorec.
```

A.2 Format of .ecode files

The following attributed EBNF grammar describes the format of ecode files generated by the *TDL* compiler. Note that there is no white space between any symbols. Integers (int4) are written in big endian byte order, strings are written as zero terminated character sequences and booleans are encoded as 1 (true) and 0 (false). byte1 is stored as a single byte. Terminal and non-terminal symbols may contain an optional name attribute written as `name:` followed by the structure or value of the symbol. Byte values are denoted as in Java or C by using `0x` as prefix of the hexadecimal value. Character values are written under single quotes (`'`). All time values (e.g. mode period, task wcet, ecode future delay) are given in microseconds. This means that the maximum time value is about 35 minutes, if signed 4 byte integers are used by an E-machine. Unused operands of E-code instructions have value -1, unused comments in E-code instructions are empty strings.

```
ECodeFile = 'E' 'C' '0' '1'
  pkgName:string moduleName:string moduleKey:int4
  0x80 Modules
  0x81 Constants
  0x82 Types
  0x83 Ports
  0x84 Tasks
  0x85 Drivers
  0x86 Guards
  0x87 Modes
  0x88 Ecodes.

Modules = nofModules:int4 {name:string key:int4}.

Constants = nofConstants:int4 {name:string ConstVal}.

ConstVal =
    0x0 val:int4
  | 0x1 val:boolean
  | 0x2 val:string.

Types = nofTypes:int4 {name:string Struct}.

Struct = form:byte1 opaqueTypeName:string.

Ports = nofPorts:int4
  {name:string kind:byte1 Struct (0x0 ConstVal | 0x1 driver:string | 0x2).

Tasks = nofTasks:int4
  {name:string wcet:int4 inputs:PortList outputs:PortList states:PortList.

PortList = nofPorts {portID:int4}.

Drivers = nofDrivers:int4
  ( (setter:0x0 | getter:0x1) portID:int4 driver:string
  | actUpdate:0x2 srcPortID:int4 actPortID:int4
  | copy:0x3 srcPorts:PortList dstPorts:PortList
```

```
  | commit:0x4 taskID:int4
  | schedule:0x5 taskImpl:FunCall taskID:int4
  ).

Guards = nofGuards:int4 {FunCall}

FunCall = name:string args:PortList.

Modes = nofModes:int4
  {name:string start:boolean pcBegin:int4 pcStart:int4}.

Ecodes = nofEcodes:int4
  {opcode:byte1 arg1:int4 arg2:int4 arg3:int4 comment:string}.
```

A.3 Example *TDL* Module

```
module Test {

  const c1 = 77;
  const c2 = true; c3 = false;
  const c4 = "x"; c5 = c4; c6 = -1;

  type Struct1;

  sensor
    int s1 uses gets1;
    Struct1 s2 uses gets2;

  actuator
    int a1 uses seta1;
    Struct1 a2 uses seta2;

  task t1 {
    input
      int i;
      Struct1 i2;
    output
      int o := c1;
      Struct1 o2 uses gets2;
    state
      int s uses myinit;
      Struct1 s2 uses gets2;
    [WCET=2500 ms]
    schedule t1Impl(i, o, s);
  }

  task t2 [1000ms] {
    input int j; int k;
    output int o := 0;
    schedule t2Impl();
  }

  start mode main [5000 ms] {
    task
      [2] t1(s1, s2)
      [1] if t2guard(s1) then t2 (j := s1; k := t1.o;)
    actuator
```

```
      [1] a1 := t1.o;
      [10] if actguard(s1, t1.o) then a2 := t1.o2;
    mode
      [1] if failure() then stop()
  }

  mode stop [1000ms] {
    mode [1] if restarted() then main()
  }

  mode freeze [1000ms] {
  }
}
```

A.4 Generated static class for Java Platform

The following text is the auxiliary Java class generated for module 'Test'. It consists of 3 sections: ports, drivers and guards and provides the table of drivers and the table of guards to the E-machine interpreter. In addition it implements the interface `ModuleBase`.

In principle, the Java based E-machine would also work without this class by falling back to a reflection-based mechanism, which is, however, slower and requires the reflection API to be available. On small embedded systems this may not always be the case.

The only complication for a Java-based E-Machine arises from the fact that Java does not support reference parameters, which are required for output and state ports. Therefore, auxiliary *ref_* classes are used in order to emulate reference parameters as close as possible.

```
import emcore.tools.emachine.types.*;

/**
 * This class has been generated automatically by emcorec -java on
 * Mon Oct 06 13:21:47 CEST 2003 from module 'Test'.
 * Compile this file with a Java compiler and make the generated .class
 * files available to the Java based E-machine in order to speed up execution.
 * Do not modify this file.
 */
public class Test$ implements emcore.tools.emachine.ModuleBase {

  //Ports
  static int port$0; //sensor s1
  static Struct1 port$1; //sensor s2
  static int port$2; //actuator a1
  static Struct1 port$3 = new Struct1(); //actuator a2
  static int port$4; //input i
  static Struct1 port$5 = new Struct1(); //input i2
  static int port$6; //output o
  static ref_int port$6$out = new ref_int(); //actual output o
  static {
    port$6 = 77;
    port$6$out.val = 77;
  }
  static Struct1 port$7 = new Struct1(); //output o2
  static Struct1 port$7$out = new Struct1(); //actual output o2
  static ref_int port$8 = new ref_int(); //state s
  static Struct1 port$9 = new Struct1(); //state s2
  static int port$10; //input j
  static int port$11; //input k
  static int port$12; //output o
```

```
static ref_int port$12$out = new ref_int(); //actual output o
static {
  port$12 = 0;
  port$12$out.val = 0;
}

//Drivers
static emcore.tools.emachine.Driver[] drivers$ = new Driver$[] {
  new Driver$(0),
  new Driver$(1),
  new Driver$(2),
  new Driver$(3),
  new Driver$(4),
  new Driver$(5),
  new Driver$(6),
  new Driver$(7),
  new Driver$(8),
  new Driver$(9),
  new Driver$(10),
  new Driver$(11),
  new Driver$(12),
  new Driver$(13),
  new Driver$(14),
  new Driver$(15),
  new Driver$(16),
};

static class Driver$ implements emcore.tools.emachine.Driver {
  private int n;
  Driver$(int n) {
    this.n = n;
  }
  public void call() throws Exception {
    switch (n) {
      case 0: //init o2
        port$7.copyFrom(Test.gets2());
        port$7$out.copyFrom(port$7);
        break;
      case 1: //init s
        port$8.val = Test.myinit();
        break;
      case 2: //init s2
        port$9.copyFrom(Test.gets2());
        break;
      case 3: //mode switch main
        break;
      case 4: //commit output t1
        port$6 = port$6$out.val;
        port$7.copyFrom(port$7$out);
        break;
      case 5: //commit output t2
        port$12 = port$12$out.val;
        break;
      case 6: //actuator update a1
        port$2 = port$6;
        break;
      case 7: //set a1
```

```
        Test.seta1(port$2);
        break;
      case 8: //get s1
        port$0 = Test.gets1();
        break;
      case 9: //actuator update a2
        port$3.copyFrom(port$7);
        break;
      case 10: //set a2
        Test.seta2(port$3);
        break;
      case 11: //mode switch stop
        break;
      case 12: //get s2
        port$1 = Test.gets2();
        break;
      case 13: //task input t1
        port$4 = port$0;
        port$5.copyFrom(port$1);
        break;
      case 14: //schedule task t1
        Test.t1Impl(port$4, port$6$out, port$8);
        break;
      case 15: //task input t2
        port$10 = port$0;
        port$11 = port$6;
        break;
      case 16: //schedule task t2
        Test.t2Impl();
        break;
      default: throw new Exception("invalid driver number");
    }
  }
}

//Guards
static emcore.tools.emachine.Guard[] guards$ = new Guard$[] {
  new Guard$(0),
  new Guard$(1),
  new Guard$(2),
  new Guard$(3),
};

static class Guard$ implements emcore.tools.emachine.Guard {
  private int n;
  Guard$(int n) {
    this.n = n;
  }
  public boolean eval() throws Exception {
    switch (n) {
      case 0: return Test.restarted();
      case 1: return Test.actguard(port$0, port$6);
      case 2: return Test.failure();
      case 3: return Test.t2guard(port$0);
      default: throw new Exception("invalid guard number");
    }
  }
```

```
  }

  //implement ModuleBase
  public int getKey() {return 0;}
  public emcore.tools.emachine.Driver[] getDrivers() {return drivers$;}
  public emcore.tools.emachine.Guard[] getGuards() {return guards$;}

} //end Test$
```

References

[1] Mössenböck, H.: Coco/R for Java. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/Java/

[2] Henzinger, T., Horowitz, B., Kirsch, Ch.: Giotto: A Time-Triggered Language for Embedded Programming. Proceedings of the IEEE, Vol. 91, No. 1, January 2003.