TDL

Timing Definition Language 1.2

Specification

Josef Templ

C. Doppler Laboratory
Embedded Software Systems
University of Salzburg
Austria

Technical Report T020
(revises T004)
February 2007

Abstract

This report defines the syntax and semantics of the timing definition language *TDL*, which has been developed as part of project MoDECS at the Paris Lodron University of Salzburg (Austria). *TDL* allows one 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 extended features, a more convenient syntax, and an improved set of programming tools.

Contents

1 Introduction

This document defines the syntax and semantics of the timing definition language *TDL*, which has been developed as part of project MoDECS [1] at the Paris Lodron University of Salzburg (Austria). *TDL* represents a domain specific language (DSL) for the target domain of dependable hard real time systems that may be run on a variety of platforms including distributed systems connected via a time triggered communication medium.

TDL allows us to *describe* the timing behavior of a hard real time application and thereby separates the timing aspect of such applications from the functionality. *TDL* programs are purely declarative, ie. all imperative parts of a control application must be provided separately using an imperative programming language such as Java, C or C++. This separation leads to 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.

Please note that this report is not an introduction into the emerging field of time triggered control systems and model based development but assumes some familiarity with the established concepts and terminology of this realm.

1.1 Relation to Giotto

TDL is conceptually based on the time triggered modelling language *Giotto*[1], but provides a more convenient syntax and an improved set of programming tools. The *TDL* compiler and the runtime system needed for the execution of *TDL* programs (E-machine [2]) resulted from a clean room 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 7 for a list of differences.

1.2 Acknowledgements

I would like to thank Prof. Christoph Kirsch, the author of the original Giotto tools, 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 Prof. Wolfgang Pree and the members of the MoDECS team (Emilia Farcas, Claudiu Farcas, and Gerald Stieglbauer) as well as the members of the TDL4FlexRay team (Andreas Naderlinger, Johannes Pletzer, and Stefan Resmerita) 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 our needs.

2 Lexical Structure

A *TDL* specification 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 as well as comments are ignored. Tokens may be keywords, operators, identifiers or 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.

[1]Project MoDECS (**Mo**del-Based development of **D**istributed **E**mbedded **C**ontrol **S**ystems.) was supported by the FIT-IT Embedded Systems grant 807144 of the Austrian government (`www.fit-it.at`). For details please visit `www.MoDECS.cc`.

Examples:

```
//this is a line comment

/* this is a block comment
   //that contains a line comment
   and some other lines.
*/
```

2.3 Identifiers

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

Examples:

```
MyModule s1 _test_task
```

2.4 Keywords and Operators

The following set of keywords is defined in *TDL*. Keywords must not be used as identifiers.

```
actuator as const false if import init input mode module output
public 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 decimal digits, a string is a sequence of arbitrary characters enclosed in single or double quotes. The enclosing quote character must not occur inside the string. Character literals are strings of length one.

Examples:

```
0, 123, 3.14159, 'abc', "bob's house", 'x'
```

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 in order 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).
`( )`	overrides 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. The TDL grammar is designed to be parsed by a top down recursive descent parser, as produced, for example, by the compiler generator Coco/R. Thus, it fulfills the LL(1) rule for context free grammars. For the sake of explaining the syntax, however, we do not always use the LL(1) version of the grammar, which is presented in the appendix.

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

3.1 Module

A module has a name (after keyword ”module”) and provides a namespace for the definition of constants, types, sensors, actuators, tasks and modes.

The name of a module may be composed of a sequence of identifiers separated by dots, called a *qualified identifier*. In general, a qualified identifier consists of a qualifier and an identifier. The qualifier my be empty, though.

In order to create globally unique module names, we recommend to use the vendor's internet domain name in reverse order (most significant part first, e.g. `com.mycompany`) followed by a project name as the qualifier and then a module identifier as the right most part of the module name.

```
TDLModule ::=
  "module" qualIdent "{"
    {"import" {importDecl ";"}}
    {attr "const" {constDecl ";"}}
    {attr "type" {typeDecl ";"}}
    {attr "sensor" {sensorDecl ";"}}
    {attr "actuator" {actuatorDecl ";"}}
    {attr "task" taskDecl}
    {modeDecl}
  "}".
qualIdent ::= [qualifier] identifier.
qualifier ::= {identifier "."}.
attr ::= ["public"].
```

The namespace introduced by a module is enclosed within braces. Names declared within this namespace are visible from the point of declaration up to the end of the module. There may only be a single module per compilation unit.

Declarations may be preceded by the specification of a visibility attribute. All names which are declared `public` are visible to client modules outside the declaring module. Names which are not declared `public` are private.

A name n declared `public` in a module m can be referred to in client modules by using the notation $m.n$. A public task (cf Sec. 3.7) implicitly exports all of its output ports. An output port o of task t of service module m can be accessed in client modules using the notation $m.t.o$. It is not possible to invoke the task in client modules, but only to access its output ports. An exported actuator can actually not be used in client modules.

Examples:

```
module M {
  //import declarations ...

  //constant, type, sensor, actuator, task declarations ...

  //mode declarations ...
}
```

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

3.2 Import Declaration

A module may depend on other modules. This dependency is expressed by specifying an *import declaration*. With respect to the import relationship between modules, the imported module is called a *service module*, whereas the importing module is called a *client module*. A module must not import itself. Thus, the import relationship between modules forms a directed acyclic graph (DAG).

An exception to this rule are so called *temporal cycles*, which are allowed in *TDL*. A temporal cycle means that only the mode declarations make use of cyclic dependencies and the cycle disappears if the modes and the no longer needed import declarations are ommitted from the modules.

```
importDecl ::= simpleImport | groupImport.
simpleImport ::= qualIdent [moduleAlias].
groupImport ::= qualIdent "{" importModule {"," importModule} "}".
importModule ::= identifier [moduleAlias].
moduleAlias ::= "as" identifier.
```

A simple import declaration specifies the qualified name of the imported module optionally followed by an alias name. The alias name, if specified, is used inside a client module to refer to a service module. If no alias is specified, the module identifier is used as an implicit alias name. This allows and actually forces the usage of unqualified module names within a client module whenever an imported module is referenced.

If a group of modules with equal qualifiers is to be imported, a short hand notation may be used as an alternative to a sequence of ordinary imports. The *group import* specifies the qualifier followed by a set of module identifiers enclosed in braces. Optionally, for every module an alias may be specified.

Examples:

```
import M1; M2;
import com.xxx.yyy.M2 as M2xy;
import com.xxx.yyy{M1 as M1xy, M3, M4};
```

3.3 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 example, for initialization of ports (see below) or for timing attributes.

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

The optional identifier following a number may assume the values `ms` or `us` and denotes the timing unit milliseconds or microseconds resp., where the latter is the default. Millisecond values will be converted to microseconds, i.e. they are multiplied by 1000. Otherwise the timing unit has no effect.

Examples:

```
const C1 = 77;
const pi = 3.14159;
const C2 = C1; yes = true;
```

3.4 Type Declaration

A type declaration associates a name with a type, which may either be an existing type or a new type. A new type (also called a user-defined type) is either an array type or a structure (similar to `struct` in C or `RECORD` in Pascal).

In order to execute a control application, all user-defined types 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 (see Sec. 5).

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 they are named `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. Please note that the basic type `char` is a single-byte ASCII character as opposed to Java's Unicode `char` type.

```
typeDecl ::= identifier "="
  ( typeDesignator ";"
  | typeDesignator "[" constExpr "]" ";"
```

```
  | "struct" structScope [";"]
  ).

typeDesignator ::= qualIdent.
structScope ::= "{"
  { typeDesignator identifier { "," identifier } ";" }
  "}".
```

The first form of `typeDecl` introduces an alias name for the type denoted by `typeDesignator`. The second form introduces an array type with element type denoted by `typeDesignator` and array length denoted by `constExpr`, and the third form introduces a structure type with its members defined in `structScope`. Arrays with element type `char` are compatible with string literals if the array length exceeds the number of characters in the string.

Examples:

```
type smallint = short;
type String = char[32];
type Complex = struct {float x, y;}
```

3.5 Sensor Declaration

A sensor declaration defines a read-only variable which represents a particular value of the physical environment of a *TDL* program. During execution, sensor values may change with the progression of time as implied by the physical environment.

```
sensorDecl ::= typeDesignator identifier ["uses" extIdent].
extIdent ::= qualIdent.
```

Sensors are typed variables which may be connected with the environment by using a so-called *getter* function denoted by the external identifier `extIdent`. A getter is an external function which returns a value compatible with the sensor's type. It must be implemented according to the language binding rules and environment the program is executed in.

Examples:

```
sensor int s1 uses getS1;
sensor Complex s2 uses getS2;
```

3.6 Actuator Declaration

An actuator declaration defines a write-only variable which controls the setting of a particular value of the physical environment of a *TDL* program. During execution, actuator values may change with the progression of time as defined by the *TDL* module (see 3.8.2). Actuators may only be set within the module they are declared in.

```
actuatorDecl ::= typeDesignator identifier [initExpr] ["uses" extIdent].
initExpr ::= ":=" constExpr | "init" extIdent.
```

Actuators may be initialized either with a constant value or with an external function, called an *initializer*. Initializers are (like getters) functions, which must return a value compatible with the actuator's type. They must be implemented according to the language binding rules and environment the program is executed in. Actuators which are not initialized explicitly are assumed to be nullified at program start.

Actuators are typed variables which may be connected with the environment using a so-called *setter* function. A setter is an external function with a single parameter compatible with the actuator's type. It must be implemented according to the language binding rules and environment the program is executed in.

Examples:

```
actuator int a1 := 1 uses setA1;
actuator Complex a2 init initA2 uses setA2;
```

3.7 Task Declaration

A task declaration defines a task, which encapsulates a computation typically to be carried out periodically by a real time application (see mode declarations below). Tasks provide a namespace for the declaration of input, output and state ports. In addition, a task uses associated external procedures (including arguments), which perform the task's computation.

```
taskDecl ::= identifier [wcet] "{"
  {"input" {inPortDecl}}
  {"output" {portDecl}}
  {"state" {portDecl}}
  {"uses" [driverAnnotation] call ";"}
  "}".
wcet ::= "[" [attrName "="] constExpr "]".
attrName ::= identifier.
```

A task may have a worst case execution time (wcet) specification, which specifies the maximum time the computation is allowed to take on any platform. Optionally, this attribute may be explicitly named `wcet`. The amount of time is specified by a constant expression. Please note that the *platform specific* worst case execution time of a task is expected to be specified outside the *TDL* module where a module is associated with a particular execution platform.

```
inPortDecl ::= typeDesignator identifier ";".
portDecl ::= typeDesignator identifier [initExpr] ";".
call ::= extIdent "("[portDesignator {"," portDesignator }] ")".
portDesignator ::= qualident.
driverAnnotation ::= "[" identifier "]".
```

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 save state between repeated invocations. The details of connecting tasks will be defined in mode declarations further below.

Output and state ports may be initialized either with a constant value or with an external function, called an *initializer*. Output and state ports which are not initialized explicitly are expected to be nullified at program start.

The external procedures used by a task serve to perform the actual computation, which in the regular case (ie with a single external procedure and without any `driverAnnotation`) is executed after a task has been released and before the logical execution time of the task elapses. Output port updates will be available only after the logical execution time of the task elapses.

For the benefit of digital controller applications, a task's functionality code may be split in parts, (1) a fast step and (2) a slow step, where the fast step is executed in logical zero time right at the release time of the task and the slow step is executed regularly. The difference is expressed by means of a `driverAnnotation`, which must take the value of `release` for the fast step. In any case, an output port must not be updated by more than one step. Output ports updated in the fast step are available immediately for actuator updates in task sequences as defined under mode declarations below.

The arguments of an external procedure must be taken exclusively from the task's ports and they are treated by the external procedure as input (value) or inout (reference) parameters accordingly.

Examples:

```
task t0 {
  input double i;
  output double o;
  uses t0Impl(i, o); //i is input, o is inout parameter
}

task digitalController [10ms] {
  input int i1, i2;
  output int o := 0;
  state double s := 0;
  uses [release] controllerOutput(i1, i2, s, o); //o must be calculated here
  uses controllerUpdate(i1, i2, s, o); //o is an input parameter here
}
```

3.8 Mode Declaration

A mode declaration defines a mode, which is a particular state of operation of a real time application. In general, real time applications may consist of multiple modes[2], one of them will be the *start* mode. Starting a *TDL* program means to switch the E-machine into the distinguished start mode of the modules being executed.

A *TDL* mode consists of a set of activities to be executed periodically. The period of a mode is defined by a constant expression which may be preceded by the explicit attribute name `period`. Activities carried out in a mode include task invocations, actuator updates and mode switches.

```
modeDecl ::= ["start"] "mode" identifier period "{"
  {"task" {taskInvocation}}
  {"actuator" {actuatorUpdate}}
  {"mode" {modeSwitch}}
  "}".
period ::= "[" [attrName "="] constExpr "]".
```

Every activity is performed with a particular frequency per mode period. The mode period must be dividable by this frequency without remainder. The frequency is specified as an attribute of each activity and may be explicitly named `freq`.

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*.

3.8.1 Task Invocation

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. A task invocation may only invoke a task that is defined in the same module.

```
taskInvocation = frequency guard (taskDesignator inputParams [";"] | sequence).
frequency ::= "[" [attrName "="] constExpr "]".
guard ::= ["if" call "then"].
taskDesignator ::= identifier.
inputParams ::= (assignmentList | paramList).
assignmentList ::= "{" {identifier ":=" portDesignator ";"} "}".
paramList ::= ["(" [portDesignator {"," portDesignator}] ")" ].
```

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 logical execution time (LET) of the task has elapsed. The LET for a task invocation with frequency f in a mode with period p is defined as p/f. In case of using the output ports of a task by other activities before the task's LET 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 times (wcet) of all task invocations must not exceed the mode period.

3.8.2 Actuator Update

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. The *update period* of an actuator update with frequency f in a mode with period p is defined as p/f. Actuator updates start after the update period has elapsed, i.e. they are neither carried out at time zero nor in the target mode at the time of a mode switch, but with a delay of one update period.

```
actuatorUpdate ::= frequency guard actPortDesignator ":=" portDesignator ";".
actPortDesignator ::= identifier.
```

[2] 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.

3.8.3 Task Sequence

A task sequence combines a task invocation and subsequent actuator updates where the actuator updates are performed right at the release time of the invoked task given that the task contains a fast step that provides the required output ports. The actuator updates are executed as early as possible, after the fast step of the invoked task, which may be required by digital controller applications.

```
sequence ::= "{"
  taskDesignator inputParams ";"
  {actPortDesignator ":=" portDesignator ";"}
  "}".
```

3.8.4 Mode Switch

A *mode switch* means that the control application switches its current mode of operation to the specified target mode and performs the specified port assignments. The assignments must be to output ports of tasks invoked in the target mode and must be thought of as initializations carried out as a first step in the affected target task's functionality code. The target mode must be different from the source mode.

```
modeSwitch ::= frequency guard modeDesignator assignModePorts.
modeDesignator ::= identifier.
assignModePorts ::= assignmentList | ";".
```

A mode switch is a synchronous operation taking place in logical zero time. The *switch period* for a mode switch with frequency f in a mode with period p is defined as p/f. A mode switch must not occur during the LET of an invoked task, thus, mode switches are said to be *harmonic*. If multiple mode switches are possible at a particular time, they are evaluated in textual order and the first applicable one is taken.

Mode switches in the target mode are never evaluated at the time of the mode switch but with a delay of one switch period. This prevents mode switch cycles without any time passing. The same holds for actuator updates in the target mode, which are not carried out at the time of the mode switch but after the first update period has elapsed.

Examples:

```
start mode main [period = 100ms] {
  task [freq=2] t0(s1);
  task [freq=5] {t1(s2, s3); a1 := t1.o;}
  actuator [freq=2] a2 := t0.o;
  mode [freq=1] if switchToFreeze(s2) freeze;
}
mode freeze [period=1000ms] {
}
```

4 Distribution

The LET-based programming model of TDL provides the foundation for *transparent distribution* of modules across a network of processing nodes. Transparent distribution means that the observable behavior of a system is the same no matter if it is executed locally or distributed over several nodes. The time between finishing a physical task execution and the task's LET may be used for transmitting information to remote nodes without affecting the semantics of the TDL modules.

The definition of the topology of a distributed TDL system is beyond the scope of the TDL language specification and it is up to external tools. Calculating a proper bus schedule, that preserves the logical timing behavior of a distributed TDL system, may either be done automatically, if an appropriate scheduling tool is available or manually, if specific constraints have to be observed. Again, this is left to external tools and conventions and is intentionally left unspecified in the TDL language specification.

5 Language Bindings

Functionality code required by a *TDL* module is provided as static (global) procedures or 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 commonly used programming languages.

5.1 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.

5.1.1 Naming Conventions

The Java name for an unqualified function `f` in module `m` is `m.f`. Thus, it must be defined in a class named after the module and contained in a package as indicated in `m`. The package name of `m` is the qualifier, if there is one, otherwise the anonymous Java package is used. The class name is the identifier of `m`.

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

5.1.2 Type Mapping

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

For output and state ports of a primitive type, an auxiliary *reference* class has to be used[3]. 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 struct or array types, there is no need to provide auxiliary reference classes since objects are passed by reference in Java anyway. Struct types are treated like `struct` in C or `RECORD` in Pascal and are copied by the E-machine when assigned to a port. The same holds for array types, which are copied by means of `java.lang.arraycopy()`.

5.2 ANSI-C

External functionality code written in ANSI-C is expected to be provided in two files, a header and a body file. According to common C programming conventions the header file contains the exported function prototypes and type definitions. The body file includes the header file and defines the functions as declared in the header file. For a module `m` the header file is named `m.h` and the body file is named `m.c` where every '`.`' in `m` is replaced by '`_`'. The replacement of dots by underscores also applies wherever `m` is used in the C code as part of a qualified name which contains `m` as a prefix.

A template of the header file can be generated by the *TDL* compiler plugin for ANSI-C. This template can be renamed to `m.h` and missing parts such as comments can be filled in manually.

The basic types defined in *TDL* are available by including `tdl_types.h`.

5.2.1 Module Initialization

Every module `m` must provide a parameterless `extern void` ANSI-C *init* function named `m_init`.

5.2.2 Functionality Code

For every external function (sensor getter, actuator setter, port initializer, task implementation, guard) there must be a corresponding `extern void` ANSI-C function with appropriate parameters and void return type. 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 may be unqualified. The following naming conventions apply.

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

5.2.3 Naming Conventions

The C name for an unqualified function `f` in a module `m` is `m_f`. Thus, name spaces in *TDL* are mapped to fully expanded C names where name parts are concatenated by using the '_' character. This provides unique C function names without the need of a name space construct.

The C name for a qualified function `f` is derived from `f` by replacing all occurrences of '`.`' by '_'.

5.2.4 Type Mapping

The basic *TDL* types are mapped to primitive C types according to the following table. For *TDL* struct and array types, a corresponding C type must be available by using a `typedef` statement (or by defining a macro). The name of the type follows the naming conventions described above for functions.

A C-based E-machine passes struct and array parameters by reference no matter if they are used as input or inout parameters (for arrays this is forced by C anyway). This avoids unnecessary copy operations. In order to prevent accidental modification of such parameters, they should be marked as `const` when passed as value parameter. It should be noted that the E-machine may copy variables. Thus, there must not be any pointers to or inside parameters.

TDL type	C name	*default C type*
byte	tdl_byte	signed char
boolean	tdl_boolean	unsigned char
char	tdl_char	unsigned char
short	tdl_short	short int
int	tdl_int	long int
long	tdl_long	long long
float	tdl_float	float
double	tdl_double	double

5.2.5 Parameter Passing

Sensor getters and port initializers are void functions that provide their result via a single output parameter, which is passed by reference. Actuator setters are void functions that have a single input parameter. Guards are functions with `int` as return type and with input parameters only. They return their result as the integer value zero (false) or non-zero (true). The number and order of parameters of task implementation functions is exactly the same as in the *TDL* source code. Input ports are passed as input parameters and state and output ports are passed as output parameters. Input parameter of basic types are passed by value and structured and array types are passed by reference. Output parameters are always passed by reference.

6 Changes over previous *TDL* Version

TODO version numbers

- Syntax: Struct and array types added. In order to improve the integration of *TDL* with Simulink or similar tools, it is now possible to define struct and array types directly within a *TDL* module rather than using opaque types.
- Syntax: Opaque and String type removed. Opaque types can be expressed as either struct or array types. String can now be expressed as array of characters with explicit length.
- Language bindings: The language binding rules for Java and C have been extended and adapted in order to cover struct and array types. The new C-language binding rules drop the usage of non-void functions, ie. they prescribe the usage of reference parameters instead of function return values.
- ecode file format: adaped for struct and array types. String and Opaque removed, Alias added.
- Syntax: splitting a task function into a fast and a slow part added to task declarations (taskDecl) by means of multiple *uses* clauses with driver annotation.

- Syntax: immediate actuator port update added to mode declarations (modeDecl) by means of task sequences (sequence).
- Semantics: temporal cyclic imports are allowed, ie. task output ports used in modes may be imported cyclically.
- Compiler: the TDL compiler (tdlc) supports module groups with automatic import ordering and temporal cycle management if the modules are enclosed in parentheses, eg. `( M1.tdl M2.tdl )`
- ecode file format: every module gets two keys now, a *public* key and a *full* key. The public key provides a hash code of the publicly visible module interface, the full key provides a hash code of the full module.
- ecode file format: imports section now uses pubKey rather than key.
- ecode file format: releaseImpl+impl in task as list of impl calls with tag, no longer in start/stop driver.
- ecode file format: start/stop drivers removed from driver table.
- ecode file format: DRVTAG-SWITCH got a new numeric code.
- ecode file format: sequences added to mode
- ecode file format: qualPortID has special moduleID -2 for access to physical port value, used for actuator updates.
- ecode file format: NOP instruction got an argument and is used to delimit ecode sections for two phase execution for cyclic imports, see ECode.NOPTAG
- C language bindings: for every module there must be an initialization function in the functionality code.

7 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 a *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.

non-harmonic mode switch Giotto allows a mode switch 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 non-harmonic mode switches. 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.

deterministic mode switch Giotto requests that among all mode switch guards of a mode only one may return true at a particular point of time. In contrast, *TDL* evaluates mode switch guards in textual order from top to bottom and performs the first mode switch whose guard returns true. This definition allows a more efficient implementation without compromising determinism.

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.

mode port assignments Assignments of task output ports upon a mode switch is done as an initialization in the affected target task in *TDL*. In Giotto it is performed before the target task is invoked, thus, it is visible to clients earlier and thereby implies problems for distributed execution.

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

E-code file format *TDL* defines a binary, platform independent E-code file format and uses statically typed APIs for connecting programs with external functionality code.

E-code instructions The structure and semantics of Giotto E-code instructions has not been changed in *TDL* but one addition has been made.

A SWITCH instruction has been added to E-code. It is used to perform mode switches. In Giotto, mode switches are performed by the JUMP instruction by jumping to code of a different mode. The SWITCH instruction makes this special usage of JUMP explicit and thereby simplifies the detection of mode switches by the E-machine.

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 the execution of *TDL* programs, for example, without including visualization in the basic E-machine directly.

8 References

References

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

[2] Henzinger, T., Kirsch, Ch.: *The Embedded Machine: predictable, portable real-time code.* Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp 315–326, 2002.

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

A *TDL* Grammar

A.1 Complete EBNF Grammar

The lexical and syntactical structure of *TDL* is defined using the compiler generator *Coco/R for Java* [3]. 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 TDLModule;

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

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

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

IGNORE cr + lf + tab


PRODUCTIONS

TDLModule =
  "module" qualIdent "{"
    {"import" {importDecl ";"}} attr /* avoid LL(1) conflict with attr */
    {"const" {constDecl ";"} attr}
    {"type" {typeDecl} attr}
    {"sensor" {sensorDecl ";"} attr}
    {"actuator" {actuatorDecl ";"} attr}
    {"task" taskDecl attr}
    {modeDecl}
  "}".

qualIdent = identifier {"." identifier}.
attr = ["public"].

importDecl = qualIdent
  [ "as" identifier
  | "{" importModule {"," importModule} "}"
  ].

importModule = identifier ["as" identifier].

constDecl = identifier "=" constExpr.
```

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

constExprBoolean = "true" | "false".

typeDecl =  identifier "="
  ( typeDesignator [ "[" constExpr "]" ] ";"
  | "struct" "{" structScope "}" [";"]
  ).

structScope = { typeDesignator identifier { "," identifier} ";" }.

sensorDecl = typeDesignator identifier ["uses" extIdent].

actuatorDecl = typeDesignator identifier [initExpr]
  ["uses" extIdent].

initExpr = ":=" constExpr | "init" extIdent.

taskDecl = identifier [ "[" [attrName "="] constExpr "]" ] "{"
  {"input" {inPortDecl}}
  {"output" {portDecl}}
  {"state" {portDecl}}
  {"uses" {driverAnnotation call ";"}}
  "}".

inPortDecl = typeDesignator identifier ";".

portDecl = typeDesignator identifier [initExpr] ";".

driverAnnotation = [ "[" identifier "]" ].

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

modeDecl = ["start"] "mode" identifier "[" [attrName "="] constExpr "]" "{"
  {"task" {taskInvocation}}
  {"actuator" {actuatorUpdate}}
  {"mode" {modeSwitch}}
  "}".

taskInvocation = frequency guard (taskDesignator inputParams [";"] | sequence).

frequency = "[" [attrName "="] constExpr "]".

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

inputParams = (assignmentList | paramList).
assignmentList = "{" {identifier ":=" portDesignator ";"} "}".
paramList = ["(" [portDesignator {"," portDesignator}] ")" ].

sequence = "{"
  taskDesignator inputParams ";"
  {actPortDesignator ":=" portDesignator ";"}
  "}".
```

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

modeSwitch = frequency guard modeDesignator assignModePorts.

assignModePorts = "{"
  { portDesignator ":=" portDesignator ";" }
  "}"
  | ";" .

designator = identifier {"." identifier}.

/* renamed productions */
attrName = identifier.
unit = identifier.
extIdent = qualIdent.
constDesignator = designator.
typeDesignator = designator.
taskDesignator = designator.
portDesignator = designator.
actPortDesignator = identifier.
modeDesignator = designator.

END tdlc.
```

A.2 Example *TDL* Modules

Module `M1` defines and exports two tasks, one counting up, and one counting down. Both counters are expected to count modulo 11. Module `M2` imports `M1` and calculates the sum of the counters of M1, which is supposed to be constant (initially 10) while `M1` is in mode `m1`, and not constant otherwise.

```
module M1 {

  public const
    c1 = 0; c2 = 10;
    refPeriod = 100ms;

  sensor
    int s uses getS;

  actuator
    int a1 := c1 uses setA1;
    int a2 := c2 uses setA2;

  public task inc [wcet=20ms] {
    output int o := c1;
    uses incImpl(o);
  }

  public task dec [20ms] {
    output int o := c2;
    uses decImpl(o);
  }

  start mode m1 [period=refPeriod] {
    task
      [1] inc();
```

```
      [1] dec();
    actuator
      [1] a1 := inc.o;
      [1] a2 := dec.o;
    mode
      [1] if switch2m2(s) then m2;
  }

  mode m2 [period=refPeriod] {
    task
      [1] inc();
      [2] dec();
    actuator
      [1] a1 := inc.o;
      [2] a2 := dec.o;
    mode
      [1] if switch2m1(s) then m1;
  }
}
```

```
module M2 {

  import M1;

  actuator
    int a := M1.c2 uses setA;

  public task sum [wcet=20ms] {
    input int i1; int i2;
    output int o := M1.c2;
    uses sumImpl(i1, i2, o);
  }

  start mode main [period=M1.refPeriod] {
    task
      [1] sum(M1.inc.o, M1.dec.o);
    actuator
      [1] a := sum.o;
  }
}
```

B Format of .ecode Files

B.1 Grammar 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-first 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. All entities named `nofXXX` specify the number of elements of the subsequent list. Byte values are denoted as in Java or C by using `0x` as prefix of the hexadecimal value. Single byte 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.

```
ECodeFile ::= 'E' 'C' '0' '7' moduleName:string pubKey:int4 moduleKey:int4
  0x80 Imports
  0x81 Constants
  0x82 Types
  0x83 Ports
  0x84 Tasks
  0x85 Drivers
  0x86 Guards
  0x87 Modes
  0x88 Ecodes.

Imports ::= nofImports:int4 {moduleName:string key:int4}.

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

ConstVal ::=
    intConst:0x0 val:int4
  | booleanConst:0x1 val:boolean
  | stringConst:0x2 val:string
  | floatConst:0x3 val:string.

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

Struct ::=
    alias:0x0 StructRef
  | byte:0x1
  | short:0x2
  | int:0x3
  | long:0x4
  | float:0x5
  | double:0x6
  | boolean:0x7
  | char:0x8
  | array:0x9 length:int4 elemStruct:StructRef
  | struct:0xA nofMembers:int4 {name:string pub:boolean StructRef}.

StructRef ::=
    byte:0x1
  | short:0x2
  | int:0x3
  | long:0x4
  | float:0x5
  | double:0x6
  | boolean:0x7
  | char:0x8
  | array:0x9 moduleName:string typeName:string size:int4
  | struct:0xA moduleName:string typeName:string size:int4.

Ports ::= nofPorts:int4
  {name:string pub:boolean StructRef
    ( sensor:0x0 (0x0 | 0x1 getter:string driverID:int4)
    | actuator:0x1 Init (0x0 | 0x1 setter:string driverID:int4)
    | input:0x2
    | output:0x3 Init
```

```
    | state:0x4 Init
    | ft:0x5
    )
  }.

Init ::=
    none:0x0
  | initializer:0x1 initializer:string driverID:int4
  | const:0x2 ConstVal.

Tasks ::= nofTasks:int4
  { name:string pub:boolean wcet:int4
    inPorts:LocalPortList
    outPorts:LocalPortList
    statePorts:LocalPortList
    ftPorts:LocalPortList
    nofUses:int1 {(release:0x0 | exec:0x1) TaskCall}
  }.

LocalPortList ::= nofPorts {portID:int4}.

TaskCall ::= name:string args:LocalPortList.

Drivers ::= nofDrivers:int4
  { init:0x0 portID:int4 initializer:string
  | get:0x1 portID:QualPortID getter:string
  | set:0x2 portID:int4 setter:string
  | actuator:0x3 srcPort:QualPortID actPortID:int4
  | release:0x4 srcPorts:PortList dstPorts:LocalPortList
  | terminate:0x5 taskID:int4
  | switch:0x6 srcPorts:PortList dstPorts:LocalPortList
  }.

QualPortID ::= moduleID:int4 portID:int4.

PortList ::= nofPorts {QualPortID}.

Guards ::= nofGuards:int4 {GuardCall}.

GuardCall ::= name:string args:PortList.

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

Activities ::=
  nofInvokes:int4 {freq:int4 guardID:int4 taskID:int4 releaseDriverID:int4}
  nofSequences:int4 {freq:int4 guardID:int4 nofElems:int4
    {( task:0x0 taskID:int4 releaseDriverID:int4
     | actuator:0x1 actuatorDriverID:int4
     )
    }
  }
  nofUpdates:int4 {freq:int4 guardID:int4 actuatorDriverID:int4}
  nofSwitches:int4 {freq:int4 guardID:int4 targetID:int4 switchDriverID:int4}.

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

The individual operation codes together with their arguments are specified in the following table. Unused operands of E-code instructions have value -1, unused comments in E-code instructions are empty strings. The first argument of the dummy operation NOP may be used for delimiting sections of the generated ecode, viz. the end of terminate section (EOT) and the end of the actuator update section (EOA).

opcode	*mnemonic*	*arg1*	*arg2*	*arg3*
0x0	nop	0=NOP 1=EOT 2=EOA	-1	-1
0x1	future	0	futurePC	deltaTime
0x2	call	driverID	-1	-1
0x3	schedule	taskID	-1	-1
0x4	if	guardID	thenPC	elsePC
0x5	jump	targetPC	-1	-1
0x6	return	-1	-1	-1
0x7	switch	modeID	-1	-1

B.2 Examples for Decoded .ecode Files

The TDL tool suite provides a decoder utility, which produces the following output for the example modules defined in Sec. A.2.

```
MODULE M1 {
  version=07
  pubKey=-50064021
  key=-579190198
IMPORTS
CONSTS
  public c1 = 0
  public c2 = 10
  public refPeriod = 100000
TYPES
PORTS
  [000] actuator int a1 := 0 uses setA1, initDriverID=-1, usesDriverID=2
  [001] actuator int a2 := 10 uses setA2, initDriverID=-1, usesDriverID=3
  [002] sensor int s := null uses getS, initDriverID=-1, usesDriverID=8
  [003] public output int o := 10 uses null, initDriverID=-1, usesDriverID=-1
  [004] public output int o := 0 uses null, initDriverID=-1, usesDriverID=-1
TASKS
  [000] public dec, wcet=20000, input, output 3, state
        uses [exec] decImpl 3
  [001] public inc, wcet=20000, input, output 4, state
        uses [exec] incImpl 4
DRIVERS
  [000] tag=terminate, taskID = 0
  [001] tag=terminate, taskID = 1
  [002] tag=set, actPortID=0, uses=setA1
  [003] tag=set, actPortID=1, uses=setA2
  [004] tag=release, assign:
  [005] tag=release, assign:
  [006] tag=actuator, actPortID=0 srcQID=.4
  [007] tag=actuator, actPortID=1 srcQID=.3
  [008] tag=get, sensorQID=.2, uses=getS
  [009] tag=switch, assign:
  [010] tag=release, assign:
  [011] tag=release, assign:
  [012] tag=actuator, actPortID=1 srcQID=.3
  [013] tag=actuator, actPortID=0 srcQID=.4
```

```
  [014] tag=switch, assign:
GUARDS
  [000] switch2m2(.2 )
  [001] switch2m1(.2 )
MODES
  [000] name=m1, start=true, period=100000, pcBegin=3
        task: freq=1, guardID=-1, taskID=1, releaseDriverID=4
        task: freq=1, guardID=-1, taskID=0, releaseDriverID=5
         actuator: freq=1, guardID=-1, actuatorDriverID=6
         actuator: freq=1, guardID=-1, actuatorDriverID=7
         mode: freq=1, guardID=0, targetID=1, switchDriverID=9
  [001] name=m2, start=false, period=100000, pcBegin=22
        task: freq=1, guardID=-1, taskID=1, releaseDriverID=10
        task: freq=2, guardID=-1, taskID=0, releaseDriverID=11
         actuator: freq=1, guardID=-1, actuatorDriverID=13
         actuator: freq=2, guardID=-1, actuatorDriverID=12
         mode: freq=1, guardID=1, targetID=0, switchDriverID=14
ECODES
  [000] call 2 //actuator init: setA1(a1)
  [001] call 3 //actuator init: setA2(a2)
  [002] return
  [003] call 4 //release task: inc
  [004] release 1 //uses: incImpl
  [005] call 5 //release task: dec
  [006] release 0 //uses: decImpl
  [007] future 0, 9, 100000
  [008] return
  [009] call 1 //terminate task: inc
  [010] call 0 //terminate task: dec
  [011] EOT //end of task termination
  [012] call 6 //actuator update: a1 := o
  [013] call 2 //actuator setter: setA1(a1)
  [014] call 7 //actuator update: a2 := o
  [015] call 3 //actuator setter: setA2(a2)
  [016] EOA //end of actuator updates
  [017] call 8 //get: s := getS()
  [018] if 0, 19, 21 //mode switch guard: switch2m2
  [019] call 9 //mode switch driver
  [020] switch 1 //mode switch -> m2:0
  [021] jump 3 //next cycle: m1
  [022] call 10 //release task: inc
  [023] release 1 //uses: incImpl
  [024] call 11 //release task: dec
  [025] release 0 //uses: decImpl
  [026] future 0, 28, 50000
  [027] return
  [028] call 0 //terminate task: dec
  [029] EOT //end of task termination
  [030] call 12 //actuator update: a2 := o
  [031] call 3 //actuator setter: setA2(a2)
  [032] EOA //end of actuator updates
  [033] call 11 //release task: dec
  [034] release 0 //uses: decImpl
  [035] future 0, 37, 50000
  [036] return
  [037] call 1 //terminate task: inc
  [038] call 0 //terminate task: dec
```

```
  [039] EOT //end of task termination
  [040] call 13 //actuator update: a1 := o
  [041] call 2 //actuator setter: setA1(a1)
  [042] call 12 //actuator update: a2 := o
  [043] call 3 //actuator setter: setA2(a2)
  [044] EOA //end of actuator updates
  [045] call 8 //get: s := getS()
  [046] if 1, 47, 49 //mode switch guard: switch2m1
  [047] call 14 //mode switch driver
  [048] switch 0 //mode switch -> m1:0
  [049] jump 22 //next cycle: m2
}
```

```
MODULE M2 {
  version=07
  pubKey=-570619059
  key=-1477044379
IMPORTS
  [000]  moduleName=M1, key=-50064021
CONSTS
TYPES
PORTS
  [000] actuator int a := 10 uses setA, initDriverID=-1, usesDriverID=1
  [001] input int i1 := null uses null, initDriverID=-1, usesDriverID=-1
  [002] input int i2 := null uses null, initDriverID=-1, usesDriverID=-1
  [003] public output int o := 10 uses null, initDriverID=-1, usesDriverID=-1
TASKS
  [000] public sum, wcet=20000, input 1 2, output 3, state
        uses [exec] sumImpl 1 2 3
DRIVERS
  [000] tag=terminate, taskID = 0
  [001] tag=set, actPortID=0, uses=setA
  [002] tag=release, assign:  1:=0.4 2:=0.3
  [003] tag=actuator, actPortID=0 srcQID=.3
GUARDS
MODES
  [000] name=main, start=true, period=100000, pcBegin=2
       task: freq=1, guardID=-1, taskID=0, releaseDriverID=2
        actuator: freq=1, guardID=-1, actuatorDriverID=3
ECODES
  [000] call 1 //actuator init: setA(a)
  [001] return
  [002] call 2 //release task: sum
  [003] release 0 //uses: sumImpl
  [004] future 0, 6, 100000
  [005] return
  [006] call 0 //terminate task: sum
  [007] EOT //end of task termination
  [008] call 3 //actuator update: a := o
  [009] call 1 //actuator setter: setA(a)
  [010] EOA //end of actuator updates
  [011] jump 2 //next cycle: main
}
```

C Functionality Code

C.1 Examples for Java-based Functionality Code

The functionality code for the example modules in Sec. A.2 can be specified in any programming language supported by the E-machine being used for exeution of TDL programs. The following code examples assume that a Java-based E-machine is used and therefore the functionality code is written in Java following the Java language binding rules of TDL.

```
import com.preetec.tdl.tools.emachine.types.ref_int;
import com.preetec.tdl.tools.emachine.Out;

class M1 {

  static int getS() {
    return 0;
  }

  static void setA1(int a1) {
    Out.println("a1 = " + a1);
  }

  static void setA2(int a2) {
    Out.println("a2 = " + a2);
  }

  static void incImpl(ref_int x) {
    int h = x.val + 1;
    x.val = h <= 10? h: 0;
  }

  static void decImpl(ref_int x) {
    int h = x.val - 1;
    x.val = h >= 0? h: 10;
  }

  static boolean switch2m2(int s) {
    return (s == 2);
  }

  static boolean switch2m1(int s) {
    return (s == 1);
  }
}
```

```
import com.preetec.tdl.tools.emachine.types.ref_int;
import com.preetec.tdl.tools.emachine.Out;

class M2 {

  static void setA(int a) {
    Out.println("a = " + a);
  }

  static void sumImpl(int i0, int i1, ref_int o) {
```

```
    o.val = i0 + i1;
  }
}
```

C.2 Examples for Generated Glue Code

The following programs show the auxiliary Java code generated for the modules. For every module there is one outer class, which 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, a Java based E-machine could also work without this class by falling back to a reflection-based mechanism, which is, however, much slower, requires dynamic memory, and requires the reflection API to be available.

The presented Java code is strongly dependent on a particular E-machine implementation and subject to change at any time. It is shown here only as an example of glue code that might inspire implementations of other E-machines and it shows that the E-machine, glue code, and functionality code work together in a systematic way.

```
import com.preetec.tdl.tools.emachine.types.*;

/**
 * This class has been generated automatically by tdlc -java on
 * Fri Feb 16 15:57:57 CET 2007 from TDL module 'M1'.
 * 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 M1$ implements com.preetec.tdl.tools.emachine.ModuleBase {

  private static com.preetec.tdl.tools.emachine.Module module$;

  //ports
  private static int port$0; //actuator a1
  private static int port$1; //actuator a2
  private static int port$2; //sensor s
  private static int port$2_tick = -1;
  public static int port$3; //output dec.o
  public static ref_int port$3_phy = new ref_int(); //physical output dec.o
  public static int port$4; //output inc.o
  public static ref_int port$4_phy = new ref_int(); //physical output inc.o
  static {
    port$0 = 0;
    port$1 = 10;
    port$3 = 10;
    port$3_phy.val = 10;
    port$4 = 0;
    port$4_phy.val = 0;
  }

  private static class Drivers$
      implements com.preetec.tdl.tools.emachine.Drivers {
    public void call(int id) throws Exception {
      int _ticks;
      switch (id) {
        case 0: //terminate task dec
          port$3 = port$3_phy.val;
          break;
```

```
        case 1: //terminate task inc
          port$4 = port$4_phy.val;
          break;
        case 2: //set a1
          M1.setA1(port$0);
          break;
        case 3: //set a2
          M1.setA2(port$1);
          break;
        case 4: //release task inc
          break;
        case 5: //release task dec
          break;
        case 6: //actuator update a1
          port$0 = port$4;
          break;
        case 7: //actuator update a2
          port$1 = port$3;
          break;
        case 8: //get s
          _ticks = (int)com.preetec.tdl.tools.emachine.Interpreter.ticks;
          if (port$2_tick != _ticks) {
            port$2 = M1.getS();
            port$2_tick = _ticks;
          }
          break;
        case 9: //mode switch to m2
          break;
        case 10: //release task inc
          break;
        case 11: //release task dec
          break;
        case 12: //actuator update a2
          port$1 = port$3;
          break;
        case 13: //actuator update a1
          port$0 = port$4;
          break;
        case 14: //mode switch to m1
          break;
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  private static class Guards$
      implements com.preetec.tdl.tools.emachine.Guards {
    public boolean eval(int id) throws Exception {
      switch (id) {
        case 0: return M1.switch2m2(port$2);
        case 1: return M1.switch2m1(port$2);
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  private static class SDrivers$
```

```
      implements com.preetec.tdl.tools.emachine.SDrivers {
    public void call(int id) throws Exception {
      switch (id) {
        case 0: //start task dec
          M1.decImpl(port$3_phy);
          break;
        case 1: //start task inc
          M1.incImpl(port$4_phy);
          break;
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  //implement ModuleBase
  public void init(com.preetec.tdl.tools.emachine.Module m) {module$ = m;}
  public int getKey() {return -579190198;}
  public com.preetec.tdl.tools.emachine.Drivers getDrivers() {
    return new Drivers$();
  }
  public com.preetec.tdl.tools.emachine.Guards getGuards() {
    return new Guards$();
  }
  public com.preetec.tdl.tools.emachine.SDrivers getSDrivers() {
    return new SDrivers$();
  }
}
```

```
import com.preetec.tdl.tools.emachine.types.*;

/**
 * This class has been generated automatically by tdlc -java on
 * Fri Feb 16 15:57:57 CET 2007 from TDL module 'M2'.
 * 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 M2$ implements com.preetec.tdl.tools.emachine.ModuleBase {

  private static com.preetec.tdl.tools.emachine.Module module$;

  //ports
  private static int port$0; //actuator a
  private static int port$1; //input sum.i1
  private static int port$2; //input sum.i2
  public static int port$3; //output sum.o
  public static ref_int port$3_phy = new ref_int(); //physical output sum.o
  static {
    port$0 = 10;
    port$3 = 10;
    port$3_phy.val = 10;
  }

  private static class Drivers$
      implements com.preetec.tdl.tools.emachine.Drivers {
```

```
    public void call(int id) throws Exception {
      int _ticks;
      switch (id) {
        case 0: //terminate task sum
          port$3 = port$3_phy.val;
          break;
        case 1: //set a
          M2.setA(port$0);
          break;
        case 2: //release task sum
          port$1 = M1$.port$4;
          port$2 = M1$.port$3;
          break;
        case 3: //actuator update a
          port$0 = port$3;
          break;
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  private static class Guards$
      implements com.preetec.tdl.tools.emachine.Guards {
    public boolean eval(int id) throws Exception {
      switch (id) {
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  private static class SDrivers$
      implements com.preetec.tdl.tools.emachine.SDrivers {
    public void call(int id) throws Exception {
      switch (id) {
        case 0: //start task sum
          M2.sumImpl(port$1, port$2, port$3_phy);
          break;
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  //implement ModuleBase
  public void init(com.preetec.tdl.tools.emachine.Module m) {module$ = m;}
  public int getKey() {return -1477044379;}
  public com.preetec.tdl.tools.emachine.Drivers getDrivers() {
    return new Drivers$();
  }
  public com.preetec.tdl.tools.emachine.Guards getGuards() {
    return new Guards$();
  }
  public com.preetec.tdl.tools.emachine.SDrivers getSDrivers() {
    return new SDrivers$();
  }
}
```