TDL - Timing Definition Language Specification

Version 1.6
Syntax and Semantics

Contents

TDL - Timing Definition Language Specification 1.6

Abstract

This document defines the syntax and semantics of the Timing Definition Lnguage (TDL), which has originally been developed as part of the project MoDECS at the Paris Lodron University of Salzburg (Austria). TDL allows one to specify the timing behavior of a hard real-time 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 C or C++. TDL is conceptually based on Giotto, but provides extended features, a more convenient syntax, and an improved set of programming tools.

Introduction

This document defines the syntax and semantics of the Timing Definition Lnguage (TDL). 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 one 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, i.e. all imperative parts of a control application must be provided separately using an imperative programming language such as 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 document 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.

Relation to Giotto

TDL is conceptually based on the time-triggered modeling language Giotto [Giotto] 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 [PLDI02]) resulted from a clean room implementation without access to the Giotto compiler or the corresponding 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 Differences to Giotto for a list of differences.

Acknowledgements

We 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. Thanks to 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 we 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.

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.

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.

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.

Examples:

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

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
```

Keywords and Operators

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

```
actuator as asynchronous const false if import init
input mode module output public sensor start state
struct task then true type uses
```

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

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

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'
```

Syntactical Structure

An EBNF grammar notation is used in order to define the syntax of TDL. Keywords, operators and special symbols are enclosed in double quotes. The following meta symbols are used for defining the productions.

Symbol	Meaning
::=	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 [Coco]. 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.

Module

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

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 "output" {portDecl ";"}}

{attr "task" taskDecl}

{modeDecl}

["asynchronous" asyncDecl]

"}".

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 (see Sec. Task Declaration) 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. Actuator ports must not be public.

Examples:

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

  //constant, type, sensor, actuator, output
  //and task declarations ...
```

```
  //mode declarations ...

  //asynchronous declarations
}
```

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

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 omitted 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};
```

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 respectively, 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;
```

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. Language Bindings).

TDL provides a set of basic types, which matches those found in the programming language Java. The basic types are pre-declared 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;}
```

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;
```

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 Actuator Update). 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;
```

Output Declaration

Output ports declared at the module level represent global output ports, i.e. they may be used as output ports by any task of the module. For more details about tasks and task output ports please refer to section Task Declaration.

portDecl ::= typeDesignator identifier [initExpr].

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

Examples:

```
output int o;
output Complex p init initP;
```

Task Declaration

A task declaration defines a task, which encapsulates a computation typically to be carried out periodically by a real time application (see Sec. Mode Declaration 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.

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

portDesignator ::= qualident | functionDesignator "(" qualident ")".

functionDesignator ::= identifier.

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 the module's global output ports and they are treated by the external procedure as input (value) or in-out (reference) parameters accordingly. Output ports defined locally within a task take precedence over equally named global output ports.

For dealing with fault tolerance issues a standard function named `FTPORT` is available and described in more detail in section Fault Tolerance below.

Examples:

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

task digitalController [10ms] {
  input int i1; int 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
}
```

Mode Declaration

A mode declaration defines a mode, which is a particular state of operation of a module. In general, real time components may consist of multiple modes[1], 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. Activities carried out in a mode include task invocations, actuator updates and mode switches.

[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.

All activities carried out in a mode start and end at so-called *logical time instants*. All modules being executed are synchronized to this logical time and start at time zero.

The difference between the start and end time of a task invocation is called the task's *logical execution time* (LET) and is always greater than zero. Actuator updates and mode switches start and end at the same logical time instant and are thus called *logical zero time* activities.

An activity may be executed several times within one mode period. Therefore every activity is associated with a *frequency* that defines the *activity's period* as the quotient of the mode period and the frequency. The frequency is an integer number and the mode's period must be dividable by the frequency without a remainder.

All start and end time instants of an activity within a mode period are an integer multiple of the activity's period. An arbitrary subset of the possible executions may be selected but the following rules must be followed: (1) the LET of a task invocation must not overlap with another invocation of the same task within a mode period; (2) an actuator update or a global output port update must occur only once per logical time instant within a mode period.

modeDecl ::= ["start"] "mode" identifier period "{"

{"task" {frequency guard taskInvocation}}

{"actuator" {frequency guard actuatorUpdate}}

{"mode" {frequency guard modeSwitch}}

"}".

period ::= "[" [attrName "="] constExpr "]".

frequency ::= "[" [attrName "="] constExpr [",” slotSelection] "]".

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

The period of a mode is defined by a constant expression which may be preceded by the explicit attribute name `period`.

The frequency of an activity is specified by an attribute of the activity and may be explicitly named `freq`. The frequency defines the period of the activity, which implicitly defines the possible logical start and end times of the activity. The activity period splits the mode period into a sequence of so-called *slots* with the length of one activity period each. By default, every task slot is filled with a task invocation and logical zero time activities are carried out at the end of each slot. For fine tuning the timing of an activity, an optional *slot selection* attribute may be defined as described below.

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

slotSelection::= [attrName "="] slotSel.

slotSel ::= [slotGroup {"|" slotGroup}].

slotGroup ::= ["~"] startSlot ["-" endSlot] [""].*

startSlot ::= constExpr.

endSlot ::= constExpr.

The slot selection attribute, which may optionally be named `slots`, defines the timing of an activity explicitly by specifying so-called *slot groups*. A slot group is an interval defined by a start- and an end-slot number, where slots are numbered from 1 on and end-slot is greater than or equal to start-slot. The activity will be executed for the specified slot groups only. The LET of a task invocation may thereby extend multiple slots and there may also be slots without any activity.

In order to provide a compact syntax for specifying repeating patterns for slot groups until the end of the mode or until the next slot group defined, the character `*` may be used. The character ~ serves to mark a task slot group as being optional, i.e. at execution time it is allowed to ignore such a task invocation, which may help to find a feasible schedule. If the end slot is not specified explicitly it is the same as the start slot.

If slot selection is not used, this defaults to `slots=1*`, which means that every slot is considered a slot group and all slots are to be filled with a mandatory activity.

Examples for slot selection:

`slots=1*`	all slots are mandatory and LET=*p/f*; this is the default.
`slots=~1|2*`	the first slot is optional, the remaining slots are mandatory.
`slots=1-3*`	mandatory slot groups with LET=3**p/f* each.

Examples for mode declarations:

```
start mode main [period = 100ms] {
  task [freq=2]
    ...//see below
  task [freq=5, slots=1-2*]
    ...//see below
  actuator [freq=2]
    ...//see below
  mode [freq=1] if switchToFreeze(s2) then
    ...//see below
}

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

Task Invocation

A *task invocation* means that the task's input ports are updated according to the input parameters and the task's computation is scheduled for execution. The input parameters 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 sensor ports, task output ports, or global output ports. A task invocation may only invoke a task that is defined in the same module. Note that a task invocation does not specify the task's output ports. Those are specified in the task's declaration and may be referred to by a designator of the form *taskName.outputPortName* wherever the output port is needed.

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

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.

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 on a single processor system the sum of the worst case execution times (wcet) of all task invocations must not exceed the mode period.

Example:

```
task [freq=2] t0(s1);
```

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. Actuator updates are neither carried out at time zero nor in the target mode at the time of a mode switch.

actuatorUpdate ::= actPortDesignator ":=" portDesignator ";".

actPortDesignator ::= identifier.

Examples:

```
actuator [freq=2] a2 := t0.o;
actuator [freq=5, slots=3-5*|18] a2 := M3.t0.o;
```

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 ";"}

"}".

Example:

```
task [freq=5, slots=1-2*] {t1(s2, s3); a1 := t1.o;}
```

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 ::= modeDesignator assignModePorts.

modeDesignator ::= identifier.

assignModePorts ::= assignmentList | ";".

A mode switch is a synchronous operation taking place in logical zero time. 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 are neither carried out at time zero nor in the target mode at the time of a mode switch. This prevents mode switch cycles without any time passing.

Example:

```
mode [freq=1] if switchToFreeze(s2) then freeze;
```

Asynchronous Declarations

An asynchronous activity in TDL is an activity that is carried out in the spare time between execution of timed (synchronous) activities and thereby does not disturb the real-time properties of a system. Its execution may be triggered by a variety of events. Asynchronous activities are never preempted by other asynchronous activities but may be preempted by synchronous activities. The TDL runtime system takes care of the synchronization of the data flow between synchronous and asynchronous activities such that reading input ports, updating output ports, and performing actuator updates are atomic actions.

asyncDecl ::= "{" {asyncSequence} "}" .

asyncSequence ::="[" asyncEvent "]" guard

{ taskDesignator inputParams ";"

| actPortDesignator ":=" portDesignator ";"

} .

TDL supports the grouping of asynchronous activities into sequences that are triggered as one unit and executed strictly sequential. Any such sequence has an associated trigger event, an optional guard, and a sequence of asynchronous activities. An asynchronous activity may be a task invocation or an actuator update. A task may either be invoked synchronously or asynchronously but not both. Also, an actuator update must either be done synchronously or asynchronously but not both.

Triggering an asynchronous activity sequence means that the sequence is registered for execution at some later time at the discretion of the TDL runtime system. Any additional triggering of a registered activity sequence is ignored until the execution of this activity sequence starts. Parameter passing takes place as part of the execution not at the time of registration.

asyncEvent ::= eventAttr ["," priorityAttr] .

eventAttr ::= attrName "=" (identifier | constExpr | portDesignator) .

priorityAttr ::= attrName "=" constExpr.

The kind of event that triggers the execution of an asynchronous activity sequence is specified by the attribute name `interrupt`, `timer`, or `update`. In case of an interrupt, the attribute value must be an identifier. This logical interrupt name needs to be mapped to platform specific interrupt specifications outside the TDL source code. In case of a timer, the attribute value must be an integer greater than zero. It describes the period of a timer in microseconds. In case of a port update, the attribute value must be the (potentially qualified) name of an output port. Whenever this port receives a value it triggers the asynchronous activity sequence.

The priority is specified by the attribute name `priority` and a value greater or equal to zero where higher numbers mean higher priority. The priority attribute affects the queueing order of registered asynchronous activity sequences. The default is the lowest priority.

Examples:

```
asynchronous {
  [interupt=int3, priority=5]
  if guard1(s1) then t1(s1, t.o); a1 := t1.o;
  [timer=10ms, priority=4]
  t2(s2);
  [update=t1.o]
  t3();
}
```

Fault Tolerance

Support for fault tolerance based on the replication of a module on several nodes is provided in TDL by means of a standard function named `FTPORT`, which takes a public output port as an argument and returns the corresponding counter for the availability of the specified port. The type of the return value is `byte`. In the trivial case, i.e. without fault tolerance support, this counter always has the value of one, which means that the value is available exactly once. In the general case, i.e. with fault tolerance support and replicating a module on several nodes, *FTPORT(o)* returns the number of available replicas of *o* at runtime, ranging from zero to the number of replicas.

It is left to the application to decide what to do in case a value of zero is returned. A typical reaction, however, would be to switch to an error mode. This is possible because FTPORT can also be used as an argument of a guard function.

It is left to the TDL runtime system and platform mapping to decide what to do if a value greater than one occurs and the replicas provide different values for a port. Typical strategies are fail over, majority voting, or averaging.

Examples:

```
FTPORT(o)
```

```
FTPORT(M1.task1.o)
```

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 communication 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.

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 conventions for commonly used programming languages.

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.

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.

Type Mapping

The basic TDL types are mapped 1:1 to primitive Java types. For TDL struct 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 `com.preetec.tdl.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[2]. These classes are contained in package `com.preetec.tdl.tools.emachine.types` for all primitive types.

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

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.System.arraycopy()`.

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 plug-in 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`.

Module Initialization

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

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.

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 '`_`'.

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 in-out 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

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.

TDL Version History

Version 1.6

- Fault tolerance support added by means of the standard function FTPORT.

Version 1.5

- Asynchronous activities added.
- E-code instruction arguments reduced to two.
- E-code instruction *repeat* added.

Version 1.4

- Syntax: Flexible slot selection added. It is now possible to separate the LET from the repetition period and to specify optional task execution.

Version 1.3

- Syntax: Global Output Ports added. In addition to having output ports defined per task, it is now also possible to have output ports defined at the module level. Such output ports may be set by any task of a module. Only one task invoked in a mode may set a global output port, though.

Version 1.2

- 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, i.e. they prescribe the usage of reference parameters instead of function return values.
- E-code file format: adapted for struct and array types. String and Opaque removed, Alias added.

Version 1.1

- 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, i.e. 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, e.g. `( M1.tdl M2.tdl )`
- E-code 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.
- E-code file format: imports section now uses pubKey rather than key.
- E-code file format: releaseImpl+impl in task as list of impl calls with tag, no longer in start/stop driver.
- E-code file format: start/stop drivers removed from driver table.
- E-code file format: DRVTAG-SWITCH got a new numeric code.
- E-code file format: sequences added to mode
- E-code file format: qualPortID has special moduleID -2 for access to physical port value, used for actuator updates.

- E-code file format: NOP instruction got an argument and is used to delimit E-code 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.

Differences to Giotto

Syntactical Differences

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,
- in addition to global output ports, TDL provides also task output ports,
- 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 introduction of asynchronous activities.

Semantic Differences

The following list explains differences between TDL and 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.
- [sensor read] Giotto defines that sensors are read right before task invocations and, as a consequence, sensor values used for actuator updates or mode switches are old values. TDL uses current values for sensors in all places in order to provide deterministic behavior even in the case that multiple modules access a shared sensor.

Tool Related Differences

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 been changed slightly in TDL. In particular, the number of arguments has been reduced from three to two, which affects the IF and the FUTURE instruction.
- 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.
- The NOP instruction received an additional argument which allows it to be used as a marker that separates three different phases in E-code blocks.
- [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.

References

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

[PLDI02] 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.

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

Appendix A: TDL Grammar

Complete EBNF Grammar

The lexical and syntactical structure of TDL is defined using the compiler generator *Coco/R for Java* [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 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}
    {"output" {portDecl ";"} attr}
    {"task" taskDecl attr}
    {modeDecl}
    ["asynchronous" asyncDecl]
  "}".

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 ["," slotSelection] "]".

slotSelection = [attrName "="] slotSel.
slotSel =   [slotGroup {"|" slotGroup}].
slotGroup = ["~"] constExpr ["-" 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 ";" }
  "}"
  | ";" .

asyncDecl = "{" {asyncSequence} "}".

asyncSequence = "[" asyncEvent "]" guard
  { IF (IsTask()) taskDesignator inputParams ";"
  | actPortDesignator ":=" portDesignator ";"
  } .

asyncEvent =
```

```
  attrName "="
  ( IF (IsInterrupt()) Ident
  | IF (IsUpdate()) portDesignator
  | constExpr
  )
  ["," attrName "=" constExpr]
  .

designator = identifier {"." identifier}.

/* renamed productions */
attrName = identifier.
unit = identifier.
extIdent = qualIdent.
constDesignator = designator.
typeDesignator = designator.
taskDesignator = designator.
portDesignator = designator ["(" designator ")"].
actPortDesignator = identifier.
modeDesignator = designator.

END tdlc.
```

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. In addition, there is a periodic asynchronous task invocation that is supposed to watch the system behavior.

M1.tdl

```
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);
  }

  public task watchdog {
    input int i1; int i2;
    uses watchdogImpl(i1, i2);
  }

  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;
  }

  asynchronous {
    [timer=1000ms] watchdog(inc.o, dec.o);
  }
}
```

M2.tdl

```
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;
  }
}
```

Appendix B: Format of E-code Files

Grammar of E-code Files

The following attributed EBNF grammar describes the format of E-code 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, E-code 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. Slot selection is represented as a string according to the slot selection syntax of TDL.

ECodeFile ::= 'E' 'C' '1' '0' moduleName:string pubKey:int4 moduleKey:int4
0x80 Imports
0x81 Constants
0x82 Types
0x83 Ports

```
 0x84 Tasks
 0x85 Drivers
 0x86 Guards
 0x87 Modes
 0x88 Asyncs
 0x89 Ecodes.

Imports ::= nofImports:int4 {moduleName:string pubKey: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
 | asyncRelease:0x7 srcPorts:PortList dstPorts:LocalPortList
 | asyncActuator:0x8 srcPort:QualPortID actPortID:int4
 }.

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

Asyncs ::= nofAsyncs:int4 {AsyncEvent guardID:int4 AsyncActs}.

AsyncEvent ::=
   interrupt:0x0 name:string
 | timer:0x1 period:int4
 | update:0x2 port:QualPortID.

AsyncActs ::= nofAsyncActs:int4
 {task:0x0 taskID releaseDriverID:int4
```

```
 | actuator:0x1 updateDriverID:int4
 } .

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

The individual operation codes together with their arguments and meaning 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 E-code, viz. the end of terminate section (EOT) and the end of the actuator update section (EOA).

opcode	mnemonic	arg1	arg2	Meaning
0x0	nop	0=NOP 1=EOT 2=EOA	-1	dummy (no operation) instruction. The argument is used as a marker for identifying different sections in the E-Code
0x1	future	futurePC	deltaTime	plans the execution of the E-code block starting at futurePC in deltaTime microseconds
0x2	call	driverID	-1	executes the driver with given driverID
0x3	release	taskID	-1	marks the task with given taskID as ready for execution
0x4	if	guardID	elsePC	proceeds with the next instruction if the guard with the given guardID evaluates to true else jumps to elsePC
0x5	jump	targetPC	-1	jumps to the instruction at targetPC
0x6	return	-1	-1	terminates an E-code block
0x7	switch	modeID	-1	performs a mode switch to the target mode with given modeID, i.e. the E-machine continues at the entry point of the target mode. In addition it resets the module's repeat counter.
0x8	repeat	targetPC	n	uses a counter per module for jumping n times to targetPC and then continues with the next instruction.

The TDL compiler generates E-code for each mode of a module. The E-code covers a single mode period and is repeated then by means of a *jump* instruction to the beginning of the mode. For every logical time instant which needs to execute E-code one E-code block is generated. An E-Code block is a list of E-code instructions terminated by a *return* instruction. It specifies for one logical time instant the actions that must be taken by the E-Machine in order to comply with the LET semantics. The following sequence of actions comprises one E-Code block for a logical time instant *t*:

1. Update output ports of task invocations logically terminating at *t* with the result values from its execution.
2. EOT. Mark end of task termination phase.

3. Update actuators that are defined to be updated at *t*.
4. EOA. Mark end of actuator update phase.
5. Switch mode if a mode switch is defined at *t*.
6. (*) Update input ports of tasks that are defined to be released at *t*.
7. Release tasks that are defined to be released at *t*.
8. Advance time to the next logical time instant *t* + *deltaTime* and register the future PC.
9. Return from E-code block.

The position marked (*) of a mode's first E-Code block is the mode's entry point. Execution will start here upon a mode switch to this mode or upon startup.

Note that sensors are read whenever their value is required. However, at one particular logical time a sensor is read only once, even if it is used multiple times.

Examples for Decoded E-code Files

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

M1.ecode

```
MODULE M1 {
  version=10
  pubKey=-2080042151
  key=1605588190
IMPORTS
CONSTS
  public c1 = 0
  public c2 = 10
  public refPeriod = 100000
TYPES
PORTS
  [000] actuator int a1:=0 uses setA1, initDriverID=-1, usesDriverID=3
  [001] actuator int a2:=10 uses setA2, initDriverID=-1, usesDriverID=4
  [002] sensor int s:=null uses getS, initDriverID=-1, usesDriverID=7
  [003] public output int o:=10 uses null, initDriverID=-1, usesDriverID=-1
  [004] public output int o:=0 uses null, initDriverID=-1, usesDriverID=-1
  [005] input int i1:=null uses null, initDriverID=-1, usesDriverID=-1
  [006] input int i2:=null 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
  [002] public watchdog, wcet=0, input 5 6, output, state
        uses [exec] watchdogImpl 5 6
DRIVERS
  [000] tag=terminate, taskID = 0
  [001] tag=terminate, taskID = 1
  [002] tag=terminate, taskID = 2
  [003] tag=set, actPortID=0, uses=setA1
  [004] tag=set, actPortID=1, uses=setA2
  [005] tag=release, assign:
  [006] tag=release, assign:
  [007] tag=get, sensorQID=.2, uses=getS
  [008] tag=actuator, actPortID=0 srcQID=.4
  [009] tag=actuator, actPortID=1 srcQID=.3
  [010] tag=switch, assign:
  [011] tag=release, assign:
  [012] tag=release, assign:
  [013] tag=actuator, actPortID=1 srcQID=.3
  [014] tag=actuator, actPortID=0 srcQID=.4
  [015] tag=switch, assign:
  [016] tag=asyncrelease, assign:  5:=.4 6:=.3
GUARDS
```

```
  [000] switch2m2( .2)
  [001] switch2m1( .2)
MODES
  [000] name=m1, start=true, period=100000, pcBegin=3
        task: freq=1, slots=1*, guardID=-1, taskID=1, releaseDriverID=5
        task: freq=1, slots=1*, guardID=-1, taskID=0, releaseDriverID=6
        actuator: freq=1, slots=1*, guardID=-1, actuatorDriverID=8
        actuator: freq=1, slots=1*, guardID=-1, actuatorDriverID=9
        mode: freq=1, slots=1*, guardID=0, targetID=1, switchDriverID=10
  [001] name=m2, start=false, period=100000, pcBegin=22
        task: freq=1, slots=1*, guardID=-1, taskID=1, releaseDriverID=11
        task: freq=2, slots=1*, guardID=-1, taskID=0, releaseDriverID=12
        actuator: freq=1, slots=1*, guardID=-1, actuatorDriverID=14
        actuator: freq=2, slots=1*, guardID=-1, actuatorDriverID=13
        mode: freq=1, slots=1*, guardID=1, targetID=0, switchDriverID=15
ASYNCS
  [000] [timer=1000000, priority=0] taskID=2, driverID=16;
ECODES
  [000] call 3 //actuator init: setA1(a1)
  [001] call 4 //actuator init: setA2(a2)
  [002] return
  [003] call 5 //release task: inc
  [004] release 1 //uses: incImpl
  [005] call 6 //release task: dec
  [006] release 0 //uses: decImpl
  [007] future 9, 100000
  [008] return
  [009] call 7 //get: s := getS()
  [010] call 1 //terminate task: inc
  [011] call 0 //terminate task: dec
  [012] EOT //end of task terminations
  [013] call 8 //actuator update: a1 := o
  [014] call 3 //actuator setter: setA1(a1)
  [015] call 9 //actuator update: a2 := o
  [016] call 4 //actuator setter: setA2(a2)
  [017] EOA //end of actuator updates
  [018] if 0, 21 //mode switch guard: switch2m2
  [019] call 10 //mode switch driver
  [020] switch 1 //mode switch -> m2:0
  [021] jump 3 //next cycle: m1
  [022] call 11 //release task: inc
  [023] release 1 //uses: incImpl
  [024] call 12 //release task: dec
  [025] release 0 //uses: decImpl
  [026] future 28, 50000
  [027] return
  [028] call 0 //terminate task: dec
  [029] EOT //end of task terminations
  [030] call 13 //actuator update: a2 := o
  [031] call 4 //actuator setter: setA2(a2)
  [032] EOA //end of actuator updates
  [033] call 12 //release task: dec
  [034] release 0 //uses: decImpl
  [035] future 37, 50000
  [036] return
  [037] call 7 //get: s := getS()
  [038] call 1 //terminate task: inc
  [039] call 0 //terminate task: dec
  [040] EOT //end of task terminations
  [041] call 14 //actuator update: a1 := o
  [042] call 3 //actuator setter: setA1(a1)
  [043] call 13 //actuator update: a2 := o
  [044] call 4 //actuator setter: setA2(a2)
  [045] EOA //end of actuator updates
  [046] if 1, 49 //mode switch guard: switch2m1
  [047] call 15 //mode switch driver
  [048] switch 0 //mode switch -> m1:0
  [049] jump 22 //next cycle: m2
}
```

M2.ecode

```
MODULE M2 {
  version=10
  pubKey=1184433608
```

```
  key=606345388
IMPORTS
  [000]  moduleName=M1, pubKey=-2080042151
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, slots=1*, guardID=-1, taskID=0, releaseDriverID=2
        actuator: freq=1, slots=1*, guardID=-1, actuatorDriverID=3
ASYNCS
ECODES
  [000] call 1 //actuator init: setA(a)
  [001] return
  [002] call 2 //release task: sum
  [003] release 0 //uses: sumImpl
  [004] future 6, 100000
  [005] return
  [006] call 0 //terminate task: sum
  [007] EOT //end of task terminations
  [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
}
```

Appendix C: Functionality Code

Examples for Java-based Functionality Code

The functionality code for the example modules in Sec. Example TDL Modules can be specified in any programming language supported by the E-machine being used for execution 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.

M1.java

```
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 void watchdogImpl(int i1, int i2) {
    Out.println("i1=" + i1 + ", i2=" + i2);
  }

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

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

M2.java

```
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;
  }
}
```

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`. There is an additional Java class per node called NodeAsyncs$$.java. This class implements the glue code required for dealing with asynchronous activities.

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.

M1$.java

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

/**
 * This class has been generated automatically by tdlc -java on
 * Thu Aug 28 15:40:59 CEST 2008 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$;
  public static final com.preetec.tdl.tools.emachine.Drivers drivers$ = new Drivers$();
```

```
public static final com.preetec.tdl.tools.emachine.Guards guards$ = new Guards$();
public static final com.preetec.tdl.tools.emachine.SDrivers sdrivers$ = new SDrivers$();

//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
private static int port$5; //input watchdog.i1
private static int port$6; //input watchdog.i2
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: //terminate async task watchdog
        com.preetec.tdl.tools.emachine.Interpreter.asyncDriverID = 2;
        com.preetec.tdl.tools.emachine.Interpreter.asyncDrivers = this;
        com.preetec.tdl.tools.emachine.Interpreter.asyncDrivers = null;
        break;
      case 3: //set a1
        M1.setA1(port$0);
        break;
      case 4: //set a2
        M1.setA2(port$1);
        break;
      case 5: //release task inc
        break;
      case 6: //release task dec
        break;
      case 7: //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 8: //actuator update a1
        port$0 = port$4;
        break;
      case 9: //actuator update a2
        port$1 = port$3;
        break;
      case 10: //mode switch to m2
        break;
      case 11: //release task inc
        break;
      case 12: //release task dec
        break;
      case 13: //actuator update a2
        port$1 = port$3;
        break;
      case 14: //actuator update a1
        port$0 = port$4;
        break;
      case 15: //mode switch to m1
```

```
        break;
      case 16: //async release task watchdog
        do {
          com.preetec.tdl.tools.emachine.Interpreter.executed = false;
          port$5 = port$4;
          port$6 = port$3;
        } while (com.preetec.tdl.tools.emachine.Interpreter.executed);
        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;
      case 2: //start task watchdog
        M1.watchdogImpl(port$5, port$6);
        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 1605588190;}
 public com.preetec.tdl.tools.emachine.Drivers getDrivers() {return drivers$;}
 public com.preetec.tdl.tools.emachine.Guards getGuards() {return guards$;}
 public com.preetec.tdl.tools.emachine.SDrivers getSDrivers() {return sdrivers$;}
}
```

M2$.java

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

/**
 * This class has been generated automatically by tdlc -java on
 * Thu Aug 28 15:40:59 CEST 2008 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$;
 public static final com.preetec.tdl.tools.emachine.Drivers drivers$ = new Drivers$();
 public static final com.preetec.tdl.tools.emachine.Guards guards$ = new Guards$();
 public static final com.preetec.tdl.tools.emachine.SDrivers sdrivers$ = new SDrivers$();

 //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 606345388;}
  public com.preetec.tdl.tools.emachine.Drivers getDrivers() {return drivers$;}
  public com.preetec.tdl.tools.emachine.Guards getGuards() {return guards$;}
  public com.preetec.tdl.tools.emachine.SDrivers getSDrivers() {return sdrivers$;}
}
```

NodeAsyncs$$.java

```
/**
 * This class has been generated automatically by tdlc -java on
 * Thu Aug 28 15:40:59 CEST 2008 .
 * Compile this file with a Java compiler and make the generated .class
 * file available to the Java based E-machine. Do not modify this file.
 */
public class NodeAsyncs$$ {

  //runtime data structure for AsyncSequence
  private static class AsyncSequenceDesc {
    public boolean pending;
    public int priority;
    public AsyncSequenceDesc(int priority) {
      this.priority = priority;
```

```
  }
}

//table of all AsyncSequenceDescs
private static AsyncSequenceDesc[] asTable = new AsyncSequenceDesc[1];
static {
  asTable[0] = new AsyncSequenceDesc(0);
}

//add to priority queue unless it is already in queue
public static void enqueue(int idx) {
  asTable[idx].pending = true;
  asyncThread.interrupt();
}

//remove from priority queue
private static int dequeue() {
  int maxPriority = -1;
  int maxPriorityIndex = -1;
  for (int i = 0; i < 1; i++) {
    AsyncSequenceDesc as = asTable[i];
    if (as.pending && as.priority > maxPriority) {
      maxPriority = as.priority;
      maxPriorityIndex = i;
    }
  }
  if (maxPriorityIndex >= 0) {
    asTable[maxPriorityIndex].pending = false;
  }
  return maxPriorityIndex;
}
private static void startTimerThread(final int period) {
  Thread t = new Thread() {
    public void run() {
      for (;;) {
        try {
          switch (period) {
            case 1000000:
              enqueue(0);
              Thread.sleep(1000);
              break;
          }
        } catch (InterruptedException x) {}
      }
    }
  };
  t.setPriority(Thread.MIN_PRIORITY);
  t.setDaemon(true);
  t.start();
}

static {
  startTimerThread(1000000);
}

//execute an AsyncSequence
private static void executeAsyncSequence (int n) throws Exception {
  switch (n) {
    case 0:
      M1$.drivers$.call(16);
      M1$.sdrivers$.call(2);
      M1$.drivers$.call(2);
      break;
  }
}

private static Thread asyncThread = new Thread() {
  public void run() {
    for (;;) {
      int next = dequeue();
      if (next >= 0) {
        try {
          executeAsyncSequence(next);
        } catch (Exception x) {
          x.printStackTrace();
```

```
          }
        } else {
          try {Thread.sleep(10);}
          catch (InterruptedException x) {}
        }
      }
    }
  };
  static {
    asyncThread.setPriority(Thread.MIN_PRIORITY);
    asyncThread.setDaemon(true);
    asyncThread.start();
  }
}
```