TDL Specification and Report

Josef Templ

Technical Report T004
(revises T001)
November 2004
http://cs.uni-salzburg.at/pubs/reports/T004.pdf

Department of Computer Science
University of Salzburg
Austria

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 us 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 was developed as part of project MoDECS at the Paris Lodron University of Salzburg (Austria). This report is not an introduction into the emerging field of time triggered control systems and model based development.

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

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 6 for a list of differences.

1.2 Acknowledgements

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

2 Lexical Structure

A *TDL* module is represented as an ASCII text. Sequences of characters form words, also called tokens, and the sequence of tokens forms the text. White space between tokens 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.

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.

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 digits, a string is a sequence of arbitrary characters enclosed in single or double quotes. The enclosing character must not occur inside the string. Character literals are strings of length one.

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

3 Syntactical Structure

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

`::=`	separates the non terminal symbol (left hand side) of a production from the right hand side.
`.`	terminates a production.
`|`	separates alternatives.
`[ ]`	encloses optional parts (zero or one).
`{ }`	encloses iterated parts (zero or more).
`( )`	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}
  "}" EOF.
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 input text, which means that EOF (end of file) must follow the module.

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 ouput 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.

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).

```
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 as opposed to the fully qualified module name is used. This allows and actually forces the usage of unqualified module names within a client module to refer to imported modules, which simplifies the program text.

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 simple imports. The group import specifies the qualifier followed by a set of module identifiers enclosed in braces. For every module an alias may be specified optionally.

Examples of valid import declarations (including the keyword `import`) are

```
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 state and output variables or for timing attributes.

```
constDecl ::= identifier "=" constExpr.
constExpr ::= ["-"] 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 a time value expressed in milliseconds or microseconds resp., where the latter is the default. Millisecond values will be converted to microseconds, i.e. they are multiplied by 1000.

3.4 Type Declaration

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

```
typeDecl ::= identifier ["=" typeDesignator].
typeDesignator ::= qualIdent.
```

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.

Sensors are typed variables which may be connected with the environment by using a so-called *getter* function. The getter is a parameterless 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.

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

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* program (see 3.8.2). Actuators may only be set within the module they are declared in.

Actuators must be initialized either with a constant value or with an external function, called an *initializer*. Initializers are, like getters, parameterless functions, which must return a value compatible with the actuator's type. It must be implemented according to the language binding rules and environment the program is executed in.

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

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

3.7 Task Declaration

A task declaration defines a task, which encapsulates a computation to be carried out by a control application. Tasks provide a namespace for the declaration of input, output and state ports. In addition, a task has an associated external procedure (including arguments), which performs the computation. The arguments of the external procedure call are taken exclusively from the task's ports and must be treated by the external procedure as value or reference parameters accordingly.

A task has a worst case execution time (wcet), which specifies the maximum time the computation needs. Optionally, this attribute may be explicitly named `wcet`. The amount of time is specified by a constant expression.

```
taskDecl ::= identifier wcet "{"
  {"input" {inPortDecl}}
  {"output" {portDecl}}
  {"state" {portDecl}}
  "uses" call ";"
  "}".
wcet ::= "[" [attrName "="] constExpr "]".
attrName ::= identifier.
inPortDecl ::= typeDesignator identifier ";".
portDecl ::= typeDesignator identifier [initExpr] ";".
call ::= extIdent "("[portDesignator {"," portDesignator }] ")".
portDesignator ::= qualident.
```

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 must be initialized either with a constant value or with an external function, called an *initializer*.

3.8 Mode Declaration

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

A *TDL* mode consists of a set of activities 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 period. The mode period must be divisible 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.

```
taskInvocation ::= frequency guard taskDesignator assignList.
frequency ::= "[" [attrName "="] constExpr "]".
guard ::= ["if" call "then"].
taskDesignator ::= qualIdent.
assignList ::= "{" {identifier ":=" portDesignator ";"} "}"
  | ["(" [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 fixed logical execution time (FLET) of the task has elapsed. The FLET for a task invocation with frequency f in a mode with period p is defined as p/f. In

[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.

case of using the output ports of a task by other activities before the task's FLET has elapsed, the previous values of the output ports are used. The intermediate values of output ports are never visible to other program entities.

Note that the sum of the worst case execution 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 identifier ":=" portDesignator ";".
```

3.8.3 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 assignList.
modeDesignator ::= qualident.
```

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 FLET 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 periode has elapsed.

4 Distribution

The FLET-based programming model of TDL provides for transparent distribution of modules across a network of processing nodes. A configuration file is used for describing the topology of the network and the assignments of modules to nodes. The configuration file is based on the conventions for Java property files and consists of a sequence of (key, value) pairs written as `key = value`, where white space surrounding the '`=`' character is not significant. Lines starting with '`#`' denote comment lines. The order of lines is not significant. The property file format for the configuration file is exemplified below. Indexed properties are used to express lists. The property name without an index specifies the number of list elements and the properties indexed from 0 to $length - 1$ specify the elements.

```
tdl.bus.nodes = <number of nodes>
tdl.bus.nodes.0 = <name>:<data>
tdl.bus.nodes.1 = <name>:<data>
...
#
tdl.bus.modules = <number of modules>
tdl.bus.modules.0 = <name>:<nodeID>
tdl.bus.modules.1 = <name>:<nodeID>
...
```

5 Language Bindings

Functionality required by a *TDL* program is provided as static (global) functions in a particular programming language. In principle, there is an open set of languages which may be used by an E-machine. The following subsections define the recommended conventiones for 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 opaque *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.Opaque` 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 `emcore.tools.emachine.types` for all primitive types. The naming convention is that for a primitive type `T` there exists a corresponding reference class named `ref_T`.

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

5.2 ANSI-C

External functionality code written in ANSI-C is 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 type definitions or comments can be filled in manually.

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

5.2.1 Functionality Code

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

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

5.2.2 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 '_'.

5.2.3 Type Mapping

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

It should be noted that the E-machine may copy variables. Thus, there must not be any reference to parameters. In C, arrays are always passed by reference, therefore array types must be enclosed by a structure in order to achieve the desired call-by-value semantics for input parameters.

TDL type	C name	*default C type*
byte	tdl_byte	unsigned 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
string	tdl_string	unsigned char[24]

5.2.4 Parameter Passing

Sensor getters and port initializers are parameterless functions that return their value as function result. Actuator setters are void functions that have exactly one input parameter. Guards are boolean functions with all parameters passed by value. The number and order of parameters of task implementation functions is exactly the same as in the *TDL* source code. Input ports are passed by value whereas state and output ports are passed by reference.

6 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.

7 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 tdlc;

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

IGNORE cr + lf + tab

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

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


PRODUCTIONS

tdlc = tdlModule EOF.

tdlModule =
  "module" qualIdent<ˆSem.modName> "{"
    {"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}.

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

importModule = identifier ["as" identifier].

constDecl = identifier "=" constExpr.
```

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

constExprBoolean = "true" | "false".

typeDecl = identifier ["=" typeDesignator].

sensorDecl = typeDesignator identifier
  ["uses" extIdent] [portAnnotation].

extIdent = qualIdent.

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

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

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

inPortDecl = typeDesignator identifier ";".

portDecl = typeDesignator identifier [initExpr] ";".

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

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

taskInvocation = frequency guard taskDesignator assignList.

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

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

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

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

modeSwitch = frequency guard modeDesignator assignList.

designator = identifier {"." identifier}.

/* renamed productions */
attrName = identifier.
unit = identifier.
constDesignator = designator.
```

```
typeDesignator = designator.
taskDesignator = designator.
portDesignator = designator.
modeDesignator = designator.

/* annotation currently used */

/* annotations currently ignored */
annotation = "[" {ANY} "]".
hardwareAnnotation = annotation.
portAnnotation = annotation.
taskTimingAnnotation = annotation.
modeAnnotation = annotation.
modeConnectionAnnotation = annotation.
modeSwitchAnnotation = annotation.
taskAnnotation = annotation.

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’ ’2’ moduleName:string moduleKey:int4
  0x80 Modules
  0x81 Constants
  0x82 Types
  0x83 Ports
  0x84 Tasks
  0x85 Drivers
  0x86 Guards
  0x87 Modes
  0x88 Ecodes.

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

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

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

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

Struct ::=
    opaque:0x0 moduleName:string typeName:string
  | byte:0x1
  | short:0x2
  | int:0x3
  | long:0x4
  | float:0x5
  | double:0x6
  | boolean:0x7
  | char:0x8
  | string:0x9.

Ports ::= nofPorts:int4
  {name:string pub:boolean Struct
    (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
    )
  }.

Init ::= 0x0 | 0x1 initializer:string driverID:int4 | 0x2 init:ConstVal.

Tasks ::= nofTasks:int4
  {name:string pub:boolean wcet:int4
   inputs:LocalPortList outputs:LocalPortList states:LocalPortList}.

LocalPortList ::= nofPorts {portID:int4}.

Drivers ::= nofDrivers:int4
  { set:0x0 portID:int4 setter:string
  | get:0x1 portID:int4 getter:string
  | actuator:0x2 srcPort:QualPortID actPortID:int4
  | release:0x3 srcPorts:PortList dstPorts:LocalPortList
  | terminate:0x4 taskID:int4
  | start:0x5 taskImpl:LocalFunCall taskID:int4
  | stop:0x6 taskID:int4
  | switch:0x7 srcPorts:PortList dstPorts:LocalPortList
  }.

QualPortID ::= moduleID:int4 portID:int4.

PortList ::= nofPorts {QualPortID}.

LocalFunCall ::= name:string args:PortList.

Guards ::= nofGuards:int4 {FunCall}.

FunCall ::= 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}
  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.

opcode	*mnemonic*	*arg1*	*arg2*	*arg3*
0x0	nop	-1	-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=02
  key=−579190198
IMPORTS
CONSTS
  public c1 = 0
  public c2 = 10
  public refPeriod = 100000
TYPES
PORTS
  [ 0] actuator int a1
  [ 1] actuator int a2
  [ 2] sensor int s
  [ 3] public output int o
  [ 4] public output int o
TASKS
  [ 0] public dec, wcet=20000, input, output 3, state
  [ 1] public inc, wcet=20000, input, output 4, state
DRIVERS
  [ 0] start, taskID=0, decImpl( 3)
  [ 1] stop, taskID = 0
  [ 2] terminate, taskID = 0
  [ 3] start, taskID=1, incImpl( 4)
  [ 4] stop, taskID = 1
  [ 5] terminate, taskID = 1
  [ 6] set, portID=0, uses=setA1
  [ 7] set, portID=1, uses=setA2
  [ 8] release
  [ 9] release
  [10] actuator, 0 := .4
  [11] actuator, 1 := .3
  [12] get, portID=2, uses=getS
  [13] switch
  [14] release
  [15] release
  [16] actuator, 1 := .3
  [17] actuator, 0 := .4
  [18] switch
GUARDS
  [ 0] switch2m2(.2 )
  [ 1] switch2m1(.2 )
MODES
  [ 0] name=m1, start=true, period=100000, pcBegin=3
        task: freq=1, guardID=−1, taskID=1, releaseDriverID=8
        task: freq=1, guardID=−1, taskID=0, releaseDriverID=9
        actuator: freq=1, guardID=−1, actuatorDriverID=10
        actuator: freq=1, guardID=−1, actuatorDriverID=11
        mode: freq=1, guardID=0, targetID=1, switchDriverID=13
  [ 1] name=m2, start=false, period=100000, pcBegin=20
        task: freq=1, guardID=−1, taskID=1, releaseDriverID=14
        task: freq=2, guardID=−1, taskID=0, releaseDriverID=15
        actuator: freq=1, guardID=−1, actuatorDriverID=17
        actuator: freq=2, guardID=−1, actuatorDriverID=16
        mode: freq=1, guardID=1, targetID=0, switchDriverID=18
  [ 2] name=<stub>, start=false, period=100000, pcBegin=44
ECODES
  [ 0] call 6 //actuator init: setA1(a1)
  [ 1] call 7 //actuator init: setA2(a2)
  [ 2] return
  [ 3] call 8 //release task: inc
  [ 4] schedule 1 //schedule: incImpl
  [ 5] call 9 //release task: dec
  [ 6] schedule 0 //schedule: decImpl
  [ 7] future 0, 9, 100000
  [ 8] return
  [ 9] call 5 //terminate task: inc
  [10] call 2 //terminate task: dec
  [11] call 10 //actuator update: a1 := o
  [12] call 6 //actuator setter: setA1(a1)
  [13] call 11 //actuator update: a2 := o
  [14] call 7 //actuator setter: setA2(a2)
  [15] call 12 //get: s := getS()
  [16] if 0, 17, 19 //mode switch guard: switch2m2
  [17] call 13 //mode switch driver
  [18] switch 1 //mode switch −> m2:0
  [19] jump 3 //next cycle: m1
  [20] call 14 //release task: inc
  [21] schedule 1 //schedule: incImpl
  [22] call 15 //release task: dec
  [23] schedule 0 //schedule: decImpl
  [24] future 0, 26, 50000
  [25] return
  [26] call 2 //terminate task: dec
  [27] call 16 //actuator update: a2 := o
  [28] call 7 //actuator setter: setA2(a2)
  [29] call 15 //release task: dec
  [30] schedule 0 //schedule: decImpl
  [31] future 0, 33, 50000
  [32] return
  [33] call 5 //terminate task: inc
  [34] call 2 //terminate task: dec
  [35] call 17 //actuator update: a1 := o
  [36] call 6 //actuator setter: setA1(a1)
  [37] call 16 //actuator update: a2 := o
  [38] call 7 //actuator setter: setA2(a2)
  [39] call 12 //get: s := getS()
  [40] if 1, 41, 43 //mode switch guard: switch2m1
  [41] call 18 //mode switch driver
  [42] switch 0 //mode switch −> m1:0
  [43] jump 20 //next cycle: m2
  [44] future 0, 46, 100000
  [45] return
  [46] call 5 //terminate task: inc
  [47] call 2 //terminate task: dec
  [48] jump 44 //next cycle: <stub>
}
```

```
MODULE M2 {
  version=02
  key=−1477044379
IMPORTS
  [ 0] moduleName=M1, key=−579190198
CONSTS
TYPES
PORTS
  [ 0] actuator int a
  [ 1] input int i1
  [ 2] input int i2
  [ 3] public output int o
TASKS
  [ 0] public sum, wcet=20000, input 1 2, output 3, state
DRIVERS
  [ 0] start, taskID=0, sumImpl( 1 2 3)
  [ 1] stop, taskID = 0
  [ 2] terminate, taskID = 0
  [ 3] set, portID=0, uses=setA
  [ 4] release 1:=0.4 2:=0.3
  [ 5] actuator, 0 := .3
GUARDS
MODES
  [ 0] name=main, start=true, period=100000, pcBegin=2
        task: freq=1, guardID=−1, taskID=0, releaseDriverID=4
        actuator: freq=1, guardID=−1, actuatorDriverID=5
  [ 1] name=<stub>, start=false, period=100000, pcBegin=10
ECODES
  [ 0] call 3 //actuator init: setA(a)
  [ 1] return
  [ 2] call 4 //release task: sum
  [ 3] schedule 0 //schedule: sumImpl
  [ 4] future 0, 6, 100000
  [ 5] return
  [ 6] call 2 //terminate task: sum
  [ 7] call 5 //actuator update: a := o
  [ 8] call 3 //actuator setter: setA(a)
  [ 9] jump 2 //next cycle: main
  [10] future 0, 12, 100000
  [11] return
  [12] call 2 //terminate task: sum
  [13] jump 10 //next cycle: <stub>
}
```

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.

```
import emcore.tools.emachine.types.ref_int;
import emcore.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 emcore.tools.emachine.types.ref_int;
import emcore.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, the Java based E-machine would 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. The parts related to distribution (everything named *stub*) should be considered as work in progress.

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

/∗∗
 ∗ This class has been generated automatically by tdlc −java on
 ∗ Wed Nov 17 12:07:17 CET 2004 from 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 emcore.tools.emachine.ModuleBase {
```

```
private static emcore.tools.emachine.Module module$;

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

private static class Drivers$ implements emcore.tools.emachine.Drivers {
  public void call(int id) throws Exception {
    int _ticks;
    switch (id) {
      case 0: //start task dec
        M1.decImpl(port$3$out);
        break;
      case 1: //stop task dec
        if (module$.usage == emcore.tools.emachine.Module.USAGE_PUSH) {
          emcore.tools.emachine.bus.BusController.setInt(module$.itemIDs[0][0], port$3$out.val);
        }
        break;
      case 2: //terminate task dec
        port$3 = port$3$out.val;
        break;
      case 3: //start task inc
        M1.incImpl(port$4$out);
        break;
      case 4: //stop task inc
        if (module$.usage == emcore.tools.emachine.Module.USAGE_PUSH) {
          emcore.tools.emachine.bus.BusController.setInt(module$.itemIDs[1][0], port$4$out.val);
        }
        break;
      case 5: //terminate task inc
        port$4 = port$4$out.val;
        break;
      case 6: //set a1
        M1.setA1(port$0);
        break;
      case 7: //set a2
        M1.setA2(port$1);
        break;
      case 8: //release task inc
        break;
      case 9: //release task dec
        break;
      case 10: //actuator update a1
        port$0 = port$4;
        break;
      case 11: //actuator update a2
        port$1 = port$3;
        break;
      case 12: //get s
        _ticks = (int)emcore.tools.emachine.Interpreter.ticks;
        if (port$2_tick != _ticks) {
          port$2 = M1.getS();
          port$2_tick = _ticks;
        }
        break;
      case 13: //mode switch to m2
        break;
      case 14: //release task inc
        break;
      case 15: //release task dec
        break;
      case 16: //actuator update a2
        port$1 = port$3;
        break;
      case 17: //actuator update a1
        port$0 = port$4;
        break;
      case 18: //mode switch to m1
        break;
      default: throw new IllegalArgumentException("invalid id:" + id);
    }
  }
}

private static class StubDrivers$ implements emcore.tools.emachine.Drivers {
  public void call(int id) throws Exception {
    int _ticks;
    switch (id) {
      case 0: //start task dec
        break;
      case 1: //stop task dec
        break;
      case 2: //terminate task dec
          port$3 = emcore.tools.emachine.bus.BusController.getInt(module$.itemIDs[0][0]);
        break;
      case 3: //start task inc
        break;
      case 4: //stop task inc
        break;
      case 5: //terminate task inc
          port$4 = emcore.tools.emachine.bus.BusController.getInt(module$.itemIDs[1][0]);
        break;
      case 6: //set a1
        break;
      case 7: //set a2
        break;
      case 8: //release task inc
        break;
      case 9: //release task dec
```

```
            break;
          case 10: //actuator update a1
            break;
          case 11: //actuator update a2
            break;
          case 12: //get s
            break;
          case 13: //mode switch to m2
            break;
          case 14: //release task inc
            break;
          case 15: //release task dec
            break;
          case 16: //actuator update a2
            break;
          case 17: //actuator update a1
            break;
          case 18: //mode switch to m1
            break;
          default: throw new IllegalArgumentException("invalid id:" + id);
        }
      }
    }

    private static class Guards$ implements emcore.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);
        }
      }
    }

    //implement ModuleBase
    public void init(emcore.tools.emachine.Module m) {module$ = m;}
    public int getKey() {return −579190198;}
    public emcore.tools.emachine.Drivers getDrivers() {
      if (module$.usage == emcore.tools.emachine.Module.USAGE_STUB) {
        return new StubDrivers$();
      } else {
        return new Drivers$();
      }
    }
    public emcore.tools.emachine.Guards getGuards() {return new Guards$();}
}
```

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

/∗∗
 ∗ This class has been generated automatically by tdlc −java on
 ∗ Wed Nov 17 11:11:19 CET 2004 from 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 emcore.tools.emachine.ModuleBase {

  private static emcore.tools.emachine.Module module$;

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

  private static class Drivers$ implements emcore.tools.emachine.Drivers {
    public void call(int id) throws Exception {
      int _ticks;
      switch (id) {
        case 0: //start task sum
          M2.sumImpl(port$1, port$2, port$3$out);
          break;
        case 1: //stop task sum
          if (module$.usage == emcore.tools.emachine.Module.USAGE_PUSH) {
            emcore.tools.emachine.bus.BusController.setInt(module$.itemIDs[0][0], port$3$out.val);
          }
          break;
        case 2: //terminate task sum
          port$3 = port$3$out.val;
          break;
        case 3: //set a
          M2.setA(port$0);
          break;
        case 4: //release task sum
          port$1 = M1$.port$4;
          port$2 = M1$.port$3;
          break;
        case 5: //actuator update a
          port$0 = port$3;
          break;
        default: throw new IllegalArgumentException("invalid id:" + id);
      }
    }
  }

  private static class StubDrivers$ implements emcore.tools.emachine.Drivers {
    public void call(int id) throws Exception {
      int _ticks;
      switch (id) {
        case 0: //start task sum
          break;
        case 1: //stop task sum
          break;
        case 2: //terminate task sum
            port$3 = emcore.tools.emachine.bus.BusController.getInt(module$.itemIDs[0][0]);
          break;
        case 3: //set a
          break;
        case 4: //release task sum
          break;
        case 5: //actuator update a
          break;
        default: throw new IllegalArgumentException("invalid␣id:" + id);
      }
    }
  }

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

  //implement ModuleBase
  public void init(emcore.tools.emachine.Module m) {module$ = m;}
  public int getKey() {return −1477044379;}
  public emcore.tools.emachine.Drivers getDrivers() {
    if (module$.usage == emcore.tools.emachine.Module.USAGE_STUB) {
      return new StubDrivers$();
    } else {
      return new Drivers$();
    }
  }
  public emcore.tools.emachine.Guards getGuards() {return new Guards$();}
}
```