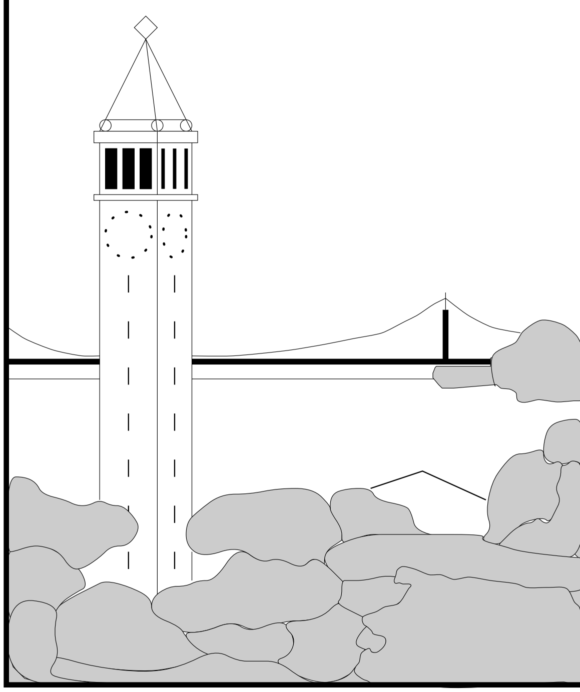
xGiotto Language Report

Marco A.A. Sanvido, Arkadeb Ghosal, and Thomas A. Henzinger



Report No. UCB/CSD-3-1261

July 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

xGiotto Language Report[1]

Marco A.A. Sanvido, Arkadeb Ghosal, and Thomas A. Henzinger
University of California at Berkeley
Berkeley, 94704, CA
{msanvido,arkadeb,tah}@eecs.berkeley.edu

June, 2003

[1]Revision: 0.2

Chapter 1

Introduction

xGiotto is an e*X*tension of the programming language Giotto [1]. Giotto was developed essentially for embedded systems with a periodical control, as example for implementing an autopilot system for a model helicopter [4]. In particular Giotto focused on the strict separation of timing and functionality. The timing allowed to simplify the temporal behavior analysis, necessary to prove system safety and reactivity [3]. Discerning the timing and functionality, exposed the functionality and made it more tractable and analyzable in respect of execution times analysis.

xGiotto builds on top of Giotto in the sense that the strict separation of timing and functionality is respected and even made more clear at the syntactical level. In Giotto the functionality was explicitly external to the Giotto program, as opposed by xGiotto where the functionality is expressed in the programm itself. This makes an xGiotto program self contained.

Another key aspect made more clear in xGiotto is the *fixed logical execution time* of a task. The logical execution time of a task is specified in the program and therefore made independent from the executing platform. This makes the timing behavior of a xGiotto program platform-independent and deterministic. Therefore, we shift the role of guaranteeing the timing behavior of a program from the programmer to the compiler. The compiler will map the program to a given platform only if the platform will guarantee its logical execution time.

Moreover, in Giotto tasks were single periodical entities, with fixed intertask connections. Each task was executed periodically, copying its input parameters and reading its generated output parameters at a formally defined point in time. In xGiotto this is more flexible, introducing the ability to start sporadic, asynchronous, and multiple instances of a tasks. The fixed logical execution time concept is therefore preserved, but made more independent from the periodical execution of xGiotto .

This report is not intended as a programmer's tutorial, and it is intentionally kept concise. Its function is to serve as a reference for programmers, and implementors.

Chapter 2

Vocabulary

- **Identifiers** are sequences of letters and digits. The first character must be a letter.

  ```
  ident  = letter {letter | digit}.
  ```

- **Numbers** are integers or real numbers. Integers are sequences of digits. A real number always contains a decimal point.

  ```
  number = digit {digit} ["." {digit}].
  ```

- **Operators** and **delimiters** are the special characters, character pairs, or reserved words listed below. These reserved words consist exclusively of capital letters and cannot be used in the role of identifiers.

  ```
  +  =   ARRAY     AT     COMPLETION
  -  ==  RECORD    ELSE   ^
  *  <   TYPE      TASK   |
  /  <=  WHENEVER  EVENT  &
  .  >   UNTIL     OF     THEN
  ;  >=  PORT      END    OUTPUT
  {  }   TIMING    WHEN   VAR
  (  )   [         ]      !
  ```

- **Comments** may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket `/*` and closed by `*/`. Comments do not affect the meaning of a program and can be nested.

Chapter 3

Declarations and Scope Rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a predefined identifier. Declarations also serve to specify certain permanent properties of an object, such as whether it is a type, a port, an event, a task, or a timing.

3.1 Type Declarations

A type determines the set of values that ports and events of that type can assume. A type declaration is used to bind an identifier to a type. There are three predefined types: `INTEGER`, `REAL`, `BOOLEAN`. A type can be constructed by combining basic types and with two structured types: `RECORD` and `ARRAY`. A record is an inhomogeneous set of values, whereas an array is a homogeneous vector with a fixed number of elements of the same type.

```
TypeDecl    = "TYPE" {Ident (ArrayDecl | RecordDecl)}.
ArrayDecl   = "ARRAY" Number "OF" TypeId.
RecordDecl = "RECORD" '{' {TypeId Ident ';' }'}'.
```

3.1.1 Predefined types

1. `INTEGER` are one-complement 32 bits wide values.
2. `REAL` are 32 bits wide floating point numbers.
3. `BOOLEAN` can have values `TRUE` and `FALSE`.

3.1.2 Structured Types

1. A `RECORD` is an inhomogeneous set of values, the number of elements are fixed with possibly different types. The record type declaration specifies

for each element, called a field, its type and an identifier that denotes the field. The scope of these field identifiers is the record definition itself.

2. An `ARRAY` is a homogeneous vector with a fixed number of elements of the same type. The array consists of a fixed number of elements called its length. The elements of the array are designated by indices, which are integers between 0 and the length of the array minus 1.

3.2 Port Declaration

Port declarations serve to introduce ports and associate them with identifiers that must be unique. They also serve to associate fixed data types with the ports. The ports scope is global for reading, i.e. ports can be read in all timing blocks, but ports can be written only at specific events (called termination event, see 4.3).

```
PortDecl = "PORT" {TypeId Ident [InitPort] ';'}.
```

Ports can be initialized with an optional predefined value. The value will be assigned to the port during program initialization. If no initial value is specified, the port will be initialized to its default value (e.g. 0 for `INTEGER` and `FALSE` for `BOOLEAN`). The format of the initialization must be compatible with the type structure of the port.

```
InitPort = '=' InitPortVal.
InitPortVal = number | "{" InitPortVal {"," InitPortVal }"}".
```

3.3 Event Declaration

Event declarations serve to introduce events and declare them as: *external*, *completion*, or *composed*. Event identifiers must be unique. The external event `TIME` is predefined and bound to the system clock. A `TIME` event is triggered every 1ms, if not specified otherwise.

```
EventDecl = "EVENT"
          {TypeId Ident
          ["AT" Ident | "WHEN" EventExpression ] ';'
          }.
```

3.3.1 External Events

External events are generated externally by interrupts. The binding is performed in a system dependent manner and specified in the program with `AT interrupt_name`.

3.3.2 Completion Events

Completion events are generated internally by the program. They are similar to software interrupts. To generate a completion event the program must bind it to a specific task at run-time. When the task terminates its computation raises the completion event. Note that is a platform dependent event.

3.3.3 Composed Events

Composed events are generated by evaluating event expressions. The evaluation of such expressions is executed by the arrival of events of the expression. If the event expression is `TRUE` then the event is raised.
Event expression are concatenation of OR-sequences or AND-sequences. An OR-sequence expression is evaluated to `TRUE` if an event of the concatenation is raised, and an AND-sequence expression is evaluated to `TRUE` if all the events of the concatenation were raised at some point in the past. Once the composed event is raised, the event expression is reset.

```
EventExpression = ETerm {"|" ETerm}.
ETerm = EFactor {"&" EFactor}.
EFactor = Ident | '(' EventExpression ')'.
```

3.4 Task Declaration

Task declarations consist of a task heading and a task body. The heading specifies the task identifier, the formal input parameters, the formal output parameters, and if any, the formal completion event, and local variables. The body contains declarations and statements and is enclosed by brackets.
All variables declared within a task body are local to the task. The values of local variables are set to their default (e.g. 0 for `INTEGER` and `FALSE` for `BOOLEAN`) upon entry to the task.

```
TaskDecl = "TASK" Ident
           FPars "OUTPUT" FPars
           ["VAR" FPars] ["COMPLETION" EFPar]
           Body.

Body = '{' StatSeq '}'.
StatSeq = Stat {";" Stat}.
```

3.4.1 Formal Parameters

Formal parameters are identifiers which denote actual parameters specified in the task call. The correspondence between formal and actual parameters is established when the task is activated (see 4.3). The input parameters are passed by value, i.e. are local ports to which the result of the evaluation of

the corresponding actual parameter is assigned as initial value. The output parameters are passed by value-reference, i.e. are local ports with the actual parameter as initial value, but their value is instantaneously copied back to the actual parameter at task termination (see 4.3).
The VAR formal parameter declaration is used to declare local variables. The formal event parameter is the completion event, which will be raised when the task computation is completed.

```
FPars = '(' [TypeId Ident {',' TypeId Ident }] ')'.
EFPar = '[' TypeId Ident ']'.
```

All formal parameters (input, output, event and local) are local to the task, i.e. their scope is the task body of the task declaration only.
The task body will be explained later in section 4.2.

3.4.2 Examples of Task Declarations:

```
TASK ReadInt(INTEGER x0; INTEGER x1) OUTPUT (INTEGER y)
                        VAR (INTEGER i) {
   i = 10*x0 + x1;
   y = i+10;
}
```

```
TASK Filter(REAL x; BOOLEAN on) OUTPUT (REAL y) COMPLETION
[INTEGER e]{
   IF (on) {
      y = 0.5*y + 0.5*x;
   } ELSE {
      y = x;
   }
   e = e + 1;
}
```

3.5 Timing Declaration

A timing is executed in logical zero time, i.e. no relevant computation is performed by the system and therefore no relevant execution time is necessary. A timing describes only the reaction of the system. The computation burden is released to the tasks.
The heading specifies the timing identifier. The body contains declarations and statements and is enclosed by brackets. In the timing body only timing statements are allowed (see 4.3).

```
TimingDecl = "TIMING" Ident '(' ')' TBody.
TBody =      '{'TStatSeq '}'.
TStatSeq =   TStat {";" TStat}.
```

3.5.1 Example of Timing Declaration:

```
TIMING Button() {
   [A]Coin();
   [B]Coin();
   Press(on)(k)[K];
}
```

Chapter 4

Statements

4.1 Expressions

Expressions denote rules of computation and whereby constants and current port values are combined to derive other values by the application of operators. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

4.1.1 Operands

Operands are denoted by designators, with the exception of number constants. A designator consists of an identifier referring to the port, and possibly followed by selectors, if the designated object is an element of a structure (i.e. array or record).
If a designates an array, then $a[e]$ denotes that element of a whose index is the current value of the expression e. The type of e must be an integer type. If r designates a record, $r.f$ denotes the field f of the record p.

```
Designator =  Ident ['.' Designator |  '[' Expression ']' ].
```

If the designated object is a port e, then the designator refers to the current value of the port.

4.1.2 Operators

Operands are composed into expressions by using operators. The binding of the operators is strictly defined by the EBNF definition.

```
Expression = SimExpr [RelOp SimExpr ].
SimExpr =    Term { AddOp Term }.
Term =       Factor { MulOp Factor }.
Factor =     Designator
            | "TRUE"
```

```
         | "FALSE"
         | Number
         | '-' Factor
         | '!' Factor
         | '(' Expression ')'.
```

The following tables list all available operators.

Logical operators:

symbol	result
|	logical disjunction
&	logical conjunction
!	logical negation

Arithmetic operators:

symbol	result
$+$	sum
$-$	difference
$*$	product
/	division

The operators $+$, $-$, $*$, and / apply to operands of numeric types. The type of the result is that operand's type which includes the other operand's type, except for division (/), where the result is the real type which includes both operand types. When used as operators with a single operand, $-$ denotes sign inversion and $+$ denotes the identity operation.

Relations:

symbol	relation
$==$	equal
$!=$	unequal
$<$	less
$<=$	less or equal
$>$	greater
$>=$	greater or equal

4.1.3 Examples of Expressions

1987	(INTEGER)
i/3	(REAL)
p | q	(BOOLEAN)
(i+j) * (i-j)	(INTEGER)
i + x	(REAL)
a[i+j] * a[i-j]	(REAL)
(0<=i) & (i<100)	(BOOLEAN)
t.key == 0	(BOOLEAN)

4.2 Task Statements

The body of a task defines the computation the task will perform in fixed logical time. In a task there are three different type of computations which can be made only on the local ports of the task (i.e. a task is side-effect free). The results of the computation stored in the local output port will be copied to the global port during the task termination (see 4.3).

```
Stat = [ Assignment | IfStatement | WhileStatement].
```

4.2.1 Assignments

The result of the evaluation of the expression replaces the actual value of the operand denoted by the designator.

```
Assigment = Designator "=" Expression.
```

4.2.2 If

If the `BOOLEAN` expression is evaluated to `TRUE` the first statement sequence is executed otherwise –if present– the statement sequence in the `ELSE` block is be executed.

```
IfStatement = "IF" '(' Expression ')' '{' StatSeq '}'
              ["ELSE" '{' StatSeq '}'].
```

4.2.3 While

The while statement will execute the statements in its block until the result of the evaluation of the `BOOLEAN` expression is evaluated to `FALSE`.

```
WhileStatement = "WHILE" '(' Expression ')' '{' StatSeq '}'.
```

4.3 Timing Statements

The body of a timing declaration defines the reaction of the system. The timing body will be executed in logical zero-time, i.e. the computation time needed to execute the statements of the timing body can be neglected. The timing body does not perform any operations, with the only exception of the evaluation of the expression of the if-statements. Ports values can only be read.
A timing statement can invoke a task (i.e. `TaskActivation`), can declare how to react to a following (i.e. `ReactionStatement`), and can conditionally execute timing statements (i.e. `IfStatement`). Timing statements can be repeated at the occurrence of an event (i.e. `WheneverStatement`).

```
TStat =  [ TaskActivation | TimingActivation
         | IfStatement | WheneverStatement ].
```

4.3.1 Event Operands

All timing statements refer to an event operand, i.e. when the event rises one or more timing statements will be evaluated in logical zero time. Event operands can also count event, i.e. the timing statement will only be executed when the number of specified events has been fired. The name of the event can be left out, as an abbreviation for the predefined event `TIME`.

```
Event = '[' [Number] [Ident] ']'.
```

Examples of Event Operands:

$[A]$	reference to event A
$[10A]$	reference to 10 times event A
[10]	10 times the event `TIME`
[]	in the next time instance

4.3.2 Task Activation and Termination

When a task is activated, the task will start performing the computation on its input, output and local ports. The computation will take a fixed logical execution time, which is specified by the termination event. When the termination event is raised, the actual output ports will be instantaneously updated and the memory used by task destroyed, and recollected.
The task will raise a completion event when the real execution time terminates.

```
TaskActivation = Ident AParsValue AParsRef Event [ComplEvent].
```

4.3.3 Timing Activation

The timing activation statements register in the system which timing block has to be executed upon raising of an determined event.

```
TimingActivation = Event Ident '(' ')'.
```

4.3.4 Whenever and Do

The `WHENEVER` and `DO` statements declare a recurrent timing block execution on a given event of a timing statement sequence until another event is raised. The `DO` timing statement sequence is performed at most once, whereas the `WHENEVER` timing statements sequence is performed only after the first event is raised. If the `UNTIL` event is specified the recurrent timing block execution is terminated at the raise of the until event.

```
WheneverStatement = "WHENEVER" Event ["UNTIL" Event ]
                    '{' TStatSeq '}'.
DoStatement       = "DO" '{' TStatSeq '}' 'WHENEVER' Event
                    ["UNTIL" Event].
```

Although WHENEVER and DO statements can be replaced by timing containing cyclic timing activations, the use of WHENEVER and DO statements are recommended in the most frequently occurring situations, because readability.

4.3.5 If

The IF statements specify the conditional execution of timing statements. The expression preceding a statement is evaluated. If the expression evaluates to TRUE the associated timing statement sequence is executed. If the expression evaluates to FALSE the timing statement sequence following the symbol ELSE is executed, if there is one. Events can also be tested as ^eventname.

```
IfStatement = "IF" Expression '{' TStatSeq '}'
              ["ELSE" '{' TStatSeq '}' ].
```

Chapter 5

Program Semantics

Arrival of an event causes a particular effect. An event combined with the effect it causes is called a *reaction*. Reactions take logical zero time for executions i.e. the effect of arrival of an event is executed instantaneously by the system. A reaction can be invocation of a block of code, deactivation of another reaction or termination of a task and is referred as *timing reaction*, *deactivation reaction*, or *termination reaction* respectively. A reaction consists of a tuple $(\varepsilon, \alpha, \eta)$ where $\varepsilon \in E$ and α implies the desired action when ε occurs for η times. The default value for η is 1.

- For a timing reaction α is an address of timing code and implies that when η repetitions of ε occurs the timing code at the address denoted by α is invoked.
- For a deactivation reaction α is a timing reaction. This implies that when ε occurs for η times the timing reaction(denoted by α) has to be removed. For example for a `WHENEVER` $[\varepsilon_A]$ `UNTIL` $[\varepsilon_B]$ $\{$*program block* $\mathbf{p}\}$ statement the deactivation reaction is
$(\varepsilon_B, (event_A, \text{start address of } \mathbf{p}))$.
- For a termination reaction α is a tuple (p_o, t, s_{p_i}) which implies that when event ε occurs the ports p_o are updated with the evaluation of task t on s_{p_i}(the valuation of the ports p_i at the instance of task invocation).

The execution of a xGiotto program yields a possibly infinite sequence of program *configurations*, called *trace*. Each configuration tracks the value of the ports and the status of the reaction sets. Formally a (*program*) configuration c is a tuple $(s, \Upsilon, \Psi, \Omega)$ where s is a valuation of ports called *port state*, Υ is timing reaction set, Ψ is deactivation reaction set and Ω is termination reaction set. The *port state* s is defined as $\vartheta(P)$ where ϑ is an evaluation function for the port set, P. Every reaction in its respective set can be identified by an index number. Thus the i^{th} reaction in Υ is $\Upsilon(i)$. The associated event, action and counter is $\Upsilon(i)\varepsilon$, $\Upsilon(i)\alpha$ and $\Upsilon(i)\eta$.

The configuration c is initial if the *port state* is initial, the timing reaction set consists of the starting reaction ($\Upsilon = (start, program)$), the deactivation reaction set is empty ($\Psi = \phi$) and the termination reaction set is empty ($\Omega = \phi$). The initial*port state* consists of initial port values as discussed in section 3.2. The *start* event is common to all xGiotto programs and on its arrival the program starts executing. Consider a configuration $c = (s, \Upsilon, \Psi, \Omega)$ at some instant. If an event ε', arrives such that

$$\begin{aligned}(\exists i.1 \leq i \leq n_1 \wedge \Upsilon(i)\varepsilon &= \varepsilon') \vee \\ (\exists j.1 \leq j \leq n_2 \wedge \Psi(j)\varepsilon &= \varepsilon') \vee \\ (\exists k.1 \leq k \leq n_3 \wedge \Omega(k)\varepsilon &= \varepsilon') \end{aligned} \tag{5.1}$$

where $|\Upsilon| = n_1$, $|\Psi| = n_2$, and $|\Omega| = n_3$ respectively, then c is updated to a new configuration *succ(c)* $= (s', \Upsilon', \Psi', \Omega')$ as follows:

1. If $\exists k.1 \leq k \leq n_3 \wedge \Omega(k)\varepsilon = \varepsilon'$:
   $\Omega(k)\eta = \Omega(k)\eta - 1$. If $\Omega(k)\eta = 0$ then $\Omega' = \Omega - \Omega(k)$ and $\Omega(k)\alpha.p_o$ is updated with the evaluation of $\Omega(k)\alpha.t$ on $\Omega(k)\alpha.s_{p_i}$.

2. If $\exists j.1 \leq j \leq n_2 \wedge \Psi(j)\varepsilon = \varepsilon'$:
   $\Psi(j)\eta = \Psi(j)\eta - 1$. If $\Psi(j)\eta = 0$ then $\Psi' = \Psi - \Psi(j)$ and $\Upsilon' = \Upsilon - \{\text{timing reaction}\Psi(j)\alpha\}$

3. If $\exists i.1 \leq i \leq n_1 \wedge \Upsilon(i)\varepsilon = \varepsilon'$:
   port state remains same and $\Upsilon(i)\eta = \Upsilon(i)\eta - 1$. If $\Upsilon(i)\eta = 0$ then $\Upsilon' = \Upsilon - \Upsilon(i)$ and the reactions sets are updated according to the xGiotto program block at the address given by $\Upsilon(i)\alpha$ as follows:

   - for every task invocation $t(p_i)(p_o)[\eta\varepsilon'']$:
     $\Omega' = \Omega \cup (\varepsilon'', \alpha, \eta)$ where α is the tuple (p_o, t, s_{p_i}) where s_{p_i} is the value of p_i at that instant.
   - for every timing call $[\eta\varepsilon'']\xi()$:
     $\Upsilon' = \Upsilon' \cup (\varepsilon'', \text{address of } \xi, \eta)$
   - for every `WHENEVER` $[\eta\varepsilon'']$ $\{$*program block* **p** $\}$:
     $\Upsilon' = \Upsilon' \cup (\varepsilon'', \text{start address of } \mathbf{p}, \eta)$
   - for every `WHENEVER` $[\eta_A\varepsilon_A]$ `UNTIL` $[\eta_B\varepsilon_B]$ $\{$*program block* **p** $\}$:
     $\Upsilon' = \Upsilon' \cup \Upsilon_{new}$ and $\Psi' = \Psi \cup (\varepsilon_B, \Upsilon_{new}, \eta_B)$
     where $\Upsilon_{new} = (\varepsilon_A, \text{start address of } \mathbf{p}, \eta_A)$
   - for every `IF` (port expression) $\{$*program block* **p**_1 $\}$ `ELSE` $\{$*program block* **p**_2 $\}$:
     There are two possible successors: *succ(c)*_1 and *succ(c)*_2 containing the information for **p**_1 and **p**_2 respectively. Depending on whether the evaluation of data expression is evaluated to `TRUE` or `FALSE`, *succ(c)* is assigned the information of *succ(c)*_1 or *succ(c)*_2 respectively.

- for every `IF` (ε'') {*program block* **p**_1 } `ELSE` {*program block* **p**_2}:
  There are two successors *succ(c)*_1 and *succ(c)*_2 due to the possibility of the events ε'' being present or absent. Any other informations that has been previously added goes to both of the successors. For *succ(c)*_1: $\Upsilon' = \Upsilon' \cup \{(\varepsilon'', \text{start address of } \mathbf{p}\_1)\}$ and for *succ(c)*_2: $\Upsilon' = \Upsilon' \cup \{(!\varepsilon'', \text{start address of } \mathbf{p}\_2)\}$.

A reaction is not repeated in its respective reaction set. The reactions are executed in sequential order. The termination reactions are executed followed by the deactivation reactions and then the timing reactions.

Chapter 6

Compiler

The compiler is divided in two parts, the front-end which parses the xGiotto program, and three back-ends which generates E code, F code, and memory layout description. The parser is a one-pass recursive-descendant parser for the LL1 grammar of xGiotto . E code [2] is the instruction set used by the Embedded Machine and is used for implementing the reactive part of the program. F code is a instruction set of a simple stack based machine for implementing the functionality code. The memory layout description consists of addresses, sizes, and system bindings for all ports and events.
The three back ends are explained in the next sections. The compiler is written in Java. The prototype implementation provides also an emulator which is able to execute the compiled xGiotto program.

6.1 Reactivity Code Generation

For each timing block the compiler generates E code and necessary F code to glue the activation and termination phases of task invocations. This is illustrated in the following example.

Example of `TIMING` Compilation:
The timing block `T1` invokes a task `P1` and reactivates itself at the arrival of the event `A`. The termination of the invoked task is the same event `A`.

```
TIMING T1() {
  P1(p)(q)[A];
  [A]T1();
}
```

The corresponding E code and F code for `T1` are as follows:

```
E code:                F code:
(0)  CALL     7        [0]  ALLOC   8,4
(1)  CALL     9        [1]  COPYR   4,4
```

```
(2)   CALL      0          [2]   COPYS    4,0
(3)   SCHEDULE  4          [3]   RETURN
(4)   FUTURETE  6,4,1      [4]   RETURNE  0
(5)   JUMP      8          [5]   DELETE   8,4
(6)   CALL      5          [6]   RETURN
(7)   RETURN               [7]   LOADP    0,4
(8)   FUTURE    0,4,1      [8]   RETURN
(9)   RETURN               [9]   PUSH     4
                           [10]  RETURN
```

E code instruction at (0) and at (1) call the F code at [7] and [9] respectively. This loads the value of the port `p` and the address of port `q` on the stack. Thereafter the call instruction at (2) calls the F code at [0]. The F code at [0] generates the task context, i.e. allocates the task ports (input, output and local) and the task stack. In particular the first argument of the `ALLOC` instruction specifies that 8 bytes are needed for the ports (`p` and `q` are `INTEGER` of 4 bytes each). The second argument specifies that 4 bytes are needed in order to store the address of the output port `q`. The termination reaction needs this address in order to update the global `q` with the local copy of `q`. The following instructions at [1] and [2] copy the values from the stack to the task context. `COPYR` copies the address of output port from the stack to the task context, and `COPYS` copies the value of the input port to the task context. At this point the the activation of the task is completed.

E code instruction at (3) then schedules the task execution. Its argument is the task entry point (i.e. address [4]). In this example the task code is empty and therefore there is just one instruction at [4], a return from the task. This is a special return instruction, which generates an event at completion of the task execution.

E code instruction `FUTURETE` at (4) activates the termination reaction for the task invocation. The first argument (i.e. (6)) is the E code address to be executed, the second argument (i.e. (4)) is the address of the event `A` and the last argument is the number of event occurrences, in this case 1. E code instruction at (5) jumps to instruction at (8), which activates the timing reaction for the timing block `T1`. The first argument (i.e. (0)) is the E code address to be executed, the second argument is the event address and the third argument is the number of occurrences. E code instruction at (9) returns from the E machine interpreter.

At this point two reactions are active in the system: one for handling the termination reaction, and the other one for handling the timing reaction. The latter starts to execute the E code at address (0) at the arrival of event `A`.

The termination reaction starts the termination phase of the task, i.e. it will copy the output port `q` to the global space and deallocate the resources used by the task. In order to deallocate the task context the `DELETE` instruction at [5] gets the context address via the stack. The special future instruction `FUTURETE`, does that. It stores the context of the task and copy it on the stack before executing the reaction.

6.2 Functionality Code Generation

The generation of the F code is illustrated by the following example. The glue code for the task invocation is not shown explicitly.

Example of `TASK` Compilation

```
TASK FACT(INTEGER a) OUTPUT (INTEGER b) {
      b = 1;
      WHILE (a > 0) {
         b = b*a;
         a = a-1;
      }
   }
```

The F code generated for the task `FACT`, which computes the factorial of the input parameter is:

```
4   PUSH    1,0
5   STOREP  4,4 b = 1
6   LOADP   0,4
7   PUSH    0,0
8   GTR     0,0 a > 0
9   NOT     0,0
10  CJUMP   20,0
11  LOADP   4,4
12  LOADP   0,4
13  MULT    0,0
14  STOREP  4,4 b = b*a
15  LOADP   0,4
16  PUSH    1,0
17  SUB     0,0
18  STOREP  0,4 a = a - 1
19  JUMP    6,0
20  RETURNE 0,0
```

The F code is a simple instruction set for a single stack machine. All partial computations are stored in the stack. Ports values are loaded or stored via their addresses in the local context of the task. During task activation all input and output ports are called-by-value, and local ports are set to their defaults. During task termination the values of the output ports are copied back to the global ports.

6.2.1 F code Instructions

In the following subsection all the F code instructions are listed (opcode name and id). The following notation is used to denote that two operands `value1` and `value2` are popped from the task stack and the value `result` is pushed back the stack: `..., value1, value2 => result`

- ADD[0]: Addition of two `INTEGER`
  `..., value1, value2 => ..., value1+value2`

- SUB[1]: Subtraction of two `INTEGER`
  `..., value1, value2 => ..., value1-value2`

- MULT[2]: Multiplication of two `INTEGER`
  `..., value1, value2 => ..., value1*value2`

- DIV[3]: Division of two `INTEGER`
  `..., value1, value2 => ..., value1/value2`

- INV[4]: Inversion of an `INTEGER`
  `..., value1 => ..., -value1`

- NOT[5]: Inversion of an `BOOLEAN`
  `..., value1 => ..., !value1`

- ADR[6]: Addition of two `REAL`
  `..., value1, value2 => ..., value1+value2`

- SBR[7]: Subtraction of two `REAL`
  `..., value1, value2 => ..., value1-value2`

- MULTR[8]: Multiplication of two `REAL`
  `..., value1, value2 => ..., value1*value2`

- DIVR[9]: Division of two `REAL`
  `..., value1, value2 => ..., value1/value2`

- INVR[10]: Inversion of an `REAL`
  `..., value1 => ..., -value1`

- EQU[11]: Returns `TRUE` if the two `INTEGER` are equal
  `..., value1, value2 => ..., value1=value2`

- NEQ[12]: Returns `TRUE` if the two `INTEGER` are different
  `..., value1, value2 => ..., value1!=value2`

- LSE[13]: Returns `TRUE` if the `INTEGER` `value1` is less or equal to the `INTEGER` `value2`
  `..., value1, value2 => ..., value1<=value2`

- LSS[14]: Returns `TRUE` if the `INTEGER` `value1` is less to the `INTEGER` `value2`
  `..., value1, value2 => ..., value1<value2`

- GTE[15]: Returns `TRUE` if the `INTEGER` `value1` is greater or equal to the `INTEGER` `value2`
  `..., value1, value2 => ..., value1>=value2`

- GTR[16]: Returns `TRUE` if the `INTEGER` `value1` is greater to the `INTEGER` `value2`
  `..., value1, value2 => ..., value1>value2`

- EQUR[17]: Returns `TRUE` if the two `REAL` are equal
  `..., value1, value2 => ..., value1=value2`

- NEQR[18]: Returns `TRUE` if the two `REAL` are different
  `..., value1, value2 => ..., value1!=value2`

- LSER[19]: Returns `TRUE` if the `REAL` `value1` is less or equal to the `REAL` `value2`
  `..., value1, value2 => ..., value1<=value2`

- LSSR[20]: Returns `TRUE` if the `REAL` `value1` is less to the `REAL` `value2`
  `..., value1, value2 => ..., value1<value2`

- GTER[21]: Returns `TRUE` if the `REAL` `value1` is greater or equal to the `REAL` `value2`
  `..., value1, value2 => ..., value1>=value2`

- GTRR[22]: Returns `TRUE` if the `REAL` `value1` is greater to the `REAL` `value2`
  `..., value1, value2 => ..., value1>value2`

- PUSH[40] value: Push the value on the stack
`...  => ..., value`

- POP[41]: Pops a value form the stack
`..., value => ...`

- LOADP[42] adr, n: Loads n values of the port at address adr and pushes it on the stack
`...  => ..., value1,...,valueN`

- STOREP[43] adr, n: Stores the n values popped from the stack in the port at address adr
`..., value1,...,valueN => ...`

- LOADE[44] adr, n: Loads n bytes of the event at address adr value and pushes it on the stack
`...  => ..., value1,...,valueN`

- STOREE[45] adr, n: Stores the n values popped from the stack in the event at address adr
`..., value,...,valueN => ...`

- COPYR[46] adr: Copies from the global memory space the output port value at address adr to the task context ctx
`..., ctx => ..., ctx`

- COPYS[47] adr: Copies from the global memory space the input port value at address adr to the task context ctx
`..., ctx => ..., ctx`

- COPYE[48] adr: Copies from the global memory space the event at address adr to the task context ctx
`..., ctx => ..., ctx`

- LOADF[49] arg: Returns on the stack if event at address adr was fired

- JUMP[60] arg: Jump to the instruction at arg

- CJUMP[61] arg: Jumps to the instruction at arg if popped value is `TRUE`
  `..., value => ...`

- RETURN[62]: Returns from the Embedded Machine

- RETURNE[63]: Fires the completion event of the task and the returns from the Embedded Machine

- ALLOC[80] lsize, rsize: Allocates lsize values for the local ports of the task and rsize values to store the address of the output ports. The allocated context reference is pushed on the stack
  `...  => ..., ctx`

- DELETE[81] lsize, rsize: Deallocates the context ctx and updates the output port in the global memory space
  `..., ctx => ...`

- WRITE[100] arg: Writes the `INTEGER` value arg on the screen (used for debugging)

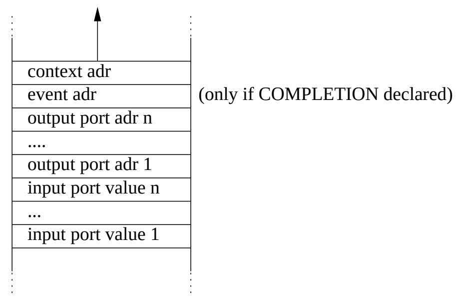
- WRITER[101] arg: Writes the `REAL` value arg on the screen (used for debugging)

- WRITEB[102] arg: Writes the `BOOLEAN` value arg on the screen (used for debugging)

6.3 The Task Context

Each task needs a private memory space in order to store its local port values and a stack to perform its operation. xGiotto does not allow any task to access the global memory space (i.e. a task is side-effect free).
The compiler assumes a given structure in which the port are stored. This is important during task activation and task termination. During activation, the actual parameters have to be passed to the task. During termination the generated task output ports have to be updated in the global memory space.
During task activation the compiler will put the actual parameters on the stack (first input ports values, then output ports addresses and the completion event address) . The compiler generates glue F code for the task activation that generates a new task context and copies the actual parameter for the stack to the task context stack. The order of the actual parameters on the stack after

the call to the ALLOC instruction (see the E code in the previous section) is shown:



6.4 Memory Layout

The memory layout is a simple description of the memory structure of the program. It contains a declaration of all ports and events defined in the program. For each port its name, address, and size is specified. When a port is an output to the system, then the address will be a platform-dependent memory mapped I/O location. Similar to a port, each event is defined by its name, address, and size. In addition, external events specify the external interrupt, and event expressions store the expression which will be evaluated at run-time.
During start-up the system loads the memory layout, allocates the ports and events as required and binds the external interrupts. The system defines a default TIME event which will be always bound to the timer interrupt.
For example the following program declaration:

```
PORT
   INTEGER p; INTEGER q;
EVENT
   INTEGER A AT int1;
```

The compiler generates the following memory layout:

```
Ports: 2
  p 4 0
  q 4 4
Events: 2
  TIME 4 0 timer
  A    4 4 int1
```

The memory layout file structure is as follows (EBNF):

```
MemoryLayoutFile = NofPorts:4 {PortElem}
                   NofEvents:4 {EventElem}.
PortElem = name:String adr:4 size:4.
```

```
EventElem = name:String adr:4 size:4 form:4
            [interrupt:String | EventExp].
EventExp = ["1" op:4 EventExp EventExp |
            "0" EventIndex:4].
```

Chapter 7

Examples

7.1 Conference

In this example we program an automatic paper writer. The program will generate 2 papers of 15 pages each (simulated by a number) as soon as the external `CallForPaper` event is raised. The first paper will be written at the arrival of the external event `Deadline`. The second paper will wait for the extended deadline (i.e. the second arrival of the `Deadline` event). The external function `MakeScience` is external to this program.

```
PROGRAM conference {

  TYPE paper ARRAY 15 OF INTEGER;

  PORT
    paper p1;
    paper p2;

  EVENT
    INTEGER C AT CallForPaper;
    INTEGER D AT Deadline;

  TASK Writing(INTEGER seed) OUTPUT (paper p)
    VAR (INTEGER i) {
    i = 0;
    WHILE (i<15) {paper[i] = MakeScience(seed,i); i++};
  }

  TIMING Call() {
    Writing(1)(p1)[D];
    Writing(5)(p2)[2D];
  }
```

```
  {Call()[C]}
}
```

7.2 xGiotto includes Giotto

In this example we demonstrate that xGiotto is a superset of Giotto . To demonstrate this we show how a single mode in Giotto is translated in a xGiotto program. To note is the fact that all events are timing events.

```
PROGRAM GiottoMode {

   PORT
     INTEGER p = 0;
     INTEGER q = 0;

   TASK P1(INTEGER a, INTEGER b) OUTPUT (INTEGER c){}
   TASK P2(INTEGER a) OUTPUT (INTEGER b){}

   TIMING ModeA() {
     P1(p, q)(p)[1000];
     P2(q)(q)[500];
      [500]ModeB();
   }
   TIMING ModeB() {
     P2(q)(q)[500];
      [500]ModeA()
   }

  {[1]ModeA();}

}
```

In the following example we extend the previous program adding a mode switch, and therefore showing that all the features of Giotto are programmable in xGiotto . The translation is semantically equivalent: it can be shown that independently from the running platform, at any point in time the value of the Giotto and xGiotto ports will be equivalent.

```
PROGRAM GiottoModeSwitch {

   PORT
      INTEGER p; INTEGER q;
      INTEGER r; INTEGER s;
      INTEGER t; INTEGER u;
```

```
EVENT
   BOOLEAN SWITCH_MODE AT buttonSwitchMode;

TASK P1(INTEGER a) OUTPUT (INTEGER b){}
TASK P2(INTEGER a) OUTPUT (INTEGER b){}
TASK P3(INTEGER a) OUTPUT (INTEGER b){}

TIMING ModeA() {
   IF (SWITCH_MODE) {[]ModeC()}
   ELSE {
     P1(p)(q)[20];
     P2(r)(s)[10];
     [10]ModeX();
   }
}

TIMING ModeX() {
    IF (SWITCH_MODE) {[]ModeInt12()}
    ELSE {[]ModeB();}
}

TIMING ModeB() {
   P2(r)(s)[10];
   [10]ModeA();
}

TIMING ModeInt12() {
   P3(r)(s)[10];
   [10]ModeC();
}

TIMING ModeC() {
   IF (SWITCH_MODE) {[]ModeA();}
   ELSE {
     P1(p)(q)[20];
     P3(t)(u)[10];
     [10]ModeY();
   }
}

TIMING ModeY() {
    IF (SWITCH_MODE) {[]ModeInt21();}
    ELSE {[]ModeD();}
}

TIMING ModeD() {
```

```
      P3(t)(u)[10];
      [10]ModeC()
   }

   TIMING ModeInt21() {
      P2(r)(s)[10];
      [10]ModeA();
   }

  {[]ModeA();}

}
```

Bibliography

[1] T.A. Henzinger, B. Horowitz, and C.M. Kirsch, GIOTTO*: A time-triggered language for embedded programming*, Proceedings of the IEEE **91** (2003), no. 1, 84–99.

[2] T.A. Henzinger and C.M. Kirsch, *The Embedded Machine: Predictable, portable real-time code*, Proc. of the International Conference on Programming Language Design and Implementation, ACM Press, 2002, pp. 315–326.

[3] T.A. Henzinger, C.M. Kirsch, R. Majumdar, and S. Matic, *Time-safety checking for embedded programs*, EMSOFT 02: Embedded Software (A. Sangiovanni-Vincentelli and J. Sifakis, eds.), Lecture Notes in Computer Science 2491, Springer-Verlag, 2002, pp. 76–92.

[4] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree, *A* GIOTTO-*based helicopter control system*, EMSOFT 02: Embedded Software (A. Sangiovanni-Vincentelli and J. Sifakis, eds.), LNCS 2491, Springer-Verlag, 2002, pp. 46–60.