Expressing Giotto in xGiotto and Related Observations on Schedulability Analysis

Arkadeb Ghosal

ABSTRACT

xGIOTTO is programming language for hard real-time event-triggered applications. Like its predecessor GIOTTO, xGIOTTO is based on the Logical Execution Time model of tasks; the programmer specifies when the outputs of a task become available, and the compiler checks if the specification can be implemented on a given platform. xGIOTTO handles both periodic time-based events and aperiodic asynchronous events. Events are the main structuring principle of the language and through a mechanism called event scoping allows a broad range of event expressiveness and environment assumption embedding. The xGIOTTO compiler performs three checks on the programs; race condition detection, memory size requirements and schedulability analysis. However the schedulability analysis is expensive (*exptime*-complete). The present works focusses on finding a suitable sub-class of xGIOTTO for which the schedulability check would be less expensive.

1. INTRODUCTION

Real-time systems for embedded applications are characterized by limited memory, distributed nodes, interprocess communication, fast context switches and concurrency [1]; the most important of them being *predictability* and *timing*. The execution of a safety critical system must be predictable (determinacy in program variables) and the evaluation of a task should be available when it is due (neither before the deadline nor after). Several programming paradigms have been used for implementing controllers for real-time systems. The traditional one is the scheduling based approach [2] where each task is assigned a priority. Another well-known approach is programming with synchrony assumption [5] where all tasks are assumed to execute in logical zero time. A third programming approach is based on task model with *Logical Execution Time*. In the LET model, when a task is released on a platform its corresponding LET is specified by a termination event. The task output is available only when the termination event occurs. Even if the task completes its execution before the termination event arrives, the task output is not released. A trace of the execution is *time-safe* if all tasks released along the trace completes their execution before the arrival of the respective termination event. A program is *schedulable* if all the traces are time-safe. While this allows an efficient way to introduce assumption on execution times of tasks and suits the need for distributed computing, the program execution is time-deterministic (no jitter) and value-deterministic (no race conditions) and thus makes program verification and analyses easier than the traditional scheduling model. GIOTTO [6] is a language based on LET model and implements real-time systems with multi-model time-periodic behavior. Timed-Multitasking [10] extends this model to event driven aperiodic systems. The language xGIOTTO [3] is based on LET model and handles aperiodic, asynchronous events like Timed-Multitasking. However xGIOTTO introduces the concept of event scoping for efficient handling of events and embedding environment constraints in an succinct manner [3, 4].

Determinacy (absence of races) and time safety (schedulability within logical execution times) can be formally proved for xGIOTTO programs. The check for determinism is formalized as a reachability over exponential state space (and is *pspace*-complete) while the schedulability check can be formalized as a game in exponential state space (and is *exptime*-complete) [3]. The target of the report is to identify a sub-class for xGIOTTO for which checking time safety is substantially simpler (at least polynomial in the size of the program).

Motivation. The present work is motivated by two results. First is the interesting case of GIOTTO programs for which the schedulability check for necessary condition is *pspace-complete* while a check for sufficiency can be done in polynomial time in the size of the program [7]. GIOTTO programs for which all modes are reachable the sufficiency check is also necessary. Second is the result presented for typed E programs in [9]. GIOTTO runs on the virtual machine E Machine [8]. Typed E programs (programs for E Machine) allow simple schedulability analysis and any GIOTTO program can be compiled into typed E programs [9].

The project targets time-triggered xGIOTTO programs. The information of time is essential to do a meaningful schedulability analysis. Given a task with release and termination event being `A` and `B` respectively, prediction on schedulability can not be performed unless the worst-case-execution-time of the task is 0; this is because in the worst case `B` can happen immediately after `A`. Only if an assumption is provided that the two events are separated by a time interval (say 3 time units), a schedulability analysis can be performed which predicts the scenario to be schedulable only if the wcet of the task is less than 3 time units. The focus is to identify a sub-class that is at least as expressive as GIOTTO and has simple schedulability analysis. The next focus would be to extend the schedulability analysis to event driven programs (possibly with some constraints).

The other formulation of the schedulability analysis is to provide constraints for the environment behavior. Hence for the above task scheduling scenario the question to be asked is: Given the worst-case-execution-time for the task, what should be the minimum time difference between the occurrence of events `A` and `B`. This would be an interesting focus for future research.

Overview. Section 2 briefly describes xGIOTTO. Section 3 presents the idea behind GIOTTO to xGIOTTO translation and the limitations of the syntax and semantics of the present definition of xGIOTTO. Section 4 presents a brief overview on the proposal for modifications in the definition of xGIOTTO. Section 5 presents the observations on schedulability test.

2. XGIOTTO

There are three basic constructs: reaction, trigger and release. The reaction `react b until e` defines a reaction block with body `b` and termination event `e`. The body `b` consists of triggers and releases. The second construct trigger `when e react r` invokes a reaction `r` at the event `e`. The reaction `r` may be a single reaction block or multiple reaction blocks composed in parallel. When the event `e` occurs all the

reaction blocks defined by the reaction `r` are invoked in parallel. The third construct is the release statement `release t` where `t` is a task. The task `t` is released as soon as the reaction block (defining the release statement) is invoked and is terminated when the reaction block terminates.

A reaction block in xGiotto defines an event-scope. An event scope consists of the until event and the when events of the triggers of a reaction block. Upon invoking the reaction block of one of the when statements, the current event scope is pushed onto a stack (i.e., it becomes passive) and a new event scope is created which becomes the active scope. In xGiotto multiple reaction blocks can be invoked in parallel; the scope of the parent block is pushed onto the stack and the scopes of all parallel blocks become active. Therefore we have a tree of scopes with the root of the tree being the initial scope, and the leaves of the tree being the active scopes. The function of the parallel reaction blocks are independent of their siblings.

An event of an active scope either, in the case of a when event, invokes a reaction, or in the case of an until event, terminates the corresponding scope. If an event `e` of an active scope can both invoke a reaction as well as terminate the scope, then the termination action has precedence. An event of a passive scope can be handled in the following three ways: it may be ignored (keyword `forget`); or it may be postponed until its scope becomes active again, once all descendent blocks have terminated (keyword `remember`); or it may disable all descendent blocks, thus speeding up their termination (keyword `asap`). Note that only active until events can terminate active tasks; in particular, active tasks cannot be prematurely terminated by passive `asap` events.

Syntactical Features. An xGiotto program defines a set of ports (or program variables), a set of events, a set of tasks and a set of reaction block definitions. A reaction block consists of a set of (possibly guarded) trigger statements and a set of (possibly guarded) task releases. This defintion has been referred to as core-xGiotto in [3]. The semantics of the core-xGiotto is provided below. Refer to [3] for the complete syntax and semantics; any xGiotto program can be translated to core-xGiotto.

Semantical Notions. The execution of an xGiotto program yields a possibly infinite sequence of configurations. Each configuration consists of the values of all program variables (*ports*) and a tree of scopes. Each *scope* contains a termination event, a trigger set and a ready set. The active scopes are the leaves.The *trigger queue* contains the enabled reactions, each associated with an invocation event: if the invocation event for an enabled reaction of an active scope arrives, then the first such reaction is invoked, and for each of its parallel reaction blocks, a new scope is added as a child to the present scope, rendering that scope passive. The *ready set* of a scope contains the tasks that have been released in the scope; their termination event is the termination event of the scope. Each when statement of a reaction block adds an event-reaction pair to the trigger set; each release statement adds a task to the ready set. The termination event of an active scope removes the scope.

The execution can be represented by a state-transition graph whose states are the program configurations, and whose transitions correspond to the occurrence of a new event, the termination of a scope, and the invocation of a reaction. When a new event arrives, first an *event transition* records the event occurrence in all scopes; then a sequence of *termination transitions* removes (possibly nested) scopes that have terminated and finally a sequence of *reaction transitions* adds (possibly nested) new scopes by invoking enabled reaction blocks. If no more reaction blocks can be invoked, the configuration is called *waiting*, and the arrival of the next event is awaited. All transitions take place in logical zero time; time advances only in waiting configurations. No two unrelated events arrive at the same time.

Implementation. The xGiotto compiler compiles the program, performs analyses (like race condition detection and schedulability check) and generates code for the run-time system. The run-time implementation consists of a virtual machine (Embedded Virtual machine or EVM), a real-time platform and the environment. The code generated by the compiler (EVM code) is interpreted by the EVM. An implementation of the EVM requires three components: a dynamical data structure to keep track of the scopes (event filter), a processor which computes the reaction to an event (reactive machine) and a scheduler. The event filter computes the event and the termination transitions on the tree of event scopes. Next it passes a set of EVM code addresses (which correspond to the invoked reaction blocks) to the reactive machine. The reactive machine interprets the EVM code (thus performing reaction transitions) and enables new triggers and releases new tasks. When all invoked reactions have been processed by the machine, the scheduler chooses a task to execute from the ready set of the active event scopes, and whenever such a task completes, the EVM is notified. The platform interacts with the environment through actuators and sensors. The actuators are driven by the task outputs and the sensors generate events which are handled by the event filter.

3. GIOTTO TO XGIOTTO

In this section the translation procedure for Giotto modes and mode switches to xGiotto has been provided using two examples. The details of the description (types of event or types of parallelism) have been ignored and the idea behind the translation has been focussed.

Consider the Giotto mode shown in Figure 1. The mode period is 6 and releases two instances of task `t1` and three instances of task `t2`. The mode period is divided into units (the idea is taken from [7]) which is the least common multiple of the frequencies; the number of units here is 6 with unit span being 1 time unit. The idea would be generate a reaction block that starts at an unit and terminates at the end of the mode period. The corresponding xGiotto modes and tasks have been shown in Figure 1; the reaction blocks are named according to the unit at which they are released and the corresponding line to a reaction block shows the span of the reaction block. Thus reaction block `R0` is released at time `0` and terminated at time `6`.

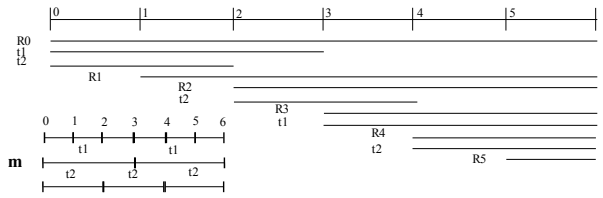

Figure 1: Giotto mode

Reaction block `R0` invokes parallel reaction blocks to release tasks `t1` and `t2`. These correspond to the first instance of

the corresponding tasks in mode `m`. `R0` invokes block `R1` at the next time unit (the units of the mode correspond to the time units here). At unit `1` there is no task release; so `R1` invokes `R2` at the next time unit. `R2` invokes a reaction to release task `t2` (corresponding to the second instance of the task in `m`) and invokes reaction `R3` at the next time unit. `R3` invokes a reaction to release task `t3` (corresponding to the second instance of the task in mode `m`) and triggers `R4` at the following time unit. `R4` releases the last instance of task `t2` and triggers `R5` (an empty reaction block) at the next time unit. Note all the reaction blocks terminate simultaneously at time unit 6 and a new instance of mode `m` is started.

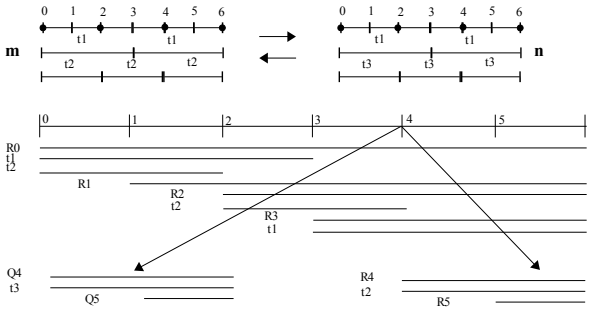

Figure 2: Giotto mode switch

A case of mode switch has been considered in Figure 2. The modes `m` (identical to the last example) and `n` (same structure as `m` except that the task `t2` has been replaced by task `t3`) switches between each other; for both the modes, the mode switch frequency is 3. The possible points of mode switches have been marked by black dots on the time line of the modes. The figure shows the implementation of the mode switch for the switch at time unit 4 for mode `m`. The reaction blocks and task release remain same for blocks `R1`, `R2` and `R3`. However the reaction block at the next time unit should check for mode switch condition and a conditional branch is added. If the switch condition is false, blocks `R4` and `R5` are invoked; if the switch condition is true, blocks `Q4` and `Q5` are invoked and task `t3` is released.

Consider the following GIOTTO program with two modes:

```
start m {
  mode m() period 6 ms {
    exitfreq 2 do n(c);
    taskfreq 1 do t1();
    taskfreq 2 do t2(); }
```

```
mode n() period 12 ms {
  exitfreq 3 do m(c);
  taskfreq 2 do t1();
  taskfreq 3 do t3(); } }
```

Lets consider the reaction blocks required to implement mode `n`; we need 6 units (so six nested reaction blocks) or 12 time units to translate the mode. At time unit 4, the switch from `n` to `m` has to be implemented. For the switch true branch: a jump to end of the mode `m` is implemented. As 12 time units have already been assigned to mode `n`, we are still left with 6 time units and a mode of `m` has to be implemented in that time. However one has to consider the mode switch case at the start of mode `m`; for the switch true case another instance of mode `n` has to be implemented with time span of 12 time units. The above scenario of mode switch has also to be implemented for this second instance of mode `n`. This causes the potential problem of allocating period of `n` infinitely. This is a problem with the present syntax. The shortcomings are mainly due to facts that all XGIOTTO modes need to have finite termination event and there can be no cyclic calls between reaction blocks. Both the constraints are required for meaningful program analysis but makes the present syntax incapable of expressing the complete GIOTTO domain.

The translation procedure outlined above works for 3 special cases of GIOTTO programs: 1, all modes have same period, 2, all modes have at least one task with frequency 1, and 3, all modes have harmonic set of tasks. The detailed procedure for such a transform has been formally presented in the extended abstract.

4. PROPOSED MODIFICATIONS

In this section i will present an outline of a modified syntax and semantics for XGIOTTO which can express GIOTTO and is amenable to simple schedulability analysis. My focus would be on purely time triggered implementations. The modifications will focus on two concepts: out-of-scope activation and use of implicit parallelism instead of explicit parallel operators.

In this new syntax, trigger statements with `when` construct is not supported. Instead only calls to reaction block is allowed; thus `react r until e` invokes a reaction block `r` with until event `e`. No parallel construct is allowed; however implicit parallelism can be defined (this would be discussed later). The reaction blocks may release tasks and /or invoke reactions sequentially. The call to reaction blocks may be guarded.

The *until* event of a reaction block has been extended to two events: *termination* event and *continuation* event. The termination event specifies the event of termination for the tasks released in the reaction block. The continuation event specifies when the next call to reaction is invoked. The continuation event may occur earlier or at the same time as the termination event but not later. Consider the code fragment in the left:

```
react R1 until 3 continue 2
react R2 until 2 continue 1
```

```
react R12 until 3 continue 2;
react R2 until 2 continue 1;
react R1 until 3 continue 1;
react R2 until 2 continue 2;
```

The tasks in block `R1` are terminated only at time 3. However the next statement is executed at time 2 i.e. `R2` is executed at time 2. This has two implications. First, the scope is defined by the continuation event and the LET of the tasks released is given by the termination event. Second, parallelism is implicit; e.g. in the above case `R1` and `R2` runs in parallel for the interval between time unit 2 and 3. To assist in simple schedulability check another constraint is added. All the nested reaction blocks in `R1` have to terminate by its continuation event and the termination event of `R2` be equal to or later than the 'pending' termination event of `R1`. The first constraints implies that the schedulability of nested reaction blocks in `R1` does not depend on the structure of `R2` and vice versa. The second constraint implies that the tasks released by `R1` has same or earlier deadline than the tasks in `R2`. The two constraints can be verified by a mix of syntactic definition and program analysis. The constraints helps in defining a simple schedulability analysis and would be further analyzed in the next section.

The code for GIOTTO mode `m` is shown in the right. Reaction `R12` releases task `t1` (the until event implies it would be terminated at time 3). A nested reaction block in `R12` (not shown) releases task `t2` with termination at time 2 (and obeys the constraint that the termiantion event of nested block should not be later than the continuation event). At time 2 the control executes the following statement (call to

R2). Now if R2 and R1 release tasks t2 and t1 then the code invokes t1 and t2 in a non-harmonic fashion similar to mode m.

The specific case of GIOTTO program which could not be translated into XGIOTTO can be translated in the new syntax and the code has been shown below:

```
/* main  */

if (mode = m) react m;
if (mode = n) react n;

--------

/* mode m */

react r0 until 6 continue 3;
if (sw12) {
   react {} until 1;
   react q5 until 2;
} else
   react r1 until 3;
```

```
/* mode n */
react q0 until 6 continue 4;
if (sw21)
 react {} until 2;
else {
 react q2 until 4 continue 2;
 react q3 until 6 continue 2;
 if (sw21) {
   react {} until 1;
   if (sw12) {
     react {} until 1;
     react q5 until 2;
   }
   else
    react r1 until 3;
 }
  else
   react q4 until 4;
}
```

Figure 3: Giotto modes in the new syntax of xGiotto

The main reaction block invokes two reaction blocks m and n depending upon the value of a port which stores the present mode. Note that no until information is used while invoking these two blocks (this potentially frees the programmer in identifying the termination event required to schedule a mode). In rest of the structure, at each step time allocated is equal to the period of the lowest frequency task.

5. SCHEDULABILITY

For a time-triggered XGIOTTO program without any branching and no parallel invocation there is always a single thread of reaction invocations. This is equivalent to scheduling a set of aperiodic tasks with varying arrival times. Horn found a scheduling algorithm for a set of independent tasks on a single processor, where tasks have dynamic arrivals and preemption is allowed [2]. This algorithm is known as *Earliest Deadline First* (EDF). For any algorithm that schedules a set of tasks with dynamic arrival times (with preemption allowed) EDF is optimal [2]. For the above class of XGIOTTO programs a complete set of task arrival and termination patterns i.e. a table of tasks with arrival time, deadline and WCET for each task instance can be derived. Then the Horn's algorithm can be used to check the schedulability and if the program is schedulable then EDF can be used as the scheduling strategy.

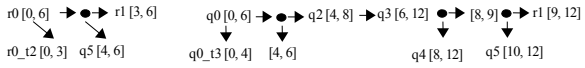

Figure 4: Giotto mode switch

However parallelism and conditional branches make the problem complex. The idea is to use a mix of utilization test and Horn's algorithm to decide for schedulability. A *reaction call graph* is generated from an XGIOTTO program. This graph bears the information of reaction calls including conditional branches. The RDG for the program in Figure 3 has been shown in Figure 4. The dots represent the branching points and the non-named intervals are the one with no task release. The procedure would be to reduce the graph to one path per branch depending on the utilization ratio. Once the whole graph is reduced such that no branches exist, it is traversed to generate the call pattern of tasks (release and termination time) and finally Horn's algorithm is used to check the feasibility of scheduling such a task pattern. However fairness of choosing the branches is an important issue. It can be shown that for unrestricted parallelism choosing the path with the worst utilization is not enough. Specifically even if the program is schedulable; a path with lower utilization may still make the program unschedulable. This is because the parallel reactions are not compositional in the sense that the schedulability of the blocks depends on the internal structure of the sibling blocks. In this situation the constraints on the termination and continuation event provides the required fairness to decide the conditional branch. This is specifically because they impose disjoint-ness in the LET of tasks for parallel reaction blocks. For parent-child pair the LETs may overlap; in this case the termination of child is guaranteed to be at the same time or later than the parent. The RDG being port-state abstracted the test would be sufficient if all reaction blocks are not reachable (the test considers all branches to be equally possible). However if all reaction blocks are reachable the test would be necessary.

Future Direction. In future the focus would be to formalize the proposed syntax and semantics and to automate the translation of GIOTTO to XGIOTTO. The other interesting question is to investigate the expressiveness of the language. The final focus would be to extend this definition to event driven systems and systemizing the schedulability check.

6. REFERENCES

[1] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[2] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publisher, 1997.

[3] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. A. Sanvido. Event-driven programming with logical execution times. In *HSCC 04: Hybrid Systems Computation and Control*, Lecture Notes in Computer Science 2993, pages 357–371. Springer-Verlag, 2004.

[4] A. Ghosal. Event-driven programming with logical execution times. Technical report, UC Berkeley, 2004.

[5] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publisher, 1993.

[6] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. GIOTTO: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.

[7] T. A. Henzinger, C. M. Kirsch, R. Majumdar, and S. Matic. Time-safety checking for embedded programs. In *EMSOFT 02: Embedded Software*, Lecture Notes in Computer Science 2491, pages 76–92. Springer-Verlag, 2002.

[8] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proceedings of Programming Language Design and Implementation*, pages 315–326. ACM Press, 2002.

[9] T. Henzinger and C. Kirsch. A typed assembly language for real-time systems. In *Proc. International Conference on Embedded Software(EMSOFT)*, LNCS. Springer, 2004.

[10] J. Liu and E. A. Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23(1):65–75, 2003.