Scheduling globally asynchronous locally synchronous systems for guaranteed response times

ABSTRACT
This paper analyzes and schedules Globally Asynchronous Locally Synchronous (GALS) programs to bound response times to input events. The proposed approach is applicable to scheduling of GALS programs for different target architectures with single or multiple processors or cores. A Satisfiability Modulo Theoretical (SMT) formulation in the quantifier free linear real arithmetic (QF_LRA) logic is used for scheduling. A novel technique to encode rendezvous protocol in the presence of synchronous parallelism into QF_LRA logic is presented. Finally, our SMT formulation is scalable, as the number of clauses is polynomial in the number of program states.

1. INTRODUCTION AND RELATED WORK
Hard real-time systems need to provide guaranteed response times to input events from their environments. For example, when a driver presses the breaks of a car, the breaks will be applied within N seconds. There are two common approaches for guaranteeing response times. One is via Real Time Operating Systems [7], wherein a scheduling policy e.g., Earliest Deadline First, is provided for scheduling the application at run-time. This approach is well suited for systems that work in an unknown environment, i.e., an environment that cannot be modeled. Many hard real-time embedded systems work in a strongly constrained environment, wherein the environment, the application and their interaction can be modeled. A significant amount of research literature exists [12, 1, 14] that targets compile time model-based scheduling and analysis of such applications to extract highest throughput and performance from the underlying execution platform – single or multi-processor systems, while guaranteeing real time properties.

In this paper our main objective is to provide an automated model-based compile-time scheduling framework for distributed control systems, programmed using Globally Asynchronous Locally Synchronous (GALS) paradigm. The motivation for carrying out this research is multiple-fold:

- A plethora of work exists [12, 1, 14, 5] for scheduling applications developed in pure synchronous languages – mainly Esterel [4] using the model-based approach. Synchronous languages due to their formal framework are good for programming real-time systems, but are not suited for distributed control systems, since the compiler removes all parallelism and compiles them into a single finite state machine. Thus, there is a need for a model-based approach for real time scheduling of distributed control software. We close this gap using our scheduling framework for GALS programs [13].

- The controller synthesis based approach to scheduling applications developed for non-synchronous languages [1], does not scale well as identified by the authors themselves [5]. In this article we propose a scalable Satisfiability Modulo Theories (SMT) [9] based scheduling approach for GALS programs. We purposely choose SMT, because we do not want to loose precision when using real-valued variables, as is the case with linear solvers. Moreover, scheduling requires ordering computations, which requires a mixed integer linear programming solution that is known to be inefficient with linear solvers.

- Finally, the recent flurry in the real-time research community targeting the worst case reaction time analysis of programs [16, 6, 15] only analyzes the so called throughput constraints – analyzing that the program reacts last enough for the environment input model (as identified by Sifakis et al [5]) and ignores the response time constraints – analyzing and scheduling the program to bound the program response to an input event.

Our major contributions in the paper are: (1) formalization of response time in a GALS setting. (2) A SMT based scheduling approach for GALS systems. (3) A rendezvous communication model amenable for real-time scheduling.

The rest of the paper is arranged as follows. Section 2 motivates the problem using an automated distributed control example. Section 3 formalizes the response time problem in a GALS setting. Sections 4 and 5 describe our approach to solving this problem. Section 6 gives the benchmark results and we finally conclude in Section 7.

2. A MOTIVATING EXAMPLE
Figure 1 shows a part of an automated ice cream manufacturing facility that is being developed in the GALS program-
The manufacturing unit consists of conveyor belts, with photo-eyes for object detection, a turntable with sauce dispenser and diverters to divert the ice cream to the correct path. Smart embedded controllers running SystemJ processes (henceforth called clock-domains (CD) – akin to Esterel programs, following SystemJ terminology) are placed across the manufacturing unit to control the manufacturing process.

Figure 2 shows the SystemJ CDs, running on the three distributed embedded controllers connected to the photo-eye (PE, in Figure 1), the diverter (DC, in Figure 1), and the turntable (TT, in Figure 1), respectively. The proposed scheduling technique can be used for such distributed multi-processor, single processor, and shared memory multi-processor systems.

The CD running on the PE controller waits for an input signal (A) from the photo-eye when the ice cream is detected. If the value of the incoming signal is 1, then the customer has asked for toppings, and this information needs to be conveyed to both: the diverter and the turntable.

A variant of CSP [10] style rendezvous on channels C and M is used to communicate this information to the CDs running on their respective controllers. SystemJ, being an asynchronous extension of Esterel, allows synchronous parallelism, lock-step execution of processes (henceforth called reactions), within CDs. We use this mechanism to synchronize with both the diverter and the turntable in parallel (C!;||M!; in Figure 2). Blocking rendezvous communication guarantees delivery of information to the other SystemJ CDs. If the value of the input signal A is not 1, then the ice cream needs to be directly moved to the pickup zone, and hence, only rendezvous with the diverter is required. The clock domain (CD-2) on DC, continuously waits to receive on channel C, and alternately moves the diverter to left or right. The turntable, controller similarly, waits to rendezvous on M and emits a signal, TT, to put the turntable in the starting position.

This system requires hard real-time guarantees. The conveyor, the diverter, and the turntable operate at different speeds. The conveyor cannot be stopped at run-time, due to backlogs and reduced throughput. There is no way to dynamically vary the conveyor speed during program execution as the conveyor speed is controlled using a knob – manually. Given these constraints, it is essential that when the ice cream is detected by the photo-eye (PE), the diverter and the turntable are in the correct position by the time ice cream reaches either of them. More formally, the time between the reception of signal A when CD-1 is in position L (see Figure 2) to the emission of signals B and TT needs to be bounded by real-time M and N, respectively, given some speed Z that the conveyor operates at.

In order to achieve this goal we need to: (1) orchestrate (schedule) the CDs on one or more processors, (2) find out the worst case reaction time for each CD on the available processors, and (3) verify that the program is functionally correct. Like others [16, 6, 5] we assume that the low-level worst case reaction time analysis and the functional correctness guarantees have been achieved. Furthermore, in this paper we also assume that the allocation of CDs onto different processors is given by the system designer.

3. PROBLEM FORMULATION

SystemJ program is a network of finite state machines (FSMs). Each state machine corresponds to an individual CD and this network of FSMs run asynchronously communicating via channels using blocking send and receive. Each state demarcates the end of a logical tick for that CD. The FSMs for CD-1 and CD-2 from Figure 2 are shown in Figures 4 and 5, respectively. The dotted circles indicate rendezvous states – states that participate in channel communication. Each transition in the FSM represents the big-step from one logical tick to the next, multiple micro-steps (micro states from different synchronous parallel reactions) might be encapsulated in a single transition. For example, every outgoing transition from state $N103$ indicates multiple micro-transitions of the individual reactions composed together in statement $C1;||M1; in Figure 2 (cf. 5.1.3).

Every transition in the FSM is guarded by signals and the result of the transition are signal updates and data computations. For example, CD-1 starts from the start state $s1$, and unconditionally moves onto the state labeled L, $\text{await}(A)$ statement in Figure 2. Three possible transitions can now be performed depending upon the presence/value of signal A as described in Section 2. CD-2 on the other hand after starting from the $s2$ state unconditionally moves to state $N21$, an internal rendezvous state, and remains there until the rendezvous with CD-1 is complete. Upon completion of rendezvous, the signal D is emitted and the correct data computation performed.

We have only shown some of the guards and outputs to avoid clutter in the figures. Furthermore, the channels C and M have been replaced with signals $C_{req}/C_{ack}$ and $M_{req}/M_{ack}$, which perform a four-phase handshake. This approach of implementing rendezvous using signals rather than channels, like in ordinary CSP, is essential to guarantee response times. We will further elaborate upon these essentials in the upcoming sections.
The real-time properties are guaranteed and specified on these FSMs. Every state demarcates the end of a logical tick of each CD. Like in Esterel, one or more `pause` constructs specify the end of a logical tick in SystemJ. A system designer (or compiler) labels any `pause` statement that participates in the response time constraint satisfaction and then uses these labels along with input and output signals to specify the parts (or whole) of the program(s) that need to satisfy different real-time bounds. For example, the response time guarantees from the PE to the DC/TT can be specified in our formulation as:

\[
\square((L \land A) \rightarrow \diamond_M D) \\
\square((L \land A) \rightarrow \diamond_N TT)
\]  

Equation (1) uses the notation from [2]. The bounded response time property in Equation (1) states that always (\(\square\)) when program is in position \(L\) and input signal \(A\) arrives, eventually (\(\diamond\)) signal \(D\) will be emitted within \(M\) units of time. Same for the emission of signal \(TT\). Our goal is to schedule the network of FSMs to guarantee these response times.

### 4. Overall Approach

The overall design flow of our approach is shown in Figure 3. The approach consists of 6 steps. Step-1, requires the user or the compiler to annotate the source program locations with labels (e.g., label \(L\) in Figure 2) and specifying the response time that needs to be guaranteed between these program locations. In step-2, the designer closes the environment with a test-bench. This so-called test-bench is programmed as a reaction (Environment reaction in Figure 3) and composed in synchronous parallel with the CD similar to [5]. Not all environments need to be closed, see Section 5.1.1 for further explanation on closing appropriate environments. Next, this composition is compiled into a network of FSMs in step-3. Step-4 requires the designer to input the worst case reaction times for individual CDs calculated using techniques described in [6]. An optional step of formally verifying the functional properties of the developed system may be carried out using a model-checker. In step-5 a SMT formulation in quantifier free linear real arithmetic (QF_LRA) logic is derived from this network of FSMs.

The resultant SMT formulation is then input into a SMT solver. If the resultant formulation is satisfied, release-times for each state in every CD are obtained from the solver, otherwise a proof of unsatisfiability is obtained. Once these release times are obtained, the original CDs in the SystemJ program are compiled into FSMs along with a tailored scheduler – compiled from the output of the SMT solver in step-6. These FSMs along with the scheduler (SCH in Figure 3) FSM are run together. The scheduler orchestrates the overall execution by sending explicit control signals to the FSM, releasing the FSM states at the calculated release-times. The result is an asynchronous interleaved execution of the states in individual CDs unlike multi-rate programs where all clocks are derived from a single super fast clock [3]. The interested reader can refer to the appendix for further details.

Before we can proceed to give our SMT formulation, we need to revisit the concept of worst case reaction time in the GALS setting.

#### 4.1 Worst case reaction time in the GALS setting

The worst case reaction time in a pure synchronous setting a la Esterel programs is well known – it is the worst case duration of an iteration of the system, from start of an iteration to the start of the next iteration [15]. In a GALS setting the worst case time in not obvious due to communication between CDs. Consider blocking rendezvous like in CSP [10], then a CD halts (stops processing) when in the send (e.g., \(C!\) in Figure 2) or receive (e.g., \(C?\) in Figure 2) state. Consequently, no input event can be captured even by the other reactions composed in synchronous parallel, using the \(1\) construct, in that CD. A chain of such dependencies can easily be created in any GALS/asynchronous program. In order to compartmentalize the worst case reaction time to a single CD it is essential that a CD does not halt even for a single instant during processing. Our solution to this problem is blocking at the logical tick level rather than halting. Hence, we rewrite all rendezvous communication to synchronous programs using signals. For example, the channel \(C\) and its corresponding
send (C1) and receive (C7) statements in Figure 2 are rewritten into signals $C_{\text{req}}$ and $C_{\text{ack}}$ to implement a four-phase handshake, see Section 5.1.2 for further details. Upon this rewrite, communicating CD acts as an environment to its partner CD, and well known worst case reaction time computation techniques [6] can be used to calculate the worst case reaction time of each CD individually.

Now, we are ready to describe the SMT formulation that results in the scheduler guaranteeing response times.

5. SMT FORMULATION FOR RESPONSE TIME GUARANTEES

First we describe the solution for a single processor system. Then we relax this constraint to shared/distributed memory multi-processor systems.

5.1 SMT formulation for a single processor system

Let us first consider the case of scheduling FSM without rendezvous communication to guarantee response times. Every FSM is a tuple $(V,E)$, where $V$ are the states and $E \subseteq (V \times V)$ are the transitions. Let, $V_1 \subseteq V$ and $V_i \subseteq V$ be the source states (states without incoming edges) and terminal states (states without outgoing edges) of the FSM. Furthermore, let $w_i$ indicate the worst case reaction time for the terminal FSM, set $C_i$, with $|C_i| = |V_i|$, be the real-time constraints for the completion of the terminal states, and for some $i \in V$, let $v_i$ denote the release-time for the state. Then, we can formulate the response time satisfaction problem in QF_LRA as:

$$\forall v_i \in V_i \; v_i \geq 0 \land \forall (i,j) \in E \; v_j \geq v_i + w_i \land \forall v_i \in V_i, c_i \in C_i \; v_i + w_i \leq c_i$$

(2)

Equation (2) states that the source states can start at anytime greater than or equal to zero. The successor states get delayed by the worst case reaction time due to the dependency from parents and finally, the terminal nodes need to satisfy the constraints specified by the designer. Note that join nodes – nodes with multiple incoming edges, e.g., $N11$ in Figure 4, get automatically delayed by the worst incoming edge.

Equation (2) can schedule only a single FSM. Multiple FSMs (denoted by the set $F$) running on the same processor need to be modeled with exclusive access to the underlying processor, as follows:

$$\bigwedge_{v_i \in F} \bigwedge_{v_j \in F, v_j \neq v_i} \bigwedge_{v_i \in V_i, v_j \in V_j} v_i \geq v_j + w_j \lor v_j \geq v_i + w_i$$

(3)

We perform a conjunction ($\land$) of Equation (2) and Equation (3) to obtain the schedule. The above formulation can schedule a network of FSMs on a single processor. We might still get unsatisfied as a solution for valid FSMs (e.g., FSM in Figure 5) due to loops.

5.1.1 Bounding temporal loops

Loops in FSMs occur in two forms: (1) self-loops, which indicate a waiting state, usually for an input signal, e.g., self-loop on node $L$ in Figure 4 waiting for signal $L$, and (2) back-edges, which indicate explicitly programmed loops, e.g., the loop statement enclosing CD-1 (see Figure 2) is indicated with edges back to node $L$ from its successor states. These loops are temporal, i.e., every iteration of the loop

(a) State dependencies between sender and receiver CDs

Figure 6: Four-phase handshake and its encoding in SMT consumes one logical time unit (tick). Temporal loops do not need any special attention in worst case reaction time analysis, because reaction time only considers a single reaction (one logical tick) of a CD, but needs special consideration in response time analysis. For example, if a designer wants to know the response time from state $L$ to state $N103$, the answer is simply $\infty$. Without knowing when input signal $L$ will be present, we can only assume that it might never be present, and hence, the answer. Bounding loops waiting on input signals from the environment is the primary reason for composing the CDs with their respective environment models, else, the response can never be guaranteed.

The back-edges, specifying explicit loops in the SystemJ CDs, also needs to be bounded or, if not bounded, can be safely ignored, because explicit loops restart the same part (or whole) of the program again and, hence, the response time between these iterations remains the same.

Finally, there are loops (e.g., self-loop on nodes $N21$ in Figure 5) that cannot be bounded simply by modeling the environment, since these loops correspond to signals created for rendezvous communication via channels. The bound of these loops can only be ensured by the scheduling policy of the CDs. For example, signal $C_{\text{req}}$ is only emitted (if ever) from CD-1 to CD-2 depending upon the scheduling policy of these CDs. But, we are building this scheduling policy itself! We tackle this chicken-egg problem in the next section.

5.1.2 Accounting for rendezvous communication dependencies

SystemJ channel implements four-phase handshake protocol through which networks of CDs (FSMs) synchronize with each other. We say CDs have _entered rendezvous state_ once the program control flow encounters a channel send or receive statement. CDs in the rendezvous state utilize a pair of rendezvous signals to perform the four-phase handshake. To illustrate the four-phase handshake protocol consider the example between the receiving and sending CDs through channels (C7 and C1) from Figure 2 with their rendezvous state representation and dependencies between those states, as solid arcs, in Figure 6a.

Nodes $N93$ and $N107$ are the rendezvous states from the sender (CD-1) and, $N21$ and $N12$ are from the receiver CD (CD-2), respectively. When the sender first enters $N93$, it waits for rendezvous signal $C_{\text{ack}}$ from the receiver. On the other hand, the receiver is able to make a transition to $N12$ from $N21$ in the absence of $C_{\text{req}}$ from the sender. In $N12$, the receiver continuously emits $C_{\text{ack}}$ until it receives $C_{\text{req}}$ from the sender. The sender is then able to make transition to $N107$ where it also continuously emits $C_{\text{req}}$ until the
On the other hand, absence of the signal $C_{ack}$. Finally, the receiver first finishes rendezvous (i.e. $C_{req} = \top$) followed by the sender (i.e. $C_{ack} = \top$).

In order to encode channel communication in the SMT formulation, we insert dependency arcs between the rendezvous states. For example, when the sender is in $N103$ it can only make a transition to $N107$ after both; its predecessor state and $N21$ in CD-2 complete their execution (i.e. $C_{ack} = \top$). On the other hand, $N21$ is only dependent on its predecessor. Dependency from $N12$ to $N107$ indicates that the sender is only able to complete rendezvous after scheduling the receiver state $N12$ (i.e. $C_{req} = \top$). Similarly, receiver is able to complete rendezvous when the sender enters $N107$ (i.e. $C_{req} = \top$). A complete encoding of the precedence constraints of the rendezvous states is shown in Figure 6b. These rendezvous constraints are automatically encoded via Equation (2), because of the added dependency arcs.

Finally, self-loops on each rendezvous state, which indicates waiting on a presence or absence of rendezvous signals, are omitted in the formula (shown as dashed arrows in Figure 6a). Instead, in our SMT formulation, dependency arcs replace the self-loops.

5.1.3 Interplay of rendezvous and synchronous parallel reactions

Although the encoding approach presented in Section 5.1.2 allows us to schedule multiple CDs with channels, it has one limitation: it cannot correctly handle rendezvous communication over distinct channels carried out in parallel using the synchronous parallel operator (\(1\)), e.g., \(CD|MT\) in Figure 2.

In Figure 7a (reproduced from Figure 4), $N103$ is one such state. Two incoming channel dependency arcs (Dep $C?$, Dep $M?$) are generated for $N103$ from CD-2 and CD-3, respectively. There are several possible outgoing transitions from $N103$: (1) only $C?$ is ready to rendezvous, (2) only $M?$ is ready to rendezvous, (3) both are ready to rendezvous (4) none of them are ready to rendezvous. In Figure 7a, we only show cases (1) and (2) due to lack of space. Suppose CD-2 enters the rendezvous state of channel C. CD-1 should be able to make transition to $N109$. However, this transition is not possible because of the dependency arc coming from CD-3 (M?). As a result, this macro state representation prevents micro transition of individual synchronous reactions.

To address this problem, we insert micro states for each synchronously composed rendezvous state with dependency arcs from multiple CDs. These additional states take zero execution time and hence do not influence execution time of the original FSM state. The updated representation of $N103$ is shown in Figure 7b. $N103$ now has only one dependency arc, which is from one of the micro states $N103_1$ or $N103_2$. $N103$ can be scheduled immediately when either of the two micro states are scheduled.

Next, we schedule every possible path due to channel communication, and based on generated schedule from SMT solver, we choose one branch: whichever is scheduled first. Our scheduler simply waits during execution slots allocated for all rejected paths. For example, when SMT solver gives a solution where CD-2 first enters rendezvous state, we then schedule N109 and discard $N79$ and its successor states from the schedule.

As a result of the introduction of micro states, Equation (2) needs to be modified. We first define $M \subseteq V$ where $M$ is a set of states, which have multiple dependency arcs coming from different CDs. Function $\forall i \in M, \lambda : i \rightarrow \Psi_i$ maps for all $i$’s in $M$ to a set of micro states $\Psi_i$. We then insert edges (dependency arcs) to these states: $\forall i \in M, E_{\Psi_i} \subseteq ((V \setminus \{i\}) \times \Psi_i)$. Since each micro state can also have multiple arcs coming from the same CD, its release time is delayed by the latest end time of its predecessor states, i.e., $\Lambda_{\forall i \in M} \Lambda_{\forall (i, j) \in E_{\Psi_i}} v_j \geq v_i + w$, where $w$ is the worst case reaction of the CD. Finally, we add the final constraint: $\Lambda_{\forall i \in M} \forall j \in \Psi_i v_j = v_i$, which states that the release time of the original state $i$ is equal to the release time of any one of its micro states in set $\Psi_i$.

5.2 Extension to the multi-processor case

Extension to multi-processor case is fairly straightforward. Equation (3) models exclusive access by states in the FSM to the underlying processor. In case of multi-processor system, Equation (3) needs to be modified. Only those FSMs (and their related states) that are allocated on the same processor should be considered for exclusive access. For example, if all the three controllers in Figure 2 are allocated on a single processor system then Equation (3) guarantees that the states in the three corresponding FSMs have exclusive access to this underlying processor. But, in case of all three CDs being allocated onto different processors, as shown in Figure 1, Equation (3) should be simply ignored.

6. BENCHMARK RESULTS

We have carried out a number of experiments to obtain schedules that satisfy response time requirements for various benchmark programs. We also show that distributing CDs on different processors decreases the response times.

Each benchmark program, Table 1, consists of up to three CDs. A dual-chamber implantable pacemaker is our case-study in safety critical system design using SystemJ adopted directly from one modelled by Jiang et al. [11]. We show that the pacemaker model generates electrical pulses within a given time constraint when it does not detect a heartbeat within a set time period. Conveyor belt control system consists of two mechatronic components, a Conveyor and a Mechanical arm each modelled as SystemJ CD, connected together into a simple mechanism for sorting items. In this example, response time for the system to generate a control signal for moving a conveyor has been verified once an item is placed on a conveyor belt. Washing machine controller is a pure synchronous system that consists of two main reactions; the washer and the drier. Each reaction awaits for an input signal from the user, which activates its washing or drying cycle. In this case, we collect the response time of the machine with specific sequences of user inputs. Robot motion controller consists of a camera and motion controllers.
which continuously locate an object, move to the location, and pick up that object. The system consists of two CDs, one for each controller. For this example, we show that once the controller detects an item, it picks up the item and generates a control signal within a guaranteed time bound. We also present the results of the response time analysis for our motivating example—the ice-cream factory and its related constraints in Equation (1).

As expected, the response times decrease, see Response time bounds in Table 1, with increasing number of processors, plateauing eventually. The reduction is because, multiple CDs can be executed in parallel. Some examples show no improvements in the response even with increasing number of processors due to dependencies.

The SMT solver (Z3 [8]) took on average 0.08 seconds for solving our benchmark programs (Figure 8). This scalability can be attributed to our SMT formulation itself. The number of generated clauses (constraints) in our formulation is polynomial in the number of variables (states in the FSM). The number of constraint clauses is bounded in worst case by: \( O(|E| \times |F|) + O(|V|^2 \times |F|) \) for Equations (2) and (3), respectively, where \( V \) are the states in the FSM, \( E \) are the edges, and \( F \) is the set of FSMs.

### 7. CONCLUSIONS

In this paper we have described a novel technique for scheduling Globally Asynchronous Locally Synchronous (GALS) programs for guaranteed response times. We first give a precise definition of response time in a GALS setting, and then perform a Satisfiability Modulo Theory (SMT) formulation in the quantifier free linear real arithmetic logic to tailor a scheduler for the GALS program that meets the response time constraints. This paper proposes novel techniques to deal with the challenges such as worst case reaction time analysis and interplay of synchronous parallel execution with rendezvous communication between asynchronous processes (the so called clock-domains). The benchmark results are very encouraging and show that SMT based approach is scalable for our purpose.

### 8. REFERENCES


Appendix

Scheduling clock-domains for the automated ice cream manufacturing system

Figure 9: A schedule for ice cream facility benchmark with response times where micro-states (e.g. \( N_{103,1} \)) are in parallel with macro-states

Figure 9 shows execution trace of the automated ice cream factory controller produced from our compiler for a single processor system. The interesting points are:

- There is no single integral multiple relating the execution speed (clocks) of different CDs. Thus, no super fast clock can be derived like in multi-rate systems. The execution of CDs is truly interleaved.

- Scheduler inserts gaps between states, corresponding to the conditional branches in the program whose results cannot be statically analyzed during compile time, e.g. any input or output signals that are not closed and/or rendezvous signals, which are regarded as inputs from the environment to CDs.