Schedulability Analysis of MSC-based System Models

Lei Ju Abhik Roychoudhury Samarjit Chakraborty
Department of Computer Science, National University of Singapore
E-mail: {julei, abhik, samarjit}@comp.nus.edu.sg

Abstract

Message Sequence Charts (MSCs) are widely used for describing interaction scenarios between the components of a distributed system. Consequently, worst-case response time estimation and schedulability analysis of MSC-based specifications form natural building blocks for designing distributed real-time systems. However, currently there exists a large gap between the timing and quantitative performance analysis techniques that exist in the real-time systems literature, and the modeling/specification techniques that are advocated by the formal methods community. As a result, although a number of schedulability analysis techniques are known for a variety of task graph-based models, it is not clear if they can be used to effectively analyze standard specification formalisms such as MSCs. In this paper we make an attempt to bridge this gap by proposing a schedulability analysis technique for MSC-based system specifications. We show that compared to existing timing analysis techniques for distributed real-time systems, our proposed analysis gives tighter results, which immediately translate to better system design and improved resource dimensioning. We illustrate the details of our analysis using a setup from the automotive electronics domain, which consist of two real-life application programs (that are naturally modeled using MSCs) running on a platform consisting of multiple electronic control units (ECUs) connected via a FlexRay bus.

1 Introduction

Message Sequence Charts (MSCs) or Sequence Diagrams are widely used by requirements engineers in the early stages of reactive system design [15, 23, 2]. MSCs can be very convenient for describing interactions between a number of agents, e.g., a bus protocol between a bus controller and a number of processing elements trying to negotiate access to the bus. MSCs are therefore a natural choice for modeling and specifying distributed real-time and embedded systems. Consequently, timing and schedulability analysis of MSC-based specifications play an important role in the high-level design of such systems.

However, a significant portion of the work directed towards analysing or reasoning about MSCs and other related specification formalisms focus on functionality validation (such as verification of safety and liveness properties). It is only recently that quantitative reasoning of such formalisms has attracted a lot of attention [8, 9]. On the other hand, there is a large body of literature on algorithms and techniques for timing and schedulability analysis of real-time systems. However, a large fraction of these techniques are applicable to system models that are essentially based on the concept of *task graphs* [3, 4, 5, 18, 19]. Although such graphs naturally represent data- and control-flow dependencies in periodically or sporadically executing applications [10, 20], they only provide *local* or *processor-centric* views of a distributed system. More specifically, the structuring mechanism used revolves around specifying all the tasks that execute on any given processing (or communication) element. As a result, they are not very suitable for specifying the *interactions* between the multiple entities of a distributed system – which is often a more natural way of specifying such systems.

Contributions of this paper: In this paper we make an attempt to reconcile the abovementioned gap between the timing/schedulability analysis techniques developed within the real-time systems community and the specification formalisms commonly used for designing large-scale distributed real-time systems. Our main contribution is a schedulability analysis technique for a standard MSC-based system specification of a distributed real-time system. Each MSC in such a specification denotes a *scenario* and captures the partial ordering between various computation and communication tasks/events constituting this scenario. Multiple such MSCs can be combined together to form what is referred to as a **Message Sequence Graph (MSG)** [1, 14], whose edges denote transitions from one scenario to the next. Multiple outgoing edges from a node in such a graph represent conditional transitions, where exactly *one* of the outgoing edges can be activated. A complete system specification consists of a set of such MSGs denoting concurrently running applications that share common resources. Examples of such applications in an automotive electron-

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.9

ics setting might be an *adaptive cruise controller*, an *advanced crash preparation system* and a *brake controller* application, all running concurrently and sharing common resources such as electronic control units (ECUs) and communication buses. It may be noted that that such a specification is completely standard [23] and is routinely used for modeling and specifying large distributed systems.

In what follows, we use certain standard MSC-specific terminology, which have been explained in Section 3 along with a formal description of MSCs. We extend the system specification described above by mapping the different *lifelines* in a MSC to different processing elements and their associated *messages* to different communication resources (e.g. buses). Further, we annotate the *events* and the *messages* constituting the different lifelines with lower and upper bounds on their execution/communication times. Such execution/communication times do not involve blocking times arising out of resource contentions, which is accounted for by our schedulability analysis. Given this system description, along with the scheduling/arbitration policies at the different resources, our analysis can be used to compute upper bounds on the end-to-end delays associated with various event (and/or message) sequences, which can then be checked against prespecified deadlines. Examples for such sequences might start with data arriving via a sensor, getting processed on several ECUs which also involves multiple transmissions over one or more buses, and then finally ending at an actuator.

Organization of this paper: The rest of this paper is organized as follows. In the next section we discuss some existing works on system-level schedulability analysis. In Section 3, we formally define our MSC-based system models and explain the difficulties involved in analyzing them. In Section 4 we provide an overview of our schedulability analysis technique, followed by the details in Section 5. A detailed case study illustrating the working of our technique is presented in Section 6. Finally, Section 7 outlines some directions for future work.

2 Related Work

There are two standard approaches for schedulability analysis of task graph-based specifications of real-time systems — worst-case response time analysis-based techniques [6, 13, 17], and the processor demand bound criteria-based analysis [4, 7]. It turns out that neither of these approaches can be applied to our setting in a straightforward manner. This is primarily because in traditional task graph-based specifications, all the vertices are mapped onto a single resource, whereas in our case each MSG (in fact even a vertex of an MSG denoting an MSC) involves multiple computation and communication resources. Hence, the semantics of MSGs are fundamentally different from the task graphs that have been studied in the real-time systems literature.

Our proposed analysis is motivated by the response time calculation algorithm presented in [27], which can handle system specifications with multiple computation and communication elements. We have adapted this algorithm to the specific context of MSCs, and in particular proposed two new extensions. (i) The algorithm in [27] is based on a response time analysis framework, which iteratively computes tighter estimates on the response times of various computation and communication tasks. However, it cannot handle conditional or non-deterministic branches which exist in MSGs. We get around this problem by combining the response time analysis-based technique in [27] with a demand bound criteria-based technique that was recently proposed in [3] to handle conditional branches in a different task model. (ii) Compared to [27], we also obtain tighter bounds on the response times of tasks by accounting for the dependencies in the preempting tasks/applications, by calculating request bound from higher priority tasks during the response time of the preempted task (event). The main novelty of our work stems from the interesting combination of response time analysis and demand bound criteria-based techniques, which is not commonly seen in the real-time systems literature. This is explained in further detail in Section 3.3 of this paper.

Analyzing system specifications with dependencies between computation and communication tasks is known to be a challenging technical problem. This is because the worst-case communication behavior depends on the traffic attempting to access the shared medium, whereas the traffic generated (by the computation inside the tasks) depends on the communication behavior encountered in the past. This naturally leads to an infeasibly expensive analysis which steps through the individual computation and communication steps inside/across tasks. It leads to a further combinatorial blow-up in the presence of conditional transitions. Our proposed analysis – using the combination of response time and demand bound criteria-based techniques – is able to contain this blow-up without leading to overly pessimistic results.

Tindell et al. [26] proposed a holistic schedulability analysis for distributed real-time systems, which bounds the worst case delays of both local computations and interprocessor communications. However, their analysis assumes only a simple static TDMA protocol for the bus communication, and the event dependencies are not taken into consideration (thereby leading to coarse analysis and pessimistic results). Finally, we would like to point out that there have been a few previous attempts towards developing schedulability analysis techniques for MSC-based system models [24, 25]. However, they either do not fully exploit the event dependencies within an MSC , or are restricted to the analysis of a single MSC (as opposed to a complete system model).

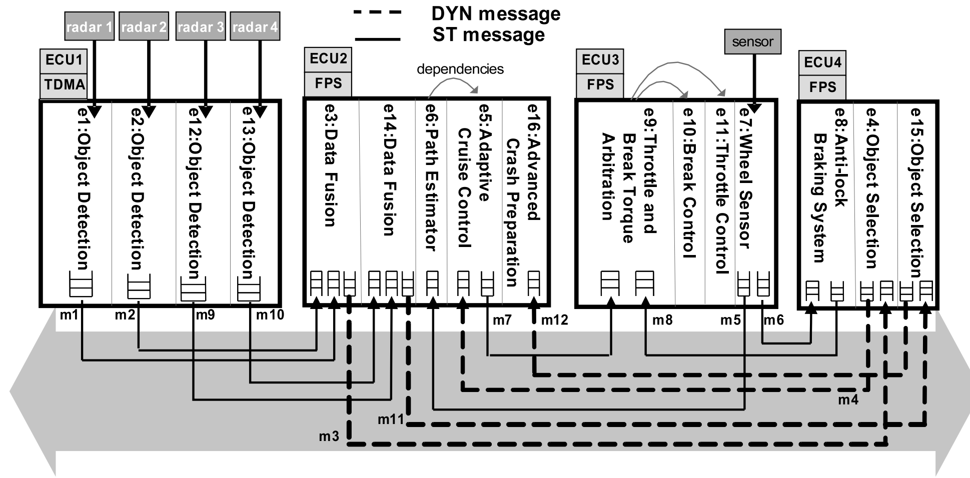

Figure 1. A FlexRay-based ECU network

3 Problem Formulation

3.1 Message Sequence Graph

Formally, an MSC is a labeled poset of the form $Ch = (L, \{E_l\}_{l \in L}, \preceq, \lambda)$ where L is the set of processes (also called lifelines) appearing in the chart as vertical lines, E_l is the set of events that the lifeline l takes part in during the execution of Ch. The labeling function λ, with a suitable range of labels, describes (a) the messages exchanged by the lifelines and (b) the internal computational steps during the execution of the chart Ch. Finally, $\preceq$ is the partial ordering relation over the occurrences of the events in $\{E_l\}_{l \in L}$. In particular, the relation $\preceq$ or $\preceq^{Ch}$ (we put Ch as the superscript when necessary to highlight that the partial order belongs to chart Ch) is defined as follows.

(a) $\preceq_l^{Ch}$ is the linear ordering of events in E_l, which are ordered top-down along the lifeline l,

(b) $\preceq_{sm}^{Ch}$ is an ordering on message send/receive events in $\{E_l\}_{l \in L}$. If e_s is a send of message m by process p to process q, and the corresponding receive event is e_r (the receipt of the same message by process q), we have $e_s \preceq_{sm}^{Ch} e_r$.

(c) $\preceq^{Ch}$ is the transitive closure of $\preceq_L^{Ch} = \bigcup_{l \in L} \preceq_l$ and $\preceq_{sm}$, i.e. $\preceq^{Ch} = (\preceq_L^{Ch} \cup \preceq_{sm}^{Ch})^\star$.

The preceding definition of MSC is an abstract one, and does not clarify the events appearing in an MSC. The complete MSC language [15] includes several types of events: message sends and receives, local actions, lost and found messages, instance creation and termination etc. However, for simplicity of exposition, we assume that the events inside an MSC is of one of the following forms — *sends, receives and local events*. A local event can denote any terminating computation within a process, i.e., a terminating sequential program.

An MSC only denotes a single scenario in a system execution, which does not form a complete system description. The purpose of the Message Sequence Graph (MSG) is to describe the control flow *between* MSCs. Each node in an MSG is a basic MSC. We also define two special nodes ∇ and $\triangle$ which denotes the unique start and end state respectively for each MSG. The edges represent the natural operation of chart concatenation. We consider the so-called *synchronous concatenation* where for a concatenation of two charts $Ch \circ Ch'$ — all events in Ch' start only after chart Ch is finished. Two outgoing edges from a single node represent non-deterministic choice, so that exactly one of the two successor charts will be executed in an execution. Finally, an execution trace is defined to be a path from the initial state (∇) to the final state ($\triangle$) in the MSG and concatenates the sequence of MSCs encountered on the way. Example MSGs are shown in Figure 2 and will be discussed in next section.

In the following we consider acyclic MSGs where there are no loops between initial state (∇) to the final state ($\triangle$). Of course, there is always an outer loop from final state ($\triangle$) to initial state (∇) denoting periodic behavior repeated forever. We can also extend our analysis to allow arbitrary loops in between the initial state (∇) to the final state ($\triangle$), provided these (inner) loops are bounded.

3.2 Running Example

A distributed system has a number of processing elements (PEs) which are connected by shared buses. A typical distributed application consists of a collection of local computations that run on different PEs and communicate with each other through message exchanging via buses. As an example, Figure 1 shows a distributed FlexRay([11]) based Electronic control unit (ECU) network from the automotive electronics domain. There are four PEs (ECUs) and one shared FlexRay bus in the system. Two concurrently running applications, an Adaptive Cruise Controller (ACC) and

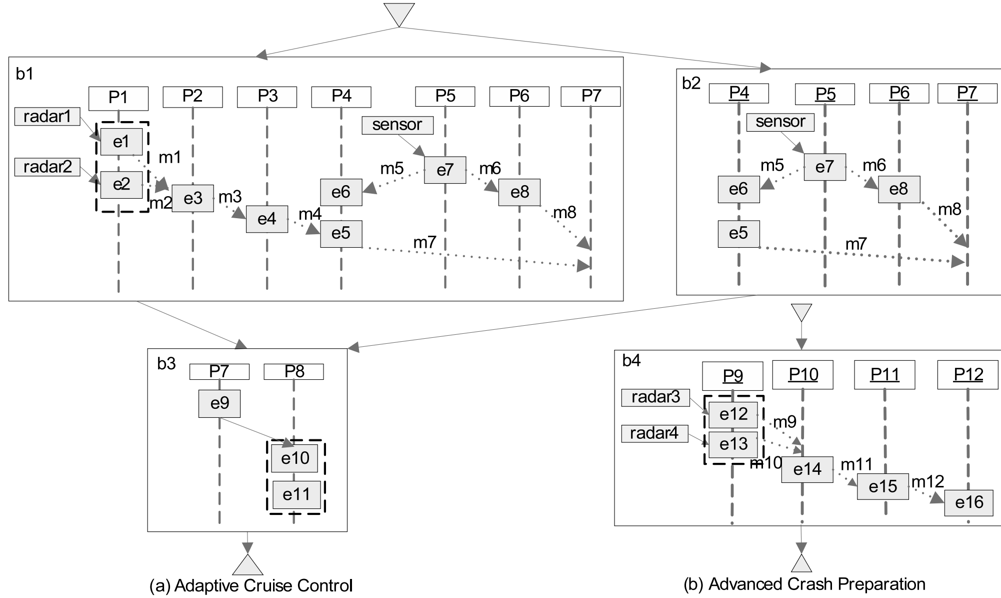

Figure 2. MSG model of the ACC and ACP applications

an Advanced Crash Preparation (ACP) system, are shown in the example.

Figure 2 shows the MSG modeling of the ACC and ACP applications in above-mentioned system. Each local computation is mapped to one local event on a lifeline (process) in a basic MSC. Note that a lifeline can represent a piece of software program which handles its corresponding event(s), or a hardware functional unit. Thus, the mapping of events onto processes can be easily obtained from the given system specification. Several processes can share a single PE, which implements its own scheduling policy (e.g. the fix priority preemptive scheduling for processes $P5$, $P7$, and $P8$ on ECU3). It may be noted here that our analysis is flexible enough to handle different scheduling policies, specified both at the MSG, and at the PE/bus level. In fact, the example shown in Figure 1 has a TDMA implemented on ECU1, fixed-priority scheduling implemented on the remaining ECUs and a FlexRay protocol implemented on the bus.

Communication between processes in an MSC can be modeled using message passing. Communication may take place via a shared bus (across PEs) or between processes running on the same PE. If the communication is done via a shared bus, we also show the message name in the MSC (e.g., $m1$ and $m2$ in MSC $b1$). We will only consider *asynchronous message passing* in our MSG modeling/analysis. Synchronous message passing, where the message sender and message receiver handshake, can be obtained as a special case of our framework. Finally, a coregion (denoted by a dashed-line box) is used to relax the strict ordering of local events along a lifeline, e.g. events $e10$ and $e11$ on process $P8$ of MSC $b3$ can be executed in any order (decided by the scheduler of $ECU3$).

In our example, the ACC application has three external triggers, namely radar1, radar2 and sensor. We assume the sensor receives input from environment twice faster than the two radars. Consequently, in the start-up stage of a complete run of the ACC application, either it receives input data from both two radars and the sensor, which corresponding to the scenario described as in MSC $b1$; or it receives only the sensor's input which triggers the scenario in MSC $b2$, and uses the old output value from the "object selection". The different system behaviors due to environment input are modeled using the indeterministic choice operation from the start of the application.

In order to perform schedulability analysis on the MSG-based system specification, we need to extend the standard MSG formalism with real-time annotations. Each MSG depicting an application is associated with the application's activation period P and deadline D. Each event is associated with the best-case and worst-case execution time (BCET/WCET) of its corresponding local computation. We assume the intra-processor communication (e.g., from $e9$ to $e10$ and $e11$ in MSC $b3$ of Figure 2(a)), as well as the local events of sending/receiving a message (implicitly denoted by the start/end of a message arrow), take zero time. Messages are labeled with their transmission time, while the actual communication time (including blocking time due to possible bus contentions) will be calculated by our analysis.

3.3 Issues in Analyzing the Model

Before proceeding to present our schedulability analysis method, let us examine the inherent difficulties in finding end-to-end delay of such an MSG model of distributed application. In order to obtain an accurate analysis for the above-mentioned model, we need to consider the effect of

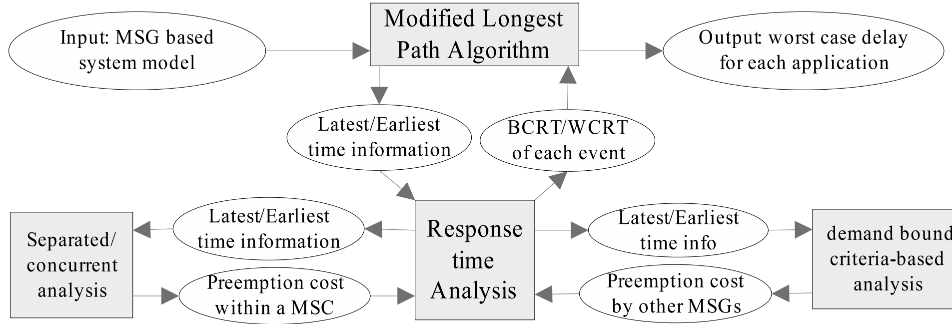

Figure 3. Overview of our analysis framework

resource contention, event dependencies, as well as conditional execution of MSCs in a MSG specification.

The possible contentions and data dependencies bring the *timing anomaly* phenomenon ([12]) when the execution times of events are not constant. In such case, the local WCET of an event may not lead to the global worst case end-to-end delay of the application. Thus, the worst-case delay of an application cannot be simply obtained by simulating the system using WCET of each individual events, over the LCM of all applications' periods.

Existing works on schedulability analysis of MSC-based specifications of distributed systems (e.g. [25] and [24]) compute the local worst-case response time for each individual event in critical instance, which assumes all events are independent. The global worst-case delay is then obtained by summing up these local worst-case response times. However, the dependencies between set of preempting events and preempted events restrict the possible preemption scenarios, which results in the critical/optimal instance assumed for worst/best case response time analysis for set of independent tasks to be too pessimistic/optimistic. For example, suppose events e_i and e_j belong to different applications in a system, and they are mapped to the same PE where e_i has a higher priority than e_j. If e_i and e_j are ready at the same time (e_i imposes the maximum interference on response time of e_j), we have the following.

- Dependency between preempting events: the successor of e_i (say e_k) cannot be ready at the same time as e_j, resulting in e_k preempting e_j fewer number of times than it could have preempted in the worst case scenario (where e_k is ready at the same time as e_j).

- Dependency between preempted events: subsequent releases of e_i may not be ready at the same time as the successor of e_j (say e_p) which also mapped on the same PE, results in less number of preemptions from e_i on e_p.

[27] proposes a schedulability analysis based on task (precedence) graph model, which captures the dependency between preempted tasks by capturing phase adjustment be-

```
1   step = 0; /*number of iterations*/
2   for (each application A_i) /*initialization*/
3     latest[∇_i^r] = earliest[∇_i^r] = 0;
4   do { /*fixed-point iteration*/
5     for (each application A_i) {
6       for (each MSC M_j of A_i in topologically sorted order) {
7         LatestTimes(M_j);
8         EarliestTimes(M_j);
9         latest[M_j^f] = max_{e∈M_j} {latest[e^f]};
10        earliest[M_j^f] = max_{e∈M_j} {earliest[e^f]};
11        for (each successor MSC M_k of M_j) {
12          latest[M_k^r] = max(latest[M_j^f], latest[M_k^r]);
13          earliest[M_k^r] = min(earliest[M_j^f], earliest[M_k^r]);
14        }
15      }
16      /*worst case delay of A_i*/
17      wcrt[A_i] = latest[△_i^f];
18    }
19    step++;
20  } while(any time instance changed and step < limit);
```

Figure 4. Delay estimation algorithm.

tween a preempting task and preempted tasks. We adopt the analysis framework from [27] and extend it to consider (a) the dependencies between preempting events, and (b) control flow, in particular non-deterministic branches, among the MSCs in an MSG. In our case, an event e in an application A can be preempted by events in a different application A'. Conditional executions of events in A' should be exploited to avoid gross overestimation of the preemption cost of e. This is done in our analysis by adapting ideas from the recurring real-time task model in real-time systems literature [3], which allows for conditional branches.

4 Analysis Overview

Figure 3 shows the overview of our feasibility analysis framework for MSG-based system models. Given a set of MSGs each representing a real-time distributed application and annotated with required timing information, our analysis will return an upper bound on the end-to-end delay for each MSG. We present our analysis method in two levels. In this section, we present the top-level analysis for computing end-to-end delay of MSG-based distributed real-time applications, which is a modified longest path algorithm adopted from [27] with necessary modifications to handle MSC concatenation and conditional branching in the MSG model. In the next section, we will present response time analysis of

```
1   LatestTimes(MSC) {
2   /*compute latest[e_i^r] and latest[e_i^f] for all e_i in MSC*/
3    for (each source event e_i in MSC) /*initialize*/
4      latest[e_i^r] = latest[MSC^r];
5    for (each event e_i respecting the partial order ⪯^MSC) {
6      w_i = WCRT of e_i /* See Section 5 for details */
8      latest[e_i^f] = latest[e_i^r] + w_i;
9      for (each immediate successor e_k of e_i) {
10       if (latest[e_k^r] < latest[e_i^f])
11         latest[e_k^r] = latest[e_i^f];
12     }
13   }
14  }
```

Figure 5. The LatestTimes algorithm.

individual events.

To facilitate the analysis, four time instances are defined for each event e and MSC M in a MSG.

- earliest ready time ($earliest[e^r]$, $earliest[M^r]$)
- latest ready time ($latest[e^r]$, $latest[M^r]$)
- earliest finish time ($earliest[e^f]$, $earliest[M^f]$), and
- latest finish time ($latest[e^f]$, $latest[M^f]$).

Figure 4 presents the top-level iterative algorithm for computing worst case end-to-end delay ($wcrt[A_i]$) for each application A_i. It uses information of individual events' response times to generate latest and earliest time instances, which in turn will be used to refine the results of the response time analysis in the next iteration. The algorithm terminates when (a) no time instance for any of the events is changed (the fixed point is reached), or (b) the maximum number of iteration steps are executed. The top level framework captures the dependencies between individual MSCs. Since exactly one of the conditional edges are taken for each branch, the earliest ready time of a MSC is set to be the minimum value of the earliest finish times of its predecessors, while the latest ready time of a MSC is set to be the maximum of the latest finish times of its predecessors. The algorithm begins with a very coarse approximation for the start and completion times of the events, and the worst/best case delay it may suffer. The results are refined in each iteration based on the information computed in last iteration. The algorithm is safe in the sense that it never produces underestimation for the worst case delays or over-estimation for the best case. For an application A_i with deadline D_i, our analysis considers it schedulable if $wcrt[A_i] \leq D_i$.

The `LatestTimes` calculation, shown in Figure 5, is similar to the `LatestTimes` algorithm in [27]. Basically, the algorithm in Figure 5 uses a modified longest-path algorithm to take into account data dependencies within a single MSC. Based on dependencies between events of the MSC being analyzed, the main purpose of the algorithm is to update the latest ready and finish times for each event. This updating is independent of the resource scheduling policies on the PEs. The scheduling policy is only taken into account in the calculation of the WCRT of an event; this calculation is elaborated in the next section.

Given the `LatestTimes` algorithm, we can easily transform it in to the `EarliestTimes` algorithm, which updates the earliest ready and finish times by calculating the best-case response time for each event.

5 Response Time Calculation

The procedure for computing the earliest/latest ready and finish times of MSC events, as discussed so far, only provides an algorithmic framework. In particular, it depends on worst case and best case response time (WCRT/BCRT) estimates of individual events inside MSCs. We now elaborate the WCRT/BCRT calculation of MSC events. Clearly, this will require us to consider the scheduling policy inside the PEs on which these events are executed. We use fixed priority preemptive scheduling for our response time calculation in this section.

The standard WCRT calculation for fixed-priority scheduling of independent periodic tasks is given by the following recursive equation [17].

$$w_i^{n+1} = c_i + \sum_{t_j \in hp(t_i)} c_j \cdot \lceil \frac{w_i^n}{P_j} \rceil \tag{1}$$

Here w_i, c_i, and P_i are the response time, computation time, and period for task t_i respectively. The set $hp(t_i)$ denotes the set of higher priority tasks mapped to the same PE as t_i. The fixed point computation starts with $w_i^0 = c_i$, and terminates when the response time calculated in $n + 1$th iteration (w_i^{n+1}) equals to the value in previous iteration (w_i^n). Equation 1 computes the WCRT of a task t_i in its *critical time instance* (i.e. all higher priority tasks are ready when t_i is ready).

The BCRT calculation is proposed in [22] as

$$b_i^{n+1} = c_i + \sum_{t_j \in hp(t_i)} c_j \cdot (\lceil \frac{b_i^n}{P_j} \rceil - 1) \tag{2}$$

for the same setting. It is based on the best case phasing (or optimal instance) where t_i finishes simultaneously with the release of all its higher priority tasks.

However, in our distributed MSC-based system model, we can obtain far more accurate WCRT/BCRT estimates by taking into consideration the dependencies between preempting events as discussed in Section 3.3. We divide the worst and best preemption cost on the execution of any event e_i as follows — (a) preemption on e_i by other events in the same application (denoted as WS_i and BS_i), and (b) preemption on e_i by events from other applications (denoted as WD_i and BD_i), respectively. Thus, our WCRT and BCRT equations are given as follows.

$$w_i^{n+1} = c_i + WS_i^n + WD_i^n \tag{3}$$

$$b_i^{n+1} = c_i + BS_i^n + BD_i^n \tag{4}$$

We now elaborate the calculation of these four quantities — WS_i, BS_i, WD_i, BD_i.

5.1 Preemption within an MSC

Equation 1 and 2 assume deadline less than or equal to period for all tasks ($D \leq P$). This guarantees that, for a schedulable task set, a task instance will not get delayed by any its previous instances. In our analysis, we also assume that deadline is less than or equal to period for all the applications being analyzed. Thus, to show that application A is schedulable ($wcrt(A) \leq D$), interference from events in previous instances of A need not be considered for the critical and optimal time instances. Suppose e_i and e_j are events in the same application A, and there is no dependency between them (neither $e_i \preceq e_j$ nor $e_j \preceq e_i$). For e_j to possibly preempt e_i, the events e_i, e_j cannot be events in different MSCs of the MSG model of A, since MSCs in a MSG are synchronously concatenated. Moreover, e_j may preempt e_i at most once owing to assumption that deadline is less than or equal to period for all the applications.

Furthermore, for event e_j to preempt an event e_i in the same MSC M, there must be overlap between their execution intervals. Let event $NCP(i,j)$ be the nearest common predecessor event for e_i and e_j in M. If such a predecessor event does not exist, we set $NCP(i,j)$ to be the start of M. Using the notion of NCP, we define the following quantities.

- smallest time interval between $NCP(i,j)$ finishing and e_i becoming ready

  $$SFR_i^{NCP(i,j)} = earliest[e_i^r] - earliest[NCP(i,j)^f]$$

  which corresponds to the scenario that all events on path from $NCP(i,j)$ to e_i execute in their BCRT.

- largest time interval between $NCP(i,j)$ finishing and e_i becoming ready

  $$LFR_i^{NCP(i,j)} = latest[e_i^r] - latest[NCP(i,j)^f]$$

  which corresponds to the scenario that all events on path from $NCP(i,j)$ to e_i execute in their WCRT.

- smallest time interval between $NCP(i,j)$ finishing and e_i finishing,

  $$SFF_i^{NCP(i,j)} = earliest[e_i^f] - earliest[NCP(i,j)^f]$$

- largest time interval between $NCP(i,j)$ finishing and e_i finishing,

  $$LFF_i^{NCP(i,j)} = latest[e_i^f] - latest[NCP(i,j)^f]$$

Executions of two events e_i and e_j from the same vertex are guaranteed to be separated in one execution of the MSG if and only if

$$separated(i,j) = e_i \preceq e_j \ \vee \ e_j \preceq e_i \ \vee (LFF_i^{NCP(i,j)} \leq SFR_j^{NCP(i,j)}) \vee \ (LFF_j^{NCP(i,j)} \leq SFR_i^{NCP(i,j)})$$

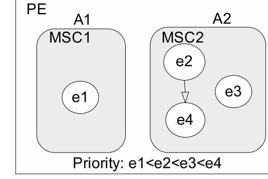

Figure 6. Projection of Events on same PE.

evaluates to true, i.e. either there is a dependency between e_i and e_j (as per the partial order for the MSC), or e_i always finishes before e_j releases, or vice versa. Note that the instances of e_i and e_j involved in the preemption belong to the same run of the MSG. Thus, the above intervals should be measured w.r.t their nearest common predecessor event (instead of start of the MSG ∇), which gives a much more accurate estimation.

Finally, the worst case preemption cost imposed on event e_i by events from same application can be calculated as follows: let c_j^u be the WCET of event e_j.

$$WS_i = \sum \{ \, c_j^u \mid contend(j,i) \wedge \neg separated(i,j) \}$$

where $contend(j,i)$ is true if and only if the events e_j and e_i are mapped to the same PE and e_j has higher priority than e_i (as per the scheduling policy of the PE).

For the BCRT calculation of e_i, we find the events e_j that are guaranteed to be ready during e_i's execution.

$$concurrent(i,j) = \neg(e_i \preceq e_j) \wedge \neg(e_j \preceq e_i) \wedge (LFR_i^{NCP(i,j)} \leq SFR_j^{NCP(i,j)}) \wedge \ (LFR_j^{NCP(i,j)} \leq SFF_i^{NCP(i,j)})$$

The best case preemption cost imposed on event e_i by events from same application can be calculated as follows: let c_j^l be the BCET for event e_j.

$$BS_i = \sum \{ \, c_j^l \mid contend(j,i) \wedge concurrent(i,j) \}$$

5.2 Preemption by Other Applications

In this section, we will briefly present how to compute WCRT and BCRT of individual events accurately using demand bound approach [3] and earliest / latest time information. An expanded version of this section can be found in our technical report [16]. Let G be a recurring real-time task graph, the **request bound function**, $G.rbf(t)$, accepts a non-negative real number t, and returns the maximum cumulative execution requirement by releasing of nodes in G that have their ready times within any time interval of duration t. To utilize the request bound calculation presented in [3], our goal is to construct, for each application A in the system, a preemption graph $PG_{e_i}^A$ that captures dependencies and timing information for all events in A that can preempt event e_i.

We start by building a *preemption chain* for a single MSC in the preemption application. Considering a scenario where an event $e1$ in application $A1$ get preempted by

```
1   ConstructPC(ps_{e_i}^M){
2     /*initialize*/
3     PC = empty; /*the preemption chain*/
4     for(each e_j in ps_{e_i}^M as the partial order ⪯^M of MSC M){
5       create a node n containing e_j;
6       /*insert n into PC*/
7       if(PC is empty) PC.insert(n);
8       else if(¬separated(n, source(PC))
9         merge(n, source(PC)); /*merge n into source(PC)*/
10      else if(earliest[n^r] > earliest[source(PC)^r])
11        insertAfter(n,source(PC)); /*insert n after source(PC)*/
12      else /*n is ready before source(PC)*/
13        insert n as the source node of PC;
14    }
15    for(each edge E(n, n_1) in PC)
16      W(n, n_1) = earliest[n_1^r] − earilist[n^r];
17    W(sink(PC), source(PC)) = P(M) − latest[sink(PC)^r]
18              +earliest[source(PC)^r];
19  }
```

```
1   merge(n,n_1){
2     C^u(n_1) = C^u(n_1) + C^u(n); //update computation time
3     earliest(n_1^r) = min{earliest(n_1^r), earliest(n^r)};
4     latest(n_1^r) = min{latest(n_1^r), latest(n^r)};
5     earliest(n_1^f) = max{earliest(n_1^f), earliest(n^f)};
6     latest(n_1^f) = max{latest(n_1^f), latest(n^f)};
7   }

1   insertAfter(n, n_1){
2     if(succ(n_1) not exist)
3       insert n as the sink node of PC;
4     else if (¬separated(n, succ(n_1))
5       merge(n,succ(n_1));
6     else if(earliest[n^r] > earliest[succ(n_1)^r])
7       insertAfter(n,succ(n_1));
8     else
9       insert n between n_1 and succ(n_1);
10  }
pred(n)/succ(n) denote immediate predecessor,
and successor of n in the preemption chain.
```

Figure 7. Constructing a preemption chain.

evnets in MSC $MSC2$ of another application $A2$. Figure 6 gives the projection of the events in $MSC2$ executed on the same PE as $e1$, including dependencies and priority assignments for the fixed-priority preemptive scheduling on PE. Note that there might be events in between e_i and e_j, which are executed on other PEs. Assume that the set of events within an MSC M that can preempt an event e_i is denoted as $ps_{e_i}^M$. For instance, in the example given in Figure 6 we have $ps_{e1}^{MSC1} = \{e2, e3, e4\}$.

Preempting events in $ps_{e_i}^M$ may either have dependencies (e.g. $e2$ and $e4$) or execute concurrently with other events (e.g. $e3$). In order to explore number of preemptions they may impose on a particular preempted event ($e1$), we first construct a *preemption chain* to capture the possible release times of events in $ps_{e_i}^M$. A *preemption chain* $PC_{e_i}^M = \{\hat{N}, \hat{E}\}$, is a sequence of nodes $n \in \hat{N}$, and each directed edge $E(n_1, n_2) \in \hat{E}$ is labeled with weight $W(n_1, n_2)$ representing the minimum time interval between release times of nodes n_1 and n_2. A node n contains a set of events from $ps_{e_i}^M$. Similar to our handling of events in Section 5.1, four time instances $earliest[n^r]$, $latest[n^r]$, $earliest[n^f]$, $latest[n^f]$ are defined for each node n in the preemption chain. The upper and lower bound computation time of a node n are denoted as $C^u(n)$ and $C^l(n)$ respectively; these estimates are obtained from summing up the WCET/BCET of the events in node n.

The algorithm to construct *preemption chain* is shown in Figure 7. Events that *may* execute concurrently are grouped into one node - the release of any of these events may cause all of them to preempt e_i in the worst case. Thus, a node is ready when any of its events is ready (see line 4 of the `merge` procedure in Figure 7). Intuitively, for the WCRT calculation of e_i, events in a node n will have the same number of preemptions on the preempted event e_i in the worst case if their execution intervals are not *separated* (as defined in Section 5.1). In Figure 6, suppose $e2$ executes between time interval $[3, 6]$, and $e3$ executes between [4,7]. Then every time $e2$ preempts $e1$, it is also possibly for $e3$ to preempt $e1$ before $e1$ resume its execution. Thus, when considering the worst-case preemption scenario, we can group $e2$ and $e3$ into a single node n_1, which has an earliest ready time of 3, and execution time of $c_2^u + c_3^u$. On other hand, suppose $e4$'s earliest ready time is 10. In this case, $e1$ could finish its execution in the interval between — (a) $e2$ and $e3$ finishing execution, and (b) $e4$ getting released. Thus, number of preemptions caused by node n_1 and the node containing $e4$ could be different. Finally, the distance between two nodes will be the minimum time elapsed between their ready time (line 16 of `constructPC`).

A *preemption graph* for a full-fledged MSG model of application A can be constructed by connecting individual preemption chains for each MSC with conditional branches defined by the MSG. If M' is a successor MSC of M in the MSG for application A, we create a directed edge $E(M, M')$ from $sink(PC_{e_i}^M)$ to $source(PC_{e_i}^{M'})$ with weight of

$$earliest[source(PC_{e_i}^{M'})^r] - earliest[sink(PC_{e_i}^M)^r]$$

that is, the minimum distance between ready time of $sink(PC_{e_i}^M)$ and $source(PC_{e_i}^{M'})$. Our *preemption graph* has a similar semantics as the graphical representation of a recurring real-time task. Thus, our problem of finding $PG_{e_i}^A.rbf(w_i^n)$, the maximum cumulative execution requirement by releasing of nodes in $PG_{e_i}^A$ over e_i's nth iteration response time w_i^n, can be converted to the problem of computing the *request bound function* of a recurring real-time task over a given time interval. In Equation 3, we have

$$WD_i^n = \sum_{A \ s.t. \neg(e_i \in A)} PG_{e_i}^A.rbf(w_i^n)$$

The best case preemption cost from other application BD_i^n can be calculated similarly with necessary changes in *preemption graph* construction and *rbf* computation.

	ACC	ACP
Proposed analysis	48 ms	95 ms
Saksena and Karvelas [24]	60 ms	110 ms

Table 1. End-to-end delay (from sensor/radar to actuator) for the ACC and ACP applications shown in Figure 1.

6 Case Study

6.1 Experimental Setup

In this section, we illustrate our analysis method by applying it to a setup from the automotive electronics domain. The system architecture of a FlexRay-based ECU network and two distributed applications (ACC and ACP) were presented in Section 3.2. The underlying system architecture consists of four ECUs communicating via a shared FlexRay bus, as shown in Figure 1. We assume ECU1 implements a Time Division Multiple Access (TDMA) scheduler, while the remaining three ECUs use preemptive fixed-priority scheduling.

Communication on the FlexRay bus takes place in periodic cycles (or bus cycles), where each cycle is partitioned into a static (ST) and a dynamic (DYN) segment. The ST segment is divided into several fixed static slots, and messages can only be sent during their allocated slots. The DYN segment implements an event-triggered bus protocol based on fixed priority scheduling. Further details of the FlexRay communication protocol can be found in [11, 21]. We compute the best and worst response times for each FlexRay message between its ready time (generated by the sender) and finish time (available to the receiver). For a ST message m_i with a transmission time of C_i, we have

$$b_i = C_i; \;\; w_i^{n+1} = C_i + T + St(w_i^n) \times T;$$

where T is the length of the bus communication cycle, and $St(w_i^n)$ is the number of occurrences of higher priority ST messages using the same ST slot as m_i, within a time interval of length w_i^n. For a DYN message m_i, the response time is calculated as

$$b_i = C_i; \;\; w_i^{n+1} = C_i + T + Dyn(w_i^n) \times T;$$

where $Dyn(w_i^n)$ is the number of occurrences of higher priority DYN messages m_j within w_i^n time units, such that m_j and m_i are not allowed to be transmitted in the same bus cycle (due to size restriction of the DYN segment).

The two applications receive data periodically from the external environment (i.e. radars and sensors), and are required to complete before the next arrival of their input data (i.e. deadlines are equal to periods). We assume input data received by the four radars and the sensor every 100 ms and 50 ms respectively. Thus, the period/deadline of the ACC and ACP applications are 50 ms and 100 ms respectively. Furthermore, we assume the FlexRay bus has a communication cycle of 5 ms. The detailed BCET, WCET, and priority for each individual events are listed in our technical report [16].

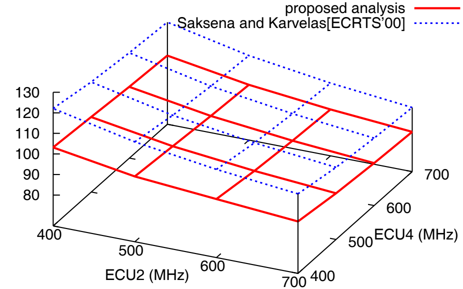

Figure 8. Delay bound for ACP obtained using our proposed analysis and the technique presented in [24].

6.2 Results

In this section we present the results obtained by analyzing the setup described above using our proposed analysis technique. Further, we compare these results with those obtained from response time analysis techniques for UML-based system models of multi-threaded implementations of objects/processes [24], where the dependency between events are not considered. Our proposed analysis as well as the one in [24] are *safe* (*i.e.*, if analysis returns "schedulable" then it is guaranteed to be so).

Table 1 shows the results obtained using the two techniques when all the ECUs run at a clock frequency of 500 MHz. Note that while our analysis returns a "schedulable" result (i.e. the end-to-end delays of the two applications are lower than the sampling periods of the radars/sensors that feed data into them), the analysis proposed in [24] returns "not schedulable".

Figure 8 shows the estimated end-to-end delays of the ACP application using the two analysis techniques when the clock frequencies of ECU2 and ECU4 are chosen between 400 to 700 MHz at a scale of 100 MHz, with the execution times of the associated tasks being scaled accordingly. The frequencies of the remaining ECUs are kept at 500 MHz. Clearly, the delay estimates obtained using our technique are considerably tighter than those obtained using [24] (12% to 16% improvements). Such tighter estimates immediately translate into better resource dimensioning and system design. In Figure 8 the clock frequencies are scaled in steps of 100 MHz. It may be noted that our analysis returns "not schedulable" only for two different combinations of frequency settings, viz. (ECU2:400 MHz, ECU4: 500 MHz) and (ECU2:400 MHz, ECU4: 400 MHz), from our underlying design space. On the other hand, the analysis proposed in [24] marks a much larger portion of the design space as "not schedulable". In particular, only (ECU2:700 MHz, ECU4: 600 MHz) and (ECU2:700 MHz, ECU4: 700 MHz), are estimated to be feasible clock frequencies. In our technical report [16], some reasons behind

our tighter delay estimates for this application are discussed in details.

7 Concluding Remarks

In this paper, we have presented a schedulability analysis technique for MSG-based modeling of distributed real-time systems. This makes schedulability analysis techniques accessible to formal system specifications such as MSCs which have long been studied in the context of the Unified Modeling Language (UML). We show the utility of our modeling and response-time analysis with real-life applications from the automotive electronics domain. Our experiments show that our method can consider the event dependencies as prescribed by an MSC partial order as well as sequencing and branching between MSCs in a MSG to produce tight response time estimates of MSG-based system models.

While we focus on synchronously concatenated MSCs within a MSG, our proposed analysis framework can be extended to asynchronous concatenation between MSCs, where events across MSCs are synchronized at process-level instead of MSC-level for synchronous concatenation. For synchronous concatenation, we needed to keep the latest / earliest time for start and finish of each MSC. On the other hand, we will need to track the latest / earliest time instances for each process across MSCs if MSCs are concatenated asynchronously.

Another popular mechanism of presenting a collection of MSCs is called high-level MSCs (HMSCs) [15], where MSCs are grouped together in a hierarchical manner. To apply our schedulability analysis to a HMSC-based system model, we could simply flatten the HMSC to an MSG and employ the techniques described in this paper. However, such an analysis would not properly exploit the hierarchical structure of the HMSC. We are currently trying to build such a hierarchical analysis for HMSCs.

Acknowledgments This work was partially supported by NUS research grants R252-000-286-112 and R252-000-321-112.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Realizability and verification of MSC graphs. In *ICALP*, 2001.

[2] R. Alur and M. Yannakakis. Model checking message sequence charts. In *CONCUR*, 1999.

[3] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Systems*, 24(1), 2003.

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1), 1999.

[5] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2, 1990.

[6] A. Burns. *Advances in Real-Time Systems*, chapter Preemptive priority based scheduling: An appropriate engineering approach, pages 225 – 248. Prentice-Hall, 1994.

[7] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[8] A. Chakrabarti et al. Verifying quantitative properties using bound functions. In *CHARME*, 2005.

[9] K. Chatterjee et al. Compositional quantitative reasoning. In *QEST*. IEEE Computer Society, 2006.

[10] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *DATE*, 1998.

[11] The flexray communications system specifications, ver 2.1. `www.flexray.com`, 2005.

[12] R. Gerber, W. Pugh, and M. Saksena. Parametric dispatching of hard real-time tasks. *IEEE transactions on computers*, 44(3), 1995.

[13] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering*, 20(1), 1994.

[14] D. Harel and P. Thiagarajan. *UML for Real: Design of Embedded Real-time Systems*, chapter Message Sequence Charts. Kluwer, 2003.

[15] ITU-T. 120: Message sequence chart (msc). *ITU-T, Geneva*, 1996.

[16] L. Ju, A. Roychoudhury, and S. Chakraborty. Schedulability analysis of MSC-based system models. Technical Report TR20/07, NUS, 2007. `http://www.comp.nus.edu.sg/~abhik/pdf/MSGsched-TR.pdf`.

[17] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, 1990.

[18] S. Manolache, P. Eles, and Z. Peng. Schedulability analysis of applications with stochastic task execution times. *ACM Transactions in Embedded Computing Systems (TECS)*, 3(4), 2004.

[19] A. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10), 1997.

[20] P. Pop, P. Eles, and Z. Peng. Schedulability analysis for systems with data and control dependencies. In *ECRTS*, 2000.

[21] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing Analysis of the FlexRay Communication Protocol. *ECRTS*, 2006.

[22] O. Redell and M. Sanfridson. Exact best-case response time analysis of fixed priority scheduled tasks. In *ECRTS*, 2002.

[23] M. Reniers. *Message Sequence Chart: Syntax and Semantics*. PhD thesis, Technical University of Eindhoven, Netherlands, 1999.

[24] M. Saksena and P. Karvelas. Designing for schedulability: Integrating schedulability analysis with object-oriented design. In *ECRTS*, 2000.

[25] F. Slomka, J. Zant, and L. Lambert. Schedulability analysis of heterogeneous systems using performance message sequence chart. In *CODES*, 1998.

[26] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2), 1994.

[27] T. Yen and W. Wolf. Performance Estimation for Real-Time Distributed Embedded Systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(11), 1998.