SCHEDULABILITY ANALYSIS OF REAL-TIME MULTI-FRAME CO-SIMULATIONS ON MULTI-CORE PLATFORMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUHAMMAD UZAIR AHSAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

JANUARY 2020

Approval of the thesis:

SCHEDULABILITY ANALYSIS OF REAL-TIME MULTI-FRAME CO-SIMULATIONS ON MULTI-CORE PLATFORMS

submitted by **MUHAMMAD UZAIR AHSAN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Ali H. Doğru
Computer Engineering, METU _____

Prof. Dr. Halit Oğuztüzün
Computer Engineering, METU _____

Prof. Dr. Ahmet Coşar
Computer Engineering, THK University _____

Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering, METU _____

Assoc. Prof. Dr. Kayhan İmre
Computer Engineering, Hacettepe University _____

Date: 17/01/2020

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Muhammad Uzair Ahsan

Signature :

ABSTRACT

SCHEDULABILITY ANALYSIS OF REAL-TIME MULTI-FRAME CO-SIMULATIONS ON MULTI-CORE PLATFORMS

Ahsan, Muhammad Uzair
Ph.D., Department of Computer Engineering
Supervisor: Prof. Dr. Halit Oğuztüzün

January 2020, 110 pages

For real-time simulations, the fidelity of simulation does not depend only on the functional accuracy of simulation but also on its timeliness. It is helpful for developers if we can analyze and verify that a simulation will always meet its timing requirements while keeping an acceptable level of accuracy. Abstracting the simulated processes simply as software tasks allows us to transform the problem of verifying timeliness into a schedulability analysis problem where tasks are checked if they are schedulable under real-time constraints or not. In this paper we extended a timed automaton based framework due to Fersman and Yi for schedulability analysis of real-time systems, for the special case of real-time multi-frame co-simulations. We found that there are some special requirements posed by multi-frame simulations which necessitate changes and improvements in the existing framework. We made the required theoretical extensions to the framework and then implemented our extended framework in *UPPAAL*, a tool for modeling, simulation and verification of real-time systems modeled as timed-automata, and tested on an example.

Keywords: Schedulability Analysis, Real-time Simulations, Co-simulations, Task Automaton , Timed Automaton

ÖZ

ÇOK-ÇEKİRDEKLİ PLATFORMLARDA GERÇEK ZAMANLI ÇOK-ÇERÇEVELİ EŞ-BENZETİM İÇİN ÇİZELGELENEBİLİRLİK ÇÖZÜMLEMESİ

Ahsan, Muhammad Uzair
Doktora, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Halit Oğuztüzün

Ocak 2020 , 110 sayfa

Gerçek-zamanlı benzetimlerde, benzetimin doğruluğu sadece işlevsel doğruluğa değil benzetim adımlarının zamanında tamamlanmasına da bağlıdır. Benzetimin zamanlama gereksinimlerini, kabul edilebilir doğruluk seviyesini tutturarak, her durumda karşıladığını, bir çözümleme işlemi sonucunda göstermek, geliştiricilere yardımcı olacaktır. Bu çalışmada benzetilen süreçler yazılım görevleri olarak soyutlanmıştır. Bu bizim gerçek-zaman kısıtlarını sağlama problemini çizelgelenebilirlik problemine dönüştürmemizi sağlar. Bu problem, görevlerin gerçek-zaman kısıtları altında çizelgelenebilir olup olmadığına karar verilmesini içerir. Bu çalışmada, Fersman ve Yi tarafından ortaya konulmuş olan, zaman-devingeni tabanlı bir çizelgelenebilirlik çerçevesini, çok çerçeveli eş-benzetimler için genişletmekteyiz. Bu çalışmada, çok-çerçeveli benzetimlere özel gereksinimleri karşılamak üzere mevcut çerçevede genişletmeler yapıldı. Gereken kuramsal genişletmeler yapıldıktan sonra yeni çerçeve; zaman-devingeni olarak modellenen gerçek-zamanlı sistemler için bir modelleme, benzetim ve doğrulama aracı olan UPPAAL kullanılarak gerçekleştirildi ve

bir örnek üzerinde sınandı.

Anahtar Kelimeler: Servis Odaklı Mimari, Koreografi Modeli, Koreografi Dili, Değişkenlik Yönetemi, Model Tabanlı, Yazılım Üretim Bantları

To my family and people who are reading this page

ACKNOWLEDGMENTS

SI would like to thank my supervisor Professor Dr. Halit Oğuztüzün for his continuous support, encouragement and guidance. It was a great pleasure and honor to work with him for all these years. This work would not have been possible without his mentoring, experience and knowledge.

There are numerous other people who helped me in one way or the other during my studies in METU. First of all my teachers at METU who enriched my knowledge greatly and which include wonderful people like Dr. Ayşenur Birtürk, Assoc. Prof. Dr. Murat Manguoğlu, Prof. Dr. Ismail Hakkı Toroslu, Prof. Dr. Ahmet Coşar, Prof. Dr. Sibel Tari and Dr. Umut Durak to name a few. I would especially like to thank Dr. Sibel Tari whose encouragement and praise were a source of pride and self-belief that helped me cross the line eventually. I am also thankful to Assoc. Prof. Dr. İlkay Yavrucuk from the Department of Aerospace Engineering, METU for providing the technical details about the helicopter simulation used as a case study in this thesis. I would also thank my lab mates and all other Turkish and Pakistani friends whose friendship, love and help made my stay very comfortable and pleasant.

I would also like to express my gratitude to Government of Pakistan who provided me the opportunity to study in Middle East Technical University, one of the leading university in the region.

Me and my family are indebted to the Turkish nation in general and our Turk neighbours at Öveçler Mahallesi in particular whose friendly attitude and love for Pakistanis was a source of unlimited joy and happiness. The love that me, my wife and children experienced here has become an indelible part of our memory which we shall cherish all our lives. Thank you all.

Lastly, I would like to thank my family for supporting and believing in me. It would have been near impossible to survive this long journey without their help and support.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

CH	Cancellation Handler
DAG	Directed Acyclic Graph
DL	Deadline
FMI	Functional Mockup Interface
FSA	Finite State Automaton
FY	Fersman & Yi
HWIL	Hardware-in-the-Loop
RT	Real-Time / Response Time
ST	Simulation Time
SV	Schedulability Verifier
TA	Timed Automaton / Task Automaton
WCCT	Worst Case Cancellation Time
WCET	Worst Case Execution Time

CHAPTER 1

INTRODUCTION

Over the period of last several years, Computer simulation has become one of the key methods for the verification and validation of functional as well as timing properties of engineering systems, especially the ones which are very large and distributed in nature. A Computer simulation provides convenient and cost effective way to safely test and analyze critical engineering systems even under conditions that are prohibitively hazardous or costly to be applied in real life. For example, analyzing the performance of an aircraft during a storm, or testing the effects of leak in a nuclear reactor. Moreover, traditional integrated system testing is possible only after the system is built completely, but computer simulation enables integrated testing of a system even during its development phases by using simulated models in place of sub-systems that are not built yet. Hence, simulations have become an integral part of the design and development process of modern engineering systems. Computer simulations are not limited to engineering systems only and are commonly used in a variety of social, economical and scientific disciplines to analyze a system or a process, study its evolution in time and evaluate the possible outcomes.

To simulate any system or process, the primary requirement is development of a simulation / mathematical model for the system or process that is being simulated. The model defines dynamic behavior of the simulated system using mathematical equations. The simulation model is then executed by a simulation engine generating the behavior of the actual system or process. We may refer to these simulation models simply as models, system models or sub-system models wherever the meaning is clear from the context. Typically, the term system model is used for models that contain several sub-system models within themselves. Since every simulation is required

to be functionally accurate to certain extent, known as the level of fidelity, it is the responsibility of the simulation model designer to develop a model that is functionally and logically accurate enough to achieve the desired level of fidelity. There is a special set of computer simulations, however, which have an additional requirement to meet; a requirement which does not depend upon the accuracy of the simulation model but rather on the performance of the simulation engine. This additional requirement is that of generating the correct timed behavior of the actual system being simulated. This additional time constraint may be imposed for the purpose of verifying the timing correctness of the simulated system or it may be an inherent requirement arising because of some particular simulation setup; for instance a Hardware-in-the-Loop (HWIL) simulation where some of the components in the simulation are actual hardware components. These simulations are called "real-time simulations" and their fidelity does not only depend upon the functional accuracy but also on the *timeliness* of the simulation execution. To explain this *timeliness* property, we need to define two more terms: the *Simulation Time* and *Wall-clock Time*.

Simulation Time Simulation time refers to a virtual time that is maintained by the simulation software / engine itself. Since computer simulations proceed in small time steps, the simulation-time is also incremented in discrete steps. The software maintains this time for each sub-system model as well for the whole system. Individual model's simulation time is incremented with each time step corresponding to the step size, while system simulation time usually follows sub-system models' simulation time and reaches a certain time instant only when all the sub-systems have either reached or surpassed that instant. The rate of increment of simulation time is usually not constant.

Wall-clock Time Wall-clock time also called real-time is the physical time that is elapsed during which a simulation is running. Since it is the actual time, it increases continuously with a constant rate.

Now, the timeliness property of real-time simulations mentioned above can be explained as the property which dictates that simulation time of a system model as well as its constituent sub-system models must be greater than or equal to the wall-clock

time at least at instants where the model has to interact with an entity external to itself. Let us call this constraint between simulation-time of models and the wall-clock time as the *real-time constraint*. The simulation- and wall-clock times referred here are measured with respect to start of simulation instant when both times are taken as zero. Since the progression of time in a simulation is managed by the simulation software itself and not by the system model that is being simulated, real-time simulation is possible only if the simulation software has the capability to do it.

If we can somehow prove that, during a simulation run of a system, the execution of each sub-system model is carried out in a way such that none of them ever violates its real-time constraint then such a proof enhances our confidence in the simulation results and so helps us in making better design choices which results in more efficient and safer systems. The process of obtaining such a proof is termed as schedulability analysis.

In this work, we developed a schedulability analysis framework for the case of real-time multi-frame co-simulations: a special kind of real-time simulations which are distributed in nature with sub-system models being possibly simulated on distinct machines. Real-time multi-frame co-simulations are introduced in detail in section 1.1. Since schedulability analysis is only concerned with timing correctness, determining functional accuracy of a simulation run is beyond the scope of this work. Our work include presentation of a model for real-time multi-frame co-simulations, development of a schedulability analysis framework for the presented model based on an existing timed-automata based framework for schedulability analysis of real-time systems [22, 24] , and finally implementation and demonstration of the new framework.

In the remainder of this chapter, sections 1.1 and 1.2 are dedicated to definition and explanation of some of the terms and concepts that formed the basis of our work followed by a description of the schedulability analysis problem and our approach to its solution in sub-section 1.3. The chapter ends with section 1.4 describing some related works that have been done on similar problems.

1.1 Real-time Multi-frame Co-Simulation

Consider an actual engineering system that is composed of physical as well as various software components.The distinct physical components or processes in this system naturally run independently and concurrently without the need of any computing resources. The software components or tasks of the same system, however, require computing resources and need to be scheduled such that they can monitor and/or control the physical components of the system in real time. Now consider a real-time simulation counterpart of the same engineering system. In this simulated environment, majority of physical components or processes, being simulated using respective simulation models, are just software processes or tasks which require processors and computing resources to run. Let us call these software tasks as *Simulation Software Tasks* as their existence is only possible in a computer simulation environment and not in actual systems.

To understand the nature of these simulation software tasks, note that a simulation model provides the differential or difference equations that describe the dynamic behavior of the modeled system. We know that a simulation is run one small step at a time and that all the continuous state variables of every sub-system model in the simulation need to be updated at each such step by a numerical solver or integrator. For each simulation step, the numerical solver uses the values of state variables and possibly theirs derivatives at the start of the step to calculate the state variable values for the time instant that marks the end of that step. These software processes that perform numerical integration of the mathematical simulation models are precisely what constitute the set of simulation software tasks. These processes are computationally intensive requiring considerable processor time. For the purposes of our discussion, we may refer to these simulation software tasks simply as software tasks or just tasks.

These tasks that are related to numerical solution of models in addition to the actual software tasks of the embedded software greatly increases the total number of tasks that need to be scheduled by real-time simulation scheduler. A modern real-time simulation system, e.g. a flight simulator, may consist of hundreds of tasks running in parallel. While in older days it was not deemed feasible to have as many processors as the number of tasks in a simulation to enable true parallel execution; with

the introduction of manycore technology, the possibility of each task having its own dedicated thread or core for execution has become very much realistic. There are processors now available which have as much as 72 cores each capable of running 4 parallel threads resulting in total of 288 available threads. In fact with manycore processors being slated as the future of computing, it is even desirable to make efficient use of multiple cores available and run softwares task on a dedicated thread as much as possible. However, to realize such multi-core executions, the simulation tasks, i.e. numerical solvers for each simulated sub-system model need to run in parallel on separate cores.

There are two possible ways to distribute these numerical solvers to multiple cores or processors; either the sub-system models are developed such that they are equipped with a solver within their code in addition to the dynamic equations for the modeled system or, alternatively, separate instances of specialized simulation softwares with specialized numerical solvers can be run on separate cores doing the numerical solution for their respective models independently. This kind of simulation architecture where each model is solved independently in parallel, possibly on a dedicated core, and where one simulation tool acts as the master of the simulation, executing the simulation tasks and performing coordination among them, is called *Co-Simulation architecture*. The first case, where a model has its own solver, is often termed *Co-Simulation with runnable code* [26] while the case where distinct simulation softwares, often referred to as simulation tools, are run on multiple processors within one integrated simulation is called *Co-Simulation with Tool coupling* [26]. Figures 1.1 and 1.2 show the architectures of these two kinds of co-simulations. In the first case, the master communicates with the runnable code directly while in the tool coupling case, API wrapper functions are usually needed that translate the APIs provided by target simulation tool into a form usable by the co-simulation master.

Co-simulation does not only enable efficient distributed simulation but there are situations where co-simulation offers the most natural choice of simulation architecture. Consider a simulation that involves sub-system components from a wide variety of domains including cyber (i.e. computational), electrical, mechanical, hydraulic and thermal etc. In such a simulation, the system models belonging to different domains may need to run at different frequencies, for example, mechanical component models

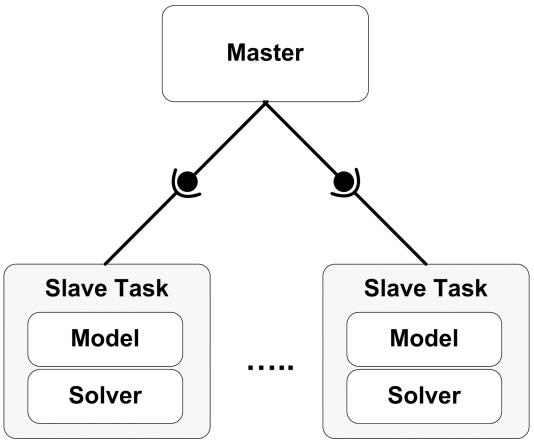

Figure 1.1: Co-simulation with runnable code

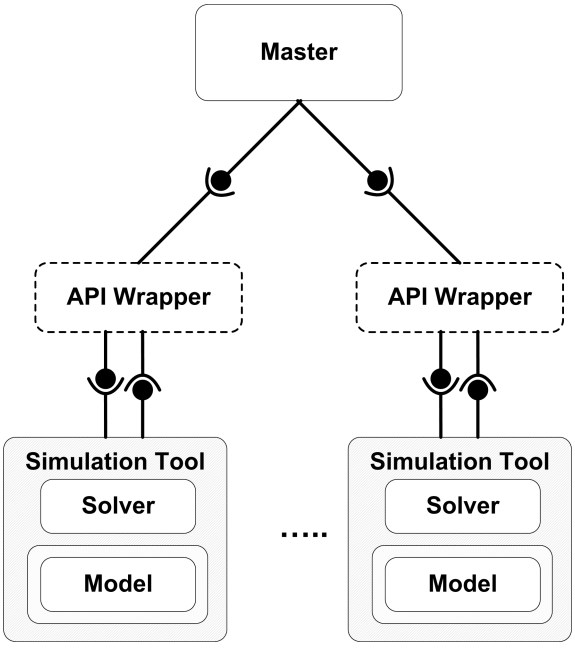

Figure 1.2: Co-simulation with Tool coupling

might not require to be solved as often as a fast electronic system models. If a single numerical solver is used for all the subsystems in a simulation with different execution frequencies, it would be highly inefficient. So, using different solvers for each subsystem and running them separately with different time-step sizes, i.e. *co-simulating* them seems to be the favorable solution.

Before proceeding further, let us define some terms that will be used throughout this work as follows,

- The simulation time steps mentioned in the above discussion are commonly called *integration steps* which refer to the fact that numerical integration algorithms are used to advance the time of a simulation model.
- *The integration step size* is simply the amount of time by which the current simulation time is advanced and for real-time simulations, it is normally constant [17].
- The *frame time* or *frame* of a model is the time interval between data transfers to or from other simulation entities that are external to that model. These external entities can either be a hardware component or other sub-system models. A *frame* can contain one or more integration steps [17].
- The simulations that use different step sizes for different models are called *multi-rate simulations* [27] or *simulations with multi-framing* [34].
- We can now redefine *real-time constraint* introduced earlier in terms of the frame as *the constraint that simulation time of a simulation model must be greater than or equal to the wall-clock time at the end of each of its frame.*

Real-time Multi-frame Co-Simulation: Now that we have defined all the relevant terms, the composite term of *Real-time Multi-frame Co-Simulation* becomes almost self explanatory and can be defined as a simulation where sub-system models with distinct frame length requirements are executed in parallel possibly on dedicated processors such that each and every simulated sub-system satisfies the real-time constraint.

Up till recently, researchers developed co-simulation solutions that are either application specific or simulation tool specific [5, 4, 39, 14, 15]. Consequently, there was no common interface standard that is both generic enough to support communication with models from every domain and powerful enough to be used in any simulation; be it real-time or non-real-time, multi-rate or with single step-size. Functional Mockup Interface (FMI) is one such interface standard designed to bridge this gap. FMI is described as an interface standard being developed for exchange of dynamic models among different simulation tools and co-simulation [26]. This means that FMI not only supports co-simulation but also solves the problem of model- exchange between different simulation tools. Although, our work does not emphasize on how to implement the co-simulation architecture, we certainly recognize FMI as an important step toward co-simulation; independent of tools or application domain.

1.2 Timed Automaton and Task Automaton

As will be discussed in section 1.3, the schedulability analysis framework that we shall propose is extension of an existing framework which is based on task automata; which in turn is based upon theory of timed automata. Therefore, It seems suitable to define basic concepts related to timed automata theory and task automaton in this introductory section before using them to build our framework.

1.2.1 Timed Automata Theory

The theory of timed automaton was first presented by Alur and Dill in 1994 [6] and soon after its introduction it became a popular formalism to model real-time systems. Various tools were then developed using the timed automaton theory to model-check real-time systems; a well-known tool among these is UPPAAL [31]. Although UPPAAL is based on timed automata theory, the syntax and semantics of timed automaton implemented in UPPAAL is a little different than that of original theory presented in [6]. However, since almost all of the timed automata based real-time schedulability research uses UPPAAL as implementation tool and hence UPPAAL's timed automaton model, we would also stick to this convention and use UPPAAL's semantics as

the basis of our work. The syntax and semantics of timed automaton as presented in UPPAAL are described below.

A timed automaton is defined a finite state automaton (FSA) with a set of non-negative real valued clocks. Clock constraints or *guards* are used to constraint the transitions or *edges* of a timed automaton. Hence the edges of a timed automaton can be annotated with guards in addition to the discrete symbols used in any ordinary FSA. The allowable form of clock guards is defined as conjunctions over conditional expressions of the form $c \sim n$ or $c - \acute{c} \sim n$, where c and $\acute{c}$ are clocks, n is an integer and $\sim$ is one of $\{<, \leq, =, >, \geq\}$. The automaton nodes or *locations* can also have associated clock constraints called *invariants* which determine the amount of time the automaton can pass in a particular location. Every clock defined in the automaton increments with the same rate but any of them can be reset to zero during a transition between locations. Thus the value of each clock actually equals the amount of time that has elapsed since that clock was last reset.

Following are the formal definitions of a timed automaton and its semantics [11]. Here,$B(C)$is used to represent the set of clock constraints over a set of clocks C.

Timed Automaton (TA): A timed automaton is a tuple $\langle L, l_0, C, A, E, I \rangle$, where

- L is a set of locations
- $l_0 \in L$ is the initial location
- C is the set of clocks
- A is the set of actions
- $E \subseteq L \times A \times B(C) \times 2^C \times L$ is a set of edges between locations with an action, a guard and a set of clocks to be reset
- $I : L \rightarrow B(C)$ assigns invariants to locations

Semantics of a TA: The semantics of a TA $\langle L, l_0, C, A, E, I \rangle$ is defined as a labeled transition system with possibly infinite states of the form (l, u) where $l \in L$

and $u : C \rightarrow R_+$is the clock assignment, (l_0, u_0) is the initial state where u_0 is a clock assignment that maps every clock in C to zero.There are two types of possible transitions,

- *Delay transitions*: $(l, u) \xrightarrow{d} (l, u + d)$ *if* $\forall \acute{d} : 0 \leq \acute{d} \leq d \Rightarrow u + \acute{d}$ satisfies $I(l)$, where $d \in R_+$ and $u + d$ increments the value every clock in C by d.

- *Action transitions*: $(l, u) \xrightarrow{a} (\acute{l}, \acute{u})$ *if* $\exists e = (l, a, g, r, \acute{l}) \in E$ such that u satisfies the guard g, $\acute{u}$ is equal to u for all the clocks except the clocks in r which are reset to zero and $\acute{u}$ satisfies $I(\acute{l})$

1.2.2 Task Automaton

The works of Norstrom [38] and Fersman [23] were the first to propose the use of an extension of timed automaton to define task arrival patterns in a task schedulability problem. The main idea was to assign to each *action transition*[38] or alternatively, a *location*[23], a task so that each transition or location represents the release of the associated task. This extended timed automaton was later considered for real-time tasks' scheduling analysis and was renamed as *task automaton* [20]. These two slight variations of this task automaton can be formally defined as follows:

Task Automaton: A task automaton with a task set P is defined as a tuple $\langle L, l_0, C, A, E, I, T \rangle$ where,

- $\langle L, l_0, C, A, E, I \rangle$ is the standard time automaton

- In [38], $T : A \rightarrow P$ is a partial function that assigns actions to tasks, while in [23] $T : L \rightarrow P$ is a partial function assigning locations to tasks

T is defined as a partial function which means that there can be action transitions or locations that are not associated with any task.

1.3 Schedulability Analysis Problem and the Proposed Approach for Solution

In a real-time system, a set of software tasks is said to be *schedulable* if it is possible for every task in the set to meet its deadline. Schedulability analysis then can be defined as a process that determines if a task-set is schedulable given the properties (execution-times, arrival times etc.) of the tasks and the imposed constraints (task deadlines, resources requirements etc.)

Since we consider the execution or numerical solution of simulation models in a real-time co-simulation simply as software tasks (see sub-section 1.1), this assumption transforms the problem of ensuring the timing correctness of real-time co-simulation into a problem of real-time schedulability analysis of the given simulation task-set. Similar to the case of real-time systems, a *schedulable* real-time simulation is one in which every simulation task is guaranteed to satisfy its *real-time constraint*.

It is generally accepted that no amount of testing can guarantee that the proposed schedule is infallible under *all* circumstances. Therefore, in order to verify that a particular scheduling algorithm will never cause any task to miss a deadline, a proof using some formal method is required. There are a number of formal methods for schedulability analysis available in the literature based on different task models, system configuration (uni- or multi-processor) and other scheduling characteristics [36, 45, 46].

Many of these analysis methods are ad-hoc techniques without any formal theory to support them. But there is one class of analysis methods that are based on timed automata [6, 1, 2] or an extension of them termed as task automata [24, 22, 20]. These works use task automata to define a system's task model, a fundamental notion in the theory of real-time schedulability analysis. A *task model* basically represents an abstraction of different tasks' behaviors that are considered important for the problem at hand. A task model based on task automaton is considered as the most general and most expressive form which encompasses almost all other task models in the literature [44]. The primary advantage of using an established automata theory based analysis technique is that the analysis procedure can be automated by using the model checking techniques and tools already developed for that theory. This automation becomes

more desirable when the number of tasks is very large. In addition to verification, the automata model can be used to synthesize an optimal scheduler as well [2, 7].

Our goal in this work is to perform schedulability analysis for real-time multi-rate co-simulations to prove the timing correctness of these simulations. As mentioned earlier, verification of functional accuracy of simulations is beyond the scope of this work. Due to the advantages discussed above, we selected a timed automata based approach as the basis of our analysis technique. Although timed automata based techniques may suffer from problem of state explosion or in some cases high complexity and even undecidability [20], but we simplify our problem by making the assumption of having a dedicated processor core for each task. As result, unlike a general scheduler which inevitably deals with queues of tasks for one or more processors and may also be required to handle preemptions and resumptions, there will be no tasks (other than instances of same task type, see section 2.1) in our co-simulation system model waiting in the queue for processor core and instead of preemptions (followed by resumptions), we will be dealing with task cancellations only. The co-simulation system model is described in detail in section 2.1.

We identified one particular timed automata based schedulability analysis framework by Fersman and Yi [24] (FY framework) as our basis for further analysis. The FY framework transformed the schedulability analysis problem into a state reachability problem of a timed automaton. We worked on same principle and transformed our problem as the state reachability problem. After the necessary theoretical extension of the FY framework, we implemented the schedulability analyzing timed automaton in UPPAAL [31], which is a popular timed automata based tool for model-checking. The implemented framework was verified on a toy problem before being applied to real world case study of a helicopter simulator.

1.4 Related Work

To the best of our knowledge, this work is the first attempt in presenting a framework for schedulability analysis of a real-time simulation. All the existing works are concerned with schedulability of actual real-time systems and not the real-time simu-

lations. However, this existing work is definitely related to and in fact forms the basis of our work. Hence, the rest of this section will describe the research done on the schedulability analysis of actual real-time systems.

Research on real-time schedulability analysis techniques has a long history. Liu and Layland's work [36] is often considered as one of the earliest and most prominent work that not only introduced the, still popular, fixed priority scheduling algorithms like *Rate Monotonic* and *Earliest Deadline First* but also analyzed these scheduling techniques. After this seminal work, researchers have developed schedulability analysis techniques that range from devising exact analytical tests [29, 8, 12] that determine schedulability of a real-time system, to providing feasibility tests whose performance is judged by the acceptance ratio [12, 32] ; and from techniques developed for uni-processor systems [40] to the ones that are applicable to multi-processor systems [30, 16, 9] or distributed systems [46, 33].

One important aspect of any scheduling analysis technique is the task model or models for which it is applicable. Typically, a task model provides information about task arrival patterns, e.g. whether they are periodic or sporadic, and task resource requirements, e.g. best/worst case execution times or any other shared resource requirements. Task models are important because, given a task model, a designer can work out what scheduling strategy is feasible for the given constraints, also called *Scheduler synthesis*. On the other hand, given a scheduling technique, a task model can be used to determine if a set of tasks is schedulable or not. This is called *Schedulability analysis*.

One of the earliest task models was a periodic task model introduced back in 70s by Liu and Layland [36] and it became the basis for many subsequent works. In a periodic task model, tasks are generated strictly at fixed intervals of time but this assumption of periodicity cannot be considered true for all the real-time systems and so different task models were introduced to model non-periodic and irregular tasks arrivals. These include sporadic models [10, 35], multi-frame models[37] and graph based models[43, 19]. For a detailed survey of task models, please, see [44].

As described in section 1.2, researchers were quick to realize that timed automaton based formalism can be extended to model task arrival patterns. These extended

timed automata were termed task automata [20] and as it turned out, task automata were categorized as the most expressive task model in the literature [44]. Task automata are expressive enough to encompass all other task models like periodic or sporadic real-time tasks and can also describe concurrency and synchronization. But with expressiveness comes the problem of increased complexity. There have been some studies that analyze the complexity and decidability of schedulability analysis techniques based on task automata [20, 22]. Besides task-automata based research, which we will describe in more detail in next two paragraphs, other prominent works on scheduling analysis that used timed-automata include [18, 13].

Our work is based on task automata which are used in a number of schedulability analysis research works. The authors of [20] analyzed the real-time schedulability of tasks on a single processor. A task was represented as a 3-tuple (B, W, D), where B is the best-case execution time of the task, W is the worst-case execution time (WCET) and D is the deadline for the task relative to its time of release. A number of scenarios were analyzed for decidability and the authors concluded that the scheduling problem for many scheduling strategies is decidable unless all three of the following conditions hold true,

1. Scheduling strategy allows preemption
2. The best-case and worst-case execution time of tasks is different
3. A task release time is based on the precise finishing time of some other task

In [21, 22], the authors examined the schedulability of fixed-priority systems using task automata. They also studied the case of data dependent control, where release time of a task may depend on a specific value of some shared variable and hence on the time-point when some previous task finished execution. In this work the task were represented as pair (W, D), where W is the WCET and D is the relative deadline. It was showed that without data dependent control, the scheduling problem for a fixed priority system can be solved by using only two extra clocks, in addition to the clocks already present in task automata. However, for the case of data dependent control the solution of scheduling problem for the same system requires $n + 1$ additional clocks, where n is the number of tasks in the system. In a further improvement of

[21], the same authors introduced a more generic framework for real-time schedulability analysis using task automata [24]. The new model was able to handle more general precedence relations and resource constraints and they showed that schedulability analysis problem for this extended case can be solved using the same technique that was introduced in [21]. In another research, the scheduling analysis using task automata was extended to multi-processor setting [30]. And the authors found one more negative result as compared to the single processor case in [20]. More precisely, they showed that in a multi-processor setting, the truth of only the first two conditions mentioned above is sufficient for the solution of scheduling problem to fall in undecidable category. This undecidability property is still an open problem for the case of single processor scheduling. A popular tool used in some of the above mentioned works and that uses timed automata to model-check real-time systems is UPPAAL [31]. The syntax and semantics of timed automata used in UPPAAL is a little different than that of original timed automata theory in [6]. But almost all of the timed automata based real-time schedulability research uses UPPAAL's model of timed-automata and so we also used the same semantics in our work. See section 1.2 for the details of these semantics.

Rest of this thesis is organized as follows,

Chapter 2 first describes a system model for multi-frame co-simulation before presenting the details of our proposed framework for schedulability analysis of multi-frame real-time co-simulations. The shortcomings in the initial proposal and the solutions are also discussed in this chapter.

Chapter 3 deals with the implementation details of the proposed framework. The implementation is done in UPPAAL and the chapter discusses details of main schedulability analysing automaton, termed *CheckingAutomaton*, as well as the supporting automata and functions. The verification of the implemented framework using a toy problem is also discussed in this chapter.

The last chapter 4 presents the application of our proposed schedulability analysis framework on a real life case study of a helicopter simulator.

CHAPTER 2

THE FRAMEWORK FOR SCHEDULABILITY ANALYSIS

This chapter constitutes the core of our work and describes the framework developed for schedulability analysis of real-time multi-frame co-simulations. As mentioned in the previous chapter, the proposed framework is an extension of FY framework [24] which was developed as a generic approach for schedulability analysis of real-time tasks. This chapter discusses in detail the proposed extensions to the FY framework, why the extensions are required in the first place and how to handle the issues arising due to the new extensions. The chapter ends with description of a special timed-automaton called *Schedulability Verifier (SV)* that actually performs the schedulability analysis of a co-simulation.

To begin with, the first thing to consider for the development of a framework that is meant for schedulability analysis of real-time multi-frame co-simulations is the precise model of a real-time multi-frame co-simulation itself. Hence, the next section is dedicated to definition of the required co-simulation model which will then serve as the foundation on which our whole schedulability analysis framework is built.

2.1 Multi-Frame Co-Simulation Model

In an effort to ensure that our defined co-simulation model reflects the industry standard co-simulations, we based the model on the co-simulation model presented in Functional Mockup Interface (FMI) standard 2.0 [26]. The FMI presents an interface standard for model exchange between different simulation platforms as well as for co-simulations. But before describing the model of a system that we feel accurately captures a multi-frame co-simulation, let us first list the assumptions that we make

for our model.

1. The system executing the co-simulation has as many processor cores available as there are number of software tasks.
2. Each processor core has been assigned a task at design time which is fixed for the entire period of simulation.
3. Software tasks can be aborted at any time during their execution.
4. The aborted task is responsible for leaving its acquired resources clean.

The software tasks mentioned here include the simulation tasks that perform the numerical integration of state variables for simulated physical systems as well as the pure software tasks that are actually executed in some embedded computer in the system. Purpose of the first assumption is to avoid cyclic dependencies among different task types that can arise as a consequence of processor core sharing. Similarly, cyclic dependencies occurring due to precedence constraints or data dependencies among tasks assigned to different cores also need to be avoided (see section 2.2.1). The cyclic dependencies among tasks are undesirable in our proposed framework because they make our analysis approach, discussed in section 2.2.2.3, unusable. However,it is worth mentioning here that the restriction imposed by this first assumption above can be relaxed if there is a scheduling mechanism where multiple task types are assigned to the same core but it is ensured that instances of any task type will be released at times such that their execution can never interfere with the execution of instances of another task type that are assigned to the same core. This means that, for each core, at any single point of time there is only one such task type whose instances are either executing or ready to execute on that core, thus avoiding cyclic dependencies.

An important thing to note is that our model allows multiple instances of the same task type to be active at the same time and therefore there can be a scenario where multiple instances of the same task type are waiting for the same processor core. This kind of interdependency among instances of same task type does not cause our analysis procedure to fail and is handled in the proposed framework by using waiting task queues for each core that keep the waiting instances of the task type assigned to that core. But what scenario can cause the multiple instances of the same task type to be

active at the same time with one executing and others waiting in the task queue? The intuition says that since our system model assumes that we have a dedicated processor core for each task, there should be no waiting queue. This intuition is especially true if the scheduling algorithm is a conservative one so that the co-simulation master does not schedule future instances of a task unless all previous instances of that task type have finished their execution. However, if we consider a scheduling algorithm that can optimistically schedule task instances without waiting for earlier instances of the same type to finish then there can be a situation where waiting queue is required. An example of such a scenario is illustrated in section 2.2.2.2

Assumption no. 2 allows the tasks to be only statically assigned to processor cores and thus relieves the scheduler of selecting a core for execution at runtime and also prohibits task migration between different processors during a simulation.

Task abortion or cancellation discussed in assumption no. 3 is a concept that is different than that of task preemption in general scheduling problems. A preemption is eventually followed by a resumption of the task from the same state at which it was preempted. In task abortion, however, the task can never resume execution from the same execution point but can only be restarted by re-performing the canceled integration step again. This task cancellation feature appears in the FMI standard as well, however, unlike our model, FMI does not allow re-performance of current integration step but restricts that the simulation must be canceled altogether. We feel this restriction to be too strict and allow the simulation to continue after a task cancellation. However, to ensure smooth simulation run in an event of task cancellation, assumption no. 4 makes the canceled task responsible to leave the system in a clean state. This means that the task cancellation requires processor time itself because of the housecleaning steps it needs to take in order to leave the system resources clean.

The task cancellations are usually caused by certain events during a simulation. An *event* can be defined as any unplanned occurrence in a simulation, such as a failure of a hardware component in a HWIL simulation or an input from a human in Human-in-the-Loop simulation. These unplanned occurrences or events can affect the simulation in different ways including cancellation of current integration steps of some tasks, change in periodicities of certain tasks and/or release of special event handling tasks

etc.

As described in the FMI standard [26], our system will be a co-simulation with a master that orchestrates and synchronizes the running of the simulation. Tasks are allowed to communicate with the master only and not directly with each other, the master is then responsible for forwarding the communicated data to the destination task, if any. This communication between the tasks and master does not necessarily have to occur at the end of each integration step but are only allowed at fixed communication time-points as required by the master. These communication time-points, however, must coincide with end time of an integration step. Fig. 2.1 shows this co-simulation architecture. For the sake of simplicity, we assume the communication between the master and the task to have a constant delay and that this delay is included in the worst case execution time of each task.

Besides communication between the tasks, the master is also responsible for scheduling or invoking each task for execution. A scheduler is therefore a part of the master and is responsible for executing tasks in a way such that no task misses its deadline while at the same time ensuring that the precedence constraints between them are not violated.

But a scheduler can only schedule a task after it is released or become active. There are a number of task models mentioned previously that can be used to describe the release or activation pattern of tasks in a system. In our system model, tasks are usually periodic during normal system operation with possibly each task having a different periodicity but at the same time there can be sporadic tasks as well which may become active in response to some particular event. Moreover for periodic tasks, we do not restrict the task periodicities to be fixed and allow them to be changed dynamically at runtime. To represent such a complicated task arrival pattern, we selected the task automata based task model for our work which is the most expressive task model in the literature and can incorporate almost any task arrival pattern. It has an added advantage of being backed by sound formal theory and availability of task automata based tools for model-checking and code generation [7]. The downside of being the most expressive task model is that the complexity of scheduling analysis based on it is the highest as well. But coupled with the simplifying assumption described above,

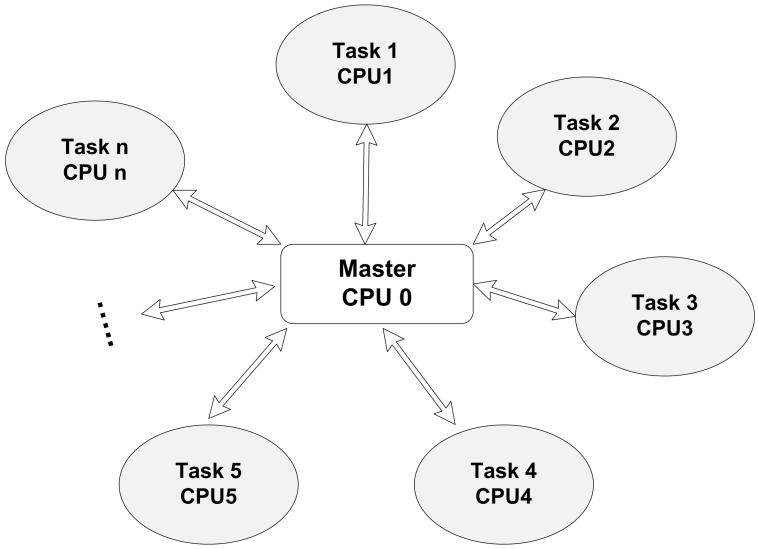

Figure 2.1: Co-Simulation System Model

we believe that the complexity of scheduling analysis for our system model using timed automata based task model will be tractable.

2.2 The Proposed Framework

The real-time schedulability analysis in the case of multi-frame real-time, possibly FMI-based, co-simulation actually amounts to analyzing the scheduler component of the co-simulation master to check if it is possible for it to schedule the tasks in real-time for a given task arrival pattern and its constraints. Therefore, for our purpose of real-time scheduling analysis of a real-time co-simulation, we can assume the co-simulation master as being composed of two distinct components. First is a scheduler component that will schedule the tasks such that no task will violate the real-time constraint while allowing for precedence constraints to be met. It is worth mentioning here that while every real-time constraint is absolutely essential to meet; many, but not all, of the precedence constraints can be relaxed in real-time co-simulation in order to ensure the mandatory task deadlines. We will come back to this issue later and discuss it in detail. The second component of the master can be considered as being composed of the rest of the master's functionality but the feature of our interest in this second part is the one that defines the *task model*. A task model includes definition

of the tasks' arrival patterns, worst-case execution times, deadlines, and precedence relations between them. The scheduler part uses this information from the task model to make its scheduling decisions. We will call these two parts of co-simulation master as the*scheduler* and the *task model*.

The two parts described above are precisely the building blocks of our schedulability analysis framework as well. Hence, following sub-sections 2.2.1 and 2.2.2 will respectively describe *task model* and *scheduler*, i.e. the building blocks of the proposed framework. However, in sub-section 2.2.2, where we define the scheduler part, our goal will not be to actually schedule a task but to just simulate the scheduler decisions and check if there is a scenario where a scheduler decision leads to unschedulability. Therefore, we shall not be defining the actual scheduler but a mechanism to mimic the scheduling decisions and monitor the task instances for any violation of deadline.

Since we intend to extend the timed-automata based framework of real-time schedulability analysis introduced by Fersman et al. in [24] and [21], timed automata will naturally play a major role in our proposed framework as well. More specifically, a pair of timed automata will be defined that will simulate scheduling decisions of the co-simulation master and monitor unschedulability while a slight variation of timed automata, i.e. task automata [24], will be a major part of our task model description.

2.2.1 The Task Model

The task model component of the master is defined as a 3-tuple $\langle S, A, G \rangle$ where,

- S is a set of task types. Each member of the set S defines a single type of task
- A is a task automata that defines the task arrival patterns
- G is a Directed Acyclic Graph (DAG) that defines precedence constraints between tasks

Following is the detailed description of each component of the task model tuple.

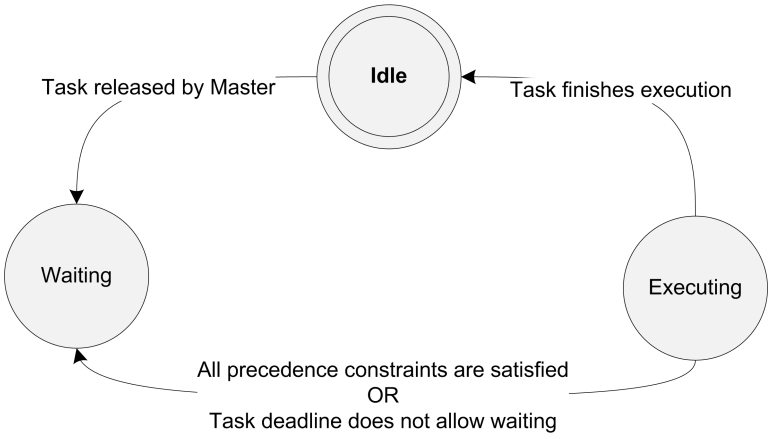

Figure 2.2: Task States

2.2.1.1 Task Type (Or Task)

Any simulation can be considered as being composed of different tasks each performing a specific function in the simulation. For our purpose of schedulability analysis, it does not matter what a task actually does, instead, the properties that matter are the task's timing and precedence constraints. Therefore, we will represent a task as an abstraction termed as *Task Type* that will contain only those attributes that are of our interest. Multiple instances of a task type can exist in a system at the same time. We shall use the term task to refer to both task type and its instance wherever the actual meaning is clear from the context. Before proceeding further, let us define the three possible states of task instances in our framework,

- *Idle*: When a task instance is waiting to be released by master
- *Waiting*: When a task is released but is waiting either for some precedent task to finish execution or for processor time. In a multi-core platform such as ours, waiting for processor time means that a task has to wait for a previous instance of the same task type as itself to finish execution. To see how this scenario is possible, refer to section 2.2.2.1
- *Executing*: When the task is actually executing on a processor

The three states and transitions possible between them are shown in fig. 2.2. A task

is termed active during waiting or executing states, so, any reference to an active task means that it is either waiting or executing. The response time (*ResponseTime*) of a task is the time elapsed between the instant when it was released and instant when it finishes execution. Therefore, *ResponseTime* is equal to the time during which a task is active which in turn is equal to the sum of times the task spends in waiting and executing states.

In most of the schedulability analysis works, a task type is usually defined by two static attributes: task's worst-case execution time and its relative deadline. But in our case, the deadline cannot be fixed at the time of task type definition simply because our system model allows the tasks to have dynamic periodicities. And since the deadline for a periodic simulation task is dependent on its periodicity and hence its current integration interval (both are equal in real-time simulations), it cannot be defined as a static attribute. More specifically, the greater the integration interval, the greater time is allowed for the task to finish, i.e. greater relative deadline and vice versa. We will, therefore, define this *Deadline* attribute as a dynamic property for our task types which will be assigned a value by the co-simulation master at runtime whenever a task instance is released for execution.

Recall that we have assumed in our real-time co-simulation system model that a task can be canceled at any time after its release by the master. We also assumed that a canceled task will leave the system in a clean and stable state. To reach this clean state, a canceled task typically needs to perform some additional tasks like relinquishing of resources, resetting state variable, garbage collection etc. These housekeeping tasks needed for cancellation need processor time and so we defined a term called Worst Case Cancellation Time ($WorstCaseCancelTime$). $WorstCaseCancelTime$ is the maximum amount of time that a task may need to perform these housekeeping tasks in an event of cancellation. We will add this cancellation-time attribute to our task type definition with the name $WorstCaseCancelTime$ and define it as the largest time that can elapse between a cancellation request from the master and completion of that cancellation by the task. Both $WorstCaseExecTime$ and $WorstCaseCancelTime$ are static task attributes and are assumed to be determined beforehand. Two more attributes are added in task type definition that will be required for real-time schedulability analysis. These are the tasks' current

simulation time ($SimulationTime$) and its current integration interval ($Current-IntegrationInterval$). The $CurrentIntegrationInterval$ is required to maintain $SimulationTime$, by continuously accumulating it, and also to predict the next arrival time of a task while $SimulationTime$ is needed in handling precedence constraints among tasks, as explained in 2.2.2.1.

The formal definition of task type is therefore a 5-tuple $\langle WCET, WCCT, DL, CII, ST \rangle$, where

- $WCET$,$WCCT$ are static attributes representing the task's $WorstCaseExecTime$ and $WorstCaseCancelTime$ respectively
- DL,CII represent the task's $Deadline$ and $CurrentIntegrationInterval$ and are assigned values at runtime by the master
- ST is the task's current $SimulationTime$ being constantly updated using the $CurrentIntegrationInterval$

2.2.1.2 Task Automata

The next thing required in our task model is the task arrival patterns. To define the arrival pattern of tasks, a task automata will be used which is actually a timed automaton with some locations annotated with task(s) [24]. A transition to a location annotated with a task means that the task used in annotation now leaves the idle state and moves to waiting state. One minor modification in our case is that before each task getting ready for execution, the master will set the $CurrentIntegrationInterval$ and $Deadline$ parameters of that task.

Typically, task automata is a collection of automata where each task automaton can define arrival pattern of one or more tasks. For example, a task automaton for a simple periodic simulation task is shown in figure 2.3. The figure shows that the first instance of the periodic simulation task, TaskX, is released after an initial offset time once the system is initialized. From then onwards, the automaton remains in one location and waits for a duration equal to the task periodicity at the end of which it loops back to the same location to release the next instance of TaskX. This continues until the task's

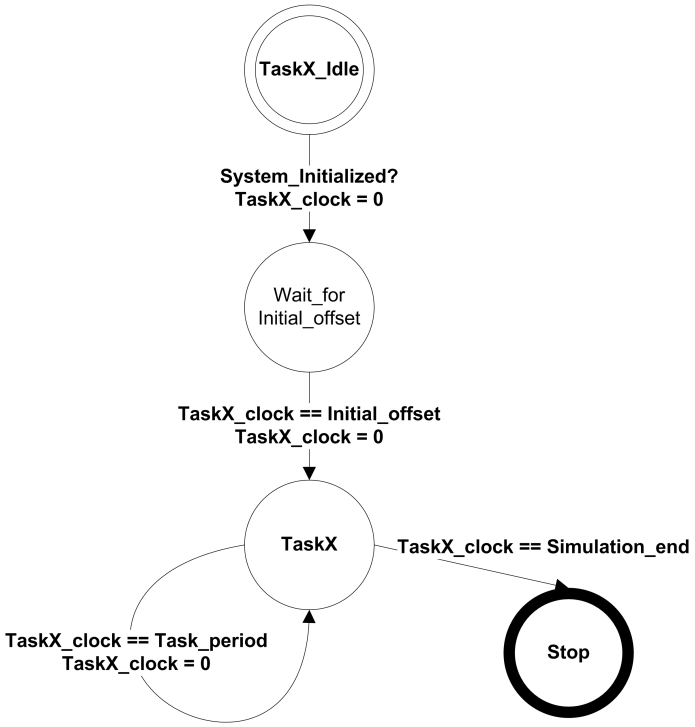

Figure 2.3: Task Automata for a simple periodic simulation task TaskX

clock variable indicates that end of simulation time is reached at which the automaton terminates at *Stop* location.

2.2.1.3 Precedence DAG

The precedence constraints can be naturally represented as DAG where the nodes represent the tasks and the edges between them denote the presence of a precedence constraint. Use of DAG means that cyclic constraints are not allowed. The importance of restriction on cyclic precedence relations is explained in subsection 2.2.2.1. As discussed in subsection below, the proposed analysis framework have two kinds of precedence constraints and so the precedence DAG for our framework needs to have two types of edges to represent the two kinds of constraints.

2.2.2 The Schedulability Analyzer

We shall now turn our attention to the second building block of our framework that corresponds to the second part of co-simulation master, that is the scheduler. As mentioned earlier, our goal in this work is not to schedule tasks but to simulate scheduling decisions and verify schedulability. Hence, it is more precise to term this part as *Schedulability Analyzer* instead of scheduler. This *Schedulability Analyzer* part will primarily consist of two timed automata: one that simulates majority of scheduling decisions and also checks the condition of unschedulability, and the second that simulates just one critical scheduling decision; that of handling cancellation of an active task instance due to release of some binding precedent task (See sub-section 2.2.2.1 for definition of binding precedent task). We shall call the first automaton as *Schedulability Verifier (SV)* and the second as *Cancellation Handler (CH)*.

Before proceeding to describe the construction of *SV* and *CH*, it is necessary to describe how our framework handles precedence constraints among tasks. This new precedence handling mechanism is the basic difference between the FY framework and our proposed framework, therefore, we shall build upon this difference and introduce the related concepts and terms as we go forward.

2.2.2.1 Precedence Handling in Proposed Framework

We shall first briefly describe how precedence constraints are handled in the existing FY framework. This description will be followed by a discussion on shortcomings of this precedence handling mechanism before presenting our proposed method to handle precedence constraints for the case of multi-frame real-time co-simulations.

Precedence Handling in FY Framework

The FY framework uses a matrix of Boolean variables to define precedence between tasks of a given task set. The framework defined just one kind of precedence constraint that was mandatory for a dependent task to satisfy. The $n \times n$ Boolean matrix represented a precedence DAG G, where n is the number of tasks and each

matrix entry g is either $true$ or $false$ with $g_{i,j} = true$ if a task P_i must be preceded by task P_j and $false$ otherwise. The precedence constraint for a task P_i is therefore, just the conjunction of all $g_{i,j}$, such that $1 \leq j \leq n$. Another matrix of size $n \times n$ is used where every time a task P_j finishes execution, all $g_{j,i}$ are set to $true$ and $g_{i,j}$ to $false$, where $1 \leq j \leq n$, indicating that all tasks that are dependent on P_j can proceed for execution while all tasks preceding P_j must execute at least once before the next execution of P_j.

Shortcomings of FY Framework

The problem with this relatively simple precedence handling is that it forces each precedence constraint to be satisfied every time and that the real-time schedulability under these mandatory precedence constraints does not depend only on the $Worst\text{-}CaseExecTime$s of tasks but also on the fact that all related tasks either have same periodicity, i.e. they become ready for execution all at the same time, or the precedent tasks have periodicity shorter than that of dependent task. If this is not true, that is, a dependent task P has a shorter periodicity and hence shorter deadline than its preceding task Q; then as shown in Fig. 2.4, no matter how short is the $WorstCase\text{-}ExecTime$ of P, a mandatory precedence constraint between P and Q will cause each successive P instance to finish more closer to its deadline and eventually causing a P instance (P_4 in Fig. 2.4) to miss its deadline. The vertical lines in the figure denote the release time of each task instance and also serve as the deadlines for the previous task instance.

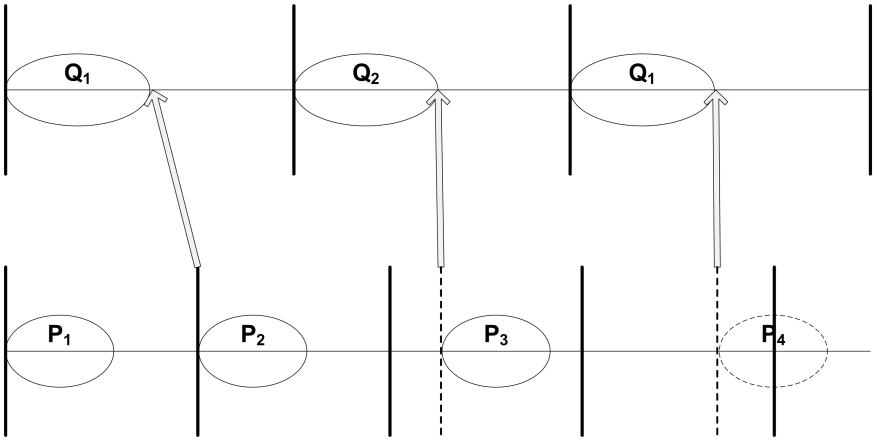

Figure 2.4: Binding precedence between tasks with different periodicities

Proposed Precedence Handling

The target of our proposed framework are real-time co-simulations with tasks having different frame sizes. In such multi-frame real-time co-simulations, contrary to the assumption in the FY framework, the tasks can, and usually do, have different periodicities. Another point of concern is that normal data dependency kind of precedence constraints among periodic simulation tasks need not be a binding precedence constraint since in many cases it is possible to advance the simulation with extrapolated old data to ensure that no task misses its deadline. However, this should not mean that the co-simulation master will always ignore the data dependency constraints among periodic simulation tasks because it is always desirable, for the sake of accuracy of the simulation, to wait for fresh outputs from the preceding tasks if the dependent task's deadline permits it. In other words, *if there is a possibility of improving the simulation accuracy by delaying a task's execution without violating its deadline, then the master should go for the delayed execution*.

Of course, not all the precedence constraints will give co-simulation master the freedom to choose between delayed or immediate task executions. That is, beside the non-binding precedence constraints, there will be some precedence constraints that are mandatory and for these constraints the master doesn't have the option of going ahead with the execution of the dependent task before the precedent task finishes execution. We envision these binding precedence constraints to be restricted to the

case where a task is dependent upon some sporadic task or a periodic task that produces random outputs at each invocation. The examples of such tasks include event handling tasks or one-time tasks required during mode-transitioning. Since it is not possible to use old outputs from these tasks, the dependent task must wait until their completion.

So, now we need to have two kinds of precedence constraints in our real-time co-simulation system: one that defines a non-binding relation and is commonly used to represent precedent constraints among periodic simulation tasks, and the other that defines a binding precedence relation where the precedent task is usually a sporadic task. We will differentiate these two precedence constraints as *non-binding constraint* and *binding constraint*.

Like the FY framework, an $n \times n$ matrix is used to represent the two kinds of precedence constraints where n is equal to the number of tasks in the system. Each entry $E_{i,j}$ in the matrix denotes whether or not there is a precedence constraint between tasks P_i and P_j and also the type of constraint, if any. Therefore, $E_{i,j} \in \{B, N, \emptyset\}$, where $\emptyset$ denotes *No constraint* while symbols B and N denote the existence of a binding or non-binding constraint respectively.

However, the information represented in the above matrix is not enough for our purposes. There must be a way for co-simulation master to determine how much the execution of the task under analysis has to be delayed because of some binding constraint; or for the case of non-binding constraints, whether or not it is suitable to wait at all for a non-binding precedent task to finish, and if yes, then how long. The FY framework does not provide any information for the master to make such a decision. Our framework will improve upon this and will help the master in deciding between accuracy and real-time requirements by providing the necessary information described below.

Additional Information Required for Proposed Precedence Handling

1. Remaining Response Time: One important piece of information is the Remaining Response Time ($RemResponseTime$) of each precedent task Q at the time of

arrival of the dependent task P. The master can use this information to determine if execution of P can be delayed - to get fresh outputs from the precedent tasks without violating P's deadline - by checking the following condition,

$$Deadline(P) \geq RemResponseTime(Q) + WorstCaseExecTime(P) \quad (21)$$

The amount of time that a task P can wait is calculated by considering the largest $RemResponseTime$ of a precedent task Q for which condition (21) is true.

But the notion of $RemResponseTime$ presumes that the actual response time (***ResponseTime***) is already known and so the $RemResponseTime$ of an active task Q can be calculated as,

$$\begin{aligned} RemResponseTime(Q_{active}) =& ResponseTime(Q_{active}) \\ & - TimeSinceActive(Q_{active}) \end{aligned} \quad (22)$$

And for the tasks that are currently *idle*, the $RemResponseTime$ will be,

$$\begin{aligned} RemResponseTime(Q_{Idle}) =& TimeToNextArrival(Q_{Idle}) \\ & + ResponseTime(Q_{Idle}) \end{aligned} \quad (23)$$

The important question now is that is it possible to find the values of all the variables on the right-hand sides of the above two equations? Fortunately, the fact that precedence constraints are restricted to be represented as DAG makes it possible to calculate the $ResponseTime$s of all the precedent tasks. Since a DAG means there are no cyclic precedence constraints among the tasks, it is possible for us to sort the tasks topologically and list them such that any task Q that is defined as precedent of a task P will come ahead of P in the list. This sorted list resembles a list of tasks with fixed priorities and enables us to use the technique employed for fixed priority

tasks by Fersman et al. in [21] which was actually based on a much older technique used for calculating worst case response times using rate-monotonic analysis for periodic tasks [29]. In this technique, a task's worst case response time is calculated by adding its execution time and the time that it spends while waiting for other higher priority tasks to finish. We can use a technique similar to that of [21] and use our framework to analyze the tasks in order of the topologically sorted list so that at the time of analyzing a task P, we can be sure that all its precedent tasks have already been analyzed (since all of them are ahead of P in the sorted list) and their respective $ResponseTime$s have been calculated. A task should be analyzed for the entire simulation scenario defined by the given task automata before moving on to the analysis of next task in the list.

A clock variable is required in our *SV* timed automaton to determine $TimeSinceActive$ by accumulating the time starting from the instant when a task Q became active. For determining the $TimeToNextArrival$ of an idle task, the task's $CurrentIntegration-Interval$ parameter can be used since for periodic tasks in a real-time simulation $CurrentIntegrationInterval$ is equal to their periodicity. Therefore,

$$\begin{aligned} TimeToNextArrival(Q_{Idle}) =& CurrentIntegrationInterval(Q_{Idle}) \\ & - TimeSinceActive(Q_{Idle}) \end{aligned} \tag{24}$$

Since we need a separate clock for each active and idle precedent task, it seems that number of clocks in our timed automaton for analysis will be large despite the fact that we were hoping to use as less clocks as possible in our timed automaton just as in [21]. This is undesirable because the complexity of performing model checking on a timed automaton increases rapidly as the number of clock variables increases. Fortunately, we were able to minimize the number of clocks in our automaton and the actual number of clocks required is discussed in section 2.2.2.3.

2. Simulation Time: Besides $RemResponseTime$ of the precedent task, there is one other parameter that needs to be taken into consideration by the master of a multi-frame real-time co-simulation before deciding to delay the execution of a dependent

task. This parameter is the $SimulationTime$ of tasks. We know that in any simulation, $SimulationTime$ of a simulation task is advanced in steps equal to its current periodicity or integration interval. But in a multi-frame co-simulation, tasks have different periodicities and therefore, their $SimulationTime$s are not in synchrony with each other. For example, consider tasks Q and P running with the periodicities of $3msec$ and $5msec$ respectively. At the start of the simulation, both tasks will start with simulation-time zero. But in subsequent simulation loops, the simulation time of Q will increment as $3msec$, $6msec$, $9msec$ and so on, while the simulation time of P will increment with step size of 5 as $5msec$, $10msec$, $15msec$. The two tasks will re-synchronize with each other at the least common multiple (LCM) of their periodicities. Now, if task P is dependent upon task Q then it means that it will require Q's output every time it executes, i.e. at simulation times $5msec$, $10msec$ and so on. But the Q only produces outputs at multiples of $3msec$. In such a scenario, the master is responsible to handle this precedence relation in a way that can counter this lack of synchronization. As discussed earlier, the master can either decide to use old or extrapolated outputs from the precedent task Q in order to meet P's deadline or it can wait for fresh Q outputs. This decision depends upon three possible scenarios at the time of task P's arrival. These scenarios are described below,

Case 1: If $SimulationTime$ of Q has already moved past the $SimulationTime$ of P then the master should use interpolation which provides better approximation than extrapolation.

Case 2: If Q's ***SimulationTime*** is less than P's at the time of latter's arrival then there can be two sub-scenarios based on task P's deadline,

- If P's deadline is such that it satisfies the condition given in (21) then the master should delay execution of task P and employ interpolation afterwards.
- If P's deadline does not satisfy condition (21) then the master should proceed ahead with extrapolated outputs.

Case 3: In the last case, where $SimulationTime$s of Q and P are equal, the master can use the outputs of Q directly without any need of extra- or inter-polation.

Above discussion shows that the only case where a master needs to check if delaying execution of P for a periodic precedent task $Q_{periodic}$ will help in simulation accuracy is when,

$$SimulationTime(Q_{periodic}) < SimulationTime(P) \tag{25}$$

This means that condition (21) needs to be evaluated only for those periodic precedent tasks which satisfy this condition (25).

However, condition (25) needs to be modified for the case of *sporadic* precedent tasks. It is because sporadic tasks do not have a notion of *simulation-time* defined for them. Therefore, instead of simulation-time, we consider the release-time for the sporadic precedent tasks and define the condition as follows,

$$ReleaseTime(Q_{sporadic}) \leq SimulationTime(P) \tag{26}$$

For *non-binding* precedent tasks, whether periodic or sporadic, condition (25) or (26) serve as the pre-condition that must be satisfied before checking condition (21) to decide if the execution of dependent task should be delayed. On the other hand, for *binding* precedent tasks, condition (25) or (26) alone provide the *"necessary and sufficient"* condition to delay execution of the dependent task.

Fig. 2.5 below can help to understand how a master will handle non-binding constraints among periodic simulation tasks. Execution trace of two task types P and Q are shown in the figure. The real-time instants on the of P and Q's execution timeline are denoted as RTQ_i and RTP_i respectively while the simulation-time instants are shown on the same timelines as STQ_i and STP_i. It is important to note that the real-time instants and simulation-time instants, in each task's execution trace, with the same subscripts denote the same time instant in their respective timelines, e.g. $RTQ_2 = STQ_2$. As shown in the figure, instances of a task Q, shown as Q_1, Q_2 and Q_3 are released for execution at real-time instants RTQ_1, RTQ_2, and RTQ_3 respectively. Instances of another task P, represented as P_1, P_2, P_3 and P_4 are released for execution at real-time instants RTP_1, RTP_2, RTP_3 and RTP_4 respectively. Each release time instant is the deadline for the previous task instance, i.e. previous task instance should have completed execution before the new instance of the same task is released for execution. The width of a task ellipse is equal to the task's

$WorstCaseExecTime$ and so represents its executing state. The time between the points when a task instance finishes execution and when a new instance is released by the master is when the task is considered to be Idle. Task $SimulationTime$s are also overlapped on the time line but $SimulationTime$ values are shown only at points where real-time and task $SimulationTime$ are not equal to each other. Assume that task P uses some data produced by the task Q and is therefore dependent on it while task Q has no precedent constraints.

- To start with, both Q_1 and P_1 use the initial values of their inputs and start execution immediately.
- By the time instance P_2 is released for execution at RTP_2, task Q_1 has already reached $SimulationTime$ value STQ_2 such that $STQ_2 > RTP_2$, therefore, P_2 does not need to wait for anything and it can proceed by interpolating Q's outputs at $SimulationTime$s RTQ_1 and RTQ_2 for the time RTP_2.
- When instance P_3 is released at real-time instant RTP_3, it is determined that $RemResponseTime$ of already executing Q_2 and $WorstCaseExecTime$ and $Deadline$ of P_3 satisfies the condition (21). So P_3's execution is delayed and it remains in the waiting state for a duration d at the end of which Q's output values at $SimulationTime$ STQ_2 and STQ_3 are interpolated to calculate the input values at RTP_3 for the task P_3.
- For task instance P_4, waiting for instance Q_3 to finish execution is not possible because in that case instance P_4 will result in violating its deadline as shown by the dotted task P_4 ellipse. Therefore, in this case only the outputs at time RTQ_3 are used and are either extrapolated or used as is for P_4 instance to start execution immediately.
- For the case where instances of both Q and P are released at the same real-time instant (not shown in this figure), neither extrapolation nor interpolation is required and P's instance can directly use Q's outputs.

The next section presents some possible simplifications to the framework in order to reduce its time and space requirements when implemented in UPPAAL. The part of the work discussed up till now has been published in [3].

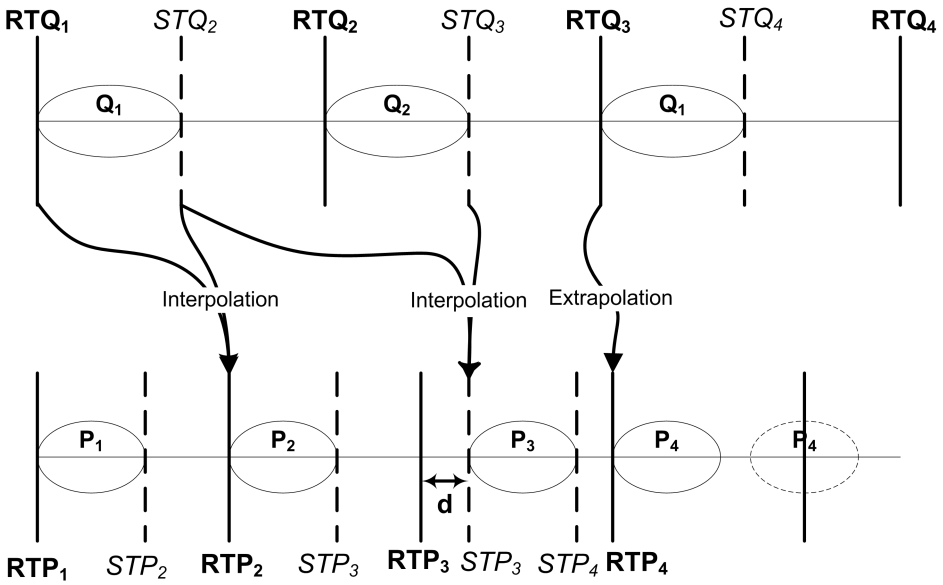

Figure 2.5: Handling of Non-binding constraints

2.2.2.2 Simplifications

We shall now see if it is possible to re-visit and simplify some of the concepts defined above in order to get an analysis framework that has same features as described above but is lesser in complexity.

1. Precedence Handling Simplification

Recall that in previous sub-section we argued that to handle precedence constraints the *SV* automaton needs to calculate $RemResponseTime$ of all precedent tasks using equations (22), (23) and (24). Consequently the automaton needs to define a clock variable to continuously maintain $TimeSinceActive$ for each precedent task. Since number of clock variables are a major reason for increase in time and space requirements of UPPAAL's verification engine [47], we wondered if it is possible to reduce the requirement of these clock variables. This leads us to the question: do we need to calculate the $RemResponseTime$s of *every* precedent task while analyzing a dependent task for real-time schedulability? Fortunately, the answer to this question is negative, that is, we do not need to calculate $RemResponseTime$s of all

the precedent tasks of the task under analysis. In the following, we shall fist prove that a dependent task does not need to wait for any precedent task, binding or non-binding, that is idle at the time of its arrival. Consequently, we do not need to calculate $RemResponseTime$ of any idle precedent task.

Proof: *No instance of a task under schedulability analysis is ever required to wait for any precedent tasks that are idle at the time of its arrival.*

To prove the above statement, we refer to a couple of facts given below

1. *Fact 1*: The condition (25) serves as the pre-conditions which a precedent task that needs to be waited upon must satisfy
2. *Fact 2:* In real-time simulations, the simulation-time of a task must keep ahead or be at least equal to the real-time at all times

We claim that the condition (25) of precedent task's simulation-time being less than the simulation-time of dependent task is only true when a precedent task is *active* at the time of dependent task's arrival. Proof of our initial statement logically follows the proof of this claim. Hence, if we can prove this claim then the inteded statement also stands proven. The arguments provided below prove our claim,

1. Consider a task S that has reached a simulation-time t_2 by starting a numerical integration step at an earlier time t_1
2. *Fact 2* implies that it is not possible for the master to pass through time instant t_2 and execute any task other than S at some real-time instant t_3, where $t_3 > t_2$, without first executing task S's next instance at t_2 such that this new instance will bring S's simulation-time to a value which is either greater than or equal to t_3
3. This means that at real-time instant t_3, task S can either be *idle* having already reached some simulation-time greater than or equal to t_3 or it can be *active* such that the active instance will take S's simulation-time from t_2 to a value greater than t_3

4. The conclusion in (iii), reached for real-time instant t_3 and task S, can be generalized for any real-time instant and any task in a simulation. So, in general terms, at any real-time instant t during a simulation, every task in that simulation is either active with its simulation-time less than t or is idle having reached a simulation-time greater than or equal to t

5. This generalized result can again be re-stated for the special case of tasks that are related with precedent constraints as,

   At any real-time instant at which a dependent task is released, every precedent task is either active with its simulation-time less than that of dependent task's simulation-time or is idle having reached a simulation-time greater than or equal to that of dependent task.

   This conclusion is same as our original claim and proves that *only active precedent tasks have simulation-time less than a dependent task at the time of dependent task's arrival*

Fig. 2.6 graphically shows how the simulation-times of active and idle precedent tasks are related with simulation-time of dependent task. The figure is similar to fig. 2.5 in that it shows execution traces of tasks P, Q and R over their timelines. The timelines represent continuous real-time and are drawn such that they are in sync with each other which means that any vertical line cuts the timelines at exactly same real-time instant. Simulation-times are also shown on the same timelines, however, these are not continuous and increment in discrete steps whenever a task finishes execution. Solid vertical lines denote the release time of a task as well as serve as the deadline for previous execution of the same task. Other conventions regarding real-time instants and simulation-time instants are also same as in Fig. 2.5, that is,

- The label for real-time instants of a task start with *RT* while simulation-time instants start with *ST*

- Each time instant shown has a subscript and the real-time and simulation-time of a task with same subscript denote the same time-instant value

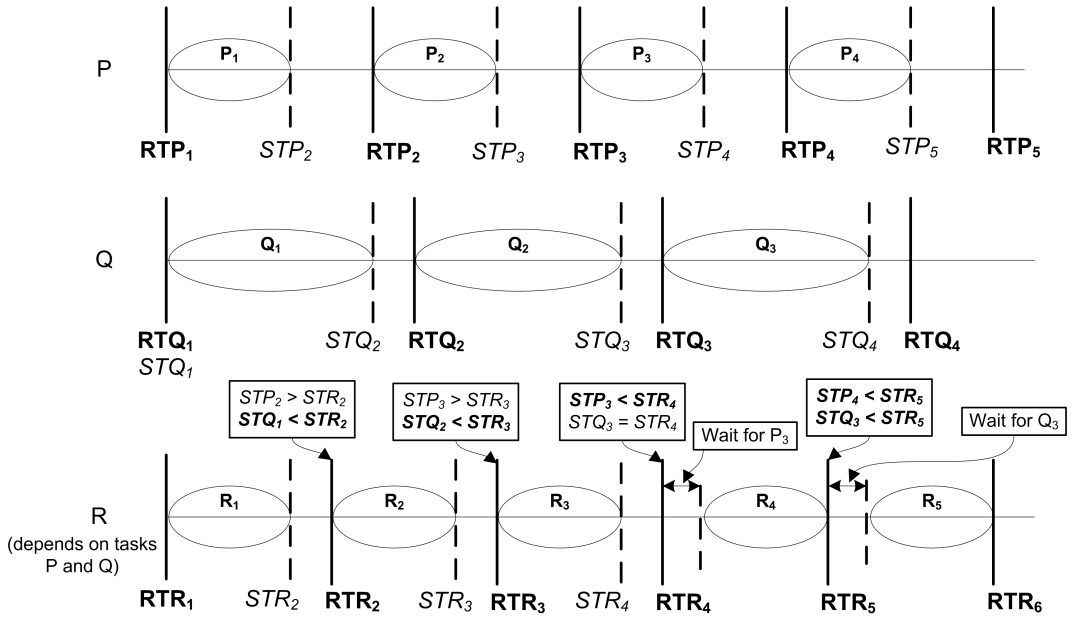

Figure 2.6: Only active precedent tasks satisfy pre-condition (25)

While reading Fig. 2.6, it is important to remember that a simulation-time with subscript i might appear earlier in the timeline but it has a value that is actually equal to the real-time instant shown with subscript i on the same timeline.

Let us assume that task R is currently under analysis and it uses some inputs from two non-binding precedent tasks P and Q. The execution of R_1, the first instance of R, does not need any explanation as it simply uses default inputs. However, at time RTR_2 when R_2 is released for execution, the master needs to see if it can wait for any of the precedent task P or Q. The relationships between current simulation-time of task R with those of Q and P at release of each R's instance other than R_1 are also shown in the figure. These relationships, according to condition (25), determine which precedent tasks need to be checked by the master for a wait time calculation. It is clear that at each instant of R's release, our claim that simulation-time of only active precedent task can be less than that of dependent task stands true.

Now that we have established that $RemResponseTime$ of only active precedent tasks is relevant for the purpose of schedulability analysis, a careful look reveals that we can do away with some active precedent tasks as well. It is easy to note

that in case of binding precedent tasks, calculating the $RemResponseTime$ of only one precedent task, which is active at the time of dependent task's arrival and is going to take the longest time to finish, is sufficient since it gives master the required information to calculate the mandatory waiting time. Thus we can safely disregard the binding precedent tasks that are active but are going to finish earlier than some other binding task.

This is where our effort to simplify precedence handling to reduce the required number of clock variables ends. Unlike in the case of binding constraint, where only the longest running binding precedent task was important, we may need to test $RemResponseTime$ of every non-binding precedent task for condition (21). This is necessary since there can be a situation where in order to ensure the dependent task's deadline, it is only possible to wait for few non-binding precedent tasks to finish while ignoring other precedent tasks with higher $RemResponseTime$s.

2. Task Type Definition Simplification

Recall that in section 2.2.2.1, the sole purpose of including $SimulationTime$ attribute in Task type definition was mentioned as to determine which precedent tasks need to be checked for wait time calculation of a dependent task. But now the above discussion about idle and active precedent tasks leads us to conclude that we do not need to maintain $SimulationTime$s of the tasks since the purpose for which these were used was automatically served by considering only those precedent tasks that are active at the time of arrival of task that is under analysis. At this point, we can safely strike the $SimulationTime$ parameter out of task type definition so now we have just four attributes in it; the task $WorstCaseExecTime$, task $WorstCaseCancelTime$, $Deadline$ and the $CurrentIntegrationInterval$. But recall again that $CurrentIntegrationInterval$ was only used to maintain $SimulationTime$ of tasks and to calculate $TimeToNextArrival$ of *idle* precedent tasks. Now $SimulationTime$ is not needed and there is no need to check $TimeToNextArrival$ as well as we do not need to check the *idle* precedent tasks any more. Therefore, $CurrentIntegrationInterval$ is no longer required and can be removed from Task type definition. So, the final task type definition will have just three indispensable attributes of $WorstCase$-

$ExecTime$, $WorstCaseCancelTime$ and $Deadline$. The new simplified definition of task type is then just a 3-tuple $\langle WCET, WCCT, DL \rangle$.

***Precedent Task sets*: A Solution to a problem with the proposed simplifications** In the last sub-section, we have concluded that there is no need to maintain simulation-times of every precedent task and it is sufficient, at the start of execution of a dependent task, to check only the precedent tasks that are active at that time. There was no other qualification mentioned in the discussion leading to this conclusion, therefore, *all* active precedent tasks are qualified to be considered for waiting time calculation. However, as we will see below, for a particular system state, considering all precedent tasks for waiting time calculation may not only be redundant but can also lead to incorrect schedulability analysis. In the discussion below, we will explain this special system state and the reason why it is problematic to consider all active precedent tasks when system is in that state. Afterwards, we will propose a solution using the concept of *Precedent Task sets*.

The Problematic System State Recall that in section 2.1 we said that our system model allows multiple instances of the same task type to be active at the same time and that a task waiting queue keeps track of these multiple active instances. This system feature may lead to the *problematic system state* where some active precedent tasks should not be considered for waiting time calculation. The problematic system state is defined as the state having following two particular properties,

1. There are one or more instances of dependent task waiting in the queue
2. There is at least one active precedent task in the system whose simulation time (or in case of sporadic precedent task, release time) is greater than the simulation time of some dependent task instance in the queue

Here, we present an example scenario which results in the problematic system state. As mentioned in section 2.1, this system state can be reached when the scheduling algorithm optimistically releases new instances of a task without waiting for previous instance to finish. Consider there are three tasks in a system with following attributes,

Task P:	**Task Q:**	**Task S:**
Periodicity: 1.2 msec	*Periodicity:* 600 msec	*Periodicity:* Sporadic
Framelength: 2.4 sec	*Framelength:* 1.2 sec	*WCET:* 300 msec
WCET: 200 msec	*WCET:* 500 msec	
WCCT: 50 msec		

Assume task P depends upon the task Q and an event handling sporadic task S and that we are currently analyzing the schedulability of task P.

We know that in real-time simulations, the scheduler or simulation master must synchronize the simulation-time of a task with real-time at the end of each of the task frame. This means that in our example scenario, the master needs to synchronize the two times at the end of each 1.2 second interval. We assume that the master in our example system schedules tasks optimistically and, between the time-instants where it needs to synchronize with real-time, it runs the simulation as fast as possible.

We shall define the current system state by the current set of active task instances, both analyzed and precedent. For the dependent task type that is under analysis, we differentiate between two stages of active task instances: one that is currently executing and others that are waiting in a queue. Therefore, we have three kinds of active task instances that define the system state: *Executing task instance*, *Tasks in waiting queue* and *Active precedent tasks*.

Let us now assume a particular simulation scenario and see how the system state changes as the simulation proceeds. The following simulation trace shows the task instances coupled with the simulation-time value at the time of their release, also referred to as release-time.

- At the start, instances P_1 and Q_1 are released at simulation-time = real-time = 0.0

  State 0 at $ST = RT = 0msec$**:**

  - *Executing task instance:* (P_1, 0.0)
  - *Waiting task queue:* EMPTY
  - *Precedent task list:* (Q_1, 0.0)

- The master takes the next steps almost instantly, proceeding the simulation-time to 600 msec and scheduling the next Q instance optimistically.
  State 1 at $ST = 600msec$ **;** $0.0sec < RT$ **:**
  - *Executing task instance:* (P_1, 0.0)
  - *Waiting task queue:* EMPTY
  - *Precedent task list:* (Q_1, 0.0), (Q_2, 0.6)

- Now as the master waits for real-time to catch up with simulation time at 1.2 secs assume that an event "s" occurs when real-time reaches 200 msec triggering an event handling task S. Since P is dependent on task S, the master now needs to rollback its simulation-time back to 200msec, canceling all tasks that were released at simulation-time greater than 200msec. The master also cancels the currently executing P instance and while the cancellation is in progress, it adds two more P instances to the queue; one which takes P's simulation-time up to the event-instant and the second which further proceeds P's time till the end of canceled integration interval. Note that the second P instance must be preceded by an instance of binding precedent task S. The system state at this point is,

  State 2 at $ST = 200msec$ **;** $RT = 200msec$**:**
  - *Executing task instance:* (P_{cancel}, 0.0)
  - *Waiting task queue:* (P_1, 0.0), (P_2, 0.2)
  - *Precedent task list:* (Q_1, 0.0), (S_b, 0.2)

- The master again proceeds the simulation-time without waiting for previous task instances to finish and schedules the next tasks optimistically,
  State 3 at $ST = 600msec$ **;** $200msec < RT < 250msec$**:**
  - *Executing task instance:* (P_{cancel}, 0.0)
  - *Waiting task queue:* (P_1, 0.0), (P_2, 0.2)

– *Precedent task list:* (Q_1, 0.0), (S_b, 0.2), (Q_2, 0.6)

At this point it can be easily checked that system states 2 and 3 in the above simulation scenario exhibit the two properties defined for a state to be considered problematic. Thus, states 2 and 3 both are problematic system states.

To understand the problem with such system states, recall that while presenting precedence handling simplifications we proved that all precedent tasks that satisfy condition (25) or (26) *must* be active at the time when dependent task starts execution. As a result of this proof, it was decided that we do not need to evaluate either condition (25) or (26) explicitly and only need to see if a precedent task is active at the time of waiting time calculation and then proceed with its $RemResponseTime$ calculation followed by evaluation of condition (21). But the observations made below show that when the system is in problematic state the reciprocal of the property proved above is not true. That is, if the system is in problematic state then there can be some precedent tasks that are although active but still do not satisfy condition (25) or (26). The said observations can be made by considering system state 2 or 3 described earlier,

- **At State 2:** Tasks' release times show that for task instance P_1, both active precedent tasks Q_1 and S_b do not satisfy conditions (25) and (26) respectively and therefore should not be considered for waiting time calculation.
- **At State 3:** The precedent tasks Q_1 , S_b, and Q_2 do not satisfy conditions (25) or (26) for dependent task P_1 while Q_2 does not qualify condition (25) for P_2. Again, these precedent tasks should not be considered for waiting time calculation despite of them being active at the time.

Hence evaluating all active precedent tasks at the time of dependent task's arrival may lead to incorrect schedulability analysis and, therefore, the analysis approach needs to be rectified accordingly.

Precedent Task sets The above discussion suggests that we need to differentiate between a precedent task that was released before the release of a particular dependent task instance and the one that was released afterwards. An obvious approach

for the sporadic precedent tasks is to tag each released task with its release time and evaluate condition (26) to decide if an active precedent task should be considered for waiting time calculation or not. This solution, however, is not possible because of an implementation issue in UPPAAL. The *clocks* in UPPAAL are special variables which cannot be saved in any other kind of variable. This means that the simulation clock value at release instant of an task cannot be saved as a tag, making this solution unimplementable. Another possible solution is to maintain a separate clock for each task to maintain every task's simulation-time. Simulation-times then can be used to see if a precedent task satisfy condition (25) for a dependent task instance in the queue. However, we still want to avoid simulation-times as maintaining these clocks for all precedent tasks as well as for the task under analysis adds unnecessary complexity to SV.

A simpler solution to differentiate among precedent tasks is to divide them into sets such that each set is composed of only those tasks whose release-times lie either between the release times of two consecutive dependent task (say P) instances in the queue or are greater than that of last P instance in the queue. For the case of active non-binding precedent tasks, let us name these different sets as $ANBTaskset_k$, where $0 \leq k \leq maxQsize$. Here, $maxQsize$ is a pre-defined parameter that defines the maximum size of the queue that holds the waiting dependent task instances. Each non-binding task, when becomes active, is then placed in the set $ANBTaskset_i$ where i is the current number of P instances waiting in the queue and therefore $ANBTaskset_i$ represents a set of active non-binding precedent tasks which became active after the i^{th} P instance in the queue but before the $(i+1)^{th}$ P instance. Clearly, i cannot exceed $maxQsize$ and so the number of $ANBTasksets$ is also bounded by $maxQsize + 1$ for queue sizes ranging from 0 to $maxQsize$.

Also, note that the need for differentiating between active precedent tasks means that we have to change the handling of active binding precedent tasks as well. Now we cannot accomplish our analysis goal by keeping track of just one active binding precedent task, i.e. the longest-running one. Rather, just like non-binding precedent asks, we need to divide the binding precedent tasks into task sets termed as $ABTaskset_k$, where again $0 \leq k \leq maxQsize$. Again, members of each binding precedent task set $ABTaskset_i$ have the property of having been arrived at a time when i instances of

P are already waiting in the queue. This means that we need to keep track of multiple active binding tasks such that each tracked binding task is the longest-running one selected from a set of active binding precedent tasks. The longest-running task from any $ABTaskset_n$ is referred to as $LRBindingTask_n$.

Although there can be multiple P instances waiting in the queue for execution, but the waiting time calculation is only done for the P instance which is at the head of the queue. This means that every waiting time calculation depends only upon the precedent tasks in $ANBTaskset_0$ and the $LRBindingTask_0$; since these are the only active precedent tasks that arrived before the P instance at the head of the queue. After performing the waiting time calculation, as the P instance is serviced out of the queue, the $ANBTAskset_0$ and $LRBindingTask_0$ are cleared and the precedent tasks in $ANBTaskset_m$ and $LRBindingTask_m$ are moved to $ANBTaskset_{m-1}$ and $LRBindingTask_{m-1}$ respectively.

Figure 2.7 helps in visually explaining the implementation of precedent task sets. The problematic states 2 and 3 are shown in the figure but with one difference; the active precedent tasks are now associated with a *TasksetNo* tag instead of their release times. This new tag, maintained with each active precedent task instance, has the value equal to the number of P instances waiting for execution at the time of corresponding precedent task instance's release. We termed this tag as $TasksetNo$ since it represents the task set the associated task belongs to. The tag also represents the number of P instances in the queue that should be executed before the precedent tasks in this task set can be considered for waiting time calculation. The $TasksetNo$ of each active precedent task decrements as the P instances leave the queue for execution and so a time comes when its value becomes equal to *zero* indicating that the this task-set can now be considered for waiting time calculation of the current P instance at the head of the queue.

The sequence of actions performed to implement precedence handling using the concept of precedent task sets is explained below as our example system moves from State 3 to State 6 shown in figure 2.7.

- **State 3:** $ST = 600msec$; $200msec < RT < 250msec$:

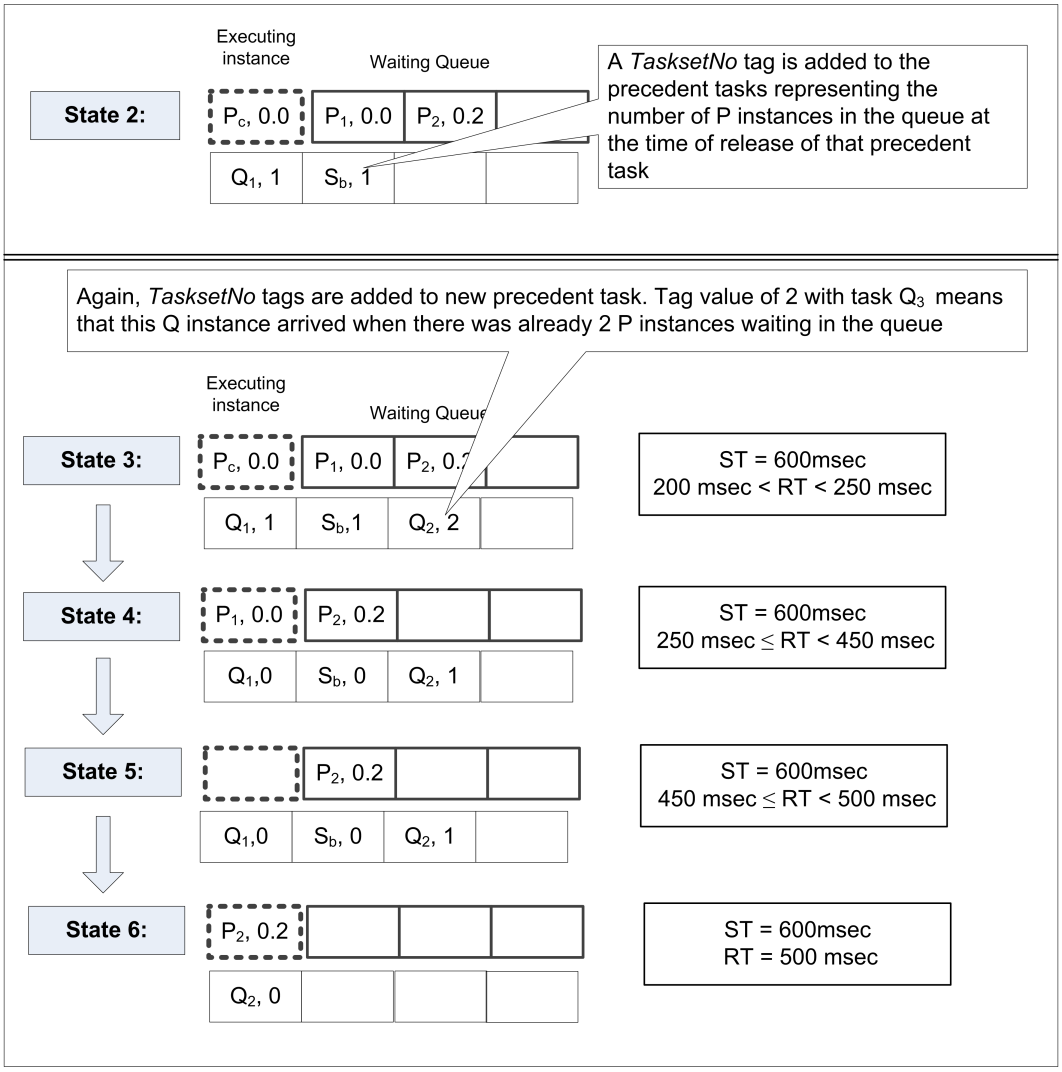

Figure 2.7: Solution to the problem of determining which active precedent tasks to consider using Task-sets

- State 3 ends when P_c finishes execution at $RT = 250msec$
- Waiting time for P_1 is then calculated without considering any precedent task since none of them has *TasksetNo* tags as zero.

- **State 4:** $ST = 600msec$; $250msec \leq RT < 450msec$:
  - P_c is serviced out and P_1 goes into execution
  - *TasksetNo* tags of precedent tasks are decremented by 1.
  - The updated *TasksetNo* of Q_1 and S_b is now 0, therefore, these tasks becomes eligible to be considered for wait time calculation of next P instance, P_2

- **State 5:** $ST = 600msec$; $450msec \leq RT < 500msec$:
  - P_1 is serviced out at $RT = 450msec$
  - $RemRsponseTime$ of Q_1 and S_b is evaluated to check if execution of P_2 should be delayed. Since S_b was released when $RT = 200msec$ and S_b's WCET is 300 msec, P_2 must wait for another 50 msec.

- **State 6:** $ST = 600msec$; $RT = 500msec$:
  - P_2 goes into execution
  - At $RT = 500msec$ both Q_1 and S_b finishes execution and so they are deleted from the list of active precedent tasks
  - *TasksetNo* tag of remaining active precedent task, i,e, Q_2 is decremented by 1
  - Q_2 can now be considered for delayed execution of any future instance of task P

2.2.2.3 Other Considerations in *SV* Automaton Design

Scope

The timed automaton that we define for schedulability analysis is termed as a *Schedulability Verifier (SV)* as it basically verifies schedulability of a task type and checks

if it is possible for any of its instance to have a response time greater than its assigned deadline, and therefore is unschedulable. The SV described here is actually a template automaton that needs to be instantiated for schedulability analysis of each task type. Therefore, it is obvious that the schedulability analysis performed by any instantiated SV checks the schedulability of only a single task type.

Like in the FY framework [21], SV encodes the problem of unschedulability as a reachability problem but the difference between SV and the automaton defined in [21] is that the SV checks all the task instances of the task type it is instantiated for and an *ERROR* state is reached whenever any one of the task instance is determined to be violating its deadline. Whereas the authors of [21] defined an automaton that checks a single instance of a task type for unschedulability which was selected non-deterministically for the analysis.

This SV feature of being able to analyze one task type at a time means that the analysis procedure has to be repeated for every task type in order to ensure the schedulability of the entire system. However, an instantiated SV analyzes the entire simulation scenario in a single step, that is, all the instances of the analyzed task type that are executed in the given simulation scenario are checked for schedulability in single analysis iteration and any task instance can be declared as unschedulable if it causes the SV to reach the ERROR location. If a task is found schedulable then we can proceed with the next task in the topological sorted order of precedence DAG. But before proceeding to next step, the SV needs to save an important piece of information from current analysis step. It needs $WorstCaseRespTime$ of the analyzed task type in order to use it in the analyses of its dependent task types as discussed earlier. Therefore, if all the task instances are found schedulable then a companion automaton, termed *RTCalc* automaton, keeps track of response times and at the end gives us an integer value which is the ceiling of the maximum response time experienced among all the instances of the task under analysis, i.e.

$$WorstCaseRespTime(P_i) = \lceil max_{1 \leq i \leq n}(ResponseTime(P_i)) \rceil \qquad (27)$$

where n is the total number of instances of task type P executed in the analyzed

simulation scenario.

Number of Clocks

There are some constants defined in the each instantiation of SV which determine the maximum number of clocks defined in that instance of SV. These constants are,

- **maxQsize:** This defines the maximum number of instances of the task under analysis that can be in the queue at one time waiting for execution. A clock is required for each waiting task.
- **maxActiveNonBindingTasks:** This constant defines the maximum number of non-binding precedent tasks that can be active at any given moment. A clock is required for each active non-binding task.

A clock is required for each binding precedent task as well. But the maximum number of active binding precedent tasks that we may need to maintain is equal to $maxQsize$ as mentioned above. Therefore, the total number of clocks defined in a SV is,

$$nClocks = 2 \times maxQsize + maxActiveNonBindingTasks + 2 \qquad (28)$$

One of the two additional clocks is used to maintain the time since execution of the longest running precedent task while the other is used in SV for multiple purposes; e.g used in invariants to restrict the time spent in waiting or executing states or used as guards when a task finishes waiting or execution.

Clock Count Growth

The complexity of a timed-automaton greatly depends on the number of clock used in it. To determine the practical usability of any timed-automata based framework, it is important to know the factors that cause the clock count to grow and also understand how these factors affect the count. We shall now discuss the two parameters that according to equation 28 effects the number of clocks in our framework i.e. $maxQsize$ and $maxActiveNonBindingTasks$.

i. maxQsize This parameter defines the maximum number of instances of the task under analysis that can be waiting in the queue for execution. All the tasks in the queue will be of the same type but may be with different relative deadlines. The intuition says that since our system model assumes that we have a dedicated thread for each task, therefore there should be no waiting queue. This intuition is especially true if the scheduling algorithm is a conservative one so that the co-simulation master does not schedule future instances of a task unless all previous instances of that task type have finished their execution. However, if we consider a scheduling algorithm that can optimistically schedule task instances without waiting for earlier instances of the same type to finish; and combine it with our other three assumptions described in section 2.1 above then there can be a situation where waiting queue is required. These three system assumptions allow the tasks to have different frame lengths with no restrictions and also allow task cancellations with the possibility that the task cancellation process may need processor time for itself.

A frame of a task type can be single or multiple integration-steps long. Let us first consider the case where a frame of the task under analysis P is one integration-step long. Since in real-time simulations the simulation-time of a task must be greater than or equal to the wall-clock time at the end of each frame, therefore in case of tasks with one integration-step long frames, the deadline for each task instance is equal to one integration-step which in turn is equal to task periodicity. This means that that the real-time constraint dictates that a P instance must finish execution before arrival of next P instance and so, it seems, that queue will always be empty. There is, however, one exception. Suppose the integration interval of the currently executing P instance, p_1, is from t_1 to t_2. Now during the p_1's execution an event, that also affects the task P, occurs in the system at time instant t_e. A normal event handling mechanism will now cause the p_1's execution to be canceled so that after cancellation the task's simulation-time is again t_1. The scheduler will again schedule the same task but with integration interval from t_1 to t_e followed by a task with integration interval from t_e to t_2. This second task instance will incorporate the effects of the event that caused the cancellation. The frame length during this event handling doesn't need to change in terms of time interval but is temporarily

changed to two integration-steps long if measured in terms of integration-steps. These two task instances are scheduled by the scheduler at a time when instance p_1 is in cancellation process, so both of these two P instances will be queued for execution. So we see that even in cases of tasks with one integration-step long frames, there can be a situation where task instances are waiting in the queue. Specifically, to be able to handle one event during an integration interval, the $maxQsize$ must be equal to 2. This maximum length can, however, increase if we allow handling of multiple tasks during one integration interval. Formally, for a task whose frame length in one integration step,

$$maxQsize_1 = nEvents + 1 \tag{29}$$

Where,

- Subscript of $maxQsize$ depicts the frame length of task in terms of number of integration steps
- $nEvents$ is the number of events allowed to be handled in a single integration interval

However, the maximum queue size for tasks with frame lengths of more than one integration interval can increase; but only if the task instances are again scheduled optimistically. Since a designer can have maximum relaxation in task deadlines when all task instances or integration-steps in a frame have the end-of-frame as their deadlines. Therefore, theoretically it is possible for a scheduler to schedule a task with frame length of n integration-steps n times without violating any deadlines and so there can be an instant where waiting queue might contain all n task instances. Also note that any event occurring in the system after the scheduling of $m \leq n$ task instances can only affect the m^{th} instance. So, $maxQsize$ for the case of task with frame length of n integration-steps can be defined as the maximum queue size required to hold n waiting task instances with the possibility of event handling in the last instance, i.e.

$$maxQsize_n = (n-1) + nEvents + 1$$
$$= n + nEvents \tag{210}$$

Where subscript of maxQsize and nEvents are same as defined for eq 29

ii. maxActiveNonBindingTasks $maxActiveNonBindingTasks$ defines the maximum number of non-binding precedent tasks that can be active at the time of task instance's arrival. This parameter can be inferred from the precedence DAG given in task model by counting the number of non-binding incoming edges to the node that corresponds to the task under analysis.

Schedulability Analysis Stage: Before or After Scheduler Design? We have deliberately looked over one practical problem with $RemResponseTime$s up till now. Despite the fact that it is possible to calculate and maintain the $ResponseTime$s of all instances of all precedent tasks during analysis, it would not be possible for the co-simulation master at runtime to know the accurate value of the $ResponseTime$ of a precedent task instance which is currently in Waiting state. This is simply because $ResponseTime$ of each instance of a task is not a constant and depends on the state of the system during the time it is active. Therefore, the precise value of $ResponseTime$ of an instance of a precedent task is not known when it is time for master to calculate its $RemResponseTime$ before scheduling a dependent task. Now, there are two ways to look at the position of our real-time schedulability analysis in the entire real-time co- simulation process.

- First, we can assume that a scheduler component of the master is designed first and then we analyze it for real-time schedulability. In this case, the scheduler has no other choice but to predict the $ResponseTime$ of a precedent task in waiting state. This prediction can be based on the available history of the task instances and can be estimated as the maximum $ResponseTime$ that has been encountered so far. But such a prediction is not risk free as there is no guarantee that the $ResponseTime$ of current instance will not exceed the historical maximum $ResponseTime$. Alternatively, to be on the safe side, the scheduler

may assume the *ResponseTime* of waiting task to be equal to its *Deadline*, assuming that it will not violate its deadline. This approach is safe but can result in a very pessimistic estimate of $ResponseTime$ for tasks with higher deadlines but lower $WorstCaseExecTime$ and consequently affects the accuracy of the simulation negatively. The equation 22 in this case will be re-written as follows,

$$RemResponseTime(P_{active}) = Predicted\_RT(P_{active}) - TimeSinceActive(P_{active}) \quad (211)$$

For the analysis step in this first scenario, we need to emulate the $ResponseTime$ predicted by the actual scheduler and cannot use the $ResponseTime$s of precedent tasks found in the previous analyses steps.

- Alternatively, we can view the analysis stage to be taken place before the scheduler design. This will allow us to use the previous analyses of the tasks that are defined as precedent to the one under analysis. We can find the worst-case $ResponseTime$ ($WorstCaseRespTime$) of a precedent task as we now have a task's $ResponseTime$s for the entire simulation scenario. Since the $ResponseTime$ of particular task instance in a scenario can never surpass its $WorstCaseRespTime$ in that scenario, therefore we can safely use the $WorstCaseRespTime$ in place of predicted $ResponseTime$ in equation above and so that it becomes,

$$RemResponseTime(P_{active}) = WorstCaseResponseTime(P_{active}) - TimeSinceActive(P_{active}) \quad (212)$$

These $WorstCaseRespTime$ of each task can then be provided to next stage of scheduler design which can use these to make more accurate scheduling decisions. But if a task in a particular scenario is found to be unschedulable because of some non-binding precedent tasks, this is probably because of the fact that although $RemResponseTime$s of the non-binding precedent tasks were calculated using their $WorstCaseRespTime$s and it was checked that these calculated $RemResponseTime$s will not result in a violation of deadline using condition 21 but some event occurring after

the calculation of non-binding wait time rendered this wait time it to be too big and caused the unschedulability. To schedule in such a scenario, we may need to adjust the calculation of $RemResponseTime$s in the analysis stage such that the non-binding waiting time calculated at actual runtime can never cause unschedulability. Since the non-binding waiting time is equal to the largest $RemResponseTime$s value for which condition 21) is true, therefore, we need to adjust the $RemResponseTime$s of problem causing precedent tasks to a higher value so that for them the condition 21, after adjustment, evaluates as false. The $RemResponseTime$s calculation in equation 212 suggests that the only parameter that can used to increase the $RemResponseTime$ values of a precedent task is its $WorstCaseRespTime$ for this scenario. Therefore, what we need to do is to find the problem causing precedent task and increase its $WorstCaseRespTime$ value such that it doesn't satisfy condition 21 at that point of time anymore. In one of the subsequent sections, it is mentioned that we actually maintain a list of active non-binding precedent tasks in descending order of their $RemResponseTime$s. This increasing of $WorstCaseRespTime$ will continue for each subsequent task in the sorted list until the simulation scenario becomes schedulable. Let us introduce a new term, $ResponseTime$ for Schedulability ($RespTmForSchedulability$), and by $RespTmForSchedulability$ we mean the value that should be used instead of $WorstCaseRespTime$ in eq 212 for determining its $RemResponseTime$ to ensure schedulability of its dependent task in a particular simulation scenario. RespTmForSchedulability of a task can be equal to a task's $WorstCaseRespTime$ in that scenario or it can be a tuned up $WorstCaseRespTime$. So the equation that will actually be used for the calculation of $RemResponseTime$s will be,

$$\begin{aligned} RemResponseTime(P_{active}) = & ResponseTmForSchedulability(P_{active}) \\ & - TimeSinceActive(P_{active}) \end{aligned} \tag{213}$$

Instead of providing $RespTmForSchedulability$ of each task for each scenario separately to the scheduler design stage, it seems more practical to calculate the maximum of these $RespTmForSchedulability$ of a task in every simulation scenario which can be termed as the maximum $RespTmForSchedulability$ ($MaxRespTmForSchedulability$) of a task. This $MaxRespTmForSchedulability$ of each task can then be supplied as a parameter to the scheduler, which can then use these in its

$RemResponseTime$ calculations with confidence. We would like to take the second view for our analysis effort and therefore, the scheduler in this case will be parametric and will use the $MaxRespTmForSchedulability$ values of the tasks supplied by the analysis stage in making runtime decisions.

Finally, we are in a position to describe the construction of the timed automaton that will analyze the real-time schedulability of a real-time co-simulation. Although real-time scheduling in a system where each task has a dedicated execution thread seems hardly any problematic but this seemingly trivial scheduling problem becomes interesting when we allow tasks to change periodicities at runtime and also take precedence constraints into account in addition to the usual timing constraints. The precedence constraints expressed as DAG can be used to topographically sort the tasks which can be considered equivalent to a sorting according to fixed priorities. It is therefore possible to use the same principle as in [21] to define a timed automaton for each task type that calculates the maximum response time of its instances. The main result of [21] was that if tasks are executed according to fixed priorities then the schedulability problem can be solved for each task type separately and for a single processor can be encoded simply as the reachability problem of a timed automaton using only two extra clocks (other than the ones already used in task automata definition).

2.2.2.4 The Schedulability Verifier (SV) Automaton Construction

Before proceeding with description of SV construction, let us first describe few data structures that are used in the definition of SV.

- *queue*, a queue whose maximum size is defined by a SV parameter $maxQsize$ and has elements which are a tuple $(clock, int)$. This queue keeps the instances of P while they are waiting for execution. The $clock$ member of each queue element represents the time spent after the associated task instance became active while int is the relative deadline value set by the co-simulation master at the time of release of this task instance.

- *ANBTasks*, an array that holds all the active non-binding precedent tasks. As

discussed before, the active precedent tasks are divided into a number of task sets, therefore for each element of $ANBTasks$ array, we need to maintain a 3-tuple $(ANBTasksetNo., TaskId, clock)$. The array is maintained in a doubly sorted manner with elements first stored in ascending order of their $ANBTasksetNo$, and then the elements with the same $ANBTasksetNo$ are stored in descending order of their remaining response times.

- *ABTasks*, an array that holds active longest running binding tasks for each $ABTaskset$. Since number of $ABTasksets$ is bounded by $maxQsize$, therefore, size of $ABTasks$ array is also equal to $maxQsize$. The fact that number of $ABTasksets$ and size of $ABTasks$ have the same value at all times allows us to use the index of each $ABTasks$ entry as its $ABTaskset$ number. Therefore, the elements of the array are a tuple $(TaskId, clock)$, where the $clock$ again maintains the time spent by the associated task while being active.

A timed automaton consists of nodes called locations and edges denoting transition between locations. Below is the description of all the locations and the edges between them as defined in the SV. We shall refer to the task type which is under analysis as P while the precedent tasks that will feature in our discussion shall be referred to as Q. Similarly, the automaton location names in the SV that are subscripted by p refer to the task under analysis whereas other subscripts are used for locations dealing with precedent tasks. See fig. 2.8 below for a graphical representation of the automaton.

1. Idle Location To start with, the SV is in $Idle$ location. At this location there are two possibilities for SV to change its state.

1. If an instance of the task under analysis, i.e. P releases for execution. This event will cause the automaton to immediately move to $Executing\_p$ location where it will remain for a duration which is either equal to P's WCET or in case of task cancellation, the sum of the duration the task executed before cancellation and its WCCT.

2. If an instance of one of the precedent tasks releases. In this case, the SV moves to location named $Active\_q$. This location is where the SV remains until an

instance of P arrives or till all the active precedent tasks finish execution.

2. Active_q Location In $Active\_q$ location, the SV keeps track of all the active non-binding precedent tasks and maintains them in appropriate $ANBTasksets$. Moreover, it also tracks one, the longest-running, active binding task from each $ABTaskset$. The time spent by these precedent tasks while being active is measured by a clock variable, separate for each precedent task. The SV keeps a check on the clock associated with the active precedent task that has the largest response time ($WorstCaseRespTime$), calculated in previous SV analysis iterations (referred to as rq in figure 2.8). As soon as this clock surpasses its associated task's $WorstCaseRespTime$, the SV moves back to Idle location, clearing all precedent task lists indicating that there are no more precedent tasks that are active now.

The other outgoing transition from $Active\_q$ is taken when an instance of the analyzed task P is released with a relative deadline d. At this event, the SV moves to a new location called $BindingWait$.

3. BindingWait Location This is the location where the SV emulates the delay due to a binding precedence task that a co-simulation master must introduce before allowing the next instance of P to execute. The duration of this mandatory delay equals the remaining response time of the longest-running binding precedent task in $ABTaskset_0$. Therefore the action taken with all incoming transitions to $BindingWait$ location is to call $getBindingWait()$ function. This function checks the $LRBindingTask_0$, the longest-running binding precedent task for P's instance at the head of the queue, and returns its $WorstCaseResponseTime$ calculated in one of the earlier analyses steps as $LRBindingTask\_WCRT$

As mentioned in discussion of $Active\_q$ location, a clock variable for each precedent already tracks the time since that task is active. Therefore, using the clock variable corresponding to $LRBindingTask_0$ and the $LRBindingTask\_WCRT$ value returned by the $getBindingWait()$ function call, we can easily calculate the $RemResponseTime$ of $LRBindingTask_0$ according to equation 212. However, to realize this delay in UPPAAL, we didn't perform this $RemResponseTime$ calculation

but rather used the following invariant for $BindingWait$ location that ensures that the time spent in this location equals the $RemResponseTime$ of $LRBindingTask_0$. The invariant used is,

$$LRBindingTask_0.clk < LRBindingTask\_WCRT$$

where $LRBindingTask_0.clk$ is the clock variable that tracks the time since LR-$BindingTask_0$ is active

The two events that can cause SV to leave the $BindingWait$ location are,

- If clock $LRBindingTask_0.clk$ progresses to eventually cause the above invariant to become false. In this case the SV moves to check if it can wait for any active non-binding precedent task to finish execution.
- When the number of tasks in the waiting queue is one and if that only waiting P instance is canceled by the co-simulation master. The SV, in this event, moves to a location called $Finish_p$.

4. CheckNBIndexToWait Location This is a location which can only be arrived at via $BindingWait$ location. When SV exits the $BindingWait$ location due to the location invariant becoming false, it needs to check if there are any active non-binding precedent tasks that can be waited for before proceeding with the execution of next P instance in the queue. The outgoing transition from $BindingWait$, which is also incoming to $CheckNBIndexToWait$ location, performs this action of checking the active non-binding precedent tasks by calling $getNonBindingWait()$. This function call returns the index of a non-binding precedent task from the list of active precedent tasks. i.e. $ANBTaskset_0$. The task at the returned index has a remaining response time value such that it is the greatest by which the waiting P's instances can be delayed without violating any deadline. The $getNonBindingWait()$ also returns the $ResponseTmForSchedulability$ of the same non-binding precedent task.

$CheckNBIndexToWait$ location is defined as a committed location in UPPAAL which means that no time can be passed while the automaton is in this location. Therefore, the location is immediately exited by simply checking if the index returned by the $getNonBindingWait()$ function is a valid one which means that there is non-binding precedent task that is worth waiting for, or if the returned index is invalid, i.e. equals to -1, which shows that there is no non-binding precedent task to wait for. In the first case, the SV moves to $NonBindingWait$ location while in the second case, the SV moves to $Executing\_p$ location.

5. NonBindingWait Location This location is entered if the index returned by $getNonBindingWait()$ function call, described above, is greater than -1, i.e. valid. As mentioned earlier, this index refers to a non-binding precedent task, within $ANB\text{-}Taskset_0$, which has the largest $ResponseTmForSchedulability$ value, also returned by the same function call, such that it is safe for all the instances of P in the queue to wait for its completion. The invariant used is similar to the one in $BindingWait$ location but this time the above mentioned $ResponseTmForSchedulability$ is used on the right of the inequality instead of the $WorstCaseResponseTime$. The invariant, therefore, is,

$$NBTasks[index].clk < ResponseTmForSchedulability$$

There are three conditions on which the SV exits this location.

1. The waiting time expires, i.e the above invariant become false. In this case, the SV simply starts executing the P instance at the head of the queue by moving to $Executing\_p$ location.

2. A new instance of P is released such that it cannot tolerate to delay its execution till the expiration of the waiting time. This condition is tested by making transition to a new committed location, $CheckNewTaskDL$, which decides to move back to $NonBindingWait$ location if the condition is false or move to $Executing\_p$ if the condition is found true.

3. If there is only one task in the waiting queue and that P instance in the queue gets canceled. This will cause the SV to move to $Finish_p$ location.

6. Executing_p Location This location emulates the time spent while executing an instance of the task P. It can be entered either from $Idle$ location when an instance of task P gets released with no active precedent tasks or after exhausting all the waiting periods, binding and/or non-binding.

The SV remains in $Executing\_p$ location either for a duration that is equal to $WCET$ of P or, in case of the executing instance being canceled, for a duration that is equal to the sum of time spent before the cancellation and P's $WCCT$ (Worst Case Cancellation Time).

The exit from this location will cause the SV to move to $Finish\_p$ location

7. Finish_p Location $Finish\_p$ location is entered whenever an instance of task P, which is either at the head of the queue or is executing, got canceled or when a P instance finishes execution. As shown in figure 2.8 below, the cancellation can be done while waiting for precedent tasks to finish execution or while a task is executing. On the other hand, a P instance can complete execution in $Executing\_p$ location only. Therefore, the incoming edges to $Finish\_p$ location are from $BindingWait$, $NonBindingWait$ or $Executing\_p$ locations.

It is again a committed location and so the SV exits this location immediately and move to one of the three other locations; $Idle$, $Active\_q$ or $BindingWait$ based on the following three mutually exclusive conditions.

1. If there is no P instance in the queue AND there is no active precedent task, the SV moves to $Idle$ location

2. If there are no P instance in the queue but there are one or more active precedent tasks, the SV moves to $Active\_q$ location

3. If there are P instances waiting in the queue, the SV then moves to $BindingWait$ location

8. CheckNewTaskDL Location This location was mentioned above and described as a committed location that checks the deadline of a new instance of P, released while the system is waiting for a non-binding precedent task to finish, and determine if the new P instance can safely wait for the remaining of the non-binding wait period or not.

The only incoming edge to this location is from $NonBindingWait$ location while the two outgoing edges lead the SV to either $Executing\_p$ or back to $NonBindingWait$ based on the result of above mentioned condition being false or true respectively.

9. ERROR Location $ERROR$ is a special location which is only entered when an instance of P is found to be violating its deadline.

The SV enters $ERROR$ location only when a P instance actually passes its allowed deadline before completing its execution. That is to say that, even though it seems possible, we did not employ any predictive technique to foretell a deadline violation or unschedulability. The reason for not predicting the unschedulability is the uncertainty that comes with allowing task cancellations. Clearly, one or more cancellations of active P instances will alter any unschedulability prediction made for the canceled or the waiting P instances.

Therefore, the places where a P's instance's clock can surpass its deadline are either when SV is in $BindingWait$ location or in $Executing\_p$ location. It cannot happen in $NonBindingWait$ location because,

- Firstly, the SV moves to $NonBindingWait$ only when it determines that all the P instances already in the queue can safely wait
- The deadline of any new P instance is checked immediately to foresee any possible deadline violation. If a possible future deadline violation is found, the SV immediately moves to $Executing\_p$ location

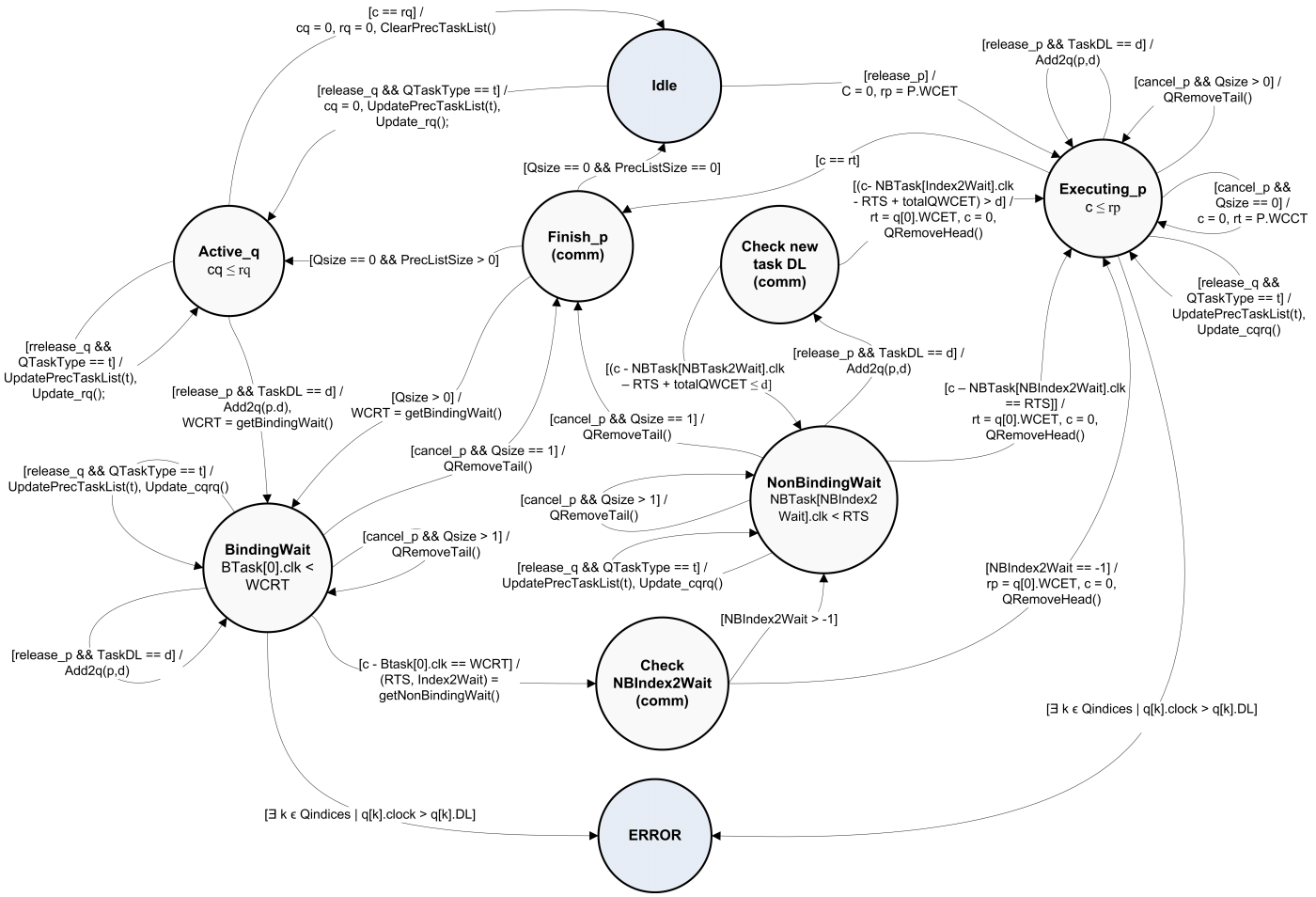

Figure 2.8: The Schedulability Verifier (SV)

2.2.2.5 The Cancellation Handler (CH) Automaton Construction

As mentioned before, the Cancellation Handler (CH) automaton mimics the decisions taken by a simulation master while handling cancellation of an active instance of task under analysis. Like SV, CH is also designed as a template automaton which needs to be instantiated for a particular task model. More specifically, transitions guards that are subscripted with q are to be replaced with transition guards referring to binding precedent tasks. Similarly, transitions which use a guard or action with subscript p are replaced with corresponding guard or action for the task under analysis.

As shown in figure 2.9, CH automaton has 4 locations. The description of each location and the transitions coming into or out of theses locations is given below.

1. Start Location $Start$ is obviously the initial location of the CH automaton. Here the automaton waits for release of a binding precedent task Q. A released task Q can result in cancellation of the latest active instance of analyzed task, P. To handle this scenario there are two transitions defined as coming out of $Start$ location. At the time of Q's release

- If $qSize > 0$ then this means that the latest active instance of P resides in the waiting queue and so the CH moves to $CancelLastQueued$ location
- If $(qSize == 0)\&\&(IsExecuting_p == true)$ then this shows that latest P instance is currently executing. In this case the CH automaton makes a transition to $CancelExecuting$ location

There is no transition for the condition when $(qSize == 0)\ \&\&\ (IsExecuting_p == false)$ because in this case there is no active instance of task P to be canceled.

2. CancelLastQueued Location As mentioned above, this location is reached when an instance of task Q is released for execution and $qSize > 0$. The location handles the cancellation of latest instance of task P that is present in the waiting queue. The decisions made during this cancellation handling involves checking two boolean conditions with three possible outcomes. The three outgoing transitions

corresponding to three outcomes and the enabling condition for each transition is described below.

- If no time has passed between release of last P instance and release of Q AND Q is not an sporadic task then a transition leads the CH back to $Start$ location without taking any cancellation action. This is because it is determined that latest instances of both P and Q are released at the same simulation time instant with Q not being sporadic means that its outputs at this instant are already available.
- If no time has passed between release of last P instance and release of Q AND Q is an sporadic task then another transition is defined that again takes CH back to $Start$ location. However, this new transition also performs the action of canceling the last P instance and then releasing another instance. This new P instance executes after the execution of Q as no Q outputs are available because of it being sporadic.
- If some simulation time has passed between release of last task P and release of task Q then this means that task P needs to be canceled and the canceled task is to be replaced with two P instances. One that takes P up to the instant where task Q was released and the second that progresses P after Q finishes execution. The transition defined in this case takes CH to $ReleaseExtra$ location

3. CancelExecuting Location As the name suggests, this location handles the cancellation of an executing instance of a task P. The three outgoing transitions from this location are similar to the three transitions going out of *CancelLastQueued* location. That is, all transitions are annotated with same actions and lead to same corresponding locations. However, the only difference is in transition guards where instead of checking the release time of last P instance, execution time of currently executing P is checked against release time of task Q.

4. ReleaseExtra Location The purpose of this location is to simply release an extra P instance and so the only transition out is back to $Start$ location releasing an instance of task P.

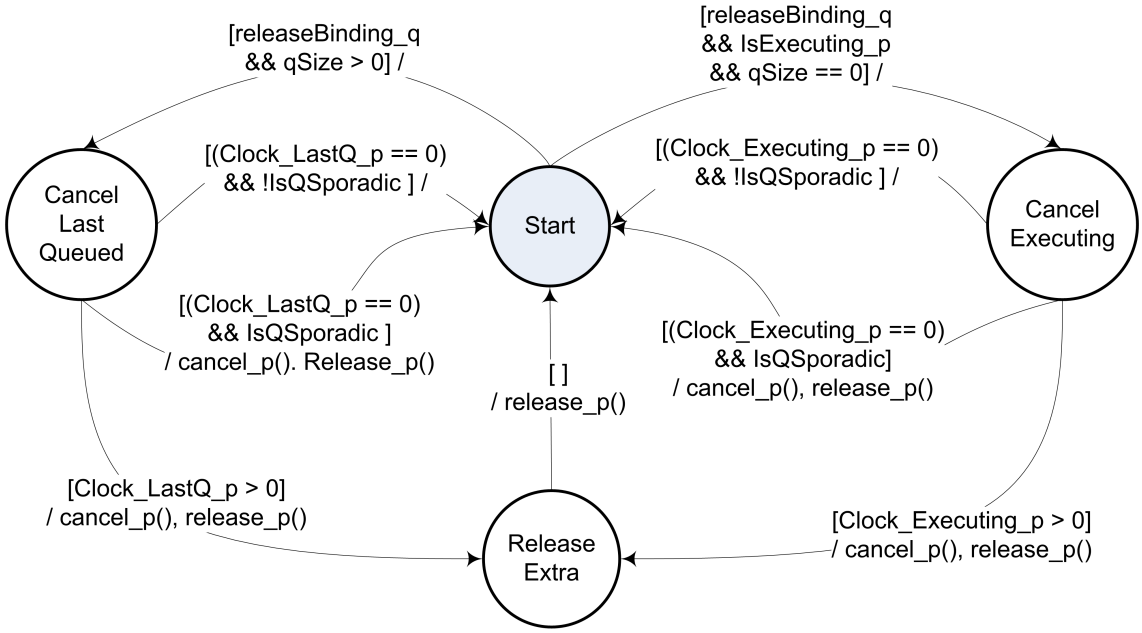

Figure 2.9: The Cancellation Handler (CH)

CHAPTER 3

FRAMEWORK IMPLEMENTATION

This chapter describes the implementation details of our proposed schedulability analysis framework. As mentioned earlier in section 1.3, the tool used for the implementation of our framework is Uppaal 4.1, 64-bit version (`www.uppaal.org`).

Chapter 2 outlined the main components of the proposed schedulability analysis framework. This design outline serves as the basis for actual framework implementation in Uppaal along with additions and modifications that are naturally required while translating any conceptual design into actual implementation. The most important part of our framework design was a couple of timed automata that we termed as *Schedulability Verifier (SV)* and *Cancellation Handler (CH)* (see section 2.2.2). Both SV and CH were defined as template automata that need to be instantiated for schedulability analysis of a particular task type. Recall that in SV and CH template designs, we followed the convention of using letter *P*, as standalone or as a subscript, to refer to the task type that is under analysis and *Q* for tasks that are defined as precedents to task *P*. For example, *release_p* was used as guard for transitions that were meant to synchronize with release event of the task under analysis. Since such synchronization between Uppaal automata is achieved through use of *Channels* (described below), our framework implementation will replace these *release_* guards with appropriate channel labels. A channel named *release_AnalyzedTask* is defined for synchronization of automata upon release of task under analysis and an array of channels, *release_PrecTask[]*, is defined to achieve synchronization for multiple precedent tasks. Each element of *release_presTask[]* array is used for synchronization over release event of one of the precedent tasks in task model. Hence, to instantiate SV or CH, all the transitions which mention *release_p* as the synchronizing event are replaced

by transitions that use *release_AnalyzedTask* as synchronization channel. Similarly, all transitions with *release_q* as firing event are replaced by a set of transitions where each transition is annotated with an element of *release_precTask[]* channel array.

Since only an instantiated automaton can be implemented, following steps must already be completed before proceeding with the our proposed framework implementation

- A task model has been defined
- A task type P has been identified in the task model as the one that will be analyzed for schedulability
- The SV and CH templates have been instantiated using task P and its precedent tasks Qs

A sample Uppaal implementation of an SV for analyzing schedulability of a task defined with two precedent tasks is shown in figure 3.1. Figures 3.2 and 3.3 are the zoomed in versions of the parts A and B of figure 3.1 respectively.

Apart from SV, CH and task automata that define the release pattern of the tasks, the implementation of schedulability framework contains some helper automata as well. These helper automata are invoked from SV to perform different actions just like function calls in other programming languages. Besides automata Uppaal supports function definition in the form of normal text based code as well. However, the difference between the two is the inability of text coded functions to handle clock variables. A clock variable is a special variable defined in Uppaal's automaton and can only be compared, reset or used as location invariants in an automaton definition. So helper automata are defined instead of simple function calls when the action to be performed requires access to clock variables.

Invocation of helper automaton requires a mechanism for communication between different automata. To achieve this communication or synchronization between automata Uppaal uses a concept of *Channels*. Synchronization is done by annotating the transitions in an automaton with *Synchronization labels*. These synchronization labels are of the form *c!* or *c?*, where *c* is the name of a channel. Two transitions of

different automata can synchronize if their guards are satisfied and if they are annotated with synchronization labels *c1!* and *c2?* respectively, where *c1* and *c2* evaluate to the same channel name. The transition labeled with *c!* can be thought of as a 'calling transition' and the one with label *c?* as the 'listening transition'.

A list of these helper automata along with a brief description of their purpose is given below. The implementation details of helper automata will be given in section 3.3

- **UpdatePrecedentLists:** This automaton, when invoked by the SV, updates the active precedent task lists. It goes through the lists of active non-binding and binding precedent tasks and removes those tasks whose time since being active has surpassed their corresponding worst case response times. The channel that performs synchronization between SV and *UpdatePrecedentLists* is named as *UpdatePrecLists*

- **Add2ActivePrecednts:** This automaton is called from SV whenever a precedent task is released for execution. The automaton places the precedent task instance at an appropriate precedent task list with the appropriate *TasksetNo*. It uses *add2PrecedentLists* channel to listen to this call from SV

- **GetNBWait:** *GetNBWait* is called by SV to determine the time that the next task *P* waiting for execution can allow for active non-binding precedent tasks to finish. *getNonBindingWait* channel is used to listen to this call from SV

- **RTCalc:** *RTCalc* helper automaton computes the response time of instances of task *P*. The SV calls this automaton when one of the following events occur,

  - When an instance of task *P* is released for execution. This call is made through *resetRT* channel and is meant to ask *RTCalc* to start response time calculation for this *P* instance

  - When a *P* instance finishes execution. This call is made to stop *P*'s response time calculation. The channel used to perform this action is called *SaveRTifMax*. This channel name signifies the fact that on this call *RTCalc* not only stops response time calculation but also saves it as worst case response time of the task type *P* if it is the maximum among all the response time calculations up till now

– When an instance of task *P* gets canceled either during execution or while waiting for execution. This call is made through *cancelRTCalc* channel and causes the *RTCalc* to stop calculating the response time for the canceled task

The next two sections describe implementation details of the SV and CH followed by a section describing implementation of helper automata.

3.1 Schedulability Verifier (SV) Implementation

A SV implementation for schedulability analysis of a task with two precedent tasks is shown in figures 3.1, 3.2 and 3.3. For ease of explanation we have colour coded the SV so that specific type of transitions and locations are easily identifiable.

3.1.1 SV Locations

The locations in SV are shown with two colours only. The green coloured locations are same as the ones defined in section 2.2.2.4 while describing SV design. These locations are also numbered in the same sequence as they appeared in the above referred section so that the details about any of these locations can be easily traced by the reader. The other type of locations, shown in magenta, are intermediate locations that are introduced to perform required actions in specific order while transitioning from one green location to the next. Note that all magenta locations are either marked as *U (urgent)* or *C (committed)* which signify that these are only transit locations since time is not allowed to pass when an Uppaal automaton is in committed or urgent location.

3.1.2 SV Transitions

We shall now discuss the SV implementation by focusing on one distinctly coloured set of transitions at a time.

Magenta: The magenta coloured transition sequences in figures 3.1, 3.2 and 3.3 represent the actions performed when a precedent task *Q* is released for execution. These sequences wherever they appear in SV are all similar except the one that originates from initial or *Idle* location which differs in the final stage reached at the end of transition sequence. While every other magenta transition sequence ends in the same location from where it starts, the one that starts from *Idle* locationl leads to *Acivte_q* location. Every magenta sequence include two intermediate locations before reaching the final location. As mentioned before, these intermediate locations are introduced just to perform the requisite actions in correct order before reaching the final location. The first magenta transition in the sequence listens to *release_q* channel to synchronize with the release event of task *Q*. For the case of *Idle* location, the first transition also resets clock variable c_q and sets the variable r_q equal to the response time of task *Q*. The SV uses c_q and r_q variables in *Active_q* location by using the expression $c_q < r_q$ as the location *invariant*. A location invariant in Uppaal specifies a condition that must be satisfied at all times the automaton is in that particular location. So this particular invariant, $c_q < r_q$, means that SV can remain in $Active\_q$ location until clock variable c_q surpasses the precedent task's response time r_q. At the end of first magenta transition, the SV reaches the first intermediate location. At this stage, the second transition is fired and calls another timed automaton that adds the released task *Q* to an appropriate list of active precedent tasks within an appropriate task set. As mentioned above, the automaton called for this purpose is *Add2ActivePrecedents* shown in figure 3.6. The last magenta transition in sequences originating from locations other than the *Idle* location checks if the remaining response time of newly released precedent task is greatest among all active precedent tasks. If yes then it resets c_q and r_q, otherwise an alternate transition is taken that leaves the c_q and r_q values unchanged.

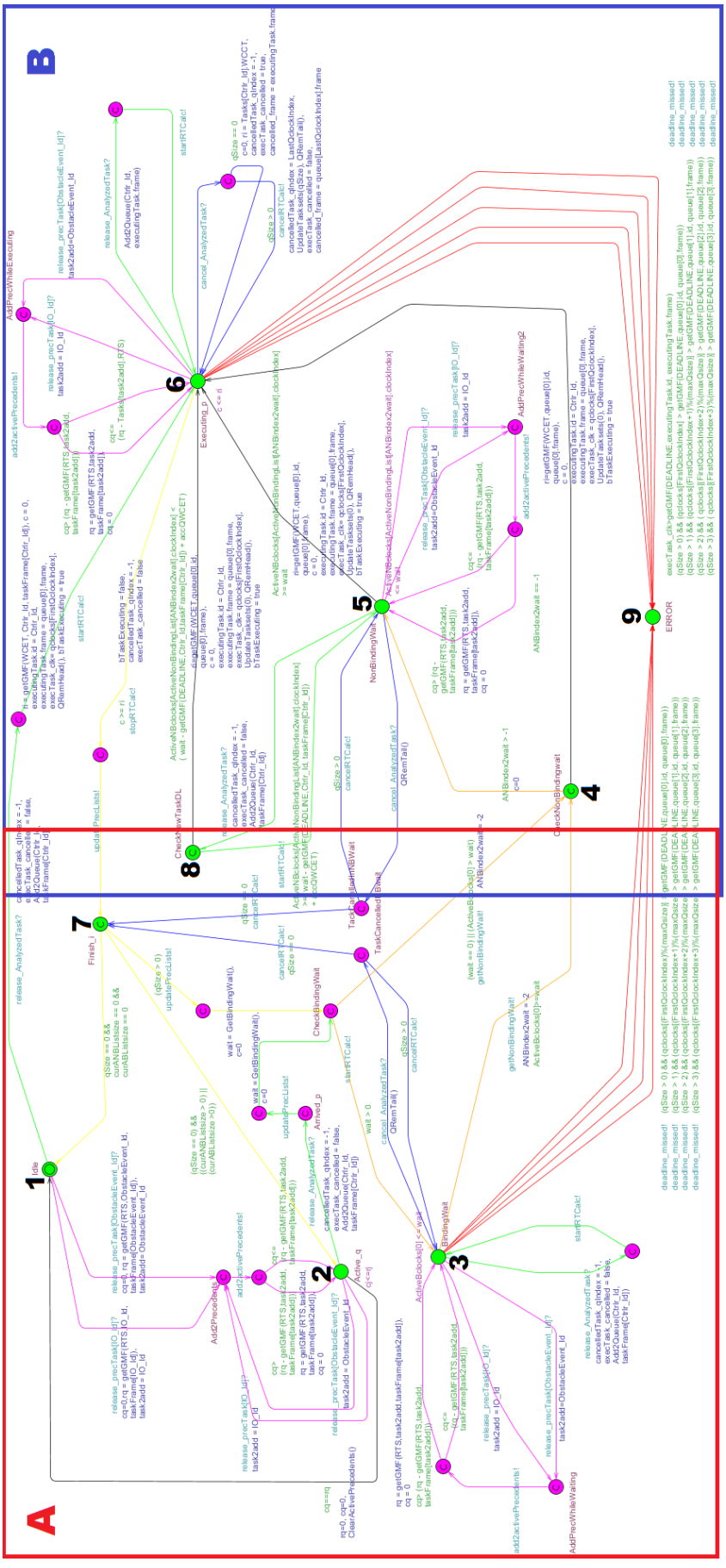

Figure 3.1: The Schedulability Verifier (SV)

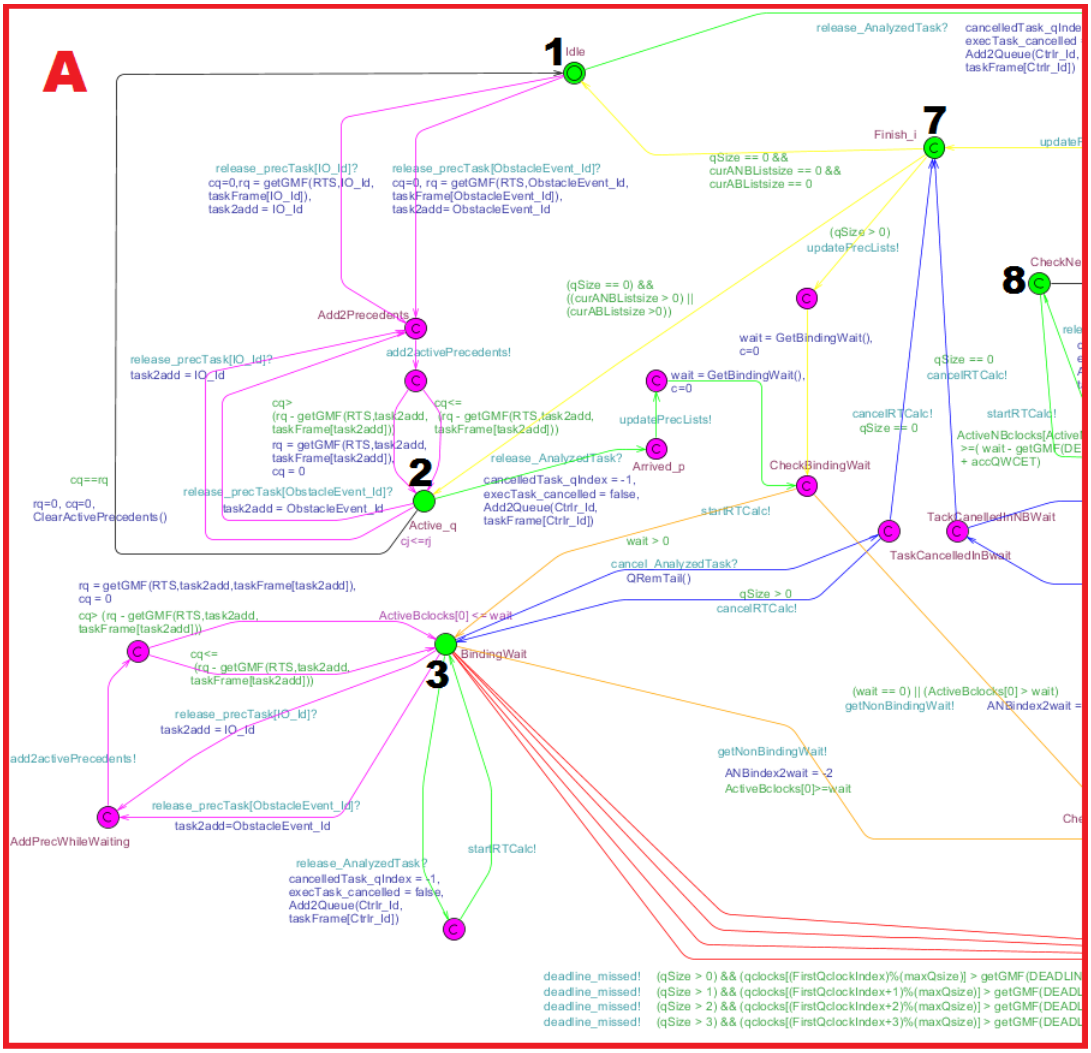

Figure 3.2: The Schedulability Verifier (SV) part A

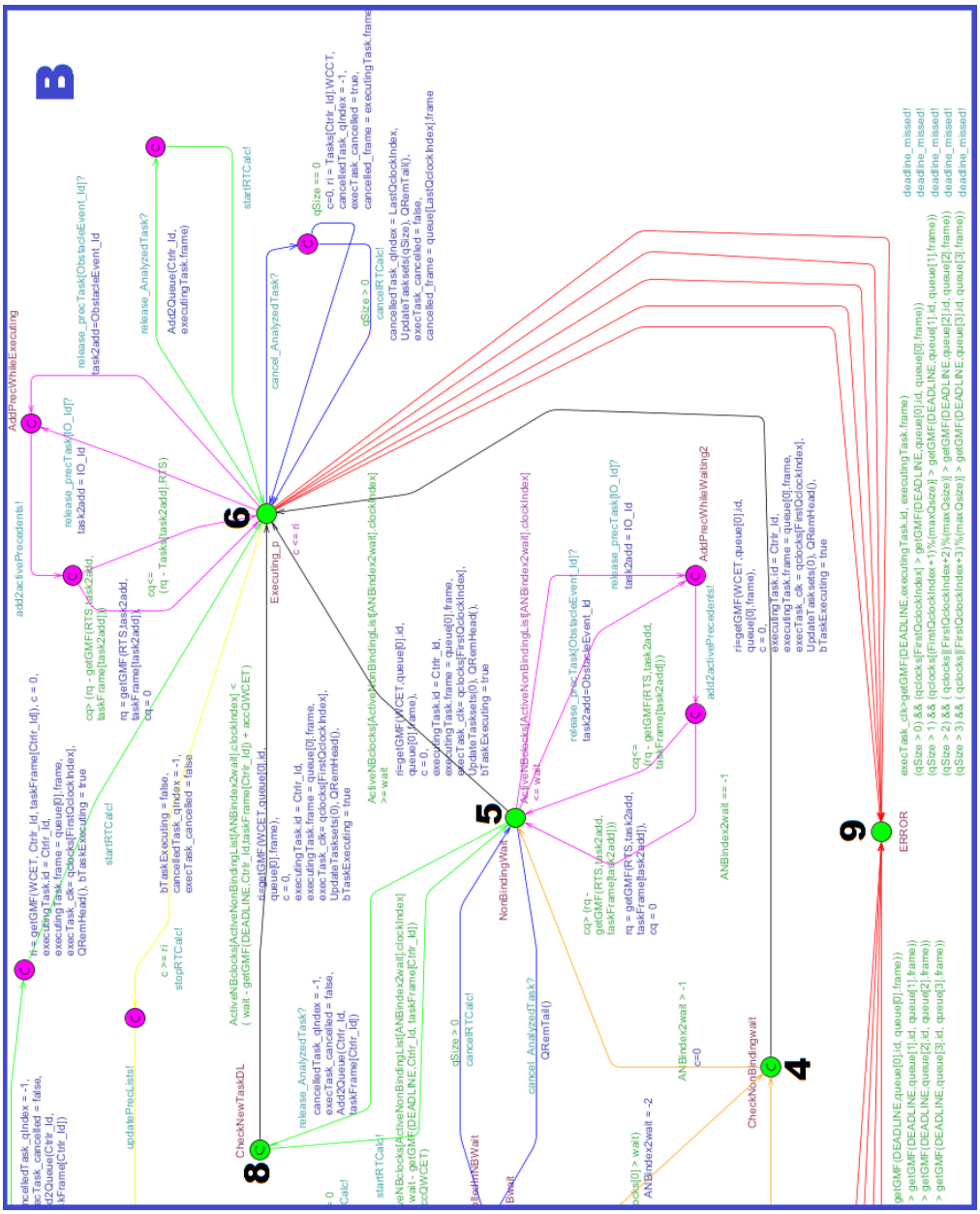

Figure 3.3: The Schedulability Verifier (SV) part B

Green: Green coloured transitions fire when an instance of task *P* releases for execution. The first transition in the sequence leads to an intermediate location from where a second transition moves SV to final location in the sequence. Action on first transition is always the same and causes the task instance to be added to a queue of task *P* instances waiting for execution. However, the action taken on second transition and the location reached after it depends upon the SV location at the time of *P*'s release. Following are the possible SV locations from where green transitions can originate. For each originating location, a description of actions taken after the first transition from that location are also given below,

- **Idle:** *Idle* is the initial location of SV. When a *P* instance releases at this location then, after the first transition, the next transition takes SV to *Executing_p* location which simulates execution of task *P*. The actions performed with the second transition include removing the analyzed task instance at the head of waiting queue, resetting clock c, setting values of r_i to analyzed task's WCET, setting task instance's deadline to a valued provided in the task model and calling *RTCalc* helper automaton to start calculating response time of the freshly released task instance

- **Active_q:** If the SV is in *Active_q* location then release of task *P* will cause the automaton to move to *Arrived_p* location from where a second transition takes SV to another temporary location invoking *UpdatePrecLists* helper automaton in the process. The last transition in the sequence ends at *CheckBindingWait* location where the automaton checks if any binding precedent task is currently active and needs to be waited upon

- **BindingWait, Executing_p:** When SV is in one of *BindingWait* or *Executing_p* location then the second transition in the green sequence takes the SV back to original location while calling the *RTCalc* automaton to start response time calculation

- **NonBindingWait:** This location simulates the delaying of a task's execution because of some non-binding precedent task. In this location if a new instance of task *P* is released for execution then, after the first transition, there are two possibilities for the next transition. Which possibility actually executes depends

upon weather new task's deadline allows for the remaining non-binding waiting time to finish or not. In first case the next transition takes SV back to *NonBindingWait* location while in second case, SV moves to *Executing_p* location. The actions performed with second transition in latter case are same as the ones performed when new task was released when SV was in *Idle* location

Blue: The blue transitions in SV represent automaton moves that are made when an instance of task *P* got canceled by the simulation scheduler. The cancellation can occur while SV is in one of wait locations, that is, *BindingWait* and *NonBindingWait* location or in *Executing_p* location. It is already understood that the task canceled by scheduler is the latest instance that is released for execution. Hence, at the time of any cancellation there may be zero or more task instances waiting in the queue for execution.

When a task is canceled while SV is in one of the wait locations then the next location depends upon whether there are any other task instances waiting in the queue for execution. If yes then SV moves back to the wait location after removing the canceled task from the queue, or in the second case it moves to *Finish_p* location.

If task cancellation occurs while SV is in *Executing_p* location then the automaton loops back to the same location after performing the cancellation actions. However, the actions performed during the transition depends again upon status of the waiting queue. If waiting queue is empty then this means that the canceled task instance is actually the one that is currently being executed. In this case the executing task is replaced by a dummy task whose execution time is equal to *WCCT* of the task *P*. On the other hand if waiting queue is not empty then the SV simply removes the last task instance in the queue and comes back to *Exeucuting_p* location.

Yellow: Few transitions leading to or originating from *Finish_p* location are shown in yellow colour in figures 3.1,3.2 and 3.3. As mentioned before *Finish_p* is reached when either a task *P* instance finishes execution or when waiting queue got empty due to cancellation of a task instance. The only yellow transition sequence that leads to *Finish_p* transition originates from *Executing_p* location. The first transition in this

sequence stops at a temporary location and calls the *RTCalc* automaton over *SaveRTifMax* channel to stop calculating the response time of the task that just finished execution and also save it as task's WCET if the calculated response time is the maximum among all the previous task instances. The second transition of this sequence calls *UpdatePrecLists* automaton and ends at *Finish_p* location. Three yellow transitions originate from *Finish_p* as well leading to three different SV locations. If the waiting queue is not empty then the SV moves to *CheckBindingWait* after passing through a temporary location location or else a decision is made either to move to *Idle* if there are no active precedent tasks or to *Active_q* location otherwise.

Orange: The orange coloured transitions represent the paths taken by SV while evaluating binding or non-binding waiting times. These transitions can originate from *CheckBindingWait*, *CheckNonBindingWait* or *BindingWait* locations. If the binding *wait* value, calculated while reaching *CheckBindingWait* location, is greater than zero then an orange transition takes SV from *CheckBindingWait* to *BindingWait* location where it will remain there until waiting time is over. However, if the *wait* value is equal to zero then the SV proceeds to evaluate non-binding waiting time by calling *GetNBWait* automaton over *getNonBindingwait* channel and then reaches *CheckNonBindingWait* location. At this location the returned value from *GetNBWait* is examined and if it is greater than -1 then the SV moves to *NonBindingWait* location where it stays until the completion of wait time or arrival of a new instance of task *P* whose deadline does not allow the remaining waiting time to complete.

Black The transitions leading to *Executing_p* are shown in black colour except the one that originates from *Idle* location which is shown in green colour. *Executing_p* is the location which represent the time spent during execution of an instance of task *P*. The actions perform with these black coloured transitions include removing the analyzed task instance at the head of waiting queue, resetting clock c, setting values of r_i to analyzed task's WCET, setting task instance's deadline to a valued provided in the task model and calling *RTCalc* helper automaton to start calculating response time of the freshly released task instance. One more transition is shown in black colour which takes SV from *Active_q* back to *Idle* location when c_q becomes equal to r_q.

Red: Red coloured transitions in SV all lead to the *ERROR* location. These transitions originate from one of *BindingWait*, *NonBindingWait* or *Executing_p* location. These transitions are fired when a clock associated with an active task *P* instance surpasses its allowed deadline.

3.2 Cancellation Handler (CH) Implementation

The implementation of CH automaton in Uppaal is shown in figure 3.4. Uppaal allows parameters to be defined with automaton definition to which values can be passed at the time of automaton instantiation. CH is also implemented as a parameterized automaton that takes two parameters named *taskID* and *bSporadic*. This *taskID* parameter is meant to refer to the binding precedent task against which the CH instance handles cancellation and the boolean parameter *bSporadic* shows if the precedent task referred to by *taskID* is sporadic in nature or not. The introduction of parameters allows us to instantiate CH automata for multiple binding precedent tasks without changing the automaton definition. Within the automaton definition, *taskID* is used as an index of *release_precTask[]* array so that each instantiated CH reacts to release of the binding precedent task that is passed as parameter to it.

3.2.1 CH Locations

Just as the SV locations in figure 3.1, figure 3.4 also shows the CH locations in two colours only. The green coloured locations are the ones that are defined in CH design described in section 2.2.2.5. The magenta locations *DecisionLoc1* and *DecisionLoc2* are introduced because of some practical considerations and has no affect on the actual working of automaton. In fact, if we can imagine two super locations: one comprising of *CancelLastQueued* and *DecisionLoc2*, and another containing *CancelExecuting* and *DecisionLoc1* then these super locations will closely resemble the *CancelLastQueued* and *CancelExecuting* locations as defined in section 2.2.2.5. An *End* location, marked as red, is also introduced to allow the automaton stop in case of deadline miss.

3.2.2 CH Transitions

We have stuck to the convention of using same colour for similar transitions, therefore we shall describe the CH transitions based on their colours.

Blue: The blue coloured transitions are fired with release of the binding precedent task whose id as passed as parameter *taskID*. These transition move the automaton from *Start* location to either *CancelLastQueued* location if $qSize > 0$ or to *CancelExecuting* location if $(qSize == 0)$ && $(bTaskExecuting == true)$.

Orange: The decision to cancel the latest instance of task P depends upon relative release times of P and binding precedent task Q instances AND whether task Q is sporadic or periodic. Orange coloured transitions are taken when no task P cancellation is required because the release times of both analyzed and precedent task instances are same and also task Q is not a sporadic task. These transitions always end up back at the $Start$ location.

Magenta: When the condition defined for orange transitions evaluates to false then this means that the latest instance of task P must be canceled. One of the two magenta transition is fired at this point and calls for cancellation of analyzed task by performing synchronization over *cancel_AnalyzedTask* channel. The transitions move the CH either from *CancelLastQueued* to *DecisionLoc2* location or from *CancelExecuting* to *DecisionLoc1* location.

Black: Black transitions are labeled with *release_AnalyzedTask* synchronization channel and are meant to release a new instance of task under analysis. These transitions can originate from *DecisionLoc1*, *DecisionLoc2* or *ReleaseExtra* locations. At *DecisionLoc1* or *DecisionLoc2*, a black transition can either lead to *ReleaseExtra* or to *Start* location depending on the decision to release two or one new P instance. As discussed earlier, this decision is taken on the basis of relative release times of tasks P and Q. The only black transition leading out of *ReleaseExtra* location moves the CH back to $Start$ location.

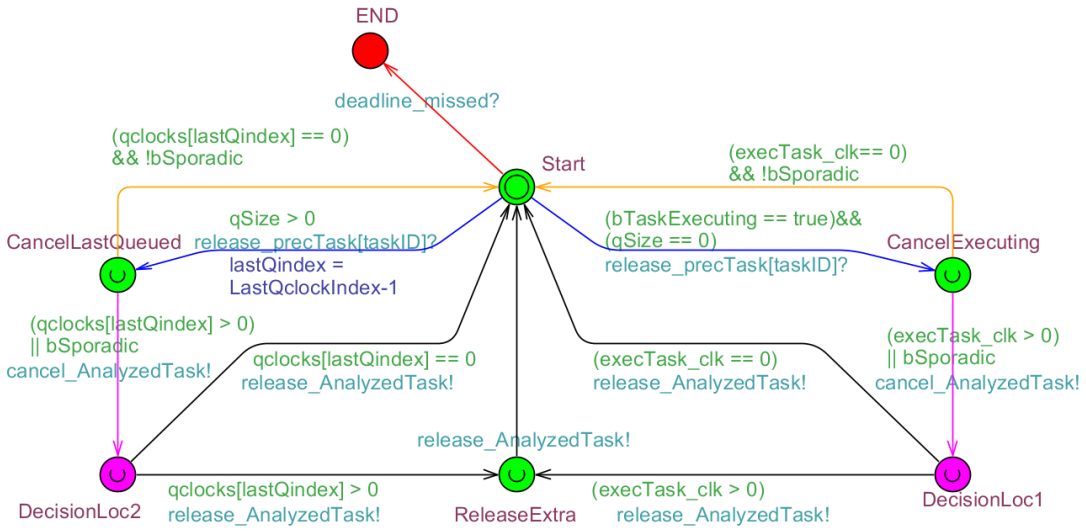

Figure 3.4: Cancellation Handler (CH) Automaton

3.3 Helper Automata

Let us now describe the implementation details of helper automata that are mentioned in the previous section. As discussed, these automata are called by SV to perform different actions just like a function or method call.

3.3.1 UpdatePrecedentLists

UpdatePrecedentLists automaton is called by SV whenever an instance of task *P* is ready for execution with active precedent tasks present in the system. On receiving the call, the automaton goes through the lists of active non-binding and binding precedent tasks to search for tasks that have completed execution or have become irrelevant. Some active precedent tasks may become irrelevant if the instance of *P* which was supposed to wait for them has already finished execution. The completed tasks are searched by comparing each task's associated clock with its WCET while tasks that are no longer relevant are identified by the value of their *TasksetNo* tag being less than zero. For a detailed discussion on *TasksetNo* tag please refer to section 2.2.2.2.

The implementation of *UpdatePrecedentTask* is shown in figure 3.5. It can be seen that after receiving the call from SV the automaton moves to *CheckANBList* location

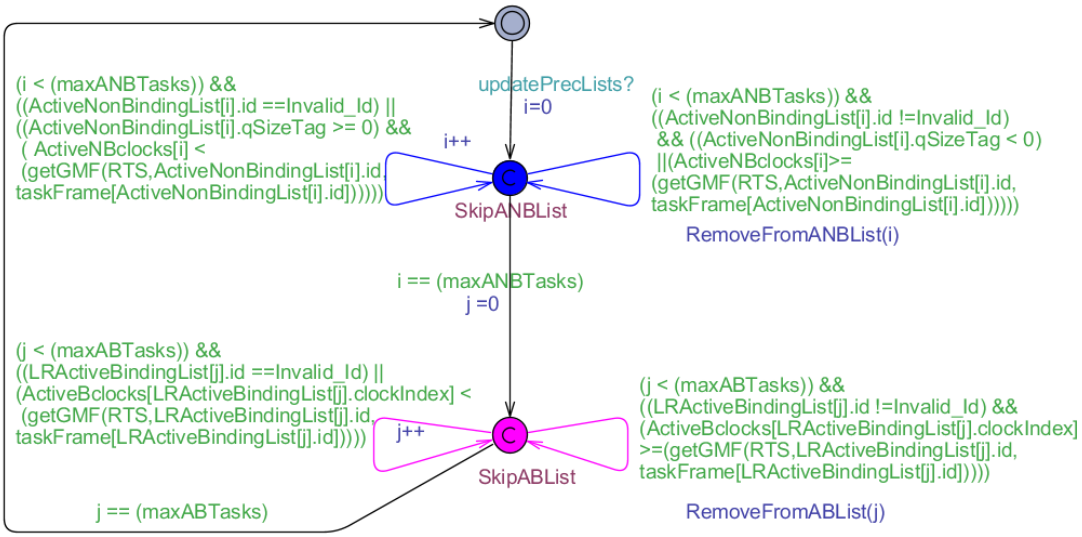

Figure 3.5: The automaton that updates the lists of active precedent tasks

where it loops through active non-binding precedent list. There are two transitions shown with this location that loop back to the same location. These transitions have complimentary guard conditions so that when a task is neither complete nor irrelevant then the left transition is taken which simply causes the automaton to skip to the next task in the list while if one of the above condition is true then right transition is taken which removes the task under consideration from the list.

After updating non-binding precedent list the automaton moves to *CheckABList* location shown in magenta colour. Here, the procedure is repeated for the list of active binding precedent tasks.

3.3.2 Add2ActivePrecedents

The *Add2ActivePrecedents* automaton adds the released precedent task either to list of active non-binding tasks or to list of binding tasks if it determines that this will be the longest running binding precedent task. As shown in figure 3.6, starting from the initial location at the top the automaton first decides if the released task is a non-binding or binding precedent task. The blue coloured locations and transitions represent the actions taken when the precedent task is non-binding while magenta coloured elements show the case of binding precedent task.

When adding a non-binding task to the active non-binding task list the automaton first loops through the list until it finds an entry that is empty or has *TasksetNo* greater than or equal to the current size of the queue containing active instances of task *P*. If the automaton finds an empty location before it finds a task with *TasksetNo* greater than or equal to current queue size then this means that there is no task previously added in the current task set so the automaton simply adds this task to empty location as the first member of new task set. However, if the search hits a task that has *TasksetNo* equal to current queue size then it again loops through all the tasks in this task set until a task is found whose remaining response time is less than that of the new task. The new task is then inserted at this position in the list. The third possibility is that in the first search the automaton hits a task that has *TasksetNo* greater than the current queue size. In this case again the new task is inserted at this position.

In cases the task to be added is a binding task then the automaton first determines if there is a task in the list at location *qSize*. If no, then simply add this new task at this location. However, if *qSize* location is non-empty then the new task will replace the existing task only if the new task has response time greater than the remaining response time of current task at list index *qSize*.

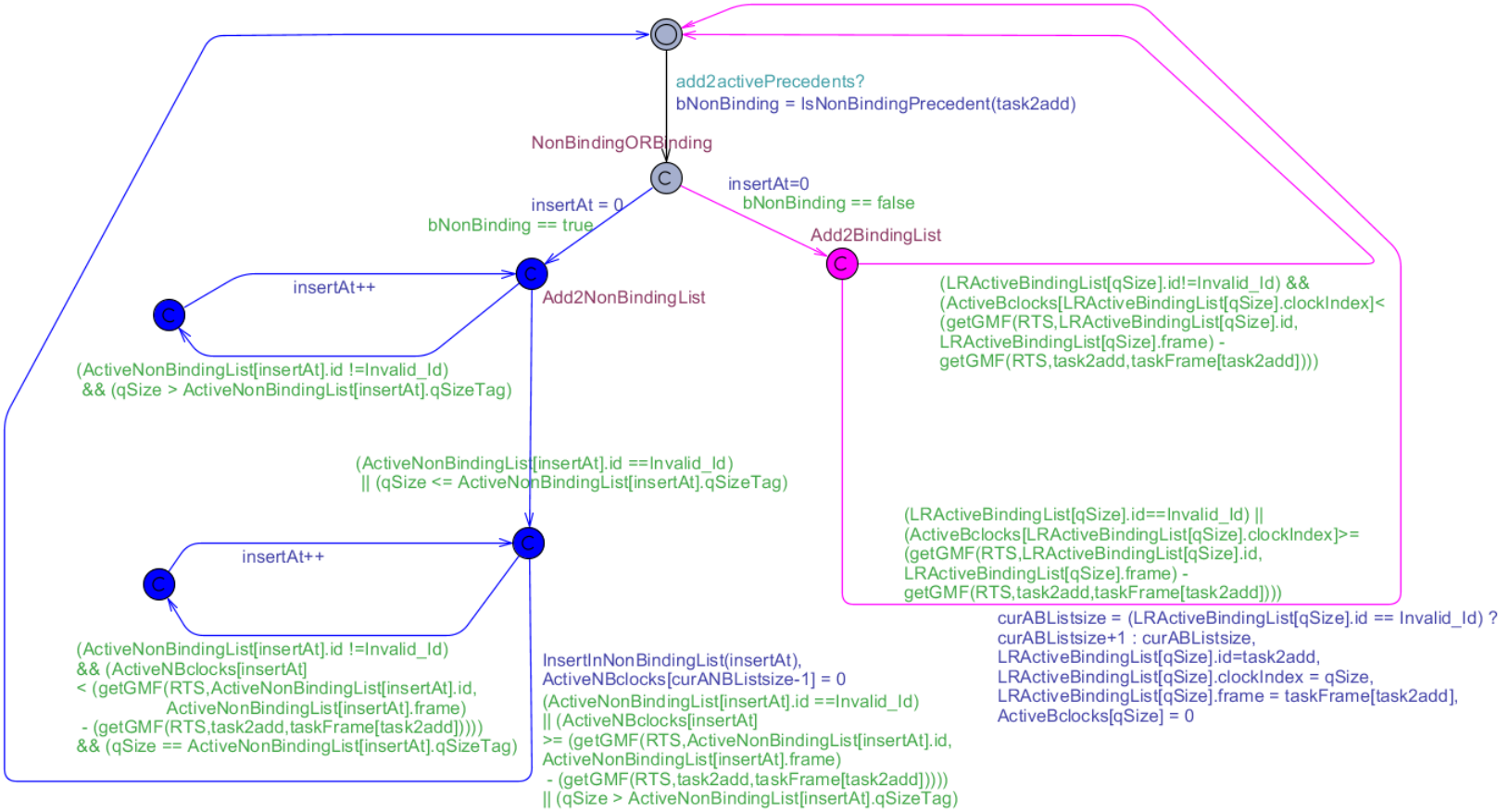

Figure 3.6: The automaton that adds active precedent tasks to appropriate lists

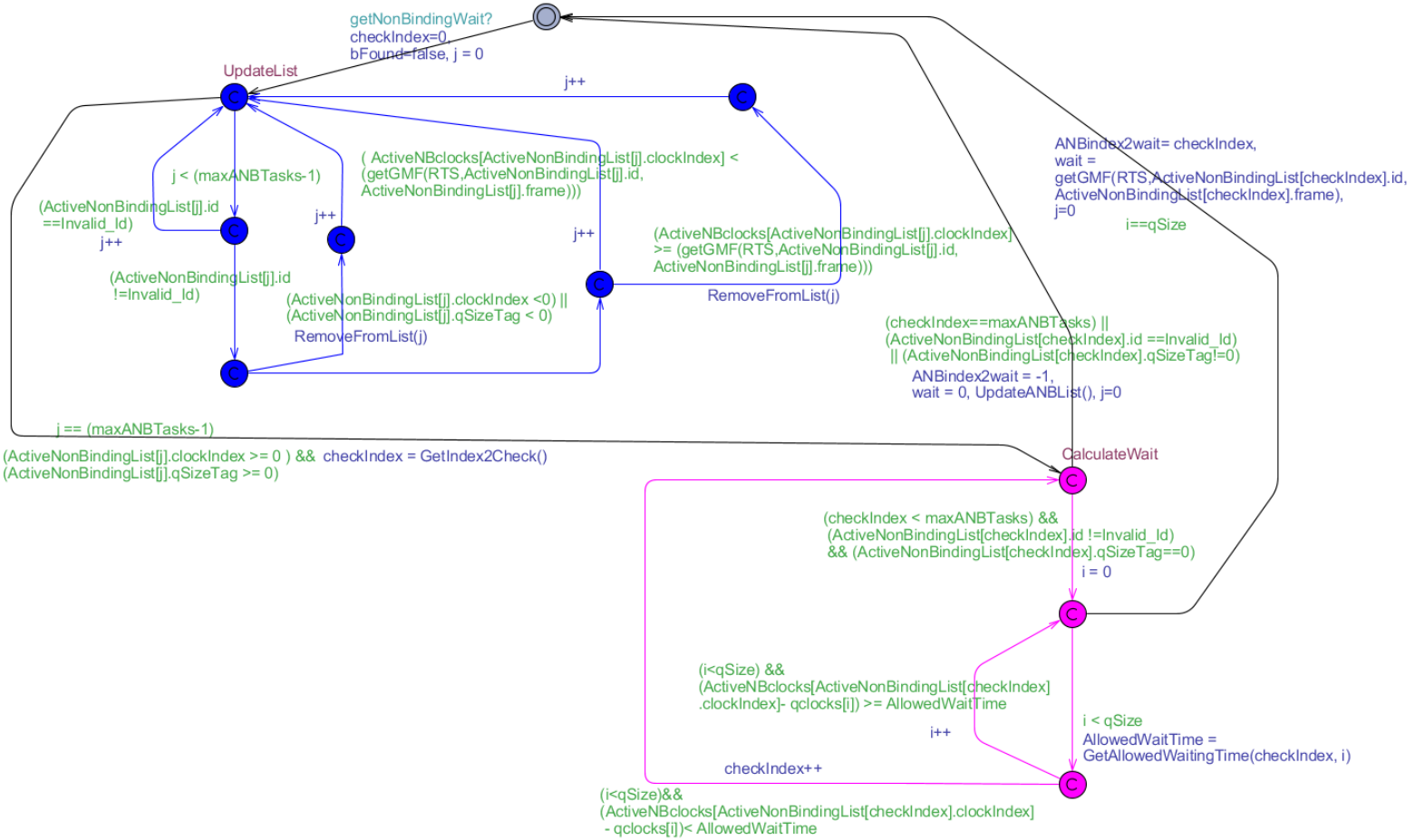

Figure 3.7: The automaton that determines the waiting time due to any non-binding precedent task

3.3.3 GetNBWait Automaton

The SV calls *GetNBWait* to decide if it is possible to wait for some active non-binding precedent task to finish execution before proceeding with the execution of task *P* instance at the head of the queue. This call is made after the SV finishes dealing with possible wait time due to some active binding precedent task. *GetNBWait* automaton upon receiving the call first performs an update operation on active non-binding list which is similar to the one performed by *UpdatePrecedentLists* automaton described in subsection 3.3.1. After update the automaton then goes through the list of active non-binding precedent tasks with *TasksetNo* tag equal to zero and search for a task with largest remaining response time value, RTS_max, such that all *P* tasks waiting in the queue can afford to wait for RTS_max time units. If such a non-binding precedent task is found then *GetNBWait* returns the index of that precedent task to SV otherwise a value of -1 is returned indicating that it is not possible to wait for any non-binding precedent task.

The figure 3.7 shows the implementation of *GetNBWait* automaton. The blue area is the one that performs the list update operation while the magenta coloured area is where the search for aforementioned precedent task is made.

3.3.4 RTCalc Automaton

RTCalc automaton is used to calculate and save the response times of the task *P* instances during the schedulability analysis process. These response times are then used in the next pass of schedulability analysis in which the next task in topological order is analyzed for real-time schedulability. As can be seen in the figure 3.8, there is only one working location (shown in blue) in *RTCalc* automaton apart from the initial location. There are four transitions that loop back to this working location. Three of these transitions synchronize with call from SV while the fourth is the one that actually increments response times of active *P* instance. The three synchronizing transitions are shown in blue, magenta and green colours and are called at the following events,

- Blue transition is fired when a new instance of task *P* is released for execution.

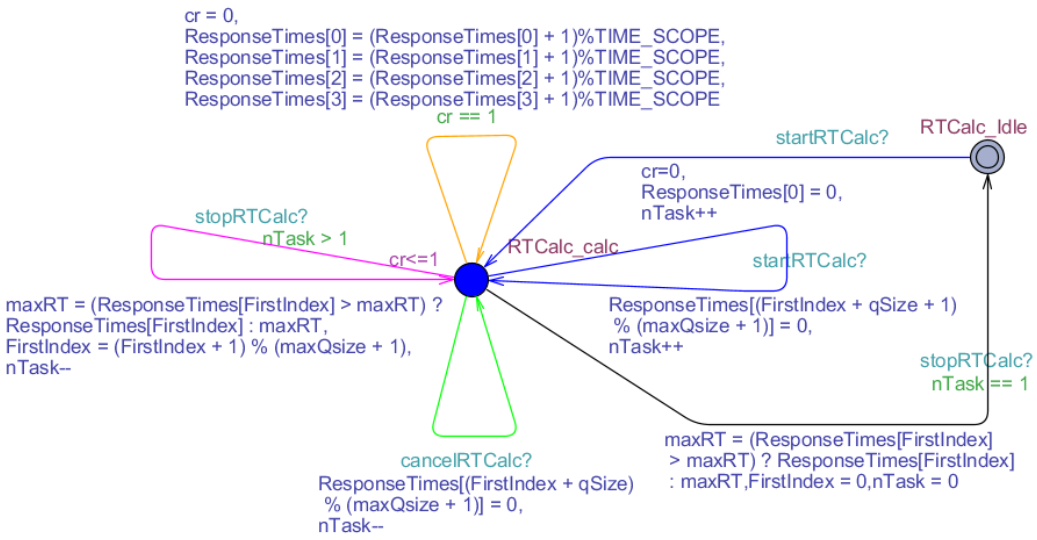

Figure 3.8: The automaton that calculates Response Times

The SV calls *RTCacl* to start response time calculation at this event

- Magenta transition is taken at the completion of a task *P*
- Green transition is called when a task *P* intance is canceled before completing successfully.

In the next section, we shall verify the correctness of our implementation by applying it on a system which is simple enough that we can analyze its schedulability manually as well. The results obtained by manual as well as automatic analysis can then be compared to verify the correctness of implemented framework.

3.4 The Functionality Verification Experiment

A simple case of a car's power window simulation was selected for this experiment [41]. The power window system works by reacting to user's pressing of window up or down button. A microcontroller reads the user input and outputs a command signal to DC motor which runs to move a scissor mechanical assembly that in turn moves the car window. However, the user input is overridden if an obstacle is detected in the path of an upward moving window. The obstacle event causes the controller to cancel its current command to move window up and instead issue a new command to move the

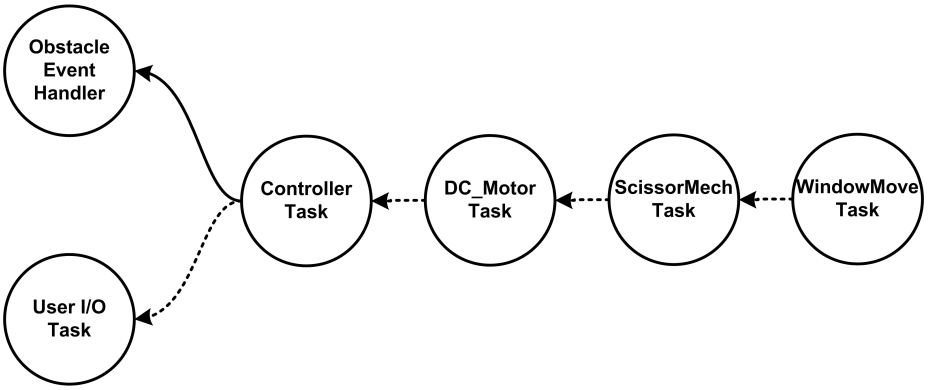

Figure 3.9: The graph showing precedent constraints among power window simulation tasks

window down a few centimeters. To realize the defined functionality, the simulation was assumed to be composed of six simulation tasks which are: *Controller Task*, *User Input Task*, *Obstacle Event Handling Task*, *DC Motor Task*, *Scissor Mechanism Task* and *Window Move Task*.

The DAG that describes the precedent constraints or dependencies among these tasks is given in Fig. 3.9. All the dependency relations in the graph are non-binding precedent constraints with the only exception of the relation between $Controller$ and $ObstacleEventHandler$ tasks which is defined as a binding constraint and shown by a solid line in the figure.

In our experiment, we analyzed the schedulability of $Controller$ task. The tasks concerned to this analysis experiment along with their description and attributes are given in table 3.1.

The CA is instantiated by taking the $Controller$ task as the $AnalayzedTask$ while considering $UserInput$ and $ObstacleEventHadndler$ as the precedent tasks. Since the problem of schedulability is now transformed into a state reachability problem, the only query that needed to be tested in order to check the schedulability is, *Is there any system state where CA is in ERROR location?*, or formally, $E <> CheckingAutomaton.ERROR$. This query is verified using UPPAAL's verifier.

Table 3.1: Tasks in Car's Power Window Simulation

Task Name	Description	Attributes
Controller Task	A task that emulates the actions of a microcontroller used in a car's power window	$WCET = 3$ time units $WCCT = 2$ time units $Period = 8$ time units
User Input Task	A task that periodically checks if the user has pressed window up or down button	$WCET = 4$ time units $WCCT = 1$ time units $Period = 5$ time units
Obstacle Event Handling Task	A task that is triggered if an obstacle is detected in the path of the car window while it is moving up	$WCET = 1$ time units $WCCT = 1$ time units Sporadic: Can arrive between 495 to 505 time units

3.4.1 Results

The schedulability analysis results of the system described above are summarized below,

- For the cases where the obstacle event occurred between 496^{th} to 498^{th} time units or between 502^{nd} to 504^{th} time unit, the $Controller$ task was found schedulable
- In rest of the cases, the $Controller$ task was found unschedulable

To understand these results, note that the $UserInput$ task with periodicity 5 arrives when simulation-time is 495 time units. The $Controller$ task which arrives at 496^{th} time unit and depends on $UserInput$ task can safely wait for $UserInput$ task to finish and still meet its own deadline at 504 time units, the next instant when new $Controller$ instance will arrive. So the scheduler will decide to delay the execution of $Controller$ task till 499 time units. Now if the obstacle event occurs between 496 to 498 time units, the scheduler has enough time to cancel the waiting $Controller$ task and then execute two new $Controller$ instances with combined WCET of 6 time

units. No cancellation time is required by $Controller$ task during this period since it is still in waiting state. On the other hand, if obstacle event between 499 to 501 time units then it does not leave enough time for two new $Controller$ task invocations and so these sub-scenarios were found unschedulable by our framework. The last case is when obstacle event occurs between 502 and 505 time units. This case is trivially schedulable since by the 502^{nd} time unit the $Controller$ task has already completed execution and so the obstacle event has no affect.

The results obtained through manual analysis match exactly with the ones produced by our implementation of schedulability analysis framework. This verifies the functional correctness of our framework implementation

CHAPTER 4

HELICOPTER SIMULATOR CASE STUDY

4.1 Introduction

After the implementation of our proposed framework and its initial testing on toy problems, it was time to test the framework's applicability for an actual real-time simulator. For this purpose a helicopter simulator [48] was selected as the subject of our case study. In the following sections we shall first describe the architecture and task model of the helicopter simulator followed by a discussion on steps that are needed to make the our schedulability analysis applicable to the selected simulator. Next we will propose an improvement in the simulator's task model and since the proposed improvement will need schedulability analysis to fine tune the task model, we shall then apply our schedulability analysis framework for this purpose and present the results for simulator designers.

4.2 Helicopter Simulator

The helicopter simulator selected for the case study is composed of several software modules or tasks communicating with each other using UDP messages over Ethernet. The execution of all the tasks is synchronized by a *Sync* task that sends a particular message to every other software task whenever that specific task needs to advance through a single simulation step. Theoretically the *Sync* task can run each of the other tasks on a different rate but in our studied system all the simulator tasks are being run at a rate of 60Hz. Hence, the *Sync* task sends a message simultaneously to each module after every 16.67 msec and upon receiving this message every task executes

and then sends its outputs to other related tasks. The tasks which are meant to be the destinations of these outputs must be able to receive and save these outputs so that it can use them as inputs for the next simulation step.

The tasks that are part of the helicopter simulator are as follows,

1. Simulator Management Console (SMC): This task is responsible for general simulator management including starting or stopping the simulation, initialization, runway selection etc.
2. Flight Module (FM): This is the main task that is responsible for the flight model calculations based on aircraft's current position, speed, attitude, environmental conditions as well as inputs from the helicopter operator.
3. Input / Output Module (I/O): This task basically detects inputs from helicopter operator and communicates them to FM and other tasks.
4. Visual System (VS): This task is responsible for the main visual display of the simulator.
5. Glass Cockpit Gateway (GCG): This task acts as a gateway for information from FM to the multifunction display described below.
6. Multifunction Display (MFD): This task takes input from GCG and displays aircraft's current speed and different sensor measurements on the screen.

The communication between the simulator tasks and the *Sync* task can be depicted as a dependency graph whose edges correspond to the communication link between tasks and the nodes represent the simulator tasks themselves. This graph is shown in figure 4.1.

Every task other than the *Sync* task is dependent on the "run" message from the *Sync* task without which it cannot proceed further. This kind of dependency is termed as binding dependency in our proposed framework since it is the case where dependent task cannot proceed until it gets the input from the precedence task, i.e. the dependency is satisfied. All the other dependencies depicted in the graph are non-binding dependencies which means that the task does not have to wait forever for new inputs from the precedent task and can proceed with old inputs.

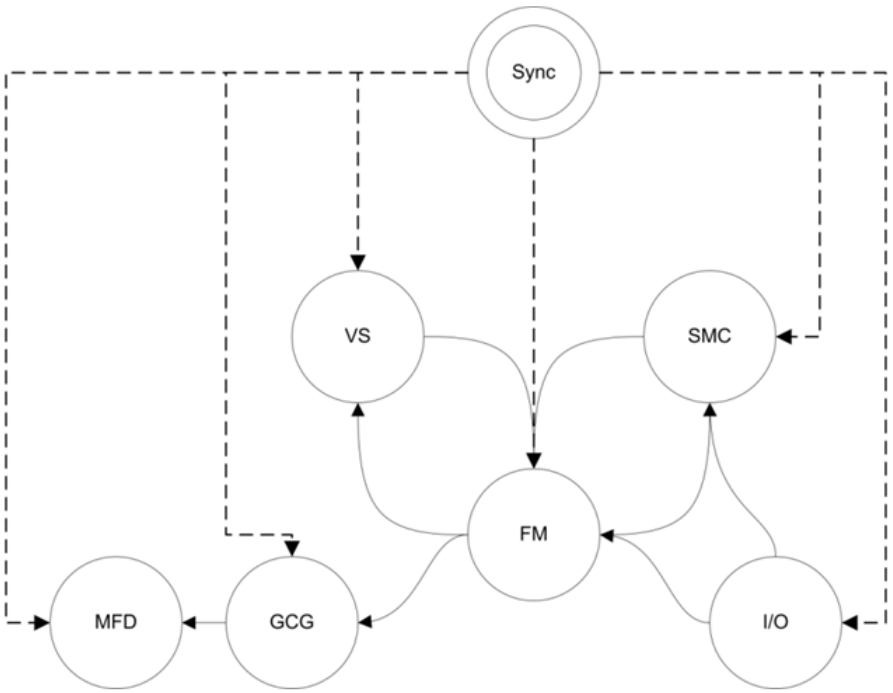

Figure 4.1: Dependency Graph among Helicopter Simulator Modules

4.3 Application of Schedulability Analysis Framework

The actual simulator architecture conforms partially to the co-simulation architecture proposed in our framework in that the execution of each task is controlled by a master called *Sync* task. However, one difference is that the communication between the tasks is independent and not through any simulation master. Moreover, the simulator master in this case is actually just the *Sync* task which serves as a kind of dumb simulation master that does not make any run-time decision based on current simulator state. We will, therefore, for our purpose of schedulability analysis assume a smarter simulation master that will make decisions on run-time to make the simulation more functionally accurate without undermining the timeliness of the simulation. The smart simulation master is also a part of simulator improvements proposed in the next section.

The dependency graph in figure 4.1 is not usable for schedulability analysis in our framework since it contains cyclic dependencies between *VS* and *FM* as well as between *SCM* and *FM*. In order to proceed further, it is necessary to resolve this problem

of cyclic dependencies. A closer examination of the messages exchanged between the cyclically dependent tasks reveals the following pattern,

- *FM* requires information from *SMC* for initialization or when the simulation status is changed, i.e. either paused or stopped.
- *FM* sends a message to *SMC* only when reporting an aircraft crash event.
- *FM* needs information from *VS* about the terrain in the vicinity of the aircraft.
- *FM* sends information to *VS* about aircraft's updated position and status after the current flight model calculations.

To break the cyclic dependencies, we split the *VS* task and *SMC* task into two. *VS* becomes a combination of *VS Terrain info* and *VS Update* while *SCM* is divided into *SMC Init/Status* task and *SMC Update*. The first part of both these tasks provides input to the *FM* task while the other reads output from the *FM*. The new dependency graph without cyclic dependencies is shown in figure 4.2.

4.4 Proposed Improvement

In flight simulators, an important factor is that of visual delay or visual transport delay [28]. Visual transport delay can be defined as the delay that occurs between the instant when the pilot performs some action and the time when the affect of this action is seen by the pilot. Although the FAA Part 60 requirement for a Level D simulator is 100 msec for helicopters [28], experience suggests that lower visual transport delays are desirable since various studies have linked simulator visual delays with increase in pilot workload [25], deterioration of pilot performance and simulator sickness.

An empirical estimation of the visual transport delay for the helicopter simulator presented above reveals that average delay is 82.87 msec while maximum delays can be up to 90.5 msec. Figure 4.3 below shows, for the case of our helicopter simulator, how to estimate the visual transport delay from the instant when a pilot takes an action to the time when this information reaches the task responsible for visual system update. The figure presents timeline of three simulator tasks: the *I/O* task, the *FM*

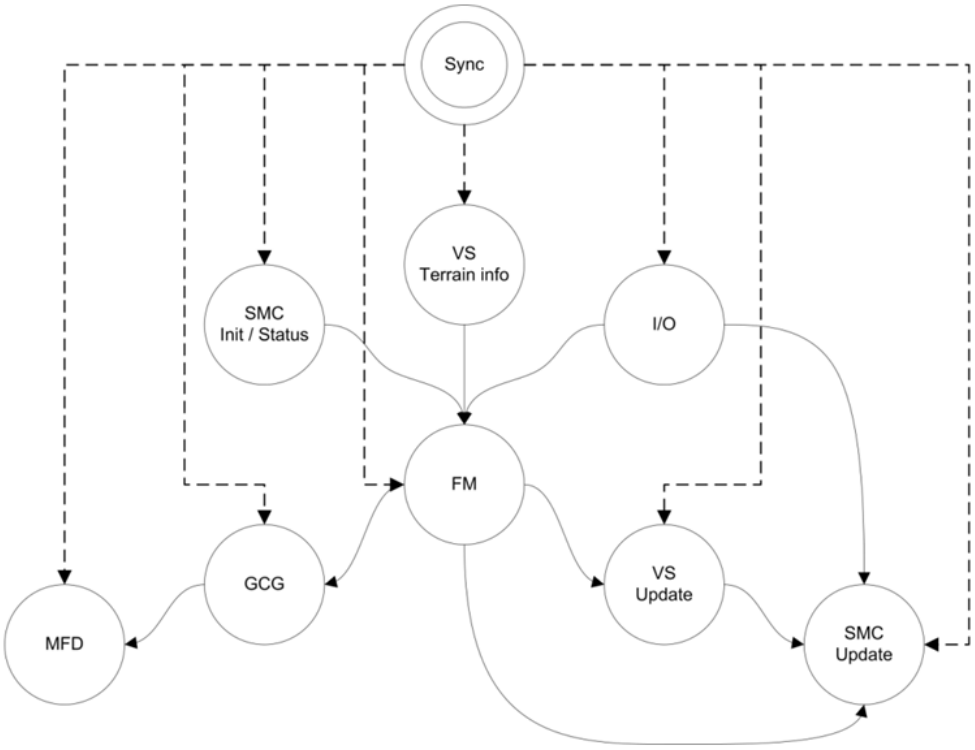

Figure 4.2: Dependency Graph without cyclic dependencies

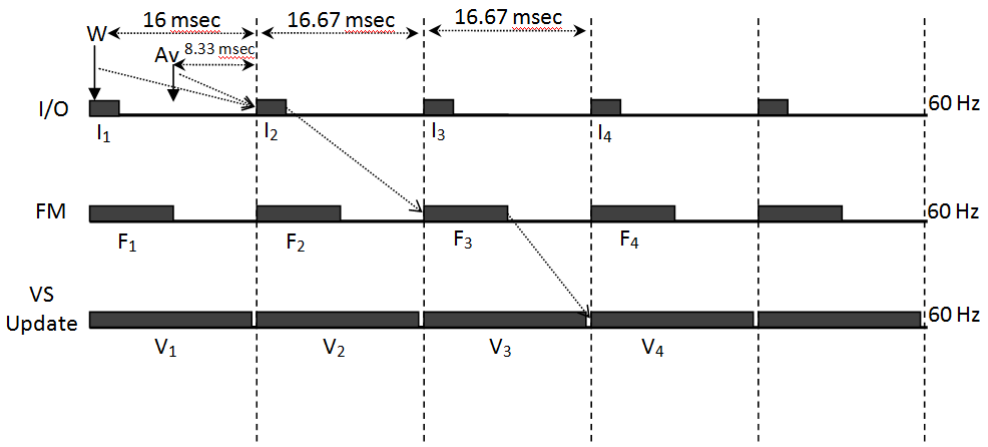

Figure 4.3: Time delay between actual pilot action and when it is read be VS update task

task and *VS Update* task. The vertical dashed lines indicate when each task becomes active. Since all the tasks currently run at 60 Hz frequency, all of them become active at the same time.

The worst case visual transport delay can occur when the pilot takes action just after the I/O task starts execution, the instant depicted in figure 4.3 as *W* while average delay, shown as *Av*, can be determined by assuming the pilot action to be taking place in the middle of the duration between two I/O tasks. Total delay in worst and average cases will be approximately 49.34 msec and 41.67 msec respectively before the information about the pilot action reaches the *VS update* task.

After the *VS Update* task gets the new data from *FM* task, there are more delays before the new scene can be displayed on the simulator screen. These delays are studied in [42] and are found out to be caused by V-Sync, graphical hardware and pixel adjustment. They calculated the total delay due to these factors to be 41.2 msec [42]. Adding all the delays we get worst case visual transport delay as 90.5 msec.

Although 90.5 msec is an acceptable delay by the official standards, however, research has shown that significantly lower delays can improve the simulator fidelity in some pilot tasks [28]. Keeping this in mind, the simulator designers wanted to reduce the visual transport delays without modifying the current simulator hardware. For this

purpose following steps were recommended,

- Increase the arrival frequencies of *FM* and *I/O* tasks
- Make *I/O* task a binding precedent task to *FM* task so that arrival of an *I/O* task can interrupt a currently executing *FM* task
- Employ a smart scheduler that can make scheduling decisions based on remaining response times of precedent tasks

With regards to proposed increased frequencies of *FM* and *I/O* tasks, a restriction was placed by simulator designers that the frequencies of both tasks should remain less than or equal to 120 Hz.

4.5 Schedulability Analysis

The problem left for us was to find a combination of *FM* and *I/O* task frequencies such that the transport delay between pilot action and its observed effect minimizes while at the same time *FM* task instances always remain schedulable. Keep in mind that an *I/O* task arrival in the proposed improvement can cancel an executing *FM* task and replace it with two instances of *FM* task; one that deals with simulation time up to the I/O instant while the second proceeds FM task time from the I/O instant onwards.

A new dependency graph of tasks that are related with visual transport delay, along with their arrival frequencies, is shown in figure 4.4. There is no line connecting the smart scheduler with *VS terrain info* task since it is noted that *VS terrain info* is actually just a sub-task, shown separately only to avoid cyclic dependency, and is run automatically after every *VS update* sub-task with an offset of 15 msec approx. Since the arrival rates of *FM* and *I/O* tasks are to be increased from 60 Hz, their frequencies are shown to lie somewhere between 60 and 120 Hz. Relevant task times of *FM*, *I/O* and *VS Update* tasks, as provided by the simulator designers, are given in table 4.1.

We shall now employ our schedulability analysis framework on this problem to determine the conditions under which the schedulability of the FM task can be guaranteed.

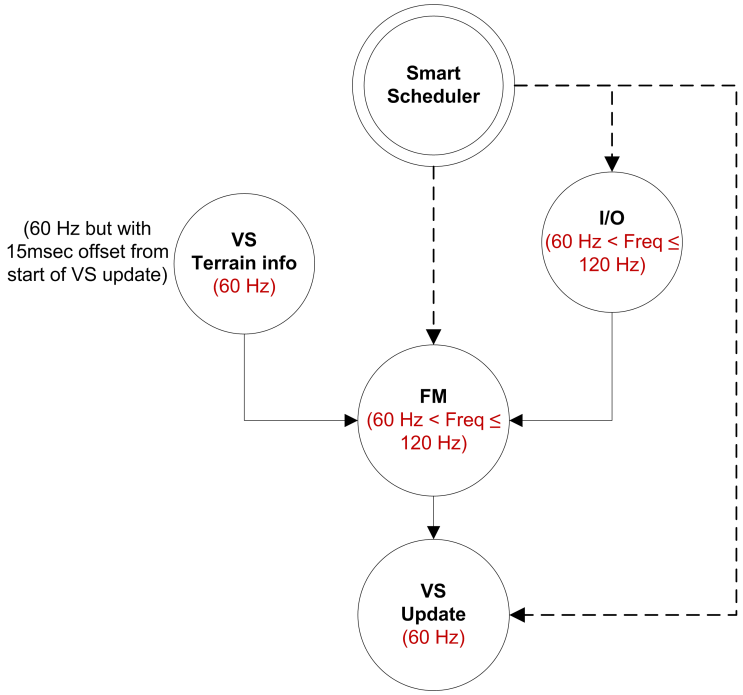

Figure 4.4: Dependency graph and arrival patterns of task related to visual display

A smart scheduler can then be designed to make use of analysis results and make the simulator faster as well more accurate.

First, a system of automata needs to be defined that contains task automata, the *CheckingAutomaton* and all other supporting automata required by the schedulability analysis framework. Hence three task automata systems were instantiated that defined the arrival pattern of each task in table 4.1 along with their time attributes. *FM* task was defined as the *analyzed task* and the other two were defined as *precedent tasks* where *VS Terrain Info* was a non-binding precedent task while *I/O* was defined as binding precedent task. *CancelationDecider* automaton was defined to monitor I/O task arrivals and decides at the arrival of each I/O instance if the analyzed task needs to be canceled or not.

Next, the schedulability of the automata system defined above was tested by varying the arrival rates of *FM* and *I/O* and running the Uppaal's verifier for each task frequency combination to check reachability of *CheckingAutomaton*'s ERROR state. The arrival frequency of *FM* task was varied from 70 Hz to 120 Hz with an interval of 10 Hz. For each *FM* task arrival rate, experiments were performed by vary-

Table 4.1: Task Attributes of FM, I/O and VS Terrain Info Tasks

Task Name	Worst Case Execution Time (WCET)	Worst Case Cancellation Time (WCCT)	Deadline
FM	4 msec	1 msec	Same as task periodcity
I/O	1 msec	NA	Same as task periodicty
VS Terrain Info	16.7 msec	NA	16.7 msec

ing *I/O* task frequencies ranging from 75 Hz to 120 Hz with an increment of 5 Hz. However, for each *FM* task frequency, schedulability was not tested for cases where $I/OFreq < FMFreq$. The results of these experiments are given in table 4.2.

Table 4.2: Schedulability Analysis Results

FM Freq (Hz) \ I/O Freq (Hz)	75	80	85	90	95	100	105	110	115	120
70	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*	*S*
80	–	*S*	NS	NS	NS	*S*	NS	NS	NS	NS
90	–	–	–	*S*	NS	NS	NS	NS	NS	NS
100	–	–	–	–	–	*S*	NS	NS	NS	NS
110	–	–	–	–	–	–	–	*S*	NS	NS
120	–	–	–	–	–	–	–	–	–	*S*

- *S*: Schedulable
- NS: Not Schedulable
- – : Not tested

The analyses results show that for *FM* arrival frequency of 70 Hz, the system remains schedulable for all *I/O* task frequencies ranging from 75 Hz to 120 Hz. However, for the case of 80 Hz only two *I/O* arrival frequencies of 80 and 100 Hz are schedulable. For the rest of the *FM* arrival rates, the only schedulable case is when *I/O* arrival frequency is same as that of *FM* task.

4.6 Conclusion

In this work, an actual real-time helicopter simulator was selected as case study to test the applicability of our proposed schedulability analysis framework. The information about the simulator architecture and simulator tasks was gathered from the simulator designers including the task arrival patterns and other time related attributes. Afterwards, a possible area of improvement was identified in the simulator design and it was suggested that a decrease in visual transport delay would improve simulator fidelity in some pilot tasks. The steps recommended to achieve the goal of decreasing visual transport delay included increase in arrival frequencies of *FM* and *I/O* and employment of a smart simulator scheduler that takes into account the remaining response times of precedent tasks before scheduling the dependent task.

The proposed improvement of increasing task arrival rates called for the application of our schedulability analysis framework to determine which combination of task arrival rates are schedulable by a smart scheduler. Hence, multiple analysis experiments were performed with different task frequency combinations and the results were then reported to the simulator designers. The results can be used by the designers to select one optimum task frequency combination or, alternatively, they can select multiple schedulable frequency combinations and run the simulator in *multi-mode* configuration. In this configuration the simulator switches to a mode with higher task arrival rates during highly dynamic simulation phases and shifts back to lower task frequency combination when simulation is in relatively lower dynamic phase.

CHAPTER 5

CONCLUSION

This research has aimed to present a framework for schedulability analysis of real-time co-simulations. To the best best of our knowledge, it is the first attempt in this direction as all the related literature surveyed for this work deals with schedulability analysis of actual real-time systems only. The framework gives the simulation designers confidence that the designed simulation is always schedulable. In case a simulation is found unschedulable, the framework can also suggest some parameters to the designer of simulation master helping him / her to make the simulation schedulable again.

The real-time co-simulation model to which the proposed framework is applicable provides freedom of choice to the simulation designers in some key areas while putting some restrictions as well. The freedom that the co-simulation model offers is in terms of frame length of the sub-system simulation models and also the activation pattern of each sub-system model. That is, the presented model not only allows each sub-system model in a simulation to have distinct frame length but also allows a single simulation model to have different frame lengths in different scenarios during a simulation run. The activation patterns of simulation models are referred to as task arrival patterns in our co-simulation model and are defined by task automata. The survey in [44] declares task automaton as the most expressive method for defining a task arrival pattern, so consequently the proposed co-simulation model allows a simulation designer to describe simulation models' activation patterns in most expressive way possible. There are a couple of limitations too. The first one requires the simulation to be executed on a multi-core platform such that each simulation model is run on a separate independent core. This is not a major restriction considering the

manycore processors that are available in the market today and the expectation that the core density will increase in future. The second assumption that the model makes is that simulation models or tasks do not have cyclic precedent constraints among themselves. This assumption is again not very restrictive as it is usually possible to break the cyclic dependencies for schedulability analysis as shown in the case study in chapter 4.

After carefully definition of the co-simulation model, a timed-automata based schedulability analysis framework for real-time systems [22] was selected as the base to start the development of new framework for schedulability analysis of real-time co-simulations. To start with, it was observed that the precedence constraint handling offered by the selected framework needs to be improved for the case of real-time co-simulations. Building upon this observation, the new framework was then developed considering special features of real-time co-simulations. Since clocks are a major source of complexity in timed automata systems, simplifications were also presented in the initial precedence handling method which reduces clock variables in the new timed automata based framework.

The development of the framework was followed by implementation in UPPAAL. The special schedulability analysing automata, *Schedulability Verifier* (SV) and *Cancellation Handler* (CH), were defined as template automata in our proposed framework. Therefore, in an implementation of the framework these automata need to be instantiated according to available task-set and the task under analysis. The implemented framework was first tested for functional verification and then was applied on a helicopter simulator to propose a possible improvement to the simulator designers.

REFERENCES

[1] Yasmina Abdedda, Eugene Asarin, and Oded Maler. Scheduling with Timed Automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[2] Y. Abdeddaim, A. Kerbaa, and O. Maler. Task graph scheduling using timed automata. *Proceedings International Parallel and Distributed Processing Symposium*, 2003.

[3] Muhammad Uzair Ahsan and Mehmet S. Halit Oğutüzün. Schedulability Analysis of Real-time Multi-frame Co-simulations on Multi-core Platforms. *Turkish Journal of Electrical Engineering & Computer Sciences*, 27(5):3599–3616, 2019.

[4] Ahmad Al-Hammouri, Vincenzo Liberatore, Huthaifa Al-Omari, Zakaria Al-Qudah, Michael S Branicky, and Deepak Agrawal. A Co-simulation Platform for Actuator Networks. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 383–384, New York, NY, USA, 2007. ACM.

[5] Ahmad T. Al-Hammouri. A comprehensive co-simulation platform for cyber-physical systems. *Computer Communications*, 36(1):8–19, 12 2012.

[6] Rajeev Allur and David L Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[7] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In Kim Larsen and Peter Niebert, editors, *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin / Heidelberg, 2004.

[8] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings.

Applying New Scheduling Theory To Static Priority Preemptive Scheduling. *Software Engineering Journal*, (September):284–292, 1993.

[9] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008.

[10] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th Real-Time Systems Symposium*, pages 182–190. IEEE, 1990.

[11] Gerd Behrmann, Alexandre David, and Kim G Larsen. *A Tutorial on Uppaal 4.0*. 2006.

[12] E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[13] Abdeldjalil Boudjadar, Jin Hyun Kim, Kim G Larsen, and Ulrik Nyman. Compositional Schedulability Analysis of An Avionics System Using UPPAAL. In *International Conference on Advanced Aspects of Software Engineering. {ICAASE}*, pages 140–147, 2014.

[14] Michael S Branicky, Vincenzo Liberatore, and Stephen M Phillips. Networked control system co-simulation for co-design. In *American Control Conference, 2003. Proceedings of the 2003*, volume 4, pages 3341–3346. IEEE, 2003.

[15] Anton Cervin, Martin Ohlin, and Dan Henriksson. Simulation of networked control systems using TrueTime. In *Proc. 3rd International Workshop on Networked Control Systems: Tolerant to Faults*, 2007.

[16] Jinchao Chen, Chenglie du, Fei Xie, and Zhenkun Yang. Schedulability analysis of non-preemptive strictly periodic tasks in multi-core real-time systems. *Real-Time Systems*, 2015.

[17] Roy Crosbie. Real-Time Simulation Using Hybrid Models. In Katalin Popovici and J. Mosterman Pieter, editors, *Real-Time Simulation Technologies: Principles, Methodologies, and Applications*, chapter Real-Time, pages 4–31. CRC Press. Taylor & Francis Group, 2013.

[18] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1. In *Model-based design for Embedded Systems*, pages 93–119. 2009.

[19] Pontus Ekberg and Wang Yi. Schedulability analysis of a graph-based task model for mixed-criticality systems. *Real-Time Systems*, 52(1):1–37, 2016.

[20] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 8 2007.

[21] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability Analysis Using Two Clocks. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619, pages 224–239. Springer Berlin Heidelberg, 2003.

[22] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301–317, 3 2006.

[23] Elena Fersman, Paul Pettersson, and Wang Yi. Timed Automata with Asynchronous Processes : Schedulability and Decidability. *Lecture Notes in Computer Science*, pages 67–82, 2002.

[24] Elena Fersman and Wang Yi. A Generic Approach to Schedulability Analysis of Real-Time Tasks. *Nordic Journal of Computing*, 11(2):129–147, 2004.

[25] Nina Flad, Frank M. Nieuwenhuizen, Heinrich H. Búlthoff, and Lewis L. Chuang. System delay in flight simulators impairs performance and increases physiological workload. In *Engineering Psychology and Cognitive Ergonomics*, pages 3–11, Cham, Switzerland, 2014. Springer International Publishing.

[26] FMI. Functional Mock-up Interface for Model Exchange and Co-Simulation, 2014.

[27] C W Gear and D R Wells. Multirate linear multistep methods. *BIT Numerical Mathematics*, 24(4):484–502, 1984.

[28] Peter Jarvis, Bernard Lalonde, and Daniel Spira. Flight simulator modeling and validation approaches and pilot-in-the-loop fidelity. In *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Honolulu, Hawaii, 2008.

[29] Mathai Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[30] Pavel Krcal, Martin Stigge, and Wang Yi. Multi-processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times. (1):274–289, 2007.

[31] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 2 2014.

[32] Sylvain Lauzac, Rami Melhem, and Daniel Mossé. An improved rate-monotonic admission control and its applications. *IEEE Transactions on Computers*, 52(3):337–350, 2003.

[33] Thi Thieu Hoa Le, Luigi Palopoli, Roberto Passerone, and Yusi Ramadian. Timed-automata based schedulability analysis for distributed firm real-time systems: a case study. *International Journal on Software Tools for Technology Transfer*, 15(3):211–228, 7 2012.

[34] Jim Ledin. *Simulation Engineering*. CMP Books, Lawrence, Kansas, 2001.

[35] John Lehoczky P., Lui Sha, and Jay Strosnider K. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments, 1987.

[36] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[37] A.K. Mok and D. Chen. A multiframe model for real-time tasks. *17th IEEE Real-Time Systems Symposium*, pages 22–29, 1996.

[38] C. Norstrom and a. Wall. Timed automata as task models for event-driven systems. *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, pages 182–189, 1999.

[39] James Nutaro, Phani Teja Kuruganti, Laurie Miller, Sara Mullen, and Mallikarjun Shankar. Integrated hybrid-simulation of electric power and communications systems. In *Power Engineering Society General Meeting, 2007. IEEE*, pages 1–8. IEEE, 2007.

[40] Risat Mahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 50(4):509–547, 2014.

[41] Sameer M Prabhu and Pieter J Mosterman. Model-Based Design of a Power Window System: Modeling , Simulation , and Validation. *Society for Experimental Machines IMAC Conference*, 2004.

[42] Bern Stegeman, Herman J. Damveld, Olaf Stroosma, Marinus van Paassen, and Max Mulder. Effects of visual delay in flight simulators. In *AIAA Modeling and Simulation Technologies (MST) Conference*, Boston, MA, 2013.

[43] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The Digraph Real-Time Task Model. *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 71–80, 4 2011.

[44] Martin Stigge and Wang Yi. Models for Real-Time Workload: A Survey. In *Proceedings of a conference organized in celebration of Professor Alan Burns' sixtieth birthday*, pages 133–159, 2013.

[45] K. W. Tindell, a. Burns, and a. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 3 1994.

[46] Ken Tindell and John Clark. Holistic shedulability analysis for distributed hard real-time systms. *Microprocessor and Microprogramming*, 40(2-3):117–134, 1994.

[47] Uppaal 4.0: Small Tutorial, 2009. Accessed: 2019-11-10.

[48] I. Yavrucuk, E. Kubali, and O. Tarimci. A low cost flight simulator using virtual reality tools. *IEEE Aerospace and Electronic Systems Magazine*, 26(4):10–14, April 2011.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Ahsan, Muhammad Uzair
Nationality: Pakistani
Date and Place of Birth: 25.02.1979, Karachi
Marital Status: Married

EDUCATION

Degree	Institution	Year of Graduation
M.S.	GIK Institute of Engineering Sciences and Technology	2004
B.E.	NED University of Engineering and Technology	2002

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
16	Ministry of Defence, Pakistan	GM (Tech)

PUBLICATIONS

Ahsan, M.U., Oğuztüzün, M.S.H., Schedulability Analysis of Real-time Multi-frame Co-simulation for Multi-core Platforms, 2019. *Turkish Journal of Electrical Engineering & Computer Sciences* 27(5), pp: 3599 - 3616.

International Conference Publications

Abbasi, A.Z., Ahsan, M.U., Shaikh, Z.A., and Nasir, Z. CAWD: A tool for designing context-aware workflows. In *The* 2^{nd} *International Conference on Software Engineering and Data Mining*, pp: 128 - 133. IEEE