Real-Time Embedded Software Modeling and Synthesis using Polychronous Data Flow Languages

Matthew W. Kracht

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

February 7, 2014
Blacksburg, Virginia

Keywords: Synchronous Languages, Real-Time Systems, Schedulability Analysis

Real-Time Embedded Software Modeling and Synthesis using Polychronous Data Flow Languages

Matthew W. Kracht

(ABSTRACT)

As embedded software and platforms become more complicated, many safety properties are left to simulation and testing. MRICDF is a formal synchronous language used to guarantee certain safety properties and alleviate the burden of software development and testing. We propose real-time extensions to MRICDF so that temporal properties of embedded systems can also be proven. We adapt the extended precedence encoding technique of PRELUDE and expand upon current schedulability analysis techniques for multi-periodic real-time systems.

This work is partially supported by the US Air Force Research Laboratory grant FA8750-11-1-0042, and the US Air Force Office for Scientific Research (grant FA8655-13-1-3049).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems are becoming increasingly prevalent within our world and are being used for ever more important tasks from avionics to control within motor vehicles. With this wide spread proliferation of embedded systems comes the need for software that has certain guarantees whether it is safety, response time, or performance. Specifically, *safety-critical* applications require these characteristics. This introduces the need for programming languages that have formal semantics and can provide the ability to analyze and verify software systems for these applications. In the past, software designers were forced to used imperative languages that were more closely bound to the hardware and offered no or little formal semantics. This lack of formal languages and verification techniques resulted in many failures such as the case of the Mars Climate Orbiter, which could have been otherwise prevented through extended type checking [2]. Although formally defined languages are important in verifying safety-critical embedded systems there exists a group of systems that are *real-time*. Such systems require safety properties to guarantee correct operation, but also require timing guarantees. In this thesis we look at real-time systems and how formal languages can be used to verify both safety and timing requirements.

Real-Time Systems

Real-time systems are usually reactive systems, meaning they continually receive stimuli from the physical environment and keep producing response. Examples of such systems span anti-lock brakes to large scale SCADA systems. Reactive systems read inputs and compute reactions within known bounds of time. The reason real-time systems are often sample-driven is because these types of systems are easier to analyze [3]. Having a known time interval between reactions greatly simplifies schedulability analysis.

Sample-driven reactions denotes the inputs to a system being read at certain time intervals but real-time systems have even more timing constraints. Not only must they conform to input constraints but they must also compute each reaction within a certain amount of time, referred to as a deadline. These deadlines can be categorized into *hard* and *soft* deadlines. Hard deadlines are those where a reaction not computed by that deadline, leads to a fatal error. An example of this might be the computation of a reaction for an ABS system within a car. A soft deadline is one where a reaction not computed in time does not lead to a complete failure. The behavior of the system is defined by the developer to handle these missed deadlines but overall the system is able to continue. Often times these types of deadlines are given to tasks that are background or maintenance tasks within a system [4].

Often times in practice it is difficult to construct one monolithic real-time task with one deadline for one reaction. Instead, systems are broken down into a set of smaller tasks each with their own characteristics such as deadline and period. These tasks define smaller reactions and can communicate with one another through a variety of constructs. In order to manage these tasks, real-time systems often utilize a real-time operating system, or RTOS. There are a variety of such RTOSs and they offer varoius features from determining which tasks are able to run to handling how communication is handled between tasks. RTOSs are also optimized to reduce latencies between events, which helps guarantee timings for operating system events such as context switches or interrupts. One key role of an RTOS is to determine which tasks are allowed to execute and when. This is known as *scheduling*, and we will discuss a few such schemes.

Real-Time Scheduling

When discussing real-time systems it is important to also discuss algorithms by which tasks are scheduled within such systems. In 1973, Liu and Layland wrote about two such algorithms: Rate Monotonic, RM, and Earliest Deadline First, EDF [5]. There are many more algorithms that exist but these two are cornerstones within commercial and academic RTOS so they are the two algorithms that we focus on in this document.

Rate Monotonic is an algorithm that uses static priorities. This means that priorities are assigned to tasks prior to run-time. Priorities are assigned inversely according to period and there is a simple test to verify that a task set is indeed schedulable [5]. Because priorities are not changed at run-time and it is preverified that tasks will meet deadlines, the overhead required for implementing such a scheduler has been seen as minimal with respect to other such scheduling schemes [6]. RM is then used for many commercial RTOS because of this reduced overhead [6]. Another benefit of RM is that it is an optimal static priority algorithm, meaning that if any other static scheduling algorithm can meet deadlines then RM is guaranteed to also meet all deadlines [5].

Unlike RM, Earliest Deadline First is a dynamic priority scheduling algorithm. Priorities are updated inversely with respect to deadlines of all currently executing tasks [5]. Where RM is optimal amongst static schedulers, EDF is optimal amongst any preemptive schedulers [6]. This means that if any preemptive scheduler can meet all task deadlines for a system then so can EDF. While EDF allows for greater utilization of processing resources it comes with complexity overhead for RTOS and their schedulers. Because of this, EDF has not been as widely adopted amongst commercial RTOS [6]. For our work, we used the ChronOS RTOS which has many possible schedulers but most importantly contains implementations of both RM and EDF [7].

1.1 An Illustrative Case Study

To better understand the types of systems targeted in this work and to briefly explain the optimizations that will be the basis of this work, we present an example embedded control system in Figure 1.1. We will refer to this system as LEU, or Location Estimation Unit. The LEU is a *sampled system* where at periodic intervals inputs are read, a reaction is computed and actuators or outputs of the system drive an external response. In the figure, the frequency, f, deadline, D, and execution time, E are given for each task and the offset of each task is assumed to be 0ms. These characteristics will be discussed in detail in later sections but they define the execution behavior and processing needs of a task. For instance, task LCU will need to execute ten times per second, given $f = 10Hz$, and will need to execute for a maximum of ten milliseconds on the CPU, given $E = 10ms$, and each execution must finish in 100ms from when the task can sample its input, given $D = 100ms$.

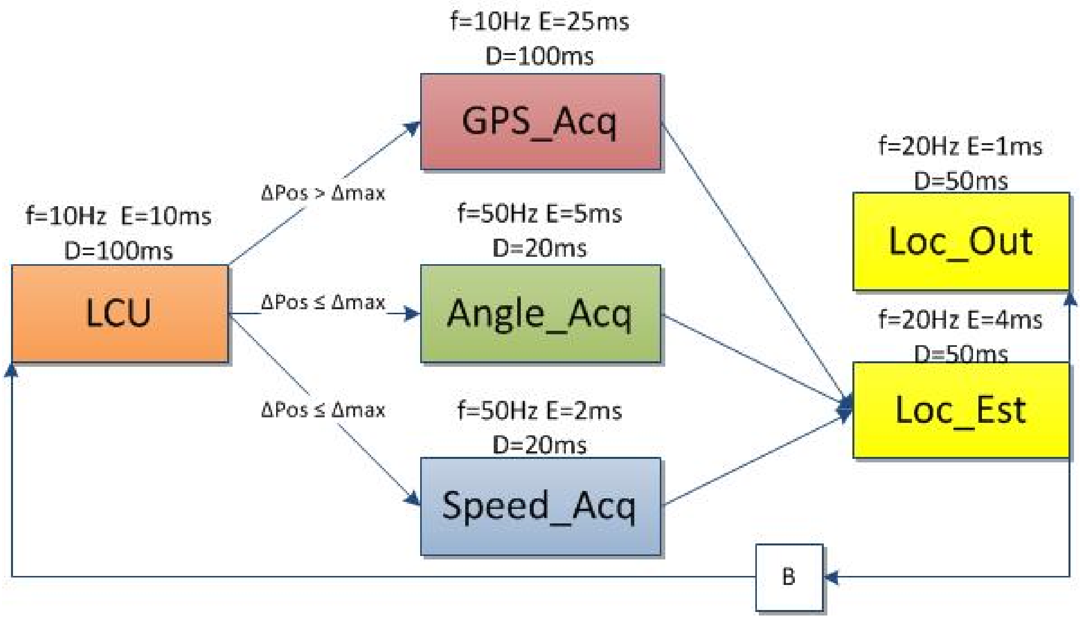


Figure 1.1: Location Estimation Unit

The LEU has two acquisition modes: obtaining multiple velocity vectors from direction and speed sensors (Angle_Acq and Speed_Acq tasks) for computing an estimated position, or directly obtaining GPS coordinates (GPS_Acq task) for a more precise location. The LocEst task then uses whichever data was obtained to calculate a location and sends that value to the Loc_Out task which outputs the value. Loc_Est also determines the error inherent in the measured value and returns that value to LCU. In order to determine which acquisition mode should be used the LCU uses the error value stored in a buffer from LocEst to determine if the current uncertainty value ΔPos has crossed a threshold given as Δmax and GPS_Acq must be triggered to return the uncertainty of the estimated location to a safe level. Conditional communication between tasks is represented on the edges of the task graph.

Using the task set of LEU, we can attempt to create a schedule for both rate monotonic, RM, and earliest deadline first, EDF. In Table 1.1, we show the utilization for each task and the assigned fixed priority for the RM schedule. It is also worth noting that the deadlines of the tasks have been adjusted due to the data dependencies within the task graph. This ensures that the dependencies are followed during the run-time execution of such a system. We will discuss how this is deadline adjustment is done in Section 2.2. There is a *utilization bound* for a set of tasks to be schedulable with RM. If the utilization of a set of tasks is lower than the utilization bound it is guaranteed to be schedulable. This bound is $B(n) = n(\sqrt[n]{2} - 1)$, where n is the number of tasks [5]. From the table, the total utilization of these tasks is .8 which is greater than the utilization bound for six tasks, $B(6) = .734$ [5]. The utilization of a task is the ratio of execution time to period and the total utilization of a model is the sum of all task utilizatinos. This means that a valid RM schedule does not exist for this task set.

Table 1.1: Rate Monotonic Scheduling for LEU

Task	Execution	Period	Deadline	Priority	Utilization
LCU	10	100	15	1	.1
GPS_Acq	25	100	44	4	.25
Angle_Acq	5	20	20	2	.25
Speed_Acq	2	20	20	3	.1
Loc_Est	4	50	48	5	.08
Loc_Out	1	50	50	6	.02

Even when using a dynamic priority scheduling algorithm, such as EDF, the utilization is under one, so we may be able to determine a valid schedule. For brevity we present the execution of this task set using EDF in Figure 1.2. It shows that some tasks will miss their deadlines. Specifically Loc_Est and Loc_Out will miss their first deadline at 50ms. Even though the utilization is under one, the execution of the GPS_Acq task must occur before Loc_Est and Loc_Out due to the dependencies, thus they are unable to meet their specified deadlines:

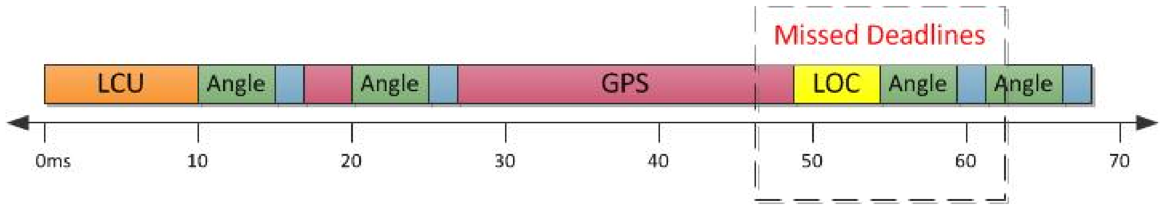


Figure 1.2: Location Estimation Unit EDF Schedule

1.2 Contributions of this Thesis

In the real-time system presented above, the system was not schedulable by a fixed priority algorithm, RM, or a dynamic priority scheme, such as EDF. This is not due to the weakness of the scheduling algorithms but in how the real-time system is modeled and what schedule is being verified. In the LEU system, it is obvious from a user perspective that not all tasks will execute every time. Specifically the data acquisition tasks are mutually exclusive. This insight though is not currently captured in formal languages that are targeted toward verification and synthesis of real-time systems.

In this thesis, we show how to leverage the analysis of both the real-time modeling language, PRELUDE, and the polychronous formal language, MRICDF, in order to better analyze real-time software execution. The MRICDF formalism, and any polychronous language, is specialized in order to determine the relative ordering of events within a system. This specialization can be combined with a sample-triggered formal language such as PRELUDE to help determine under what conditions, task precedence, or data dependence relationships are present between tasks. This allows us to further refine the task set that is used for schedulability analysis since it may be that in each computation round only a subset of the originally task set can actually execute.

Applying this idea to the LEU example will allow us to know exactly when precedences are active between LCU, GPS_Acq, Angle_Acq and Speed_Acq. We will be able to determine the two possible executions within this system: either sampling from the GPS, or estimating location via angle and speed acquisitions. When looking at these two possible task sets it is obvious that this system is in fact schedulable using RM or EDF. These two sub-schedules are shown in Figure 1.3 where EDF is used as the scheduling algorithm. When looking at these two task sets, it is clear that the angle and speed acquisitions tasks carry the highest utilization and total execution time thus this task set is worst case and is used for schedulability analysis. Since this task set is schedulable for EDF and RM then the whole system can be shown to be schedulable.

We will also present how real-time code can be synthesized from MRICDF models. For code generation steps our target OS is ChronOS, a real-time operating system based on the real-time Linux kernel. This way code can be easily ported and tested on any Linux capable platform [7].

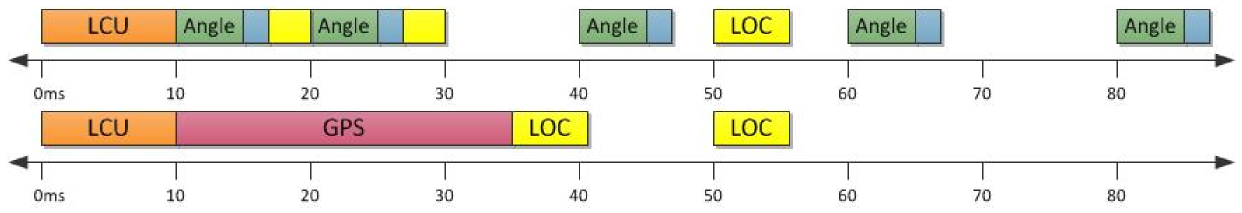


Figure 1.3: Location Estimation Unit Sub-Schedules using EDF

The rest of this thesis is organized into three sections. We will discuss background works of both PRELUDE and MRICDF. We will then present our real-time extensions to MRICDF including how real-time systems are modeled, analyzed and then synthesized into code within an MRICDF modeling and synthesis environment. Finally we will present some results as well as discuss possible improvements and future work.

Chapter 2

Background

2.1 MRICDF

Multi-Rate Instantaneous Channel connected Data Flow, MRICDF, is a formal data-flow language similar to SIGNAL [8]. With MRICDF, inputs to a system can be seen as infinite streams. A data flow network represents the computation needed in order to produce the outputs of the system which are also infinite streams [9]. Using the SIGNAL terminology, each such infinite data stream in called a signal. MRICDF models are used to synthesize correct-by-construction embedded code using a *polychronous* model of computation. In a polychronous model, the model time is relatively relaxed when compared to a synchronous model. The polychronous timing model allows for streams to be computed asynchronously to one another which fits very easily to a multi-threaded environment [10]. In order to understand MRICDF as well as its compiler and development framework EmCodeSyn [11] we will present some preliminary definitions, the formal language, and finally the epoch analysis for MRICDF models.

2.1.1 Preliminaries

Within a MRICDF specification there are individual synchronous blocks called *actors* which contain a set of input and output ports. Actors represent a computation which completes instantaneously because of the *synchrony hypothesis*. The synchrony hypothesis states that the reaction of a system can be seen logically instantaneous with respect to the latency between subsequent inputs on any of the input streams. Actors are interconnected using *channels*, which pass data instantaneously between the ports of actors. The different signals can have different rates, which are allowed given the polychronous model of time. The basic entity in a polychronous language is an *event*. Events may denote any change of value within the system: input ports, output ports, variables, etc.

Definition 2.1. (Event). An event is an occurrence of a new value. We denote the set of all events in a system by Ξ.

The relative occurrences of events can then be represented using a binary relations, over Ξ. Such relations define whether or not an event happened before, simultaneously to, or after another event in the system, or if their relative occurrences do not matter. These relations are defined below:

Definition 2.2. (Precedence, Preorder, Equivalence). Let $\prec$ be a precedence relation between events in Ξ. It is defined such that $\forall a, b \in \Xi, a \prec b$ if and only if a occurs before b. The relation $\preceq$ defines a preorder on Ξ such that $\forall a, b \in \Xi, a \preceq b$ if and only if a occurs before b or a, b occur logically simultaneously, or their order does not matter. Finally the equivalence relation $\sim$, is defined on Ξ such that $a \sim b = a \preceq b \wedge b \preceq a$, meaning that a and b are equivalent only if they occur simultaneously or their order does not matter. Thus $\sim$ represents synchronicity of events.

Even though plainly an instant means a specific point in time, we think of an instant as a maximal set of events such that all events in the set are related by $\sim$, meaning that all events in an instant are synchronous. An instant can also be seen as a maximal set of events that occur in reaction to any one or more events [12].

Definition 2.3. (Logical Instant or Instant). The set of all instants is denoted by Υ. Each instant in Υ can be seen as an equivalence class obtained by taking the quotient of Ξ with respect to $\sim$ such that $\Upsilon = \Xi / \sim$. For each set $S \in \Upsilon$, all events in S will have the property $\forall a, b \in S, a \sim b$, and $\forall a, b, (a \in S_1 \wedge b \in S_2 \wedge S_1 \neq S_2 \wedge S_1, S_2 \in \Upsilon \rightarrow a \nsim b$.

Because all instants are equivalence classes, a precedence relationship can be drawn between instants. We define the relationship $\prec$ between two sets in Υ such that, $S, T \in \Upsilon, S \prec T$ if and only if $\forall (s, t) \in S \times T, s \prec t$ [12]. Each instant contains events on signals. If a signal has no event in an instant then it is considered absent.

Definition 2.4. (Signal) Let T by the domain of values that a signal can take and let $\perp$ denote a special absent value such that $T_\perp = T \cup \perp$; then a *signal* can be defined as a total function of type $\Upsilon \rightarrow T_\perp$. This means that for each instant in Υ, a value or absence of value is implied on a signal x. We denote the set of all events in a signal x as $E(x)$.

Proposition 2.5. *A signal is a set of events that is totally ordered.*

Proof. If a signal contains two synchronous events, then a signal has been updated twice in an instant but that is not possible by definition. $\square$

We denote a specific value of a signal x by function $x(t)$ where $t \in \mathbb{N}$ and t represents the t^{th} instant in the totally ordered set of instants where signal x has a value different from $\perp$. $x(t)$ thus returns the t^{th} event value in the signal x.

Definition 2.6. (Epoch, Clock) The *epoch*, $\sigma(x)$, of a signal x is the maximum set of instants in Υ where for each instant in $\sigma(x)$, x takes a value from T. The *clock* of the signal x is a characteristic function that tells whether or not an event in x is absent or is in the set T. Clock is a function of type $\Upsilon \to [true, false]$ such that for a signal x it returns another signal $\hat{x}$ defined by $\hat{x}(t) = true$ if $x(t) \in T$.

Note that not all inputs and outputs are present or computed during every instant in Υ which means that not all signals have the same epoch or clock. This gives the multi-clocked or polychronous behavior. Using the above definitions and characteristics, three possible relationships can be drawn between any two clocks x and y: equivalent, sub-clocked, or unrelated. If the clocks of x and y are true for the exactly the same set of instants, $\hat{x} = \hat{y}$, then it is said that these two clocks are **equivalent**, and the corresponding signals are also synchronous. If the clock of a signal x is true for a subset of instants where the clock of y is true then it is said that x is a **sub-clock** of y. If the clocks of x and y are not equivalent or subset or superset of the other then the clocks are said to be **unrelated** [10]. It is obvious that some specific subsets of relationships may be drawn from clocks that are unrelated. One type of relationship is mutual exclusion, meaning that $x(t) = true$ if and only if $y(t) = \perp$ and $y(t) = true$ if and only if $x(t) = \perp$. These relationships are stored in a structure called a *clock tree* and will be presented in Section 2.1.3.

2.1.2 The MRICDF Formalism

As was previously mentioned, an MRICDF model consists of modules called actors that are interconnected via instantaneous channels [11]. An actor can be of two different types: primitive and composite. *Primitive actors* are of four types, function, buffer, merge, and sampler, which are denoted by $T_p \in \{F(n, m), B, M, S\}$, while *composite actors* are hierarchic compositions of primitive actors [11], whose type is denoted as T_c. Regardless of whether an actor is primitive or composite it can be represented by $A = \langle T, I, O, N, G \rangle$ where I, O are the set of input signals and output signals respectively, T is the type where $T \in \{T_c, T_p\}$. N is the set of internal actors, which for primitive actor types is an empty set, and G denotes the graph created by the channel connections. A primitive actor can then be described by $A = \langle T_p, I, O, \varnothing, G \rangle$ [12].

Definition 2.7. (Primitive Actor,Composite Actor) A primitive or composite actor is graphically represented by a geometric shape and can also be represented by $A = \langle T, I, O, N, G \rangle$. Each actor has a set of input and output signals, I, O, which associate input and output signals with the actor. T represents the type; for primitive actor $T \in T_p$ while $T \in T_c$ for a composite. N is the set of internal actors which for a primitive actor is null and G denotes the data flow graph created by interconnected channels.

Definition 2.8. (Primitive Actor Port) Each primitive actor has a set of input and output signals, I and O respectively. Graphically a primitive actor is represented by a geometric shape with a set of input ports P_I and a set of output ports P_O of which the input and output signals of the actor are associated. In Figure 2.1, a sampler actor can be seen with input ports i_1,b and output port o. In MRICDF, the port names and corresponding signal names are the same.

Definition 2.9. (Composite Actor Port) Composite actors are interconnected graphs of primitive actors and/or other composite actors, the ports of a composite actor are defined recursively:
For primitive actor A:

$$\text{A set of input ports } P_I^A$$
$$\text{A set of output ports } P_O^A$$

A composite actor C consisting of a set of actors A:

$$P_I^C = (\bigcup_A P_I^A) - P_{int}$$
$$P_O^C = (\bigcup_A P_O^A)$$
$$P_{int} = \bigcup_{(A,B)\in\mathcal{A}}((P_I^A \cap P_O^B) \cup (P_O^A \cap P_I^B))$$

Note that the output ports of a composite actor is not disjoint with the internal set P_{int} because any output port can be read from the outside but an input port that is internal cannot be driven from the outside.

All models when fully expanded, meaning composite actors are replaced by their sub-actors, N, and graph, G, consist of only primitive actors. Each primitive actor also gives implicit constraints for the clocks of the input and output signals. The primitive actors are described in detail below and their implicit clock constraints are shown in Table 2.1.

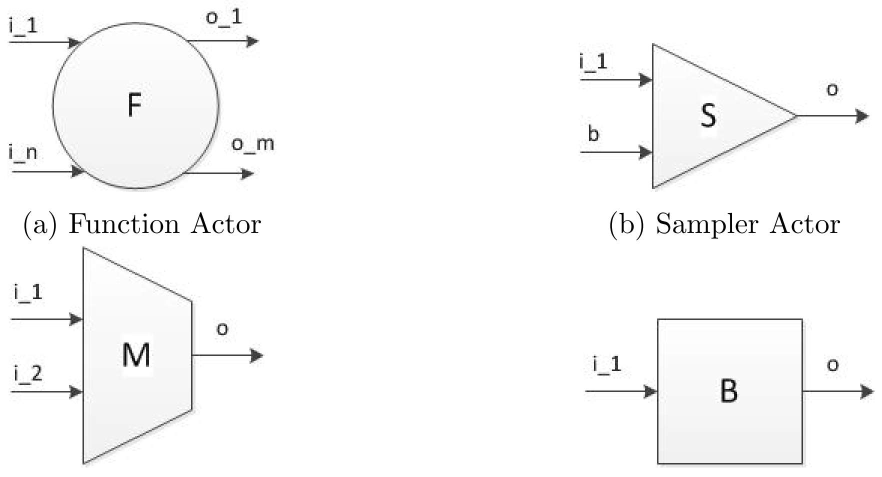
1. **Function :** $< m >= F(n)$ **:** A function actor contains n inputs and m outputs. A function actor's operation is defined by the user to handle any functional computation. A function constrains its input and output signals to all be synchronous; all inputs must arrive on the same logical instant and the outputs must be computed in that same logical instant.

2. **Buffer: B :** A buffer actor is a single input single output synchronous actor. A buffer can be initialized with n number of *delay* values. A value then takes n instants to pass from input signal to output signal, and the first n output values are taken from the initial delay values.

3. **Sampler: S :** A sampler actor can be thought of as a filter. It has two inputs and one output. The first input can be of any type and the second input type must be Boolean. If the first input signal is present while the second input signal is present with a true

value then the value of the first signal is passed to the output signal. If not, the output signal is absent. The two inputs must occur at the same logical instant, and if the second input carries the Boolean value true then at the same logical instant an output occurs.

4. **Merge: M :** A priority merge actor has two inputs and a single output. If the first input, or priority input, signal is present at a logical then the value of that signal is passed to the output port at the same logical instant. If the priority input signal is absent and the second input signal is present, then the value of the second input is passed to the output. If neither input signals are present then the output signal is absent at that logical instant.



(a) Function Actor (b) Sampler Actor (c) Merge Actor (d) Buffer Actor

Figure 2.1: MRICDF Primitive Actors

Table 2.1: MRICDF Actors and Epoch Constraints

Actor	SIGNAL expression	Epoch Relations
Function	(o_1, ...,o_m) := F(i_1, ...,i_n)	σ(o_1) = ... = σ(i_n)
Buffer	o :=i_1\$$n$ $init$ $v_1, ..., v_n$	$\sigma(o) = \sigma$(i_1)
Sampler	o =i_1 $when$ b	$\sigma(o) = \sigma$(i_1) $\cap\, \sigma([b])$
Merge	o =i_1 $default$ i_2	$\sigma(o) = \sigma(i_1) \cup \sigma(i_2)$

The implicit clock constraints along with exogenous constraints, meaning epoch relations explicitly given in the model, can be used to determine all possible epoch relations between signals of an MRICDF model [11]. These epoch relations can be translated to the Boolean domain in terms of Boolean clauses of the model. These Boolean constraints are used to determine an ordering for the clocks within a model. This technique is epoch analysis and will be described in the next section.

2.1.3 Epoch Analysis

The goal of MRICDF is to create executable reactive software from model specifications. In order to do this, the model must be implementable, meaning that the logical instants can be mapped to real computation while respecting all of the temporal relationships between events in the model. In order to schedule these events, a reference signal must be found called the *master trigger*. The master trigger is a signal in the process such that the epochs of all other signals in the process are subsets of the master trigger's epoch. This means that the master trigger initiates each round of computation that happens at each logical instant in the model [11]

In order to perform epoch analysis, a Boolean theory must be constructed for the model. This Boolean Theory consists of a set of variables B_M and a set of clauses T_M. Each Boolean variable in $b_x \in B_M$ denotes if a signal x has an event at an arbitrary instant. The clauses represent all relations that can be derived between events in M. For each actor, specific Boolean clauses are derived. These clauses can be seen in Table 2.2. For any instantaneous channel in M connecting two ports x and y, the respective signals s_x and s_y are synchronous. Therefore, for all such channels in M, there exists Boolean clause $b_x = b_y$ in T_M.

Table 2.2: MRICDF Actors and Boolean Clauses

Actor	Boolean Clauses
Function	$b_{o\_1} = ... = b_{i\_n}$
Buffer	$b_o = b_{i\_1}$
Sampler	$b_o = b_{i\_1} \wedge [b]$ $b_{i_2} = [b_{i_2}] \vee [\neg b_{i_2}]$ $[b_{i_2}] \wedge [\neg b_{i_2}] = false$
Merge	$b_o = b_{i_1} \vee b_{i_2}$

Sequential Implementation

The master trigger of a process can be determined by using the Boolean theory representation of a model M. Let B_M be the set of all Boolean variables, let T_M be the set of Boolean clauses, the master trigger, MT, can be defined as follows:

Definition 2.10. (Master Trigger) The *master trigger* of a process is the signal corresponding to the Boolean variable b_{MT} in B_M, such that $\forall b_i \in B_M, (b_{MT} = false) \rightarrow \neg b$. This means that if the master trigger does not have an event at an instant then no other signals in M will have an event at that logical instant. The Boolean variable b_x represents if a signal x has an event at an instant. Hence a signal x can be a master trigger if b_x is a unitary prime implicate of the Boolean theory [11].

In [10], the computing of the master trigger involves passing the Boolean theory of the model to an SMT solver, Yices [13], and to assign potential master trigger values to false and check whether the theory is satisfiable. There exists a trivial solution to the Boolean theory of any model: assigning all variables to be false. This solution has no consequence in determining the master trigger so the following clause is added to the Boolean theory remove such a solution:

$$\bigvee_{b_i \in B_M} b_i$$

Given the Boolean theory, an SMT solver is used to determine if there exists a unitary prime implicate, or master trigger. If that is the case an iterative approach is used to determine sub-clock relations among the clocks of all signals within a model. A clock tree is formed by determining all sub-clock relations of clocks within M. If every clock relation is known, then the presence or absence of every signal can be determined for every instant of the master trigger. This means that M is deterministic and is *endochronous* [8].

The property of endochrony in SIGNAL [8] is similar to sequential implementability in MRICDF [12], and this property guarantees that the order in which inputs must be read is known by the program and that there is a total ordering over the set of instants within the model.

It may be the case, that a model is not sequentially implementable or endochronous. In this particular case, exogenous constraints must be added to the model in order to force it to conform to this property. From this ordering of instants as well as the data dependencies found in the model graph, a schedule for the actors can be found and from this deterministic sequential code can be generated [11].

Clock Tree

Regardless of whether a model is sequentially implementable or concurrently implementable, the process for determining the precedence relations amongst the epochs of the model is the same. In [10], an SMT based method for determining the epoch relations are stored in an acyclic structure called a *clock tree*. If a model is sequentially implementable then the corresponding clock tree is singly rooted. If a model is concurrently implementable then the clock tree is multi-rooted, creating a *forest* instead of a single tree. The root of any tree is the master trigger of the model, if sequentially implementable, or the master trigger of the sub-model, if concurrently implementable. All clocks under the root represent sub-clocks of the master trigger. A single tree and data dependencies give a total order of all signals within a model and a forest of trees gives a partial order of all signals within a model.

The Polychrony tool constructs the clock tree by partitioning all clocks within a process according to clock relations seen in Table 2.1 [14]. For example a Boolean signal's clock $\hat{s}$ can be partitioned into $[s] \wedge [\neg s]$. They take all such clock partitions of a model and fuse them together in a way that maintains compactness and constructs a much larger tree containing all partition sets. Within this tree, any clock $\hat{s}$ that is a child of $\hat{p}$, carries the relation $\hat{s} \subseteq \hat{p}$. The clock tree in MRICDF is similar but is built in a top down approach using SMT equations to determine relations [10]. Here each node of the tree is an equivalence class of clocks. Each edge between nodes represents the $\subseteq$ relation as in Polychrony but also contains a Boolean predicate. This Boolean predicate defines exactly at which instants of the parent clock the child node clocks are present. An example clock tree can be seen in Figure 2.2.

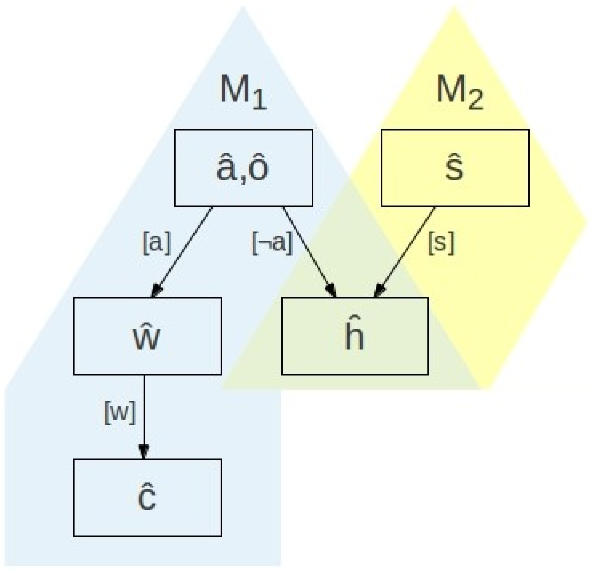


Figure 2.2: Example Clock-Tree for Model M

In Figure 2.2, there are two sub-models M_1 and M_2 in M. The master triggers of M_1 and M_2 are clocks $\{\hat{a}, \hat{o}\}$ and $\{\hat{s}\}$ respectively. The clock of any signal s in M can be described in terms of the Boolean predicates required to traverse the tree from master trigger nodes to the node that contains s. For example, $\hat{w} = \widehat{[a]}$, $\widehat{[a]} \subseteq \hat{a}$, $\widehat{[\neg a]} \subseteq \hat{a}$, $\hat{c} = \widehat{[w]} \subseteq \hat{w}$, $\hat{h} = \widehat{[s]} \vee \widehat{[\neg a]}$, etc.

The clock tree gives a compact representation of all clock relations that have previously been discussed. These relations as well as mutual exclusivity can be informally determined via the tree structure:

1. **Synchronous: =,** Two signals, s_1 and s_2 are synchronous if their clocks are contained

within the same node.

2. **Sub-Clock:** $\subseteq$**,** A the clock of signal s_1 is a subclock of the clock of signal s_2 if the graph can be traversed from s_2 to s_1.

3. **Mutually Exclusive:** $\oplus$**,** Two signls, s_1 and s_2 are mutually exclusive if $\hat{s_1} \cap \hat{s_2} = \varnothing$, and if the conjunction of all Boolean predicates from s_1 and s_2 to their respective root clocks is false.

Given these relationships it is possible to draw implication relationships between signals. These implications can help determine which signals in a model must be present in an instant, as well as what conditions cause certain signals to be present. Given the two signals, s_1,s_2, a Boolean predicate B, and the instant $t \in \Upsilon$, the implications for each type of relationship can be seen in Table 2.3.

Table 2.3: Epoch Relations and given implications between Signals

Relation	Implications
$s_1 = s_2$	$\hat{s_1}(t) \leftrightarrow \hat{s_2}(t)$
$s_1 \subseteq s_2$	$\hat{s_1}(t) \rightarrow \hat{s_2}(t)$ $(\hat{s_2}(t) \wedge\ [B]) \rightarrow \hat{s_1}(t)$ $(\hat{s_2}(t) \wedge \neg[B]) \rightarrow (\hat{s_1}(t) = \perp)$
$s_1 \oplus s_2$	$\hat{s_1}(t) \leftrightarrow (\hat{s_2}(t) = \perp)$
Unrelated	$\varnothing$

The clock tree is a pivotal structure that is created within the compilation process of MRICDF regardless of whether a model is sequentially implementable or concurrently implementable. Being able to determine relationships between clocks of signals statically via the clock tree is useful especially when determining possible ways a model may execute in one instant.

2.1.4 Uses of MRICDF

MRICDF allows to create data-flow models visually and enables algorithmic synthesis if the model satisfies implementable properties. Because it is a polychronous language, it is well suited for expressing systems with concurrency. This means that MRICDF is able to synthesize deterministic code for a larger subset of all models, than synchronous languages such as Lustre [15] or Esterel [16]. It also offers safety checks such as deadlock detection, causal loop detection, and a typing system [11] [17].

Although the language is well suited for describing concurrently implementable systems, there is currently no ability to model real-time systems within its development framework EmCodeSyn. There is a formalism, SynDEx, that targets multi-component distributed real-time systems and can accept Signal models as task definitions. This language is mostly concerned with the specification of distributed systems and finding optimal schedules for multi-task partitioned systems [18]. There are other data flow languages such as Simulink that allow for the targeting of real-time systems but lacks formally defined semantics [19]. Formally defined synchronous languages such as Esterel have been extended to include some real-time capabilities [20]. These languages though impose greater temporal restrictions by forcing models to be synchronous which may not be well suited for implementing large multi-periodic real-time software systems.

2.2 The PRELUDE Framework

Prelude is a formal language used in the development of real-time embedded systems. It is in the family of data-flow languages such as Signal [8] and MRICDF [11], but specifically focuses on defining of software architectures for multi-rate, multi-periodic systems while allowing users to import software functionality from C or from synchronous languages such as Lustre [21]. The Prelude language and tool set offers not only the ability for users to define such systems but also the ability to do schedulability analysis as well as generate executable code. In order to understand work to be presented later, we now discuss some of the major aspects of the Prelude tool set including periodic clocks, task set representation, and schedulability analysis.

2.2.1 Periodic Clocks

The Prelude synchronous real-time model relies on the Tagged-Signal Model [3]. In this model, similar to other synchronous languages, variables and expressions are represnted as *flows*. A pair, $(v_i, t_i)_{i \in \mathbb{N}}$, where v_i is a value in the domain $\mathcal{V}$ and t_i is a date in $\mathbb{Q}, \forall t_i \in \mathbb{Q}, t_i < t_{i+1}$, can be used to represent the value of a variable or expression at a specific date t_i. A flow is then a sequence of these pairs and represents a variable or expression value over the set of all dates. The *clock* of a flow is then a set of dates in $\mathbb{Q}$ in which the value v_i must be computed, and the value v_i must be computed in $[t_i, t_{i+1}[$, or in one logical *instant*. This means that flows may have different dates for when v_i must be computed, and thus can have different clocks as well as different instant durations. With different clocks comes different relations that can be defined between such clocks and flows such as equivalence; two clocks are equivalent if they are active for all of the same dates. Prelude focuses on a specific subset of clocks called *strictly periodic clocks* [22].

Definition 2.11. (Flow, Flow Clock, Flow Instant) A *flow* f is a sequence of pairs, $(v_i, t_i)_{i \in \mathbb{N}}$, where v_i is a value in the domain $\mathcal{V}$ and t_i is a date in $\mathbb{Q}, \forall t_i \in \mathbb{Q}, t_i < t_{i+1}$. The *flow clock*, $ck(f)$, is the set of dates in $\mathbb{Q}$ where one value v_i must be computed. A *flow instant* is one date in the flow clock.

Definition 2.12. (Strictly periodic clock). A clock $h = (t_i)_{i \in \mathbb{N}}, t_i \in \mathcal{T}$, is strictly periodic if and only if:

$$\exists n \in \mathbb{Q}, \forall i \in \mathbb{N}, t_{i+1} - t_i = n$$

where n is the *period* of h, denoted $\pi(h)$, and t_0 is the *phase* of h, denoted $\varphi(h)$.

Strictly periodic clocks are then able to define a flow's instants in terms of a rational valued real time clock, by giving the period and phase, while the Boolean clock of a flow gives the activation condition of a flow for a specific instant. Strictly Periodic clocks offer a way to compare and transform different clocks that are not offered with Boolean clocks alone. Strictly periodic clocks can be compared via their period and phase characteristics. Transformations on these clocks can be done as well to alter their characteristics. Three strictly periodic clock transformations are defined [3]:

Definition 2.13. (Periodic clock division). Let α be a strictly periodic clock and $k \in \mathbb{Q}$. "$\alpha/.k$" is a strictly periodic clock such that:

$$\pi(\alpha/.k) = k * \pi(\alpha), \varphi(\alpha/.k) = \varphi(\alpha)$$

Definition 2.14. (Periodic clock multiplication). Let α be a strictly periodic clock and $k \in \mathbb{Q}$. "$\alpha * .k$" is a strictly periodic clock such that:

$$\pi(\alpha * .k) = \pi(\alpha)/k, \varphi(\alpha * .k) = \varphi(\alpha)$$

Definition 2.15. (Phase offset). Let α be a strictly periodic clock and $k \in \mathbb{Q}$. "$\alpha \to .k$" is a strictly periodic clock such that:

$$\pi(\alpha \to .k) = \pi(\alpha), \varphi(\alpha \to .k) = \varphi(\alpha) + k * \pi(\alpha)$$

2.2.2 PRELUDE Language Details

A Prelude process is defined as a hierarchy of nodes. A node simply defines its output flows in terms of its input flows, where both inputs and outputs are given by the user. The node can be an imported process (C or Lustre), or can be hierarchic where the flows are defined by a series of data flow operators [21]. The node in Prelude is a synchronous block that defines a task, commonly used when referring to real-time software architecture and the period of that task is taken from the period of the input/output flows. For each flow of both input and output ports of a node must eventually be defined in terms of a strictly periodic clock where the characteristics, period and phase, can be given by the user or attempted to be inferred at compile time. Because Prelude is used to target real-time systems, it must also offer a way to define real-time constraints within the language. This is done through user defined *periodicity constraints* and *deadline constraints*. Periodicity constraints are defined on input and output flows and are given as the period and phase of an offset. The deadline constraints are defined on output flows and specify at what specific date a specific value in a flow must have been computed. Given an output flow with a deadline constraint c, the original instant of the flow would be $[t_i, t_{i+1}[$, where v_i must be computed by t_{i+1}, but the deadline constraint forces the instant to be $[t_i, t_i + c[$; the next instant still beginning at t_{i+1}. Deadline constraints do not change the clock of any flow and are only used when scheduling a process [3]. Through the defining of nodes and real-time constraints on flows, users are able to define a set of multi-periodic tasks.

Rate Transition Operators

In order to model larger subsets of real-time software, communication between tasks is required. Within multi-periodic processes, this creates situations where flows may not have equivalent periodic clocks but must pass information between each other. In order to do this PRELUDE defines *rate transition operators*, which are based on the strictly periodic clock transformations. These rate transition operators allow for users to define communication patterns between tasks or nodes in such a way that the communication will always be deterministic [3]. The basic rate transition operators are given in Table 2.4 where f is a flow and $k \in \mathbb{N}$ and $q \in \mathbb{Q}$ [21].

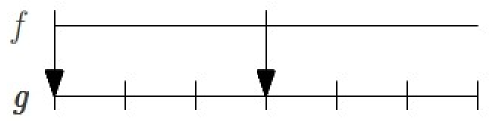
Table 2.4: Rate Transition Operators for PRELUDE

Operator	Description
$f *^\wedge k$	Produces a flow with a period k times shorter than that of f where each value in f is duplicated k times
$f/^\wedge k$	Produces a flow with a period k times longer than that of f where only each kth value in f is kept
$f \sim> q$	Produces a flow with the same period as f but the phase is $q * \pi(f)$
const fby f	(Delay Operator) Produces a flow with the same clock as f but values are delayed by one instant and the first value is *const*

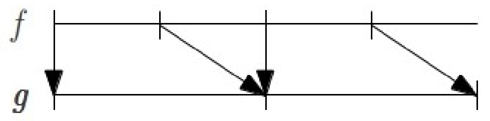
From these basic operators, users can write more complex communication schemes between the ports of nodes. While these transition operators define a way to make strictly periodic clocks equivalent, a more detailed description of the communication patterns is needed. In order to have deterministic communication between tasks, the data dependency relationship must be known for all instants of each task within a system. PRELUDE defines this through a function $g_{ops}(n)$. Let f be a producer flow that is writing to a consumer flow g, the transition operator *ops* is required on f in order to equate the two clocks. $g_{ops}(n)$ then returns the specific instant of g that consumes value at the n^{th} instant of f. The $g_{ops}(n)$ function for basic rate transition operators is defined inductively below:

$$g_{*\hat{}k.ops}(n) = g_{ops}(kn)$$
$$g_{/\hat{}k.ops}(n) = g_{ops}(\lceil n/k \rceil)$$
$$g_{\sim>q.ops}(n) = g_{ops}(n)$$
$$g_{fby.ops}(n) = g_{ops}(n+1)$$
$$g(n) = n$$

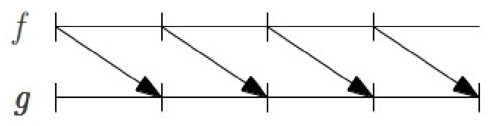
When two clocks are equivalent there is no transition operator necessary; shown as $g(n) = n$. This means that the n^{th} instant of the producer flow f is consumed by the n^{th} instant of the consumer flow g, or $f(n) = g(n)$. Other transitions are defined in terms of this basic function. For example, if flow g has a period that is half that of f the rate transition is $*\hat{}2$ and $g_{*\hat{}2}(n) = 2n$, or $f(n) = g(2n)$. Every other instant of g consumes an instant of f because of the difference in periods. A few examples of different rate transitions and corresponding $g_{ops}(n)$ functions between flows f and g can be seen in Figure 2.3.



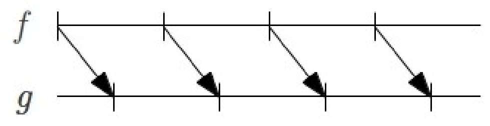
(a) $g_{*\hat{}3}(n)$ = 0,3,6,9,...



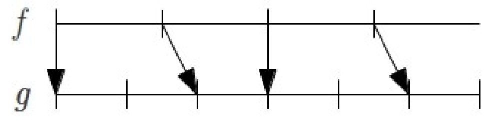
(b) $g_{/\hat{}2}(n)$ = 0,1,1,2,...



(c) $g_{fby}(n)$ = 1,2,3,4,...



(d) $g_{\sim>1/2}(n)$ = 0,1,2,3,...



(e) $g_{*\hat{}3/\hat{}2}(n)$ = 0,2,3,5,...

Figure 2.3: Variety of *ops* rate transitions between flows f and g
J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.* PhD thesis, Universit'e de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.
Used under fair use, 2014

Using the above definitions, more complex rate transitions can be constructed using the composition of the basic operator and the $g_{ops}(n)$ function can still be determined. The deterministic communication patterns given by the above functions will also be used for determining data dependencies between tasks in future sections.

Boolean Operators

Besides the strictly periodic clock operators, which only affect the period and phase of a clock, there are special operators which affect strictly the Boolean clock of the flow, which controls at which instants the flow is present. Specifically, PRELUDE offers two such operators: **when** and **whennot**. These operators are used to under-sample flows. For example, given the flow f and the Boolean flow c, the flow $g = f$ *whennot* c is going to have a clock that is only present when the clock of f and c are present and the value of c in that instant is $false$. This then creates a *under-sampling* relation between the clock of g and the the clocks of f and c. Although these operators are included in the syntax and have formally defined semantics within PRELUDE, they are not treated any differently when it comes to data dependencies and task communication. In [22], it is shown that the $g_{ops}(n)$ function for these operators are merely over approximated using the following functions:

$$g_{when.ops}(n) = g_{ops}(n)$$
$$g_{whennot.ops}(n) = g_{ops}(n)$$

This over-approximation is because of the inherent complications they present for static analysis. The Boolean conditions of these operators, and therefore communication patterns, can only be determined run-time in some cases. For static analysis purposes these conditions must be over-approximated as is done in PRELUDE. While over-approximation removes such Boolean operators for static analysis, these operators are still used in the synthesized code [3].

2.2.3 Task Set Representation

In order to perform static schedulability analysis, the hierarchy of nodes present in the user defined PRELUDE process must be translated into a *task graph*. A task graph is a collection of vertices, where each vertex represents a task, and each edge represents a precedence, or data dependence, relationship between two tasks. The first step, to create a task graph is to expand the original process. This is done recursively by replacing intermediate nodes with equivalent equations until the only nodes left are user defined or imported nodes, which were the original leaf nodes of the process hierarchy [3].

In [3], the authors provide the details of translating the expanded program into an intermediate graph and then how they reduce the intermediate graph into the final task graph. For our purposes it will suffice to understand the structure of the PRELUDE task graph. A PRELUDE process can be represented with a graph $g = (V, E)$ where V is a set of vertices or tasks and E is a set of edges or precedences. Each vertex $v_i \in V$ contains a set of characteristics (in_i, out_i, f_i), where in_i is the set of task inputs, out_i is a set of task outputs, and f_i is the relation between the inputs and outputs. Precedences or edges occur when there is a variable v such that $v \in out_i, v \in in_j$ and the precedence is represented as $t_i \to t_j$.

A vertex v_i represents a task t_i which has real-time characteristics. These are represented as (T_i, C_i, r_i, d_i). Tasks are treated as synchronous blocks, all inputs and outputs have the same periodic clock, pck_i. This clock can be used to derive the period, $T_i = \pi(pck_i)$, as well as the release date or phase, $r_i = \varphi(pck_i)$, of the task t_i. The other two characteristics C_i, worst case execution time, WCET, and d_i, deadline, are derived from user specifications for nodes. There is no analysis done to determine the WCET of a task. Instead the user provides this value by specifying execution times within the process. The deadline, d_i, is by default $d_i = T_i$ but if the user has specified a deadline for any output in out_i then d_i is the minimum specified deadline for all outputs in out_i [3].

There are two different types of precedences, or edges, in the graph g: simple and extended. Simple precedences, $t_i \to t_j$, occur between tasks that have equivalent periodic clocks, $pck_i = pck_j$. This type of precedence does not require a rate transition. Extended precedences, denoted $t_i \overset{ops}{\to} t_j$, occur between tasks that do not equivalent periodic clocks. The precedence requires a rate translation operator *ops* in order to equate the two clocks pck_i and pck_j.

Another characteristic of a task in Prelude is its *activation condition*. An activation condition, $cond_i$, is a Boolean formula describing the conditions under which the task t_i will execute. This is important when there are Boolean operators on rate transitions such as *when* and *whennot*. The definition of $cond_i$, where *pck* is a periodic clock and c is a Boolean condition, can be seen below:

$$\begin{gathered} cond(pck) = true \\ cond(pck\ when\ c) = cond(pck)\ \wedge\ c = c \\ cond(pck\ whennot\ c) = cond(pck) \wedge !c = !c \end{gathered}$$

The activation for the activation condition of task t_i is then the disjunction of $cond()$ for every input clock:

$$cond_i = \bigvee\nolimits_{ck \in ins_i} cond(ck)$$

This means that for any task with at least one input that does not contain a Boolean operator will always be active. Otherwise the task may be inactive depending on Boolean conditions. This activation condition is used to define when tasks should execute which is particularly important to determine when generating executable code from such models. While Prelude uses this activation condition during code synthesis, there is deviation during schedulability analysis. For schedulability analysis, Boolean operators are over-approximated which causes the activation condition $cond_i$ to always be true, thus the analysis is sound but not complete. These altered function definitions can be seen below [22]:

$$\begin{gathered} cond(pck) = true \\ cond(pck\ when\ c) = cond(pck) = true \\ cond(pck\ whennot\ c) = cond(pck) = true \end{gathered}$$

2.2.4 Precedence Encoding

Common scheduling policies such as Earliest Deadline First (EDF) or Rate Monotonic (RM) are not suited to handle task sets with precedences. This makes it particularly challenging to perform code generation as well as to verify the schedule of the task set. In order to handle this, Prelude uses a technique of encoding task precedences into the characteristics of each task which is extended from simple precedences in [23] to handle multi-periodic precedences inherent in Prelude. This allows for the system to retain the same deterministic execution while creating a task set that is completely independent. This independent task set is then easily handled by common real time schedulers and schedule verification techniques [3]. Another benefit to the precedence encoding approach is that it does not require the use of semaphores or synchronization in order to maintain determinism. In [21], the authors cite that the communication between a producer task and a consumer task maintain two properties: the producer must complete before the consumer begins, and that the produced data must be available as long as the consumer requires it. The first of these properties will be assured through the precedence encoding and the second property is handled during the code synthesis via buffering protocols [3].

Simple Precedences

The explanation of the precedence encoding technique is best given through the explanation of encoding of task precedences within the context of a simple precedence, meaning $t_i \rightarrow t_j$ where tasks t_i and t_j have equivalent periodic clocks, $pck_i = pck_j$. In order to respect the communication model, t_i must complete its computation before t_j can begin its computation. This means that t_i must complete its computation in time for t_j to perform its computation and still meet its deadline d_j. The adjusted deadline for t_i then becomes $d_i^* = min\{d_i, d_j - C_j\}$ and the adjusted release time for t_j becomes $r_j^* = max\{r_j, r_i + C_i\}$ [23]. From the work presented in [23], a precedence encoding can be determined for any task with only simple precedences. The definitions are as follows where $pred(t_i)$ gives the set of predecessor tasks of t_i and $succ(t_i)$ gives the set of successor tasks of t_i:

$$\begin{aligned} d_i^* &= min\{d_i, d_j^* - C_j\} \forall t_j \in succ(t_i) \\ r_j^* &= max\{r_j, r_i^* + C_i\} \forall t_i \in pred(t_j) \end{aligned}$$

Extended Precedences

In [3], the authors extends the work of [23] to handle the case of *extended precedences*. Extended precedences, denoted $t_i \overset{ops}{\rightarrow} t_j$, are any precedence relations where the periodic clocks of two tasks are not equivalent, $pck_i \neq pck_j$, and the rate transition ops is needed to equate these clocks. It is possible to apply the simple precedence encoding to an extended precedence. The extended precedence will require a different encoding for every task instant. This is because of the differences in release dates of task instants that are dependent. In the simple precedence there is only one case: both tasks release during the same point in time. In an extended precedence the number of cases that need to be encoded varies. If a simple precedence encoding is used for each of these cases then the extended precedence will be encoded. This can create a situation requiring many simple precedence calculations but the number of calculations is bounded due to the periodic nature of the communication. The *hyper-period*, HP, determines this bound. The HP of a task set is the least common multiple of the periods of every task within the set. This is the bound due to the fact that the model will repeat release dates in the HP interval over the execution of the process[24]. The issue with encoding extended precedences this way is that it causes a large deal of computation overhead since task sets and HPs can become quite large; the authors present a method to encode such precedences without unrolling the task set over the HP [3].

To encode extended precedences, the dependency relationships need to be known between all instants of both tasks within the extended precedence. These data dependencies are given with the $g_{ops}()$ function, which was previously discussed in Section 2.2.2. Using this function, for the extended precedence $t_i \overset{ops}{\rightarrow} t_j$, $\forall n, t_i[n] \rightarrow t_j[g_{ops}(n)]$ [3]. A variety of extended precedences can be seen in Figure 2.4. Note that in these examples more dependencies exist than $t_i[n] \rightarrow t_j[g_{ops}(n)]$. This happens in a case of over-sampling, meaning t_j is consuming data produced by t_i at a faster rate than t_i is producing, or $T_j < T_i$. When t_j begins in instant and new data hasn't been produced by t_i, a new instant of t_i hasn't been computed, t_j must consume the previous value of t_i. These data dependencies are shown in Figure 2.4 but are redundant. By definition an instant t of pck_j must be completed before instant $t+1$. Therefore if the dependency $t_i[n] \rightarrow t_j[g_{ops}(n)]$ is respected, meaning $t_i[n]$ is computed before $t_j[g_{ops}(n)]$ begins, so will any dependency that exists $t_i[n] \rightarrow t_j[m]$ where $m > g_{ops}(n)$.

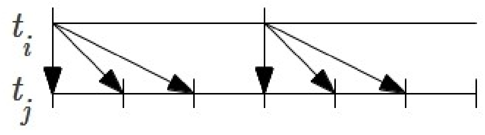
Using g_{ops} the authors in [3] define another function $\Delta_{ops}(n, T_j, r_j)$ which represents the difference, in continuous time, of release dates of related instants of an extended precedence relation. Given that each task t_i contains a set of release dates R_i and the relation $t_i \overset{ops}{\rightarrow} t_j$, the function Δ_{ops} is given below:

Definition 2.16. [3] Let $\Delta_{ops}(n, T_j, r_j) = g_{ops}(n)T_i - nT_i + r_j - r_i$. Given $R_i[n]$ is the release date for the n^{th} instant of task t_i, we have:

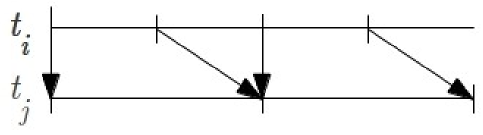
$$R_j[g_{ops}(n)] = R_i[n] + \Delta_{ops}(n, T_j, r_j)$$

In Figure 2.5, the value of Δ_{ops} can be seen for different rate translations for the extended precedence $t_i \overset{ops}{\rightarrow} t_j$. The value of Δ_{ops} for a specific instant n in pck_i gives the continuous time different between the release of that instant and the dependent instant m in pck_j. The value of $m = g_{ops}(n)$. This value can be seen is a slack or extra time an instant n in pck_i has to be computed before that data value must be present for t_j.

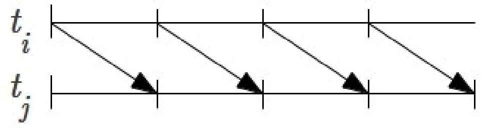
The function Δ_{ops} is ultimately periodic and some examples can be seen in Figure 2.5 [3]. This periodic pattern allows for the adjusted deadlines for tasks with extended precedences



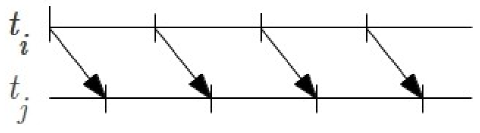
(a) $t_i \overset{*\hat{}3}{\to} t_j$



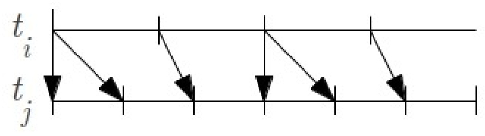
(b) $t_i \overset{/\hat{}2}{\to} t_j$



(c) $t_i \overset{fby}{\to} t_j$



(d) $t_i \overset{\sim>1/2}{\to} t_j$



(e) $t_i \overset{*\hat{}3/\hat{}2}{\to} t_j$

Figure 2.4: Examples of Extended Precedence Relations and Data Dependencies c J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.* PhD thesis, Universit'e de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.
Used under fair use, 2014

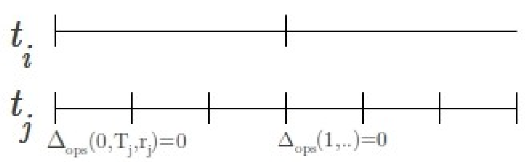
to be represented as ultimately periodic as well [3]. In order to perform *deadline calculus*, or adjusting the deadlines to encode extended precedences, the task deadline characteristics must be translated into *deadline words*, w. Deadline words are used to express the sequence of unitary deadlines, d, and they have the following syntax:

$$w = u.(u)^\omega$$
$$u = d|d.u$$

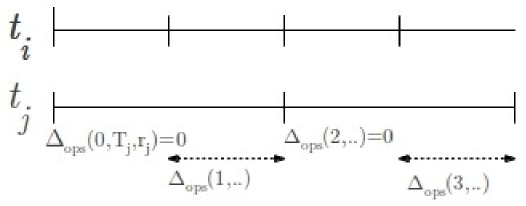
Given a deadline word $w = u.(v)^\omega$, u is the prefix and $(v)^\omega$ is the sequence of deadlines that is repeated infinitely. Furthermore, $w[n]$ gives the n^{th} unitary deadline in the sequence of deadlines [3]. Using deadline words, the deadline word for task t_i, given as w_i, can be computed for the given precedence relation $t_i \overset{ops}{\to} t_j$:

$$w_i \leq W_{ops}(w_j) + \Delta_{ops}(T_j, r_j) - C_j$$

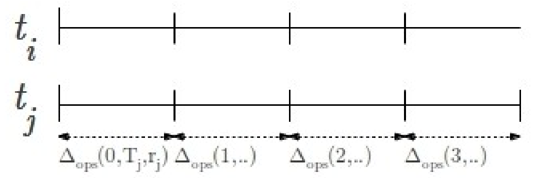
The function $W_{ops}(w_j)$ is a deadline word where $W_{ops}(w_j)[n] = w_j[g_{ops}(n)]$ and where $\Delta_{ops}(T_j, r_j)$ is word representation of the Δ_{ops} sequence, $\Delta_{ops}(T_j, r_j)[n] = \Delta_{ops}(n, T_j, r_j)$



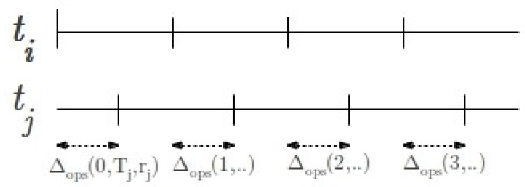
(a) For $t_i \overset{*\hat{}3}{\to} t_j$



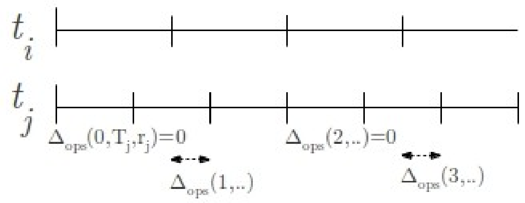
(b) For $t_i \overset{/\hat{}2}{\to} t_j$



(c) For $t_i \overset{fby}{\to} t_j$



(d) For $t_i \overset{\sim>1/2}{\to} t_j$



(e) For $t_i \overset{*\hat{}3/\hat{}2}{\to} t_j$

Figure 2.5: Examples of Δ_{ops}
J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.* PhD thesis, Universit'e de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.
Used under fair use, 2014

[3]. This gives the deadline word for a task with a single successor task. In order to determine the deadline word for a task with multiple successors the least deadline date required by any successor task is taken for each instant of the given task. The deadline word for any task is as follows:

$$w_i^* = min\{w_i, \Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j\}(\forall t_j, t_i \overset{ops}{\to} t_j)$$

In order to adjust the deadline for a task t_i and encode the precedence relationships originating from t_i, the deadline calculus for all tasks $t_j, t_j \in succ(t_i)$ must be computed first. This means that to adjust all deadlines within a graph g, a topological sort is needed starting with tasks that do not have successors and then ending with tasks that do not have predecessors. The algorithm for adjusting all task deadlines within a graph g returning a set of independent tasks q is given in Algorithm 1 [3].

Algorithm 1: Prelude Deadline Adjustment for Precedence Encoding [3]

```
Input: Task graph $g$
Output: Independent Task Set $q$
$q \leftarrow \varnothing$;
forall the $t_i \in g$ do
    if $t_i$ has a user defined deadline $d_i$ then
        $w_i \leftarrow (d_i)^\omega$;
    else
        $w_i \leftarrow (T_i)^\omega$;
    end
end
$S \leftarrow$ List of tasks without successors;
while $S! = \varnothing$ do
    Remove head $t_j$ of $S$;
    $q \leftarrow q \bullet t_j$;
    forall the $t_i, t_i \overset{ops}{\rightarrow} t_j$ do
        $w_i = min(w_i, \Delta_{ops}(T_j, r_j) + W_{ops}(w_j) - C_j)$ ;
        Remove $t_i \overset{ops}{\rightarrow} t_j$ from $g$;
        if $succ(t_i) = \varnothing$ then
            $S \leftarrow S \bullet t_i$;
        end
    end
end
return $q$
```

For an example of the precedence encoding, the task graph of a simplified version of a collision avoidance controller is given in Figure 2.6. This task graph contains a mixture of simple and extended precedence. The deadline of the task $Speed_o$ is not adjusted because it doesn't have any successors. The deadlines of all other tasks are then adjusted through a backward traversal of the graph. The characteristics of each task within the graph as well as the final adjusted deadlines can be seen in Table 2.5.

Given an independent task set q, it is easy to verify the schedulability using different algorithms such as EDF and RM. Prelude previously used a third party tool, Cheddar [25], for schedule verification [26]. They have since shifted the schedulability analysis to their complementary real-time operating system (RTOS) SchedMCore [21]. SchedMCore is also the RTOS that is targeted for code generation which creates a top to bottom tool set for the specification and implementation of multi-periodic real-time embedded systems.

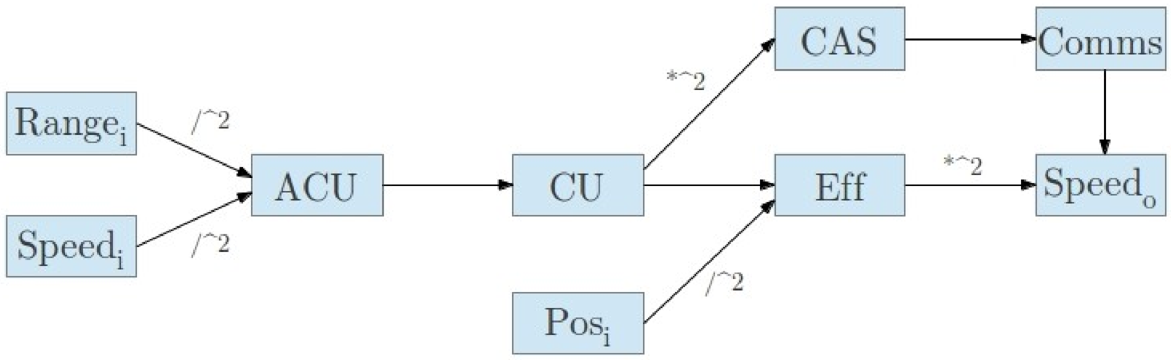


Figure 2.6: A Task Graph for a simplified Collision Avoidance Controller

Table 2.5: Real Time Characteristics and Deadline Adjustments for Figure 2.6

Task	Period	Execution	Deadline	Release	Adjusted Deadline
$Range_i$	50	5	50	0	$(36, 50)^\omega$
$Speed_i$	50	5	50	0	$(36, 50)^\omega$
Pos_i	50	3	50	0	$(43, 50)^\omega$
CU	100	3	100	0	$(42)^\omega$
CAS	50	5	50	0	$(47)^\omega$
$Comms$	50	1	50	0	$(48)^\omega$
Eff	100	5	100	0	$(48)^\omega$
$Speed_o$	50	2	50	0	$(50)^\omega$
ACU	100	3	100	0	$(39)^\omega$

2.2.5 Evaluation of Prelude

Prelude provides a novel approach to the design of multi-periodic real-time embedded systems. They offer a language that abstracts away from a strict task graph to allowing users to create a software architecture in terms of node hierarchies with deterministic execution and communication. They are then able to translate this abstraction into a task graph which can be more easily used for schedulability analysis and code generation. They extend task precedence encoding to cover multi-periodic communication in a compact and efficient manner which allows for the tool to cover more use cases since it does not require synchronization primitives such as semaphores. The language also offers some safety property techniques such as a typing system as well as causal loop detection that are not discussed here.

Although Prelude presents a novel way for the specification and analysis of multi-periodic systems, there are still improvements that are possible for the analysis of this set of real-time systems. Specifically, improvements can be made with respect to schedulability analysis and verification. The over-approximation of the Boolean clock operators such as the **when** operator could result in processes being rejected as unschedulable when in fact they are schedulable. In PRELUDE, the Boolean clocks of periodic clocks are forced to be always present during schedule analysis because the value of a Boolean clock operator cannot always be determined statically. This is true that these values cannot be determined statically but it could be the case where these Boolean conditions create clocks that are mutually exclusive. Consider the following periodic clocks:

Definition	Analysis Simplification
$f = a\ when\ c$	$f = a$
$g = a\ whennot\ c$	$g = a$

In this example it may not be possible to determine the exact value for f or g statically but it would be possible to determine that such periodic clocks were mutually exclusive. This case can be present in different control software where certain tasks are executed conditionally based on the mode or state of a system. In this situation it would be possible to represent such mutual exclusion with a *conditional task graph*, where precedence relations contain conditions under which the precedence occurs. It would then be possible to refine the final task set that is used in PRELUDE for schedule verification. Instead of all tasks executing every instant, a subset of that task set would be the worst possible execution for the process and if that subset is schedulable then the entire process would be schedulable.

In this thesis, a solution to scheduling real-time tasks with conditional behaviors will be presented. The PRELUDE style multi-periodic real-time systems will be combined with MRICDF and EMCODESYN will be extended to allow users to specify tasks as well as their real-time characteristics. We will then use the semantics of MRICDF as well as periodic clock translations to build a conditional task graph from a specification. This conditional task graph will allow us to refine the worst possible execution and reduce the number of systems incorrectly determined as being unschedulable.

Chapter 3

Real-Time EmCodeSyn

Real-Time EmCodeSyn aims to combine the modeling of real-time embedded system software within the framework of a polychronous language MRICDF. The goal of Real-Time EmCodeSYn is to extend the data-flow network formalism of MRICDF with real-time task constructs to combine the code synthesis capabilities of MRICDF with schedulability analysis and code generation of real-time software. In this chapter, we present all requirements to achieve this goal.

We begin by explaining the extensions that we made to the EmCodeSyn environment to allow for real-time system modeling. This includes how users will define tasks and characteristics as well as how the communication between tasks is defined. This leads us to discussing what types of models are valid with respect to the different timing characteristics. For instance, we are only be able to perform analysis on endochronous models which are sequentially implementable in absence of timing constraints.

We show how conditional task graphs are constructed with a specific example using the Location Estimation Unit presented in Chapter 1.1. This example will also be used to show our two methods for refining the worst case execution of an MRICDF model. Each method attempts to achieve a refinement of the execution through different approaches: one used for better refinement and the other for performance. Finally we show how real-time code is synthesized from a schedulable model.

3.1 MRICDF Extensions for modeling of Real-Time Constraints

Within the MRICDF formalism there were no constructs for defining a real-time task. Because MRICDF is a visual data-flow language, we developed the constructs for defining tasks within the MRICDF based framework, EmCodeSyn. By implementing the task definitions within EmCodeSyn, there is a separation between the data-flow model and the task-graph abstraction for real-time tasks. This allows users to alter the real-time characteristics for the software based on the target platform, without changing the underlying model.

Definition 3.1. (Task) A *task*, t_i, is given as $t_i = \langle A_c, I, O, H, c_i, B_i \rangle$ where A_c is a composite actor, I and O are sets of input and output signals respectively, and H is the set of real-time characteristics of the task. c_i is a strictly periodic clock such that $\pi(\hat{c_i}) = T_i$, and $\varphi(\hat{c_i}) = r_i$, where T_i is the period of the task and r_i is the release time of the task as denoted in the task characteristic H. B_i is a set of dependency branches.

In Prelude, the user supplies periods and phases to variables within a process and also specifies deadlines. Instead of interpreting characteristics from the variables, our tool allows the user to specify the characteristics directly when defining a task. This allows the user to have more direct control over how the tasks are defined and formed instead of abstracting the characteristics into certain variable definitions, such as *due* keyword used to specify deadline within Prelude. The task characteristics are defined below:

Definition 3.2. (Task Characteristics) A *task characteristic*, $H = \langle T_i, d_i, r_i, C_i \rangle$, is defined as a tuple: period (T_i), deadline (d_i), offset (r_i), and worst-case execution time (WCET) (C_i). These are defined in terms of number of milliseconds.

- **Period:** T_i **:** The interval of time over which one instant of the task must be computed.
- **Deadline:** d_i **:** The amount of time from when a task is released to when that task must be computed. By default $d_i = T_i$ and if d_i is user defined $d_i \leq T_i$.
- **Offset:** r_i **:** The amount of time the release of the task is shifted within the interval of the period. r_i must specified to be less than d_i.
- **WCET:** C_i **:** The maximum amount of time it will take for the task to be computed on the target platform.

A task is an extension of a *MRICDF* composite actor. A user can select a set of actors, forming A_c, which creates a task t_i within a MRICDF model. From this set of actors the set of input and output signals are derived. These input and output signals are determined based on the input and output ports of the actors that have been selected. The set of actors in A_c and data-flow connections imply a set of *task ports*. A task port is either an input or output of a task and the associated signals of these ports are task signals.

Definition 3.3. (Task Port) Each task, t_i, contains a composite actor A_c where each actor $a \in A_c$ contains a set of input ports P_I^a and a set of output ports P_O^a. The set of task input ports $P_I^{t_i}$ and a set of output ports $P_O^{t_i}$ given:

$$P_I^{t_i} = (\bigcup_{A_c} P_I^{A_c}) - P_{int}$$
$$P_O^{t_i} = (\bigcup_{A_c} P_O^{A_c}) - P_{int}$$
$$P_{int} = \bigcup_{a,b \in A_c}((P_I^a \cap P_O^b) \cup (P_O^a \cap P_I^b))$$

In Figure 3.1, we show a basic MRICDF task. This task contains three actors and each actor contains a set of input and output ports. For instance, $P_I^S = (p3, p4)$ and $P_O^S = (p5)$. To determine the input and output ports of the entire task the internal ports must be defined. An internal port is any port that is contained in output port of an actor within the task and also within an input port of an actor within the task. Simply, it is a port that connects the output and input of two actors within the task. In the example, we have $P_{int} = (p3, p4)$. The input ports of the task, P_I^t, is the collection of input ports to actors within the task that are not internal; in the example $P_I^t = (p1, p2)$. The task output ports, P_O^t, is the same idea except for output ports that are not internal, $P_O^t = (p5)$.

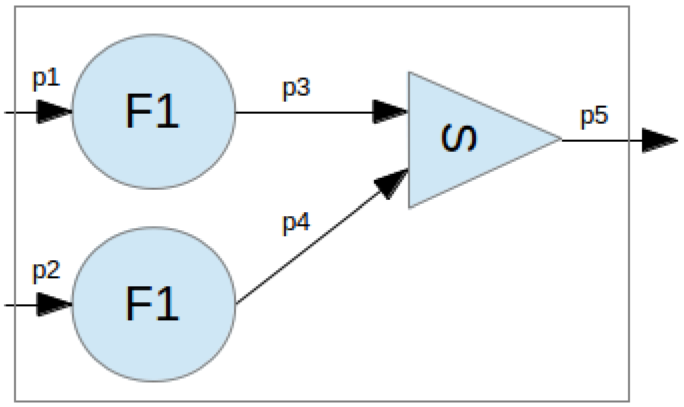


Figure 3.1: An Example Task, t, in MRICDF with ports and actors

Definition 3.4. (Task Input Signal, Task Output Signal, Open Signal) Each task, t_i, has a set of input ports, $P_I^{t_i}$, and a set of output ports, $P_O^{t_i}$. Task input signals, I, and task output signals, O, are the associated signals of these ports. An *open signal* is a signal such that it's associated port is an input or output port of an MRICDF model that receives input from its environment.

The input and output signals of a task are defined by *strictly periodic clocks*. This concept is derived from Prelude, where a signal's epoch is defined both by its Boolean clock, which is the clock in MRICDF, as well as the associated strictly periodic clock. Any strictly periodic clock, h, has a period, $\pi(h)$, as well as a phase, $\varphi(h)$. In Prelude, these are defined by the user but in our tool, these values are taken from the task that the signals are associated with. This means that every input and output signal of a task have the equivalent periodic clocks but does not mean that they have equivalent Boolean clock. This means that two signals may have a period of 20ms but may be present independently of each other within the instants defined by the periodic clock. The distinction made between the strictly periodic clock and the Boolean clock allows for the MRICDF model to remain unaltered even when real-time characteristics are added or changed by the user.

In order for a task set T for a MRICDF model M to be well formed, T must conform to the following properties:

$$\begin{gathered}\forall \text{ actors } a \in M, \exists! \; t_i \in T, a \in t_i \\ \forall t_i \in T, d_i \leq T_i \wedge \\ r_i < d_i \wedge \\ C_i \leq d_i - r_i\end{gathered}$$

In other words, if a task graph is to be well formed all actors must be associated with a unique task and each task characteristic must not automatically exclude T from being schedulable. Each task in T can be seen as its own model where it computes a reaction for each set of inputs it receives. However, within the scope of the entire MRICDF model multiple of these reactions can be contained within the total model reaction. E is the set of edges between the vertices.

3.2 Model Restrictions

Before constructing any task graph from a real-time specification using MRICDF, the clock tree of such model must be constructed. The clock tree will be used to determine all conditional relationships within the graph: all '*sample*' operators. The clock tree also gives information about whether or not a process is endochronous, weakly-hierarchic or not compilable. There are certain clock tree conditions that when they occur make it impossible to guarantee real-time deadlines.

When looking at the ordering of logical clocks within a MRICDF model, there is one key characteristic that creates issues with respect to the real-time execution of a system. This characteristic is a synchronization within a weakly-hierarchic process. In terms of the clock tree, a synchronization is an intersection of two trees. At a synchronization point, the two independently timed clock trees must synchronize for that specific instant and there are conditions under which this occurs.

In Figure 3.2 a weakly-hierarchic model is shown. In this model there are two inputs X and Y whose clocks are unrelated to each other. When $X = 10$, the clock of A is present and carries the value 10. When $Y = 20$ the clock of B is present and carries the value 20. Both A and B are inputs to the same function actor which means $\hat{A} = \hat{B}$. The two inputs are allowed to arrive at complete different instants but when $X = 10$ or $Y = 20$ the two

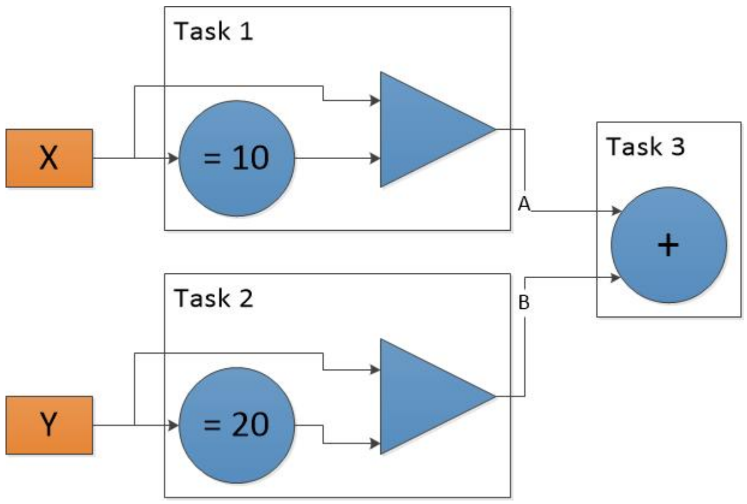


Figure 3.2: An Example Weakly-Hierarchic Model

unrelated clocks must synchronize. This is because the function actor requires both A and B to be present. The clock tree of this model is shown in Figure 3.3. Here the synchronization can be seen in the node that contains the synchronous clocks of A and B.

In a run-time execution of this model, Task 1 and Task 2 can execute in two different threads independent of each other. This is due to their unrelated clocks. Whenever a thread reaches a synchronization condition, either $X = 10$ or $Y = 20$, it must wait for the other thread to also reach its synchronization condition. These conditions must occur in the same logical instant which is dictated by the fact that A and B are synchronous. This means a thread cannot begin to compute the next instant until the function in Task 3 has executed and thus may be blocked for some indefinite period of continuous time.

In terms of a real-time model, the blocking that may occur at run-time is detrimental to a system that must execute according to predefined periods and deadlines. In most cases it is impossible to guarantee or determine that the synchronization points of a model will not cause a deadline to be missed. This is why we reject any model that has a synchronization point in its clock tree during compilation. Although we cannot currently accept weakly-hierarchic models with synchronizations we can accept some weakly-hierarchic models. In the case of a parallel endochronous models we can guarantee that they are non-blocking. These are models with multiple clock trees but none of the clocks intersection or have synchronizations.

3.2.1 Conditional Task Graph Construction

The precedence encoding methods available in Prelude take a task graph g and return an independent task set q. This independent task set represents every task executing its WCET

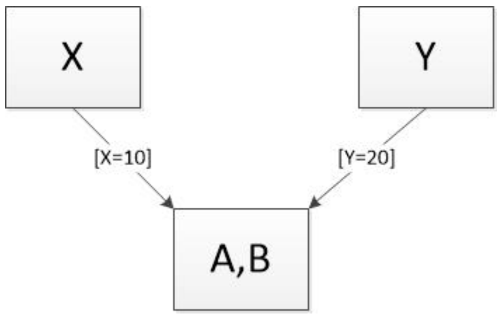


Figure 3.3: Weakly-Hierarchic Clock Tree with Synchronization

every period, which is defined by the strictly periodic clocks of the task. This is because of the over-approximation of Boolean rate operators such as **when** and **whennot** [26]. This over-approximation could easily lead to schedulable models being rejected on the basis that hard deadlines within the model would be violated. It may be the case that only a subset of q needs to execute during run-time, and among these subsets, the one that has the worst slack is called the *worst-case execution*, q'.

In order to better explain how this q' is determined we refer back to the Location Estimation Unit case study. Figure 3.4 gives the MRICDF representation of the real-time system. All blue shapes are MRICDF actors, while all orange boxes are system inputs and green boxes are constants. There are particular groupings of actors that correspond to the tasks within the LEU model. Figure 2.1, defines all actors in terms of their shapes where all circular actors are functions, all triangle are samplers, all squares are buffers, and all trapezoids are merge actors. This convention is followed within Figure 3.4.

Each task is defined by a tuple containing the period, execution time, deadline, and offset in that particular order. For instance, LCU's deadline and period are defined to be 100ms while the execution time is 10ms and the offset is 0ms. Also defined are the rate operators used in inter-task communication. Note that there are no *when* rate operators present. This is because MRICDF clock tree analysis is used to determine the if and what conditions are needed in order for that communication to occur.

In order to find q', we retain the Prelude concept of the activation condition. Each task, t_i, has its own activation condition, which can be represented as a Boolean signal c_i. Let I_{t_i} be the set of input signals of task t_i, the activation condition is given below:

$$[c_i] = \bigcup_{s \in I_{t_i}} \hat{s}$$

The clock of c_i, denoted $\hat{c_i}$, is a strictly periodic clock such that $\pi(\hat{c_i}) = T_i$, and $\varphi(\hat{c_i}) = r_i$. The activation condition is computed for every possible task release date, which is also every possible reaction of the corresponding task. If the activation is true at the release date, $[c_i]$, then a reaction is computed. We refer to the set of all task release dates as *task instants.*

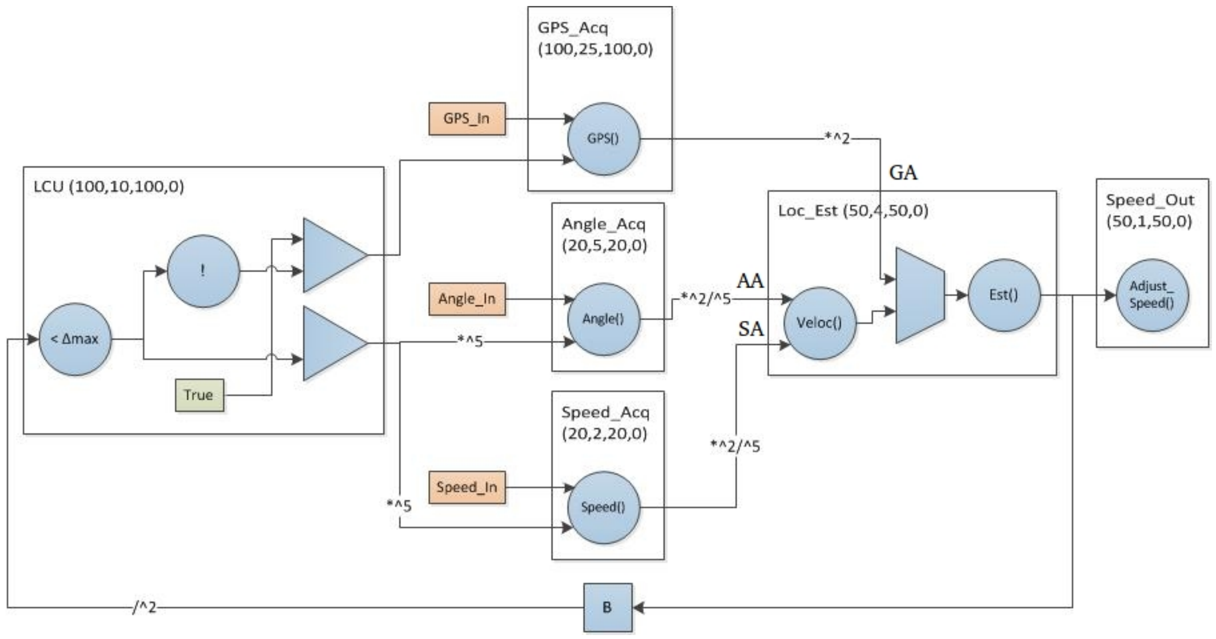


Figure 3.4: MRICDF model of Location Estimation Unit

Definition 3.5. (Task Instants) Each task t_i has an activation condition signal c_i. The set of totally ordered *task instants*, Υ_i, is equivalent epoch of c_i. A task instant is every possible point in time when a reaction may be computed.

The maximum amount of time that any task reaction will take to be computed with preemption is given by the WCET of that task. We refer to any logical instant where the activation condition signal is true, to be a *task activation* and a computational block of WCET duration must be scheduled at this point in time.

Definition 3.6. (Task Activation)Let Υ_i be the set of task instants, which represents the set of logical events where a reaction for task t_i may be computed. The t^{th} instant in Υ_i where $c_i(t) = true$, is a *task activation*, denoted $[c_i(t)]$. The set of all activations of t_i is denoted $[c_i]$. $[\hat{c_i}] \subseteq \Upsilon_i$.

To better explain task instants and task activations we refer to the LEU in Figure 3.4. The GPS_Acq task has a period of 100ms and a release offset of 0ms. That means that given LEU begins executing at $time = 0ms$ it may be computed at 0ms,100ms,200ms,etc. Each one of those points in time represents a GPS_Acq task instant. GPS_Acq however cannot compute a reaction unless it receives an input from LCU. Thus the task activations of GPS_Acq are the instants where there is also an input received from LCU.

For a task t_i, there are a finite number of task instants that are of concern. This number of instants is determined by the *hyperperiod*, HP, of the MRICDF model.

Definition 3.7. (Hyperperiod) The hyperperiod of a model is the least common multiple of all task periods within the model. Over this time interval, the relative release dates of all tasks begin to repeat.

If M is schedulable over the HP then it is also schedulable over any period of time greater than HP. This means that only a finite number of activation conditions of a task need to be determined, $hp(c_i)$. We will use the function $hp(s)$ to represent the number of instants of a signal s. Thus, $hp([c_i])$, is the number of task activations of t_i within a HP. In the case of the LEU, the hyperperiod is 100ms so we only consider five instants of Speed_Acq, Angle_Acq, one activation of LCU, etc.

All tasks within M are composed of a set of actors and ports within M. Any task t_i can be seen as a sub-model, $M_i \subseteq M$. All signals within a task must have the same periodic clock as $\hat{c_i}$ and also must have an epoch that is a subset of the epoch of c_i. Because of this, c_i can be seen as the master trigger of the sub-model M_i and the set task instants of t_i, Υ_i, is equivalent to the epoch of that master trigger.

Even though each task can be seen as its own sub-model computing reactions based on inputs, there still exist dependencies between tasks. For instance, GPS_Acq computes a reaction whenever its activation condition is met; yet the activation condition is dependent on the reaction of LCU. We use a *conditional task graph* to represent these dependencies between task activations. A conditional task graph represents each task as a node and each dependency between activation conditions of tasks as edges. We perform the Prelude precedence encoding prior to creating the task graph so all task characteristics have already been adjusted accordingly.

Let $C = \langle V, E \rangle$ represent a conditional task graph where V is the set of tasks from a given MRICDF model and E a set of task dependencies. E is a subset of $(V \times V)$. E is a set of *dependencies* between task activations and not simply a set of task precedences. Precedences are due to a specific data dependency but do not fully represent how different task activations are related. Each dependency can either be a *forward dependency* or a *backward dependency* and is annotated with operators that define how the activation conditions are related. A forward dependency is an edge in E that is taken from an original precedence relation within the MRICDF model. Forward dependencies represent that if a producer task executes then under what conditions does the consumer task receive that output; they are output signal to input signal dependencies. A backward dependency is an edge in E where the dependency is from a consumer task to one of its input producing tasks. This dependency represents that if a consumer tasks has an activation, what inputs must it have received and thus what producer tasks must also have had an activation.

Table 3.1: General Form of Inter-Task Communication

Case	General Form
Buffered	$ops = const\ fby\ *\hat{}\ l/\hat{}m \sim> n$
Non-Buffered	$ops = *\hat{}l/\hat{}m \sim> n$

Forward Dependencies

Forward dependencies are easily determined via the data-flow of the MRICDF model, M. If there is a port in M that is shared between two different tasks, then there is a forward dependency in C. The *ops* function can then be built from the task characteristics and the Boolean condition statements can be inferred from the MRICDF clock tree.

The rate operator for any edge $e \in E$ is defined using four basic operators which in turn also defines the communication pattern between tasks. These operators are inferred in our tool. Given that a task set T is well formed, it is possible to use the data-flow model M to determine which ports are shared between tasks and thus where there is a data dependence. Given that task t_i's input and output ports cannot be connected via a channel to any other signal associated with t_i then there are only two data-flow cases from which to infer an edge e in the conditional task graph C. Let t_i and t_j be tasks and let $P_I^{t_i}$ and $P_O^{t_i}$ be ports of each respective task, the cases are given below:

- A port in $P_O^{t_i}$ is also in $P_I^{t_j}$ and this port is not contained in the input or output ports of any buffer actor. An edge can be drawn between t_i and t_j in C with the operator $*\hat{}l/\hat{}m \sim> n$.
- If port in $P_O^{t_i}$ is also in $P_I^{t_j}$ and is also the input or output port of a buffer actor in t_i or t_j, then an edge can be drawn between t_i and t_j in C with the operator $ops = const\ fby\ *\hat{}\ l/\hat{}m \sim> n$.

For every edge in C, the characteristic functions given in Table 3.1 must be specified. This is done by using the task characteristics given. Given an edge $t_i \to t_j$ in C, regardless of whether or not the communication is buffered, the values for l, m and n are determined in the same manner. Given that the least common multiple of the periods of t_i and t_j is denoted $LCM_{i,j} = LCM(T_i, T_j)$ and r_j is the release date of t_j; the formulas for l, m and j are given below:

$$l = LCM_{i,j}/T_j$$
$$m = LCM_{i,j}/T_i$$
$$n = r_j/T_j$$

From the above formulas and the buffer actors used to create a buffered edge, the rate operators for all edges within C can be defined. If there is a buffer actor b used in a buffered edge, the value *const* in the rate translation operator is given by the delay values of b.

In Figure 3.5, there are two edges that require the characteristic function to be determined: $A \rightarrow B$ and $A \rightarrow C$. The task characteristics are shown within each task, $\langle T_i, d_i, r_i, C_i \rangle$.

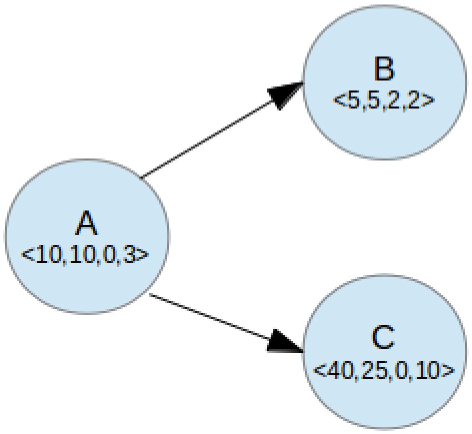


Figure 3.5: Example of defined tasks within MRICDF

For the edge $A \rightarrow C$ we know that A executes at four times the rate of C. This means C must sample one fourth the inputs provided to it, $l = 40/40$, $m = 40/10$. The characteristic function is then $/\hat{}4$. For the edge $A \rightarrow B$, B is execution twice as fast as A and has a 2ms delay on its release, $l = 10/5$, $m = 10/10$, $n = 2/5$, where the characteristic function is $*\hat{}2 \sim> \frac{2}{5}$.

While the rate operators determine the relative timings of producing and consuming data between tasks, they do not determine the Boolean clock of the output signal. The Boolean clock determines whether or not data is passed and whether the input signal of the dependent task is present. Lets assume we have an output signal s_i of LCU communicating to the input signal s_j of GPS_Acq. In order to determine the Boolean clock of s_i, the MRICDF clock tree must be used. The clock of any signal in an endochronous MRICDF model, M can be described in terms of the master trigger. Every signal's clock is either synchronous to the master trigger of a subset of master trigger. The subset relationship is determined via Boolean inputs of samplers which allows us to determine a *when* operator. In the case of the LCU and GPS_Acq, LCU outputs to GPS_Acq *when* $\geq \Delta max$, shown in Figure 3.4. We can then define both the rate translation and the Boolean condition for each forward dependency.

One special case to note is that we exclude all buffered edges within our conditional task graph. While these dependencies are important for code synthesis they can also be used to refine worst-case task sets. Instead we overapproximate and assume that any buffered edge

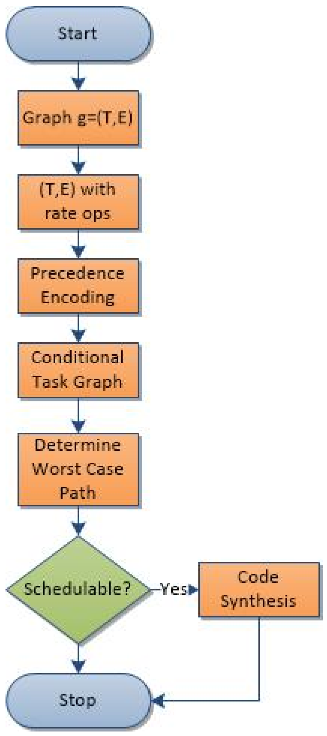


Figure 3.6: Real-Time EmCodeSyn Compilation Process

supplies an infinite stream of inputs to any consumer task.

Backward Dependencies

Backward dependencies in the set E are not taken from the MRICDF data flow as with forward dependencies. The idea of the backward dependency is that if a task activation of t_i is known to be true for an instant of t_i then what are the possible combinations of inputs that caused that activation. We can draw which activations of producer tasks must have also occured for those inputs to be present. We denote a backward dependence as $t_i \overset{[c_j]}{\to} t_k$: if an activation of t_i is caused by the activation of t_j, $[c_j]$, then task t_k must also have had an activation.

To better explain this we look at the Loc_Est task within LEU in Figure 3.4. It has three inputs which we will call GA, AA, and SA, where GA is from GPS_Acq, AA from Angle_Acq, etc. Both AA and SA must be synchronous because they are inputs to the same function. This means that if Loc_Est has an activation and SA is present then so must AA. The other signal, GA, is an input to a merge actor which does not relate its two inputs. This means that s_{GA} is present then Loc_Est will have an activation but it does not require the other two inputs to be present. In this case we would have two backward dependencies:

$Loc\_Est \overset{[c_{Speed\_Acq}]}{\rightarrow} Angle\_Acq$, and $Loc\_Est \overset{[c_{Angle\_Acq}]}{\rightarrow} Speed\_Acq$. These dependencies read "if Loc_Est has an activation and Speed_Acq was known to provide an input then Angle_Acq must also provide an input for that activation."

Algorithm 2: Conditional Task Graph C

```
Input: An MRICDF model $M$
Output: Conditional Task Graph $C = \langle V, E \rangle$
Let $V$ be the set of tasks in $C$ and $E$ be the set of dependencies in $C$
$V \leftarrow \{\}$; $E \leftarrow \{\}$;
foreach $t_i \in M$ do
    $V \leftarrow V \cup t_i$;
end
Let $e$ be a dependency;
Let $mt$ be the master trigger of $M$;
foreach $(t_i, t_j) \in V$ do
    if $\exists$ port $p \in M, p \subseteq P_O^{t_i} \wedge p \subseteq P_I^{t_o}$ then
        Let $s$ be signal associated with $p$, and $mt$ be the master trigger of the model;
        Let $conds_s$ be the Boolean signal such that $mt\ when\ [conds_s]^\wedge = \hat{s}$;
        Lets $ops$ be the computed rate operator;
        $e = t_i \xrightarrow{ops\ when\ conds_s} t_j$
        $E \leftarrow E \cup e$;
    end
end
foreach $t_i \in V$ do
    Let $s_j, s_k$ be input signals of $t_i$, and let $t_j, t_k$ to be the tasks communicating via $s_j$ and
    $s_k$ respectively, $(t_j, t_k) \in pred(t_i)$;
    foreach $(s_j, s_k) \in I_{t_i}$ do
        if $\hat{s_j} \subseteq \hat{s_k}$ and $\exists (t_k, t_i) \in V, t_k \rightarrow t_i$ then
            $e = t_i \xrightarrow{[c_j]} t_k$;
            $E \leftarrow E \cup e$;
        end
    end
end
return $C$;
```

In Algorithm 2, the steps to create the conditional task graph C from the MRICDF model is given. The first step is to create all tasks in C and then to build forward dependencies from those tasks. Finally, backward dependencies are added using the clocks of input signals. It can be seen that no dependencies are drawn for any precedence or data relation where there is a buffer. This is an over-approximation in order to make calculating the execution simpler. We assume that for all inputs receiving a buffered signal that these inputs are always present. This allows us to only have to calculate the execution of M over one HP because there are no delays.

When dealing with the graph C, we denote two helpful functions $succ(t)$ and $pred(t)$ which return successor and predecessor tasks for a specific task, t. The function $succ()$ returns the set of all tasks such that $\forall t_j \in succ(t_i), \exists t_i | t_i \rightarrow t_j \in C$. In other words it returns all tasks that are dependent on t_i. The function $pred()$ returns the set of all tasks such that $\forall t_j \in pred(t_i), \exists t_i | t_j \rightarrow t_i \in C$, meaning all predecessors of t_i produce an input to t_i.

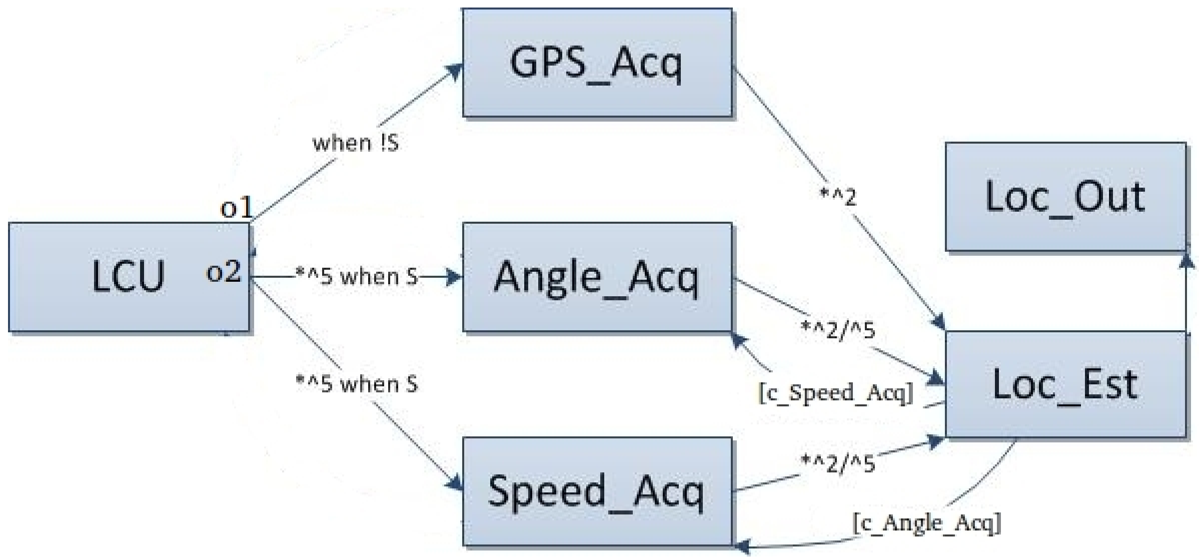


Figure 3.7: The Conditional Task Graph of Location Estimation Unit

The final conditional task graph of the Location Estimation Unit is presented in Figure 3.7. You will notice all of the forward dependencies are there with the exception of the buffered communication between Loc_Est and LCU. We do this to reduce the complexity of task set refinement and instead assume that the buffer always provides an input to any consumer task. By doing this we are able to limit the period of time that we must explore the execution of any model to the HP of that model. The only backward dependencies that exist are those between Loc_Est and Angle_Acq or Speed_Acq, which were discussed previously.

For this conditional task graph we have two methods that can be used to refine the worst-case task set of a model in order to gain a more accurate understand of it's schedulability.

3.3 Task Set Refinement

The goal of task set refinement is to reduce the task set q returned by the Prelude task precedence encoding to a task set q', the worst case task set of the MRICDF model M. The worst case task set is the subset of q that has the least amount of slack in scheduling that does not break any of the dependencies within the task graph C. While q' is the worst case task set we are able to ascertain it may not be the true worst-case task set that would execute on a real system. This could be due to unknown input relations, buffer over-approximations and many other factors. Instead, q' should be the least upper bound of possible schedules of the MRICDF model such that if q' is schedulable then all other possible subsets of q are schedulable. The closer q' is to the true worst-case execution of M the fewer number of false negatives will occur during schedulability analysis.

Dependency Branches

In order to refine q, the conditional task graph C is primarily used to determine which tasks may have activations that are mutually exclusive. Because activations of a task are determined by the tasks activation clock, $\hat{[c_i]}$, which is determined by the inputs that task receives, then the easiest way to determine mutually exclusive task activations is to understand the relationships between the clocks of task output signals.

Every task t_i in C has a set of output signals, O. These output signals may or may not be present during the same task activations, meaning they can have different clocks. The relationship between the output clocks can be determined via the clock tree. We will use an example to explain how this can be done. Let's assume we have three output clocks, $\hat{a}, \hat{b}$, and $\hat{c}$, with the relationships: $\hat{a} \subset \hat{b}$, $\hat{c} \subset \hat{b}$, $\hat{a} \cap \hat{c} = \varnothing$. Cleary $\hat{b}$ is a maximal clock, $\hat{a}$ and $\hat{c}$ are minimums of the set, and $\hat{a}$ and $\hat{c}$ are mutually exclusive. We need to determine what combinations of outputs are possible from these output signals in order to determine task activations of successor tasks. There are possible combinations of outputs at an instant t: $[\hat{b}(t)]$, $[\hat{b}(t)] \wedge [\hat{a}(t)]$, and $[\hat{b}(t)] \wedge [\hat{c}(t)]$. Each of these sets is a *signal branch* and the set of corresponding dependencies in the conditional task graph which must occur based on these signals being present is a *dependency branch*.

Definition 3.8. (Signal Branch) A *signal branch*, *sb*, is a set of output signals of a task that can be present during one possible reaction of that task.

Definition 3.9. (Dependency Branch, Branch Activation) A *dependency branch*, $b = \langle E, br \rangle$, of a task t_i is a set of dependencies, E originating from the output signals of a signal branch, *sb*. E is simply a translation of signals in *sb* to their corresponding dependencies in C. Within *sb* there is a signal l such that $\forall s \in sb, \hat{l} \subseteq \hat{s}$. The *branch activation*, *br* of b, is a Boolean Signal such that $\hat{br}^{\wedge} = \hat{c}_i, [br] \subseteq [c_i], [\hat{br}]\hat{}=\hat{l}$. *br* denotes for which activations of a task t_i the dependencies E of branch b are present.

Every task t_i contains a set of output signals, O. Let O_{ind} be the set of distinct clocks of the output signals in O: $\forall \hat{c} \in O_{ind}, \exists s \in O, \hat{s}\hat{}=\hat{c} \wedge \nexists \hat{d} \in O_{ind}, \hat{d} = \hat{c}$. For each of the clock $\hat{c}inO_{ind}$, a set of output signals, *sb*, must be present during each instant that the clock is true: $\forall s \in sb, \hat{s} \subseteq \hat{c}$. The set of signals, *sb*, is a signal branch. For every output signal in the signal branch there is a set of data dependencies that original from that output signal. These data dependencies are present within C in the form of forward dependencies. The dependency branch, b, is then the set of all forward dependencies in C that are due to the signals in sb. The branch activation of b, br, is true during each instant of $\hat{c_n}$, where $\hat{c_n}$ was previously denoted as the minimal clock of all signals within sb.

To explain signal branches and dependency branches we will refer to the conditional task graph of LEU in Figure 3.7. For task LCU there are two output signals, labeled $o1$ and $o2$ whose clocks are $\hat{o1}$`^=`$[\neg S]$, and $\hat{o2}$`^=`$[S]$; S is a Boolean signal that determines the output of the samplers within the LCU task. In this case there are two mutually exclusive clocks and thus two minimums: $\hat{o1}$, and $\hat{o2}$. The signals that must be present when $\hat{o1} = true$ are only $o1$ and vice versa for $\hat{o2}$. This means that the set of all signal branches from LCU is $\{o1\}, \{o2\}$. For output signal $o1$, there is only one forward dependency, $LCU \rightarrow GPS\_Acq$, within C. For output signal $o2$, there are two forward dependencies in C: $LCU \rightarrow Angle\_Acq$, and $LCU \rightarrow Speed\_Acq$. These groupings of dependencies are then the dependency branches of LCU:

Table 3.2: LCU Dependency Branches

Dependencies	Branch Activation
$LCU \rightarrow GPS\_Acq$	$br = \neg S$
$LCU \rightarrow Angle\_Acq$ $LCU \rightarrow Speed\_Acq$	$br = S$

From this point forward when we refer to a task's *branches*, denoted B_i, we will be referring to its dependency branches. We show both signal and dependency branches in order to clarify how the dependency branches are formed based on the output signals of a task. These branches allow us to express which task activations may be mutually exclusive. For instance, the two branches from LCU show that one reaction of LCU cannot produce an input for both GPS_Acq and Angle_Acq.

Method 1

For the first method, all total combinations of branches and task activations must be explored. From a given task t_i in C, one can use dependencies and branches of t_i to determine what activations of successor tasks of t_i are dependent on which activations of t_i. If we take the task LCU in our LEU task graph we can determine all possible activations of LCU's successor tasks. LCU has one task instant per hyperperiod, which gives two possibilities for the task activations of LCU: $[\neg c_{LCU}(0)]$ or $[c_{LCU}(0)]$. We will consider the case of one task activation. This activation can either produce an output for GPS_Acq, $[\neg S(0)]$, or both Angle_Acq and Speed_Acq, $[S(0)]$. We know this because of the branches of LCU. We can then trace dependent activations through the successors of GPS_Acq, Speed_Acq, or Angle_Acq, and continue until there are no more successors to iterate through. This is how we construct a *path*, π.

A path is a set of activations both task activations and branch activations. A path, π_i is always denoted in terms of its starting task, which is t_i for π_i. The path will describe a set of independent activations, which are the task activations for t_i and the branch activations for all branches within t_i. In the previous paragraph, the independent activations would have been $[\neg c_{LCU}(0)$, $[c_{LCU}(0)], [\neg S(0)]$, or $[c_{LCU}(0)], [S(0)]$. A path will also contain a set of dependent activations, which are both task and branch activations that must occur in all tasks in the fanout from t_i, meaning successors, successors of successors, etc.

Definition 3.10. (Path, Path Set) A *path* is denoted $\pi_i = \langle I, D, R\rangle$. A path has a set of independent activations of t_i , I, and the set of dependent activations of successor tasks, D. R is a set of tuples, $\langle [c_j(t)], t_k\rangle$, where $[c_j(t)]$ is an activation in D and t_k is a task that must provide an off path input. A *path set*, Π_i, is a collection of paths that all originate from the same task, t_i.

While there can be multiple paths from a single task, the clocks of the activation conditions are not required to be the same. Assume two paths, π_i^0 and π_i^1. Both paths must contain the same activation signals of t_i, but they represent different ways the task can execute so $[c_i^0]^\wedge \neq [c_i^1]$. This is an important distinction. We use the same signals within paths but each path represents an execution in an undefined time interval so they are not required to have the same clock.

The set R in a path is used to determine what off path task activations are necessary. These off path inputs are formed by looking at the backward dependencies of tasks within the path. A path starts with a set of independent activations I and the dependent set D is formed from the fan out of these independent activations. These dependent activations are created by traversing forward dependencies. Some tasks within the path may require more inputs for an activation than the ones that are represented in the path. R is used to represent the necessary backward dependencies that much be satisfied for the path to be feasible.

In the LEU example, there are paths which have required off path inputs. If we look at a path, π_{Speed_Acq}, from the Speed_Acq task it will contain independent activations of Speed_Acq, and will contain dependent activations of both Loc_Est and Loc_Out. There exists the backward dependency $Loc\_Est \overset{[c_{Speed\_Acq}]}{\longrightarrow} Angle\_Acq$ in the conditional task graph of LEU. It is already known that Loc_Est receives an input from Speed_Acq based on the path. The backward dependency signifies that in order for this path to be legal it requires an input from Angle_Acq as well. The tuple in R of the path is then equal to $\langle [c_{Loc\_Est}(t)], Angle\_Acq\rangle$. This tuple determines which activations of on path tasks require inputs from which off path tasks but it does not stipulate from which activation of an off path task that input must be produced.

A time line of independent activations of Speed_Acq, SA, is shown in Figure 3.8. The independent activations are for the 0^{th} and the 1^{st} instants of SA. These are shown with solid lines. In order to determine the dependent activations of Loc_Est, LE, in this path we use $g_{ops}()$ and the rate operators of this dependency. We get $g_{*\wedge 2/\wedge 5}(0) = 0$ and $g_{*\wedge 2/\wedge 5}(1) = 1$. This is shown in the graph by which instants of LE the arrows from instants in SA are directed toward. The required off path inputs in this example are $\langle [c_{LE}(0)], AA\rangle, \langle [c_{LE}(0)], AA\rangle$. This means we need at least one activation of AA to provide an input to the 0^{th} and the 1^{st} instants of LE. We show this in the graph as dotted lines from GA. We once again use the $g_{ops}()$ function to determine which activations of GA would be able to provide the necessary inputs to LE.

In this example we know that SA and AA must execute during the same instants because speed and angle are both required to build a velocity vector. However if we had no knowledge of this, the off path inputs would not force them to be synchronous. It merely requires that they both provide an input to a certain activation of a successor but does not specify which independent activation that must be.

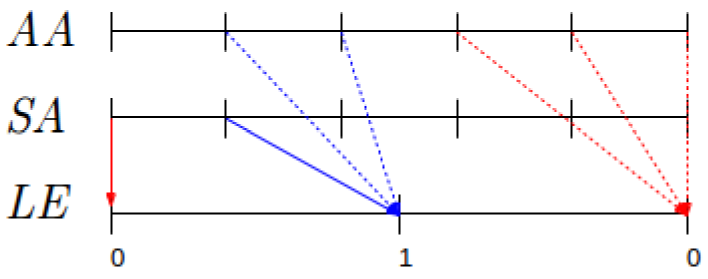


Figure 3.8: An Example of off path inputs to task Loc_Est from the LEU

There can be many paths from one single task. To refine the worst case task set of an MRICDF model we must explore all possible paths from every task within C. This is done in our first method. We begin by simply exploring the possible independent activations within each task, or the *initial path set* of each task. If we know all possible independent activations of all tasks then all of the possible task sets of a model can be easily found. Combining one path from every Path Set of each task gives one possible task set; all of these combinations for a model give all task sets. The behavior defined in the MRICDF model restricts some of these combinations. The first method will utilize the conditional task graph to reject combinations that are not feasible.

Definition 3.11. (Initial Path Set) The *initial path set*, denoted Π'_i, of task t_i is a collection of paths with the same independent task and branch activations of t_i. However the combination of instants over which these activations occur must be unique. $\forall \pi_i^k \in \Pi'_i, \pi_i^k = \langle \{[c_i^k], [br_0^k], ..., [br_n^k]\}, \varnothing, \varnothing\}$ and $\forall \pi_i^l \in \Pi'_i, k \neq l, \exists$ one activation $[s_i]$, $[s_i^k]^\wedge \neq [s_i^l]$.

The first method is presented in Algorithm 3. It computes all possible paths for every task in q. In order to simplify computation, the paths are computed through the backward traversal

Algorithm 3: Compute the Refined Task Set q'

```
Input: Conditional Task Graph $C$
Output: A Refined Task Set $q'$
Let $L$ be the set of tasks in $C$, $\forall t_i \in L, succ(t_i) = \varnothing$;
Let $F$ be the set of tasks in $C$, $\forall t_i \in F, pred(t_i) = \varnothing$;
Let $P$ be a set of path sets, $\forall t_i \in C, \exists! \Pi'_i \in P$;
foreach $t_i \in L$ do
    $P \leftarrow P \bullet computePathSet(t_i, P, C)$;
end
//Every Path from every task has now been explored. Combine input task paths
Let $Final$ be an empty set, which will eventually contain all global paths;
Let $\Pi$ be a set of paths that contains every unique combination task activations for tasks in $F$; foreach $\pi \in \Pi$ do
    $Final \leftarrow Final \bullet recurse(\pi, F, P)$;
end
$Final \leftarrow confirm(Final, C)$;
Let $\pi_w$ be a path;
foreach Path $\pi \in Final, R_\pi = \varnothing$ do
    if WCET($\pi$) > WCET($\pi_w$) then
        $\pi_w = \pi$;
    end
end
Let $q'$ be the set of all independent and dependent task activations in $\pi_w$;
return $q'$;
```

of C. The set of all tasks in C that do not contain any forward dependencies is denoted as L. The paths of these tasks are computed first. Every possible path from tasks in L is given in its initial path set due to the fact that they cannot have dependent activations. We then compute any task whose successor paths are completely explored. This eases computation since all possible dependent activations are contained in the path sets of successors and the method is shown in Algorithm 4. This continues until every task in C has a completely explored path set. The final tasks whose path sets will be computed are those tasks that do not have predecessors. We denote this set of tasks as F. The tasks in F represent the tasks which receive strictly environment inputs or buffered inputs; the refined task set q' is a combination of paths from the path sets of these tasks.

In Algorithm 4 the initial path set of a task is expanded into the path set that explores every activation of that task and tasks in its fanout. The first thing done is defining the successor task activations. Then these activations are used to expand the initial path shown in Algorithm 5.

As we compute path sets for each task, we must make sure that all off path inputs are either

```
Algorithm 4: computeCompletePathSet()
Input: $t_i$, a task within $C$ and a set of path sets, $P$
Output: $P'$, a new set of path sets
$\Pi_i$ is the path set of $t_i$ in $P$;
Let $\Pi$ be a path set;
$\Pi \leftarrow \{\}$;
foreach Path $\pi_i^n \in \Pi_i$ do
    $D$ is the set of dependent activations in $\pi_i^n$;
    $R$ is the set of required off path inputs in $\pi_i^n$;
    foreach Branch $b_q \in B_i$ do
        foreach $t_i \xrightarrow{ops\ when\ s} t_j \in br_m q$ do
            $[br_q]$ is the branch signal of $b_q$;
            $[c_j]$ is the activation condition of $t_j$;
            $\forall t \in [0, hp(c_i)), [c_j(g_{ops}(t))] = [br_q(t)]$; $D \leftarrow D \bullet [c_j]$;
        end
    end
    //$D$ now contains the set of dependent activations of all tasks $t \in succ(t_i)$;
    foreach $[c_j] \in D$ do
        foreach $t_j \xrightarrow{t_i} t_k \in C$ do
            $R \leftarrow R \bullet \langle [c_j], t_k \rangle$;
        end
    end
    //$R$ now contains the set of off path inputs for all tasks $t \in succ(t_i)$;
    $\Pi \leftarrow \Pi \bullet recurse(\pi_i^n, succ(t_i), P)$;
end
$P' = P - \Pi_i + \Pi$;
$P' \leftarrow confirm(P', C)$;
return $P'$;
```

still unknown or have been satisfied with our current path. We use the function $confirm()$, presented in Algorithm 6, to make sure every completed path set does not contain any off path input violations. As more tasks are traversed eventually off path inputs will become part of a path. When this happens all of the off path inputs in R must be satisfied. This is shown in the LEU example. The path sets of Angle_Acq and Speed_Acq have required off path inputs. When building the path set from LCU, these are no longer off path. When combining paths of Angle_Acq and Speed_Acq to build the path set of LCU, these off path inputs must be satisfied. If the off path inputs are not satisfied, then the path is invalid.

Once all complete path sets are determined for the set of input nodes F in Algorithm 3, the path sets of these tasks are combined to determine the task sets for the entire model. This final combination of paths is a merger of path sets. It is a set of paths and not a path set

Algorithm 5: recurse()

```
Input: A path $\pi$, a set of tasks $T$, and a set of path sets $P$
Output: Path Set $\Pi$
$\Pi' \leftarrow \{\}$;
Let $t_i$ be an element of $T$;
Let $\Pi_a$ and $\Pi_b$ be path sets from $t_i$;
$T \leftarrow T - t_i$;
if $T \neq \{\varnothing\}$ then
    $\Pi_a \leftarrow recurse(\pi, T, P)$;
end
$[c_i]$ is the task activation of $t_i$ in $\pi$;
$\Pi_i$ is path set of $t_i$ in $P$;
$\Pi_b \leftarrow \{\varnothing\}$;
foreach $\pi_i^n \in \Pi_i, [c_i^n]^\wedge = [c_i]$ do
    $\Pi_b \leftarrow \Pi_b + \pi$;
end
if $T = \{\varnothing\}$ then
    return $\Pi_b$;
end
foreach $\pi_a \in \Pi_a$ do
    foreach $\pi_b \in \Pi_b$ do
        $\Pi' \leftarrow \Pi' \bullet \langle I_\pi, I_a \cup I_b \cup D_a \cup D_b, R_a \cup R_b \rangle$;
    end
end
return $\Pi'$;
```

since it does not originate from a particular task. The independent activations of the paths in the final set are strictly task activations for tasks in F. If any paths in this set still have any off path input requirements, they are not considered to be a valid and are removed. Of the remaining paths, the worst-case execution times for each path is compared to determine which execution time is greatest. The function $WCET(\pi)$ where π is a path returns the total amount of processor time the path will take. This is accomplished by taking the sum of the WCET of all activations in the path; the WCET of an activation is the same as the WCET of the corresponding task. The task activations in the path with the greatest WCET are then the refined worst case task set q'.

Method 2

The second method improves upon the first method by reducing the total number of paths that are explored. The trade-off for this improvement is a loss in granularity with respect

Algorithm 6: confirm()

```
Input: A set of paths $\Pi$, task graph $C$
Output: Set of paths $\Pi'$
Let $T$ be a set of tasks, $\forall t_i \in T, \exists \pi \in \Pi, [c_i] \subseteq (I_\pi \cup D_\pi)$;
$\Pi' = \Pi$'
foreach $\pi_n \in \Pi$ do
    foreach $r = \langle [c_i(t)], t_j \rangle \in R_{\pi_n}$ do
        if $t_j \in T$ then
            Let $t_j \overset{ops\ when\ s}{\longrightarrow} t_i$ be dependency in $C$;
            if $\nexists u \in [0, hp(c_j)), [c_j^n(u)] \wedge g_{ops}(u) = t$ then
                $\Pi' \leftarrow \Pi' - \pi_n$;
            end
            else
                Let $\pi_m$ be in $\Pi', \pi_m = \pi_n$; $R_{\pi_m} - r$;
            end
        end
    end
end
return $\Pi'$;
```

to task activations. Instead of determining precisely which task activations occur for the worst case execution, only the total number of activations for each task is determined. We use a technique to over-approximate the number of activations in order to retain soundness but gain significant performance increase, which make the technique more viable during the compilation process.

Previously, we have discussed how the *ops* function can be used to determine dependencies between specific task activations. In the case of the second method, a new function must be used that expands beyond *ops*. Specifically, the method requires a function that can take the number of activations of an independent task and return the maximum number of activations of the dependent task within a data dependency. We will refer to this function as $G_{ops}(n, i)$, where n is the number of independent task activations and i is the number of task instants of the independent task, $n \leq i$. The argument i defines a time window over which the number of activations n can occur. This window is $(0, i * T_i]$ where T_i is the period of the independent task.

Let $n_i \overset{*\wedge 2/\wedge 3}{\longrightarrow} n_j$ be an dependency within C. The function $g_{*\wedge 2/\wedge 3}(n) = 0, 1, 2, 2, 3, 4, 4, ...$ for an increasing n. Using this function, the $3m$ and $3m-1$ activations of t_i provide inputs to the same instant of t_j for any integer m. If we look at three instants of t_i, the possible $G()$ values are: $G_{*\wedge 2/\wedge 3}(0, 3) = 0$, $G_{*\wedge 2/\wedge 3}(1, 3) = 1$, $G_{*\wedge 2/\wedge 3}(2, 3) = 2$, and $G_{*\wedge 2/\wedge 3}(3, 3) = 2$. There is no change between $n = 2$ and $n = 3$ because when there are two activations of t_i, t_j is already achieving its maximum activations for the number of dependent task instants. This means that the third activation is guaranteed to not cause another activation of t_j because of the overlap dependent activations in t_j. The reason the second argument is required for $G_{ops}()$ is that depending on the time window, the third activation of t_i may cause an activation in t_j. If we double the interval to six instants, then $G_{*\wedge 2/\wedge 3}(3, 6) = 3$ because there are four instants of t_j in this window and the overlap is not guaranteed to occur until $n = 5$. In actuality this overlap may occur $n > 1$. This depends on which instants of t_i are activations. However, we are concerned with the worst case number of activations, so $G_{ops}()$ returns the max activations and these non-guaranteed overlap cases are ignored.

We can simplify the different communication cases that must be handled by $G_{ops}()$. When a buffer is used there is no dependency that exists in C. When the phase shift operator, $\sim>$, is used the $g_{ops}()$ function is unaffected. This then means that there is no effect in $G_{ops}()$ since it is a function based on the grouping of instants returned by $g_{ops}()$. This means that the only case over which $G_{ops}()$ must be defined is $ops = *\wedge l/\wedge m$. If there are no activations for the independent task, $G_{ops}(0, i) = 0$. The definition of the function given $n > 0$ follows:

$$G_{*\wedge l/\wedge m}(n, i) = \sum_{j=1}^{n-1} \left\lceil \frac{\frac{i*l}{m} - j}{i} \right\rceil$$

In the second method we lose some granularity of task execution in order to increase performance. We also make changes to how the general method traverses the task graph. In the first method we built paths from a single task and included off path inputs to describe how certain unknown tasks must execute. In the second method we use a *frontier* based traversal of the graph. A frontier is a set of tasks, T, within the graph that represent a set of independent tasks where every possible combination of task activations are known for every task that is a successor of any task in the frontier. We build a set of paths, P, where the set of independent activations, I, is comprised of the branch and task activations of tasks in T, and the set of dependent activations, D, is comprised of activations of previously traversed tasks. The frontier is initially the set of all tasks in C, $\forall t \in T, succ(t) = \varnothing$. The frontier is updated every time a task, t_i, in the frontier is traversed. This means that the set of independent activations in the path will no longer contain $[c_i]$, but instead $\forall t_j \in pred(t_i), [c_j]$. This traversal ends when the frontier is comprised of only the tasks in C that do not contain predecessors. At the end of the traversal the set of paths, P, will contain the worst case number of activations for all tasks within C. We present this method in Algorithm 7.

The frontier is propagated through the tasks of the graph through the $traverse()$ method presented in Algorithm 8. A traversal of a task means replacing the activations of that task with activations of its predecessors for every path in P. We will refer to this as *splitting.* Before splitting, there is a set of activations $[c_i^n] \in I$ for any path $\pi^n \in P$. There are a number of combinations of task activations and branch activations that can cause the task activation $[c_i^n]$. Again, we use the forward dependencies to determine which combinations

Algorithm 7: Compute the Refined Task Set q'

```
Input: A conditional task graph $C$
Output: A Refined Task Set $q'$
Let $L$ be the set of nodes in $C$ such that $\forall n_i \in L, succ(n_i) = \varnothing$;
Let $F$ be the set of nodes in $C$ such that $\forall n_i \in F, pred(n_i) = \varnothing$;
Let $P$ be a set of paths and let $m, ..., n$ denote the set of tasks in $L$;
$\forall(hp([c_m]), ..., hp([c_n])) \in (\Pi'_m, ..., \Pi'_n), \exists! \pi = \langle\{[c_m], ..., [c_n]\}, \varnothing, \varnothing\rangle \in P$;
Let $T$ be a set of nodes, $T \leftarrow L$;
while $T \neq F$ do
    foreach $t_i \in T$ do
        if $\forall t_j \in succ(t_i), t_j \nsubseteq T$ then
            $P \leftarrow purge(P, t_i)$;
            $P \leftarrow traverse(P, t_i, C)$;
            $T = T - t_i + pred(t_i)$;
        end
    end
end
foreach $t_i \in T$ do
    $P \leftarrow purge(P, t_i)$;
end
Let $\pi_{wcet}$ be the path,$\pi_{wcet} \subseteq P \land \forall \pi \in Final, wcet(\pi_{wcet}) \geq wcet(\pi)$;
$q' \leftarrow deriveTaskSet(\pi_{wcet}, C)$;
return $q'$;
```

are allowed based on the model. We use the task split of Loc_Est from Figure 3.7 to clarify. The frontier of the LEU model starts as Loc_Out. It then traverses Loc_Out to the only predecessor, Loc_Est. This means that all of the paths in P will have independent activations of Loc_Est and dependent activations of Loc_Out. These paths are presented in Table 3.3.

Table 3.3: Paths in LEU, $T = \{Loc\_Est\}$

Independent {Loc_Est}	Dependent {Loc_Out}
$\{hp([c_{Loc\_Est}]) = 0\}$	$\{hp([c_{Loc\_Out}]) = 0\}$
$\{hp([c_{Loc\_Est}]) = 1\}$	$\{hp([c_{Loc\_Out}]) = 1\}$
$\{hp([c_{Loc\_Est}]) = 2\}$	$\{hp([c_{Loc\_Out}]) = 2\}$

When the splitting occurs Loc_Est is replaced by GPS_Acq, Speed_Acq, and Angle_Acq. There are a total of 72 different combinations of number of activations for these three tasks - $hp([c_{GPS\_Acq}]) \in [0, 1]$, $hp([c_{Speed\_Acq}]) \in [0, 1, 2, 3, 4, 5]$, $hp([c_{Angle\_Acq}]) \in [0, 1, 2, 3, 4, 5]$. Each of these combinations causes a specific number of activations of Loc_Est determined through $G_{ops}()$. The number of activations of Loc_Est is not a simple sum of $G_{ops}()$ for all forward dependencies. In this case, the inputs received from Speed_Acq and Angle_Acq are synchronous. This means that for one activation of Loc_Est an input from both of these tasks must be present. The number of activations is the sum of $G_{ops}()$ for all forward dependencies going to Loc_Est of each maximal input clock to Loc_Est: GPS_Acq and Speed_Acq, or GPS_Acq and Angle_Acq. The possible paths in P after the split are shown in Table 3.4.

Table 3.4: Paths in LEU, $T = \{GPS\_Acq, Speed\_Acq, Angle\_Acq\}$

Independent {GPS_Acq,Speed_Acq,Angle_Acq}	Dependent {Loc_Est,Loc_Out}
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 0, hp([c_{AA}]) = 0\}$	$\{hp([c_{Loc\_Est}]) = 0, hp([c_{Loc\_Out}]) = 0\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 1, hp([c_{AA}]) = 1\}$	$\{hp([c_{Loc\_Est}]) = 1, hp([c_{Loc\_Out}]) = 1\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 2, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 3, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 4, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 5, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 2, hp([c_{AA}]) = 3\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 3, hp([c_{AA}]) = 3\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
...	...
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 4, hp([c_{AA}]) = 5\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 5, hp([c_{AA}]) = 5\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 1, hp([c_{SA}]) = 0, hp([c_{AA}]) = 0\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 1, hp([c_{AA}]) = 1\}$	$\{hp([c_{Loc\_Est}]) = 1, hp([c_{Loc\_Out}]) = 1\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 2, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
$\{hp([c_{GA}]) = 0, hp([c_{SA}]) = 3, hp([c_{AA}]) = 2\}$	$\{hp([c_{Loc\_Est}]) = 2, hp([c_{Loc\_Out}]) = 2\}$
...	...

Some of the 72 combinations are not allowed given the specification. This is due to backward dependencies. In Table 3.4, $\{hp([c_{Speed\_Acq}]) = 1, hp([c_{Angle\_Acq}]) = 2\}$ is not allowed. Any backward dependency $t_i \xrightarrow{c_j} t_k \in C$ dictates that t_i must receive an input from t_k for at least the same activations as it does from t_j. In the case of the second method this means that $G_{ops}()$ of $t_k \rightarrow t_i$ must be greater than or equal to $G_{ops}()$ of $t_j \rightarrow t_i$. In the case of Speed_Acq and Angle_Acq, they must be equal because of the mutual backward dependency.

Every time there is a split from a task, a set of activations is appended to the paths for every predecessor task. For any task with at least two forward dependencies there will occur a situation where there are multiple task activations represented for that task within the path. We account for this situation by *merging* the activations. This is shown in *merge*() in Algorithm 9. To handle this situation we use the branch activations that are also present within the path. The branch activations represent which forward dependencies receive outputs so the only issue when merging is how to determine the number of activations of the task. Each

Algorithm 8: traverse()

```
Input: A set of paths $P$, a task $t_i$, and conditional task graph $C$
Output: An adjusted set of paths $P'$
Let $P'$ be a set of paths and let $m, ..., n$ denote the set of tasks in $pred(t_i)$;
$\forall(hp([c_m]), ..., hp([c_n])) \in (\Pi'_m, ..., \Pi'_n), \exists! \pi = \langle \{[c_m], ..., [c_n]\}, \varnothing, \varnothing \rangle \in P'$;
$\Lambda_S = P'$;
foreach $\pi^n \in P'$ do
    foreach $t_i \xrightarrow{c_j} t_k \in C$ do
        Let $ops_k, ops_j$ represent rate operators for $\{t_k \to t_i, t_j \to t_i\} \in C$ respectively;
        if $G_{ops_k}(hp([c_k^n]), hp(c_k)) < G_{ops_j}(hp([c_j^n]), hp(c_j))$ then
            $P' = P' - \pi^n$;
        end
    end
end
//Combine Paths of $\Lambda$ and $\Lambda_S$ if legal;
foreach $\pi^n \in P'$ do
    foreach $\pi^m \in P$ do
        Let $S_i$ be an input set $S_i \subseteq I_i, \forall(s_j, s_k) \in S_i, \hat{s_j} \nsubseteq \hat{s_k}$;
        Let $imp$ be the max number of activations of $t_i$ possible given $\pi^n, imp = 0$;
        foreach $s \in B_i$ do
            Let $e = t_j \xrightarrow{ops_j \ when \ conds_j} t_i$ be the forward dependency that supplies $s$;
            Let $br$ be the branch of $t_j, e \subseteq br$;
            $imp = imp + G_{ops_j}(hp([br^n]), hp(c_j))$;
        end
        $imp = min(imp, hp(c_i))$:
        if $hp([c_i^m]) = imp$ then
            $\pi^n = \langle I_{\pi^n} \cup I_{\pi^m} - [c_i^m], D_{\pi^m} + [c_i^m], \varnothing \rangle$;
        end
    end
end
$P' \leftarrow merge(P', C)$;
return $P'$;
```

```
Algorithm 9: merge()
Input: A set of paths, $P$, task graph $C$
Output: A reduced set of paths $P'$
$P' = P$;
foreach $\pi \in P'$ do
    if $\exists t_j \in C, ([c_j^n], [c_j^m]) \subset I_\pi$ then
        $[br_n]$ is branch activations of branch $n$ of $t_j$ in $\pi$;
        Let $BR$ be the set of branch activations of $c_j$ in $I_\pi$;
        if $\forall([br_n], [br_m]) \in BR, br_n \subseteq br_m \rightarrow hp([br_n]) \geq hp([br_m])$ then
            Let $C$ be the set of maximal branch activation clocks
            $\forall \hat{c} \in C, \nexists[br_n] \in \pi, [\hat{br_n}] \subseteq c\forall[\hat{br_n}] \in \pi$. Let $[c_j^l]$ represent a set of activations of
            $t_j, hp([c_j^l]) = min(hp(c_j), (\sum hp(\hat{c}), \forall \hat{c} \in C))$;
            $\pi = \langle I_\pi - [c_j^n] - [c_j^m] + [c_j^l], D_\pi, \varnothing \rangle$;
        end
        else
            $P' = P' - \pi$;
        end
    end
end
return $P$;
```

branch activation is represented as a signal $[br]$. There is a set of branch activations S within the independent activations of a path where $S \subseteq B_i$. Given any set of signals S, a set of maximal clocks C of S is defined such that $\forall \hat{c} \in C, \nexists[br_n] \in S, \hat{c} \subseteq [\hat{br_n}]$. The number of activations of t_i, $[c_i]$, within a path being merged is $[c_i] = min(hp(c_i), (\sum hp(c) \forall c \in C))$.

The methods discussed thus far are used to traverse the conditional task graph and build paths. In Algorithm 10 we present our method for reducing the set of paths during this traversal. This reduction is based on the lack of off path inputs. In the first method we propagated paths because we were waiting to merge the path with another path that satisfied the off path inputs. In the second method we never have off path inputs. There are no off path inputs for the initial frontier. Every time a split occurs all predecessors of a task, t_i, are added to the frontier and all inputs to t_i are represented within the paths from the new frontier. Before t_i is split, all of its successors must be traversed based on Algorithm 7. This requires all inputs to all successors of t_i to be represented in set of paths from the frontier, meaning there will be no off path inputs for any successors of t_i. The paths from t_i can be condensed because we do not care about every possible path from t_i but instead we only want to know the worst possible path from t_i for each number of activations of t_i. That is what is done in in Algorithm 10. In the first method we propagated paths so the number of paths grew exponentially as each node was traversed. By using the frontier the number of paths being considered is only dependent on the number of tasks in the frontier and the number of times those tasks execute per hyper period. This is another source of performance

```
Algorithm 10: purge()
Input: A set of paths, $P$, a task $t_i$
Output: A reduced set of paths $P'$
foreach $\pi \in P$ do
    $\pi = \langle I_\pi - B_i, D_\pi, \varnothing \rangle$;
end
foreach $\pi^n \in P$ do
    if $\exists \pi^m \in P, \forall c \in I, hp([c^n]) = hp(c^m) \wedge wcet(\pi^m) \geq wcet(\pi^n)$ then
        $P = P - \pi^n$;
    end
end
return $P$;
```

increase for the second method besides the reduction of state space when only considering number of activations.

When the frontier has propagated to the first tasks, F, of the model the number of paths should only be determined by the number of possible executions of each task in F. If we look at the first method, this what is done once the path sets of all first tasks are constructed. We treat the model as one monolithic block instead of considering every possible path amongst a set of blocks. We determine the worst case path set in P by taking the total of number of activations of each task and their corresponding worst case execution times. A notable difference between the methods is that once the task set is determined in the second method, the specific activations must be assigned given the number of task activations. We do this in Algorithm 7. It is a somewhat arbitrary assignment due to the fact that this worst case task set is an over-approximation of what is done in the first method. There are a variety of options on how to assign specific task activations but we simply fill activations from front tasks and propagate backwards to successors using $g_{ops}()$. From this point, regardless of method, the worst case task set can be sent to a schedule verification tool to determine whether or not the task set is in fact schedulable.

3.4 Schedulability Analysis

Regardless of which method is used to attain the refined schedule q', the schedule is exported to Cheddar for verification [25]. This tool offers a variety of analysis options from simulation to verification for many different schedules including both rate monotonic (RM) and earliest deadline first (EDF). We export our worst-case execution to Cheddar and utilize its capabilities to give feedback on whether or not the task list we determined is schedulable.

We also have two different methods that offer differing positive and negative aspects. The first method has the highest resolution with respect to activations and their dependencies.

However this comes at a major complexity cost for the number of paths possible through the conditional task graph. The second method attempts to prevent this path explosion, but in doing so, loses some of the resolution. Let $n_i \xrightarrow{ops\ when\ conds} n_j$ be a conditional data dependency. When the second method determines the number of activations of t_j for the activations of t_i it must over-approximate this value. This value is determined using $G_{ops}(n, i)$. This over-approximation leads to inaccuracy and thus a less refined q' than produced with the first method.

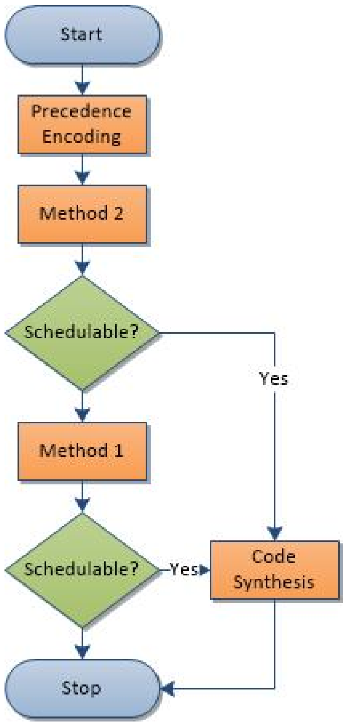


Figure 3.9: Additional Compilation steps for Real-Time Models

In order to take advantage of the second method's speed and also retain the first method's resolution we use both methods during the compilation process. These additional steps are shown in Figure 3.9. Here we append this steps to the traditional MRICDF compilation. We perform the precedence encoding and then build the conditional task graph of the model. From here we determine q' using the second method and if it is schedulable then we are able to synthesize code for the model. If this task set is not schedulable then we determine a new q' using the first method. This comes at a longer compilation time but in some cases can offer some reduction in task set.

In most of the models we encountered, the second method was able to determine a task set q' that was very close to the first method. We show these results in Chapter 4.1. This means that in most cases the second method will be able to determine whether the model is schedulable or not and not introduce the performance cost of also running the first method. If a schedule can be determined for either method, code will be synthesized for the model.

3.5 Code Synthesis

If a model is determined to be schedulable, then code is automatically generated for the given model. The current implementation targets ChronOS, a real-time operating system built around the Linux kernel with real-time extensions [7]. ChronOS relies on the original Linux primitives for all system tasks from timers, file-systems, interrupts, etc. Where ChronOS departs from the original Linux kernel is that is expands upon the original $O(1)$ scheduler to implement a real-time scheduler that can run a variety of single and multi-processor algorithms [7].

We utilize the original Linux kernel to maintain all of our program integrity. This means managing the process memory as well as managing synchronization mechanisms in between threads. In order to make the generated code as flexible as possible, the POSIX API is used to create and manage threads and synchronizations. We then utilize system calls within ChronOS to define each task within a threads execution and then allow the chosen ChronOS scheduling algorithm to determine execution order. The overall API can then be seen as POSIX with the addition of the ChronOS system calls. Before covering the code structure we will discuss some ChronOS scheduler specifics as well as the ChronOS system calls used.

A large distinction between Linux with real-time extensions and ChronOS is how scheduling entities are defined. In a non-real-time environment these scheduling entities could be referred to as tasks and they would be given processor time according to a round robin or time sliced schedule. However in ChronOS, we must define which of these scheduling entities have real-time requirements or not. In order to avoid confusion we will refer to scheduled entities without real-time characteristics as *threads* and entities with real-time characteristics as *tasks*.

Real-time extensions for Linux give an inverted priority scheduler. This scheduler creates a queue of threads at each priority level and every thread in a priority queue will be scheduled after all threads in lower priority queues are scheduled. This scheduler structure can be seen in Figure 3.10 [7].

In ChronOS, they include the ability to define real-time tasks with periods, deadlines, priorities, etc. With the addition of these tasks comes the ability to expand upon the basic Linux scheduler. This is done by creating two queues at each priority level: the Linux run queue and the new ChronOS run queue. The Linux run queue consists of all threads and tasks at that priority level. The ChronOS run queue consists solely of references to real-time tasks within the Linux run queue [7]. The order of the real-time tasks within the ChronOS queue is dependent on the scheduler selected. For instance, if EDF is selected then the first task will be the task with the nearest deadline. When Linux scheduler is to pick another entity from queue it will first take the head of the ChronOS queue and if that is empty schedule the first thread in the Linux run queue. The structure of this ChronOS additional queue can be seen in Figure 3.11.

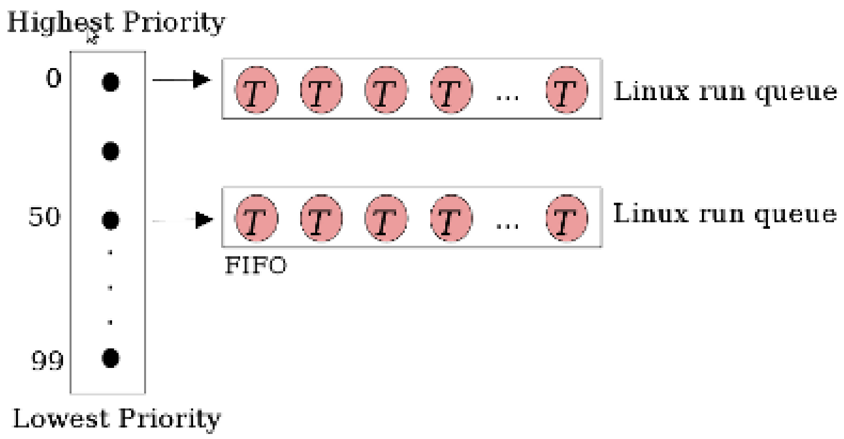


Figure 3.10: Real-Time Linux Scheduler [1]
About ChronOS Linux. URL http://chronoslinux.org/wiki/About_ChronOS_Linux.
Used under fair use, 2014

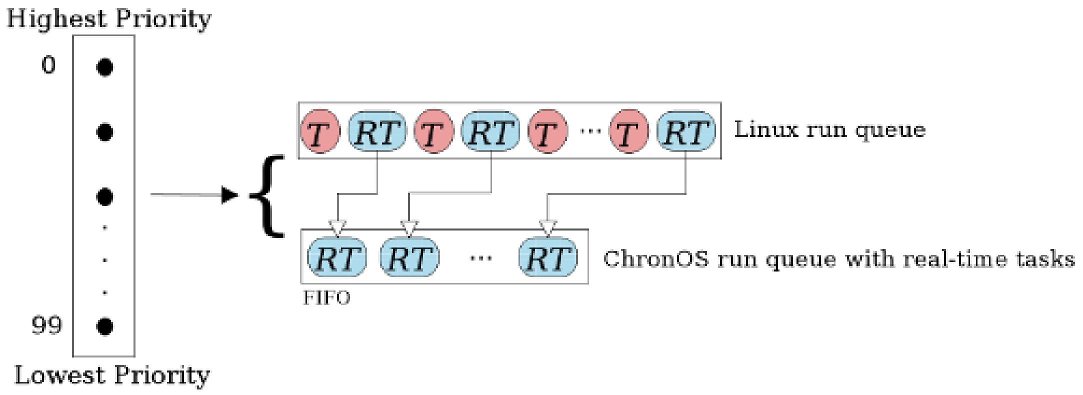


Figure 3.11: ChronOS Scheduler
About ChronOS Linux. URL http://chronoslinux.org/wiki/About_ChronOS_Linux.
Used under fair use, 2014

Although ChronOS adds this second queue to each priority level the original properties of the Linux scheduler remain. Even if there are real time tasks that are to be scheduled, they will be scheduled after any thread or task with a lower priority.

In order to fully discuss generated code we must also present how tasks are defined using ChronOS system calls. The structure of the generated code contains a POSIX thread for every task within the MRICDF model. Not every operation done within these threads are related to the real-time task specification within the model. For instance, task initialization is one such operation that must be completed but is not a real-time operation. In order to maintain this distinction, we define a *real-time section* for each thread. We make ChronOS system calls to enter and exit this section and during this section the thread is elevated to a task within the scheduler. The system call wrappers for the real-time section are given below [7]:

(1) **begin_rtseg_basic(**thread_id, priority, deadline, period**);**

(2) **end_rtseg_self();**

The first function is used to define a task's entry point into a real-time section. This system call can be used by any thread to define another thread's entry point, via the *thread_id* field. This will be important when discussing task synchronizations later on. The other three arguments, *priority*, *deadline*, and *period*, are used to characterize the real-time task that is beginning so that the ChronOS scheduler can properly determine task execution ordering. The second function is called by a real-time task to notify the ChronOS scheduler that it is leaving its real-time section. When this system call is performed it signals the scheduler to select another task or thread for execution [7].

Now that we have discussed the APIs used in the generated code we will discuss the overall structure of the code. Given a model with n number of tasks there will always be $n + 1$ threads created; one *worker thread* for each task in the Real-Time EmCodeSyn and one lightweight *synchronization thread*. We will discuss the general structure of both types of threads.

Worker Thread

For every task in the MRICDF model, one worker thread exists in the generated code. Each worker thread initiates task variables, opens system input files, handles inter-task communication, and also executes the real-time section defined in the MRICDF model. The pseudo code for worker threads is shown in Figure 3.12.

The general structure of the worker thread consists of a while loop where every iteration of the while loop represents one task instant. At the beginning of the instant, the thread pends on a semaphore. This semaphore will be posted to once the release date for the associated task has occurred. The thread does not make the ChronOS system call to promote itself to a real-time task. Instead, this is done within the synchronization thread. However, once a thread has gotten beyond the semaphore wait, it is guaranteed to be within its real-time section.

All tasks within MRICDF are treated as synchronous blocks so all real-time sections within the generated code share the same structure. Before each instant all input signals are read. These input signals can be from system inputs in which case a file is read to get these values. Input signals can also be the outputs of other tasks. For inter-task communication, buffer structures are used similar to those found in the generated code of Prelude and more information can be found in [3]. These input signals can be used to determine if the activation condition of a task has been met. If it has, then the task instant is computed, if not, then the task ends and waits for its next release date. When an instant is computed all output signals are passed on to dependent tasks and then the task is demoted back to a thread.

Overall the worker task is mostly comprised of the real-time section defined within the MRICDF model. In the next section we will cover the synchronization thread which contains subtleties to how the code structure works as a whole.

```
int threadN_id;

void *taskN(void *arg){

initialize_task();
threadN_id = gettid();

while(true){

    sem_wait(&taskN_sem); //Wait to be released

    read_input_signals();

        if( activation_condition_met() ){

            //compute reaction

            write_output_signals();

            taskN_instant++; //Next iteration is next task instant

        }

    end_rtseg_self(); //leave real-time section

} }
```

Figure 3.12: Pseudo code for MRICDF tasks

Synchronization Thread

The synchronization thread is a thread that is used to manage the different release times of tasks. This thread is not a scheduler but is needed to manage differences between how threads are treated within the original Linux kernel and how tasks are treated within CHRONOS.

The most important aspect of the synchronization thread is that the real-time Linux priority of the thread is lower than the real-time Linux priorities of the work threads. This means that whenever this synchronization thread needs to execute it will preempt all tasks that may be executing. By having these priority differences we are able to guarantee that a work thread release will not be blocked due to scheduling of threads versus tasks. For instance, consider a system where there is no synchronization thread, two work threads A and B, and the real-time scheduler is EDF. All work threads would call *begin_rtseg_basic()* for themselves in order to switch their status to a real-time task. If thread A is currently a ChronOS task and is executing and the release time comes for a thread B which a lower deadline, then thread B will be blocked until thread A downgrades from a task to a thread. This is incorrect behavior for EDF since we want thread B to become a ChronOS task as soon as the release date arrives and we want it to preempt thread A. The synchronization thread prevents this by preempting all work threads and elevating work threads to tasks which forces the ChronOS scheduler to recalculate which task should be executing.

Each task thread pends on a semaphore for each instant of the task that will occur. The synchronization thread not only posts on the semaphore to release the specific task but also sets that thread as a real-time task in ChronOS by calling *begin_rtseg_basic()*. By changing the thread status to a ChronOS task prior to posting the semaphore, it guarantees that all code executed between the semaphore pend and the *end_rtseg_self()* call will have ChronOS real-time task characteristics.

A pseudo code structure for the synchronization thread is given in Figure 3.13. The basic structure is a while loop where each iteration of the loop is one hyper-period for the MRICDF model. Within each hyper-period there are a finite number of release dates for every task in the model. The release date for each task is calculated using the period and offset characteristics that were defined for the task by the user. At each discrete release date, all worker threads are elevated to ChronOS tasks and allowed to execute their real-time segments. The synchronization thread then yields the processor, allowing the worker threads to execute, until the next release date.

Another issue that the synchronization thread addresses is the fact that tasks may drift over time. Without the synchronization thread, each task would have to calculate its own next release date and then yield the processor for that amount of time after each instant. If all tasks do this calculation independently there may be cases where over long periods of time the tasks will drift from the global time of the system. By using the synchronization thread we are able to calculate the release dates from one set time, which would be the synchronization thread time. We cannot guarantee that this time will not drift but it will keep the timings between each worker thread the same. This is largely helpful with intertask communication where dependent tasks are need on other tasks to have completed in time so they can read inputs for that task instant.

```
int main(int argc, char* argv[]){

int HPs = 0;

set_scheduler(SCHED_TYPE); //Set ChronOS scheduler type
set_priority(SYNC_PRIORITY); //Set Lower Priority

create_all_threads(THREAD_PRIORITY); //Create MRICDF task threads

while( all_threads_schedulable() ){

        //First Release date in HP
        for_each(task N to be released){
            begin_rtseg_basic(threadN_id, prior, dead, per);
            sem_post(&taskN_sem);
        }
        sleep_until_next_release_date();

            ...

        //Last Release date in HP
        for_each(task N to be released){
            begin_rtseg_basic(threadN_id, prior, dead, per);
            sem_post(&taskN_sem);
        }

        HPs++; //Another HP passed

}

join_all_threads();

return 0;
}
```

Figure 3.13: Pseudocode for Synchronization Thread

Chapter 4

Conclusion

4.1 Results

In order to fully discuss the results of this work there are several facets which must be covered. First we will cover the complexity of the algorithms. This will give a general sense of how each method compares. It will also allow us to discuss what factors affect the number of paths the most. This is important because the number of paths directly correlates to compile times so reducing specific factors such as branches or hyper-period executions can directly increase performance. We will also offer specific tests that were ran as well as timings to show how the implementations of the two methods compare.

While understanding compilation time comparisons between different methods, the main focus of this work is to increase the number of models which can be considered as schedulable. Specifically, we do this through worst-case task set refinement of models. We will discuss how our coverage of such models improves with respect to Prelude.

Finally we return to the case study presented in Chapter 1 of the Location Estimation Unit. We will show how each algorithm computes the worst-case task set and also show the generated code structure of the model.

4.1.1 Discussion on Complexity

The first method presented in Section 3.3, the number of possible executions per node is exponentially increasing. The specific number of paths per node is shown below:

$$\sum_{m=1}^{max}(C(max, m) * (m * n))$$

In this equation max is the maximum number of activations per HP of a task and n is the number of branches. Given m number of activations, there are $C(max, m)$ combinations that these activations can occur and $(m * n)$ number of branch activations that can occur. This number must be determined for all possible m which is shown in the equation. Given the two variables - maximum number of activations and number of branches - for a node, the maximum number of activations is the most significant with respect to execution space increasing. This can be seen in Table 4.1. The process used to collect these timings has 3 task, where the first task contains two branches, each branch going to one task. Leaving the process the same and only changing the HP length produces the Method 1 timings in the table. This clearly shows that the first method will cause unusable compile times as the HP executions of a process increases. Because the HP of a process is the least common multiple of the tasks' periods, the HP of a process will in general increase as more tasks are added to a process. This means that as processes become larger the compilation time will quickly become unwieldy.

In the second method, we aimed to prevent this exponential increase in compile time by reducing the rate of increase due to the max term. The max term was targeted because the hyper-period of a model can become very large creating large numbers of activations per node. On the other hand the number of branches m within a node is bounded by EmCodeSyn at 10 and in no processes did we find a model that came close to this limit.

$$\sum_{m=0}^{max}(C(m+n-1, n-1))$$

In the second method the number of paths from one node is given as $C(m + n - 1, n - 1)$, where m is the number of total activations and n is the number of branches from the node. This can be seen as the number of ways m objects can be partitioned into n possible groups where a single group can receive 0 objects. For the same example process discussed in the previous paragraph, where $m = 2$, the number of possible paths is only n, which is linearly increasing. This is a vast improvement over the first method which must still consider the max term. The improvement in compilation times between the first and second methods are shown in Table 4.1.

The number of branches from one node is bounded, but the number of branches within a process is not. With increasing number of branches in a process, an increase in the size of the front can be expected. As front sizes become larger the performance of the algorithm should deteriorate due to the number of combinations of possible paths to be quite high. To look at this performance penalty, a process was created where the basic node contained one activation and had two branches. In order to increase the number of branches and nodes in the graph these would be connected in series, creating a tree like structure, where the size of the fronts effect on timings could be seen. This is presented in Table 4.2. The both methods perform slightly worse as the number of branches, which is also the maximum number of nodes in front, increases. In order to determine if this performance penalty was significant

Table 4.1: Increasing number of Activations of one Node

Branches = 2		
Activ.	Meth. 1 (ms)	Meth. 2 (ms)
1	46.3	44.7
5	47.3	41.7
10	109.7	49.0
15	957.0	45.0
20	32037.0	53.7

Table 4.2: Increasing Branches of a total Process

Activations per Node = 1		
Branches	Method 1 (ms)	Method 2 (ms)
2	1.0	1.3
4	1.0	2.0
8	1.3	4.3
16	3.3	8.7
32	6.3	24.0
64	17.7	66.7
64	340.3	70.0
64	32673.3	77.3

with respect to the penalty of increase the HP executions per node, the number of activations in each node was increased. It is very clear that the performance of of the algorithm under large numbers of activations affects the compilation time more significantly than the front size.

The second method presents a much better approach with respect to speed of analysis but will not always give as refined of a schedule as the first method which will be discussed next.

4.1.2 Coverage

The main goal of this work is to refine the worst possible execution of a hard real-time process. By doing so, it would allow for developers to be able to implement larger and more complex processes while still being able to guarantee the temporal properties within their model. In Table 4.3 we show a few examples, giving both the overhead in our computations as well as the worst case execution timing, WCET, determined for each model by our two

Table 4.3: Comparison of Worst Case Schedules

	Compile (ms)		WCET (ms)		
Model	M1	M2	M1	M2	Prel.
Coll. Avoid	62.7	56.7	42	42	53
Switch	48.3	41.7	10	10	12
Loc Est.	110.7	64.3	90	90	110
MFG	2637.7	129.0	231	241	251
LCD Drive	160.7	88.7	65	65	79

methods and Prelude. As we have already compared the timings between our two methods, the comparison of the worst case execution can be seen in the table. While the first method tends to have higher compile times it does present a more refined worst case than the second method for some examples. In implementation, the second method is used to create worst case schedule quickly, and if that schedule is not feasible then the first method is used to create a more refined schedule for analysis.

Also, we draw a comparison between our worst case schedule and the schedule given by PRELUDE for the same model. The examples in the table all contain a task with at least one branch which allows for a lower execution time of the worst case schedule, without a branch in a process the schedules would be the same as Prelude. This lower execution time may present a developer with opportunities to take advantage of the extra cycles that can be found when modeling a real-time system using EMCODESYN and MRICDF. Utilizing this time could mean sampling inputs at a more frequent interval, or being able to include more extensive computations to reduce error within the system.

4.2 Conclusion

We proposed extensions to the polychronous formal language MRICDF for the modeling of real-time embedded systems. These extensions allow for the abstraction of real-time systems as a collection of interdependent time driven tasks. Using the language semantics of MRICDF as well as the real-time task abstractions we are were able to show that we could verify more models to be schedulable than previous attempts and methods.

While we extended the MRICDF formalism, we also incorporated many ideas from the PRELUDE language which is used strictly for defined synchronous real-time systems. Using their extended precedence encoding techniques we were able to transform the task graph abstraction into a set of independent tasks which could more easily be analyzed with existing schedulability analysis tools such as Chedddar [25]. We also incorporated some code generation improvements such as buffer optimization techniques for inter-task communication.

Although we incorporated some ideas of the PRELUDE language, we leveraged the formal semantics and compilation analysis of MRICDF to create a tool that was capable of modeling, analyzing, and generating code for a wider range of models.

Bibliography

[1] About chronos linux. URL `http://chronoslinux.org/wiki/About_ChronOS_Linux`.

[2] Douglas Isbell. Mars climate orbiter team finds likely cause of loss, 1999. URL `http://mars.jpl.nasa.gov/msp98/news/mco990930.html`.

[3] J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Universit'e de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.

[4] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720.

[5] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973. ISSN 0004-5411. doi: 10.1145/321738.321743. URL `http://doi.acm.org/10.1145/321738.321743`.

[6] GiorgioC. Buttazzo. Rate monotonic vs. edf: Judgment day. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, volume 2855 of *Lecture Notes in Computer Science*, pages 67–83. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20223-3. doi: 10.1007/978-3-540-45212-6_6. URL `http://dx.doi.org/10.1007/978-3-540-45212-6_6`.

[7] M. Dellinger, P. Garyali, and B. Ravindran. Chronos linux: A best-effort real-time multiprocessor linux kernel. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 474–479, 2011.

[8] Abdoulaye Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 1441909400, 9781441909404.

[9] B.A. Jose, J. Pribble, and S.K. Shukla. Faster software synthesis using actor elimination techniques for polychronous formalism. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 147–156, 2010. doi: 10.1109/ACSD.2010.31.

[10] Julien Ouy Mahesh Nanjundappa, Matthew Kracht and Sandeep K. Shukla. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. 2013.

[11] B.A. Jose and S.K. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 13–18, 2010. doi: 10.1109/ASP-DAC.2010.5419925.

[12] BijoyA. Jose and SandeepK. Shukla. MRICDF: A polychronous model for embedded software synthesis. In Sandeep K. Shukla and Jean-Pierre Talpin, editors, *Synthesis of Embedded Software*, pages 173–199. Springer US, 2010. ISBN 978-1-4419-6399-4.

[13] The yices smt solver - b. dutertre and l. de moura, http://yices.csl.sri.com/.

[14] Tochou Amagbegnon, Loc Besnard, and Paul Le Guernic. Arborescent canonical form of boolean expressions. Technical report, 1994.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. ISSN 0018-9219. doi: 10.1109/5.97300.

[16] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, UK, 1985. Springer-Verlag. ISBN 3-540-15670-4. URL `http://dl.acm.org/citation.cfm?id=646723.702721`.

[17] B.A. Jose, A. Gamatie, J. Ouy, and S.K. Shukla. Smt based false causal loop detection during code synthesis from polychronous specifications. In *Formal Methods and Models for Codesign (MEMOCODE), 2011 9th IEEE/ACM International Conference on*, pages 109–118, 2011. doi: 10.1109/MEMCOD.2011.5970517.

[18] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991. URL `http://www-rocq.inria.fr/syndex/publications/pubs/ecc91/ecc91.pdf`.

[19] *Simulink: User's Guide*. The Mathworks.

[20] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = esterel + kronos - a tool for verifying real-time properties of embedded systems, 2001.

[21] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In *19th International Conference on Real-Time and Network Systems*, Nantes, France, September 2011. Irccyn. URL `http://hal.inria.fr/inria-00618587`.

[22] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 251 –260, dec. 2008. doi: 10.1109/HASE.2008.47.

[23] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Syst.*, 2(3):181–194, September 1990. ISSN 0922-6443. doi: 10.1007/BF00365326. URL `http://dx.doi.org/10.1007/BF00365326`.

[24] Joseph Y.-T. Leung and ML Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3):115–118, 1980.

[25] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar: a flexible real time scheduling framework, 2004.

[26] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 301–310, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4001-6. doi: 10.1109/RTAS.2010.26. URL `http://dx.doi.org/10.1109/RTAS.2010.26`.