Semantics of Graphical Temporal and Logical Patterns

Technical Report for NSTIP Funded Project 13-INF761-10
Track - Software Engineering and Innovative Systems
Version 3.1 - September, 2016

Aamir M. Khan

PhD Embedded Systems

Consultant MODEVES Project

NSTIP, Saudi Arabia.

Acknowledgments

This project is partially funded by NSTIP (National Science Technology and Innovative Plan), Saudi Arabia under the Track "Software Engineering and Innovated Systems" bearing the project code "13-INF761-10". We would like thank the government of Saudi Arabia especially the Ministry of Science, Technology and Scientific Research for providing us an opportunity to work on the cutting-edge technologies and latest research trends in this domain.

We would like to thank Prof. Dr. Frederic Mallet from University of Nice, Sophia-Antipolis, France for his relentless support for the project. Without his ideas and suggestions, it wouldn't have been possible. We would also like to thank the reviewers of this NSTIP funding and the reviewers from various reputed conferences (like TASE, SIES) and SoSym journal for their valuable comments and suggestions. All their critical remarks helped us evolve the work to its present form.

Contents

List of Figures

List of Tables

Chapter 1

Proposed Framework

The combined features of UML and MARTE provide a working ground for developing timed-models but they still lack the specialized features to model graphical temporal and logical patterns. This section serves as the first major contribution of this research work by presenting a framework that provides set of reusable generic graphical patterns. Design flow of the proposed framework is shown in Figure 1.1. It clearly distinguishes three major design flow phases: requirements engineering, systems modeling, and system verification and validation. *Requirements engineering phase* is the one when design engineers collect system structural, functional and non-functional requirements using tradition software engineering techniques. In the *systems modeling phase*, these requirements are then used to manually build the UML model using the custom-defined Observation profile. These annotated UML models are automatically transformed into equivalent but semantically rich MARTE models. In the *verification and validation phase*, these semantic MARTE models are used to generate the CCSL constraints and Verilog observers code. There are numerous advantages to this generated information. The generated Verilog observers can be used along with system-under-verification implementation to perform simulation and verification in tools like QuestaSim [1]. On the other side, the generated CCSL code can be used in TimeSquare tool [2] for simulation, early design validation, and verification analyses. Moreover, these CCSL constraints can also be used to generate the VHDL-based observers code [3].

Here concerning the core part of the framework, two types of behavioral constraints are extracted from the system functional and non-functional requirements: state-state relations and state-event relations. The focus of the presented work is not the minimalist approach but rather the expressiveness of the property from the system designer's point-of-view. So when a property specifies occurrence of something at some point in time, then it is an event and it seems natural to represent such properties as logical clocks. But when the properties specify duration or interval (not a particular point in time), then the obvious choice of representation is state relations.

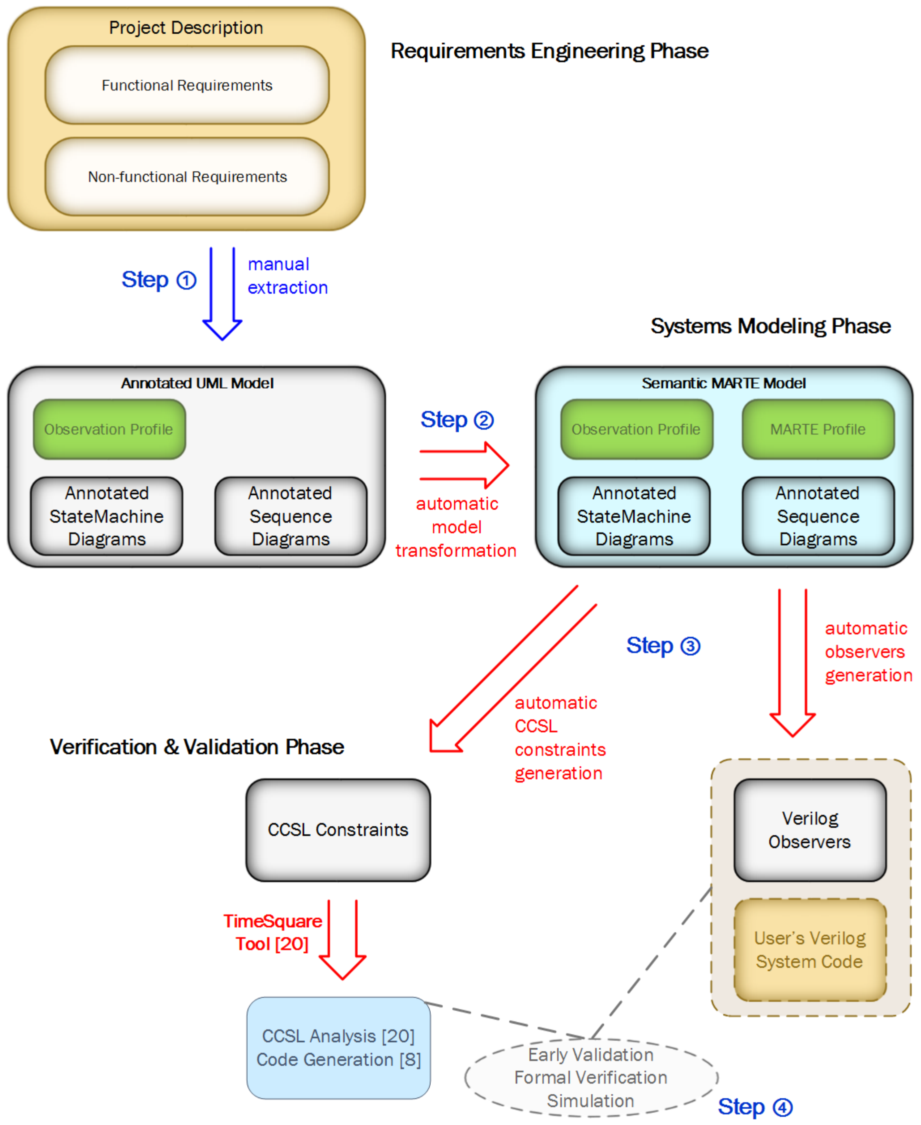


Figure 1.1: Overall Framework Design Flow

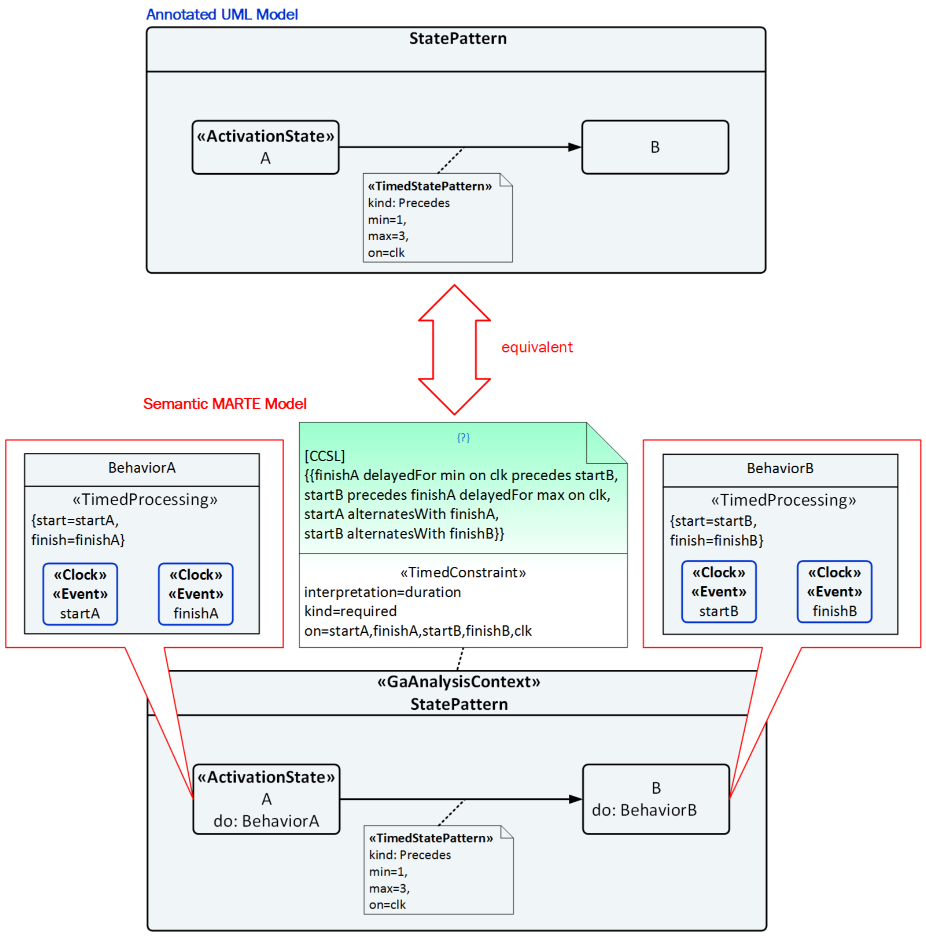


Figure 1.2: Sample State-State Timed Pattern

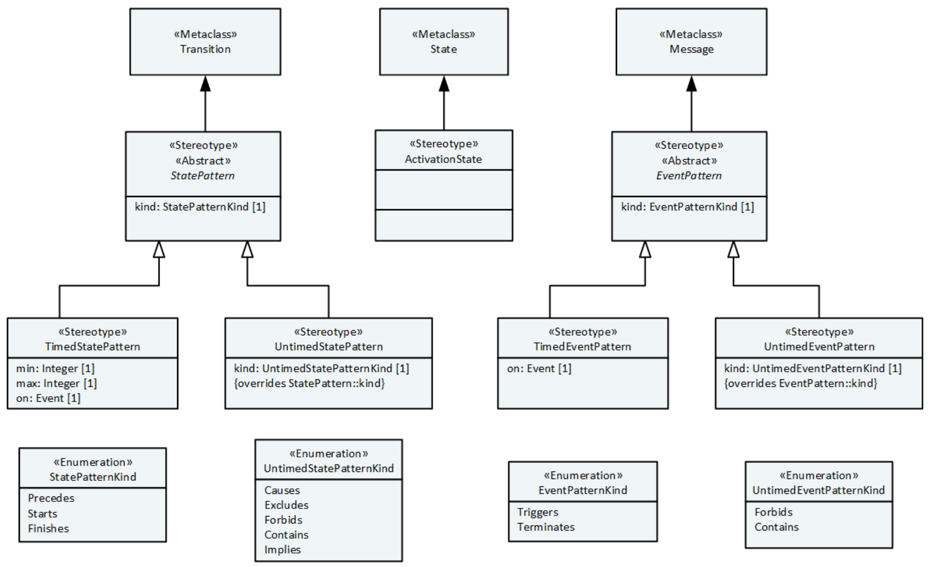

«Metaclass» Transition
«Metaclass» State
«Metaclass» Message
«Stereotype» «Abstract» StatePattern
kind: StatePatternKind [1]
«Stereotype» ActivationState
«Stereotype» «Abstract» EventPattern
kind: EventPatternKind [1]
«Stereotype» TimedStatePattern
min: Integer [1]
max: Integer [1]
on: Event [1]
«Stereotype» UntimedStatePattern
kind: UntimedStatePatternKind [1] {overrides StatePattern::kind}
«Stereotype» TimedEventPattern
on: Event [1]
«Stereotype» UntimedEventPattern
kind: UntimedEventPatternKind [1] {overrides EventPattern::kind}
«Enumeration» StatePatternKind
Precedes
Starts
Finishes
«Enumeration» UntimedStatePatternKind
Causes
Excludes
Forbids
Contains
Implies
«Enumeration» EventPatternKind
Triggers
Terminates
«Enumeration» UntimedEventPatternKind
Forbids
Contains

Figure 1.3: Proposed Observation Profile

1.1 State-State Relations

1.1.1 Representation

Extended state machine diagrams are mainly used to provide a more natural and syntactically sound interpretation of graphical behavioral patterns of system states. Figure 1.2 shows the two representations of a sample state-state temporal pattern of precedence relation (*A precedes B by [1,3] on clk*). It is achieved by altering the semantics of states and transitions from their traditional meanings. These features are beyond the existing capabilities of both UML and MARTE and for this a custom profile (named *Observation* profile) is introduced, as shown in Figure 1.3. It is designed to target the expressiveness of graphical properties and resides on top of the UML to provide a structure for building some predefined patterns. This profile presents *ActivationState* stereotype, extending the state metaclass, as a means to identify the antecedent state in relations. It is used to identify the state that activates this particular temporal/logical pattern. It is active whenever the system is in a specified condition (like in relation *A precedes B*, A is the activation state). Stereotypes *TimedStatePattern* and *UntimedStatePattern* extend the transition metaclass. It is used to specify that the given state transition represents a predefined pattern. A number of predefined state-state relation patterns are identified by the *kind* attribute of these stereotypes. The timed graphical patterns have additional attributes (min, max, on) to specify time delay on some base clock.

1.1.2 Observation Profile

The semantically rich graphical pattern (lower part of Figure 1.2) uses the MARTE profile along with Observation profile. As the state machine is used as an analysis formalism, it utilizes the existing analysis context stereotype (GaAnalysisContext) from the *Generic Quantitative Analysis Modeling* (GQAM) package of MARTE_AnalysisModel. State machine states (like A,B) are behavioral entities which can be associated with an inner behavior (*OpaqueBehavior*) using the *Do Activity* attribute. The framework proposes to use the MARTE *TimedProcessing* stereotype from the *Time* package of the MARTE_Foundations. This MARTE specialization facilitates to provide the start and finish events for the state (like A_start and A_finish events). Further, a UML constraint is introduced to the state machine specialized by MARTE *TimedConstraint* stereotype, from the *Time* package of the MARTE_Foundations. It provides semantics to the pattern in the form of CCSL specification with the use of *TimedConstraint* that provides the set of logical clocks used for semantic modeling. In short, the user specifies a pattern looking simple and easy-to-understand (upper part of Figure 1.2), that is transformed into a model packed with all the annotations and technicalities required to make it semantically correct and at the same time being able to generate accurate CCSL or Verilog observers.

1.2 State-Event Relations

1.2.1 Representation

Extended sequence diagrams are used to provide the interpretation of graphical patterns relating system events and states. Figure 1.4 shows the two representations of a sample state-event logical pattern of forbiddance relation (*A forbids B*). In a typical state-event relation, the event (like e) is represented by the asynchronous message, while the *Behavior Execution Specification* represents the state behavior (like state A). The pattern relation is specified by extending the semantics of interaction messages. The Observation profile contains the stereotypes *TimedEventPattern* and *UntimedEventPattern* that extend the message metaclass. It specifies that the asynchronous message passing between lifelines represents a predefined behavior identified by the *kind* attribute of these stereotypes. For timed patterns base clock event is also specified in the *TimedEventPattern* stereotype while the time duration is specified graphically using UML *Duration Constraint* between the occurrence specification of message reception and start execution occurrence specification of the behavior.

The semantically rich graphical pattern (lower part of Figure 1.4) uses the MARTE profile along with Observation profile. Just like the state machines, the *GaAnalysisContext* stereotype from the MARTE_AnalysisModel is used for the sequence diagram. As the state is associated behavior execution specification, its *Behavior attribute* specifies the inner state behavior (*OpaqueBehavior*) just as we did earlier for state machines. The state opaque behavior is stereotyped by MARTE *TimedProcessing* from the *Time* package of the MARTE_Foundations.

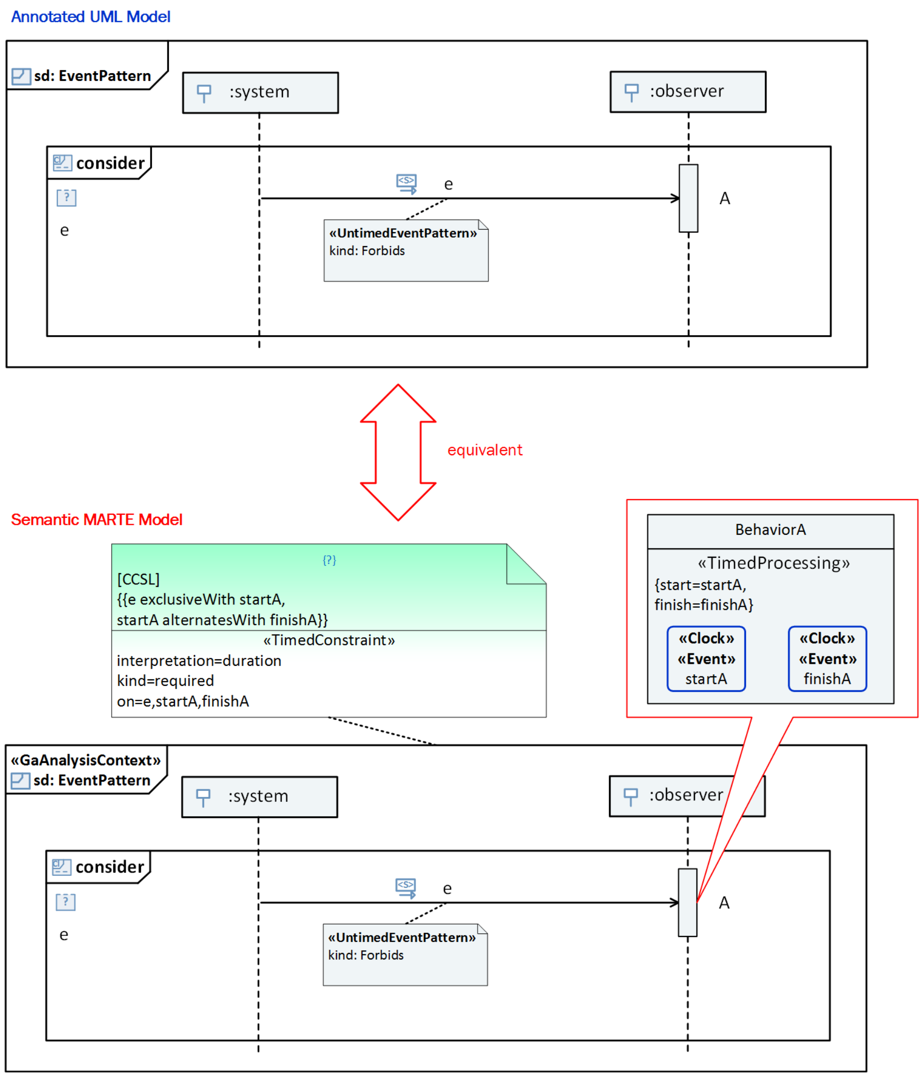


Figure 1.4: Sample State-Event Untimed Pattern

This MARTE specialization facilitates to provide the start and finish logical events for the state (like A_start and A_finish events). Moreover, a constraint is created having the context of sequence diagram to specify the CCSL semantics of the given graphical pattern. Further, it is specialized by MARTE *TimedConstraint* stereotype, from the MARTE_Foundations, to provide all the logical clocks used in the CCSL specification.

1.3 Framework Design-flow

From the end-user perspective different steps involved in using the framework, from high-level requirements to the point where verification results are achieved, are listed below. These steps are also marked on the Figure 1.1.

1. **Annotated UML Modeling** - The first step is the manual modeling of the system properties as plain annotated UML models from the requirements specification. For this purpose the state machine and sequence diagrams are used for the state-state and state-event properties respectively, annotated with Observation profile features. Requirements can be initially specified in textual form using specialized formalisms like SysML requirements diagrams which are then satisfied by graphical patterns.

2. **Semantically Annotated MARTE Modeling** - In the second step, these generic UML models are semantically annotated with MARTE formalisms from its time and GQAM sub-profiles. This step provides semantically sound models with the desired capabilities to model temporal/logical patterns. There is semantics application of these diagrams by producing UML/MARTE/CCSL information. So the textual requirements are converted into semantically sound patterns that can then further be used for verification and validation purposes.

3. **CCSL Constraints and Verilog Observers Generation** - Once the desired graphical patterns are complete and semantically sound, they can be used to generate CCSL constraints and Verilog Observers from our TemLoPAC plugin [4]. CCSL constraints can further be used to automatically generate VHDL code [3] using the TimeSquare tool [2].

4. **Simulation and Verification** - Generated CCSL code can be simulated directly in TimeSquare for an early design validation. Moreover, the generated Verilog observers can be simulated with the actual system HDL code (in softwares like QuestaSim [1]) for design verification. This system HDL code is usually available to the design engineers during the requirements engineering phase.

Chapter 2

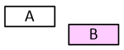
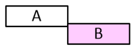
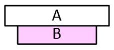
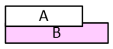
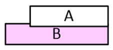
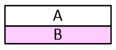
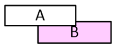
State-State Graphical Patterns

The subsequent chapters narrate the second and third major contribution of the presented framework. For identifying temporal/logical properties of systems, we started by considering several examples like the famous stream boiler case study [5, 6], railway interlocking system [7] and the traffic light controller case study [8]. Working on these diverse examples to model behavioral properties in UML, we noted that several patterns were repeated across different examples. These patterns collected across various examples were refined with some inspiration taken from the Dwyer's patterns [9] and Allen's algebra targeting intervals [10]. This practice gave us a valuable collection of generic patterns divided into three major categories of behavioral relations that may exist in a system: state-state relations, state-event relations, and event-event relations. Researchers have already shown that constraints specified in CCSL are capable of modeling logical event-event relations [11, 12]. These CCSL constraints can be represented graphically using SysML/MARTE models [13, 14, 15].

Presented framework can be used to formally model relations between two different states of a system. Almost two decades ago, Dwyer [9] identified two broader categories of patterns: occurrence patterns (absence, existence, universality) and ordering patterns (precedence and response). Moreover, Allen's algebra [10] for intervals provides a base defining thirteen distinct, exhaustive and qualitative relations of two time intervals, depicted in Table 2.1. From the comparative analysis of these relations, six primitive relations are extracted that can be applied to the state-based systems. Further two 'negation' relations are added, based on their usage and importance in the examples, to complete the set. The overlapping of states is not particularly interesting relation to dedicate a pattern for. But it can easily be modeled indirectly using the state-event relation 'A contains b_s' where b_s is the start event of state B.

Semantically a state can be considered similar to an interval. We use the nomenclature of using capital letters (A,B,...) to denote states and small let-

Table 2.1: Allen's Algebra and Proposed Relations

No.	**Graphic**	**Allen's Algebra for Intervals**	**State-State Relation**	**Symbol**
1		A precedes B	A precedes B	$\boxed{\preccurlyeq}$
2		B preceded-by A		
3		A meets B	A causes B	$\boxed{\models}$
4		B met-by A		
5		A contains B	A contains B	$\boxed{\supseteq}$
6		B during A		
7		A starts B	A starts B	$\boxed{\vdash}$
8		B started-by A		
9		A finishes B	A finishes B	$\boxed{\dashv}$
10		B finished-by A		
11		A equals B	A implies B	$\boxed{\Rightarrow}$
12		A overlaps B	See the text	
13		B overlapped-by A		
14			A forbids B	$\boxed{\neg}$
15			A excludes B	$\boxed{\#}$

ters (a,b,...) for events/clocks. Notations like a_s and a_f are used throughout the text to represent the state start and end events. Note that these events are introduced as part of the framework methodology and the system under verification only provides us relevant states. Formally defining, given a strict partial ordering $\mathbb{S} = \langle S, < \rangle$, a state in $\mathbb{S}$ is a pair $[a_s, a_f]$ such that $a_s, a_f \in$ S and $a_s < a_f$ where a_s is the start and a_f is the end of the state interval. An event or point e belongs to a state interval $[a_s, a_f]$ if $a_s \leqslant e \leqslant a_f$ (both ends included).

In this research work, we identify two types of state-state relations: Temporal and logical. The sections next discuss these patterns in detail one-by-one.

2.1 Temporal/Timed State-State Patterns

Temporal patterns are the ones associated with notion of time or clock events. State-state temporal patterns identified in this work are precedes, starts, and finishes.

2.1.1 Timed Precedes Relation

Precedence is an important state property where the relation *'A precedes B'* means the state A comes before the state B. It includes a delay/deadline clause to explicitly specify the duration between the termination of state A and the start of state B. The unit of the duration is dependent on the level of abstraction being target of the graphical specification, that is, it can be physical clock, loosely timed clock, or logical clock. Deadline defers the evaluation until some number of ticks of clk (or any other event) occur. The number of ticks of clk considered are dependent on the two parameter natural numbers min and max evaluated as:

- [0, n] means 'before n' ticks of clk
- [m, 0] means 'after m' ticks of clk
- [m, m] means 'exactly m' ticks of clk

Mathematical Notation

Mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), constants m, n and a clock clk, the equation

$$A \boxed{\preccurlyeq} B \quad by\ [m, n]\ on\ clk$$

means $a_f \preccurlyeq b_s$ and b_s occurs within the duration $a_f + \Delta$, where Δ is between m and n ticks of event clk. The last tick of clk coincides with the start the state B (i.e, b_s), as shown in Figure 2.1.

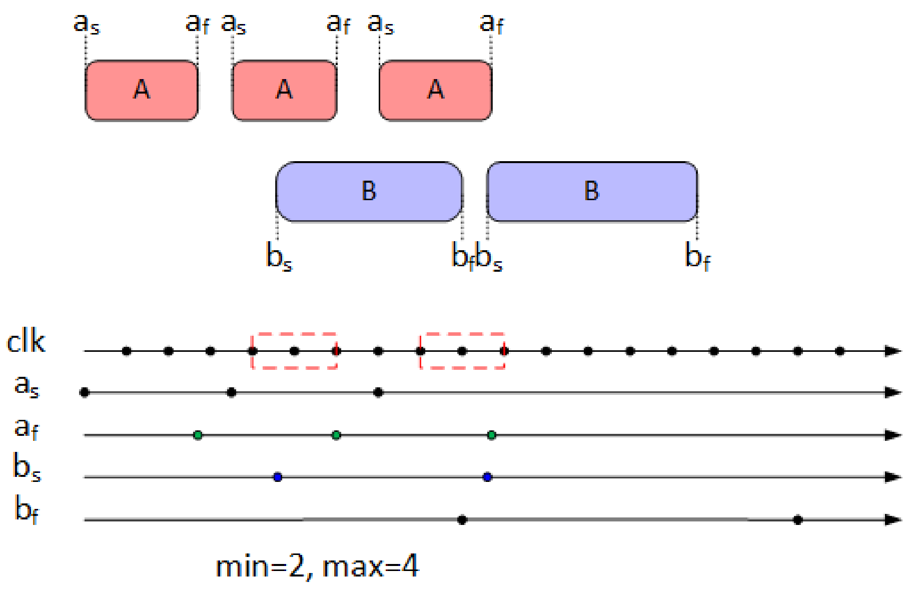


Figure 2.1: Depiction of A precedes B by [2,4] on clk

Graphical Representation

Graphically, precedence is shown in Figure 2.2 as the annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. If we need to specify duration, *TimedStatePattern* stereotype is applied. Time interval is specified using min and max attributes along with the base clock for the delay (using 'on'). Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

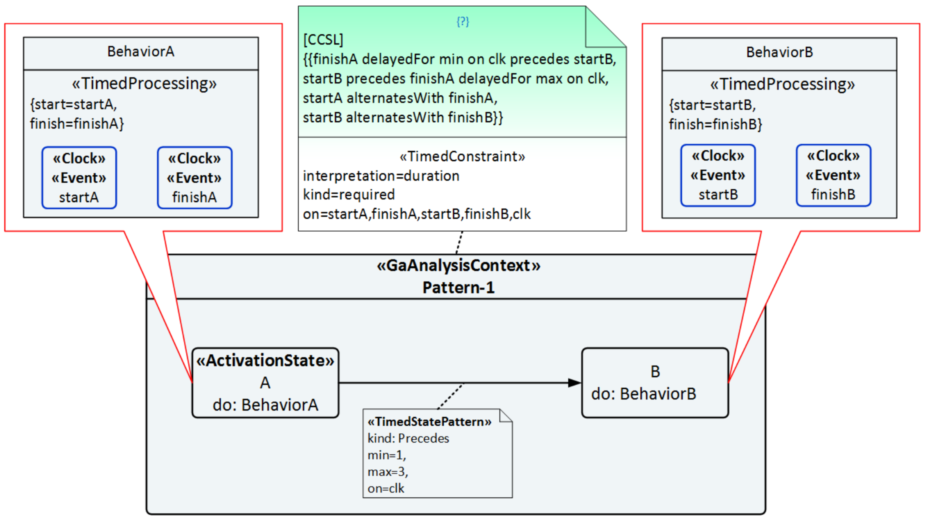


Figure 2.2: MARTE Model of A precedes B by [1,3] on clk

Representation in CCSL

$$af(min) \rightsquigarrow clk \boxed{\preccurlyeq} bs \boxed{\preccurlyeq} af(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

af delayedFor min on clk precedes bs
bs precedes af delayedFor max on clk
as alternatesWith af
bs alternatesWith bf

State Automaton

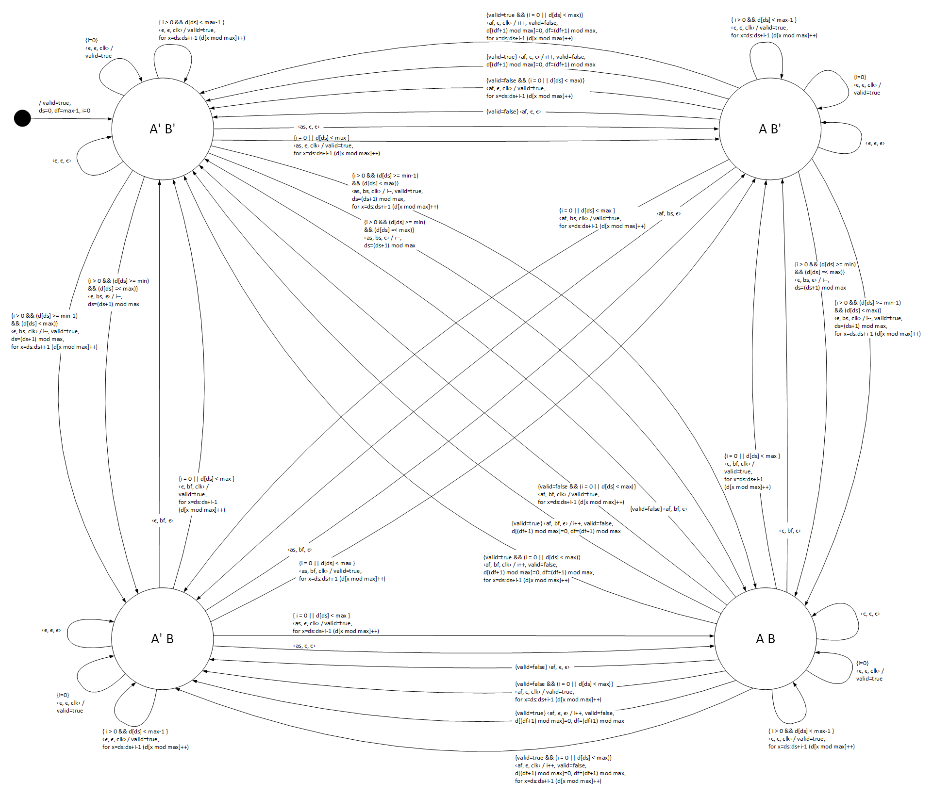


Figure 2.3: Automaton of A precedes B by [m,n] on clk

SystemVerilog Observer

```
// A precedes B after [m,n] on clk

module Precedes (
 input as,
 input af,
 input bs,
 input bf,
 input clk,
 output violation
 );
 parameter min=2,max=4;

 reg valid;
 int unsigned d[max-1];
int unsigned ds;
int unsigned df;
int unsigned i;
int unsigned FSM;
reg v;

always @ (as or af or bs or bf or clk)
begin
 case(FSM)
  // ------------------------------------------------------------
  0: // State A'B'
   if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
    FSM=0;   // empty
   else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
   begin          // clk
    if(i==0) begin FSM=0;valid=1'b1; end
    else if(i>0 && d[ds]<max-1)
    begin
     FSM=0;valid=1'b1;
     for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
    end
    else
    begin
     FSM=4;v=1'b1;
    end
   end
   else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
    FSM=2; // as
   else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
   begin          // as, clk
    if(i==0 || d[ds]<max)
    begin
     valid=1'b1;FSM=2;
     for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
    end
   end
   else if(as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b0)
   begin    // bs
    if(i>0 && d[ds]>=min && d[ds]<=max)
    begin
     i--; FSM=1;
     ds=(ds+1) % max;
    end
    else
    begin
     FSM=4;v=1'b1;
   end
  end
  else if(as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b1)
  begin    // bs,clk
   if(i>0 && d[ds]>=min-1 && d[ds]<max)
   begin
    i--;FSM=1;valid=1'b1;
    ds=(ds+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=4;v=1'b1;
   end
  end
  else if(as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b0)
  begin    // as,bs
   if(i>0 && d[ds]>=min && d[ds]<=max)
   begin
    i--; FSM=3;
    ds=(ds+1) % max;
   end
   else
   begin
    FSM=4;v=1'b1;
   end
  end
  else if(as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b1)
  begin        // as,bs,clk
   if(i>0 && d[ds]>=min-1 && d[ds]<max)
   begin
    i--;FSM=3;valid=1'b1;
    ds=(ds+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=4;v=1'b1;
   end
  end
  else
  begin
   FSM=4;v=1'b1;
  end
//-----------------------------------------------------------------
1: // State A'B
```

```
if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
 FSM=1;   // empty
else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
begin     // clk
 if(i==0) begin FSM=1;valid=1'b1; end
 else if(i>0 && d[ds]<max-1)
 begin
  FSM=1;valid=1'b1;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
 end
 else
 begin
  FSM=4;v=1'b1;
 end
end
else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
 FSM=3; // as
else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
begin    // as,clk
 if(i==0 || d[ds]<max)
 begin
   valid=1'b1;FSM=3;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
 end
end
else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b0)
 FSM=0; // bf
else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b1)
begin     // bf,clk
 if(i==0 || d[ds]<max)
 begin
  FSM=0;valid=1'b1;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
 end
 else
 begin
  FSM=4;v=1'b1;
 end
end
else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b0)
 FSM=2; // as,bf
else if(as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b1)
begin          // as,bf,clk
 if(i==0 || d[ds]<max)
 begin
  FSM=2;valid=1'b1;
```

```
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=4;v=1'b1;
   end
  end
  else
  begin
   FSM=4;v=1'b1;
  end
//-----------------------------------------------------------------------
2: // State AB'
  if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
   FSM=2; // empty
  else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
  begin   // clk
   if(i==0) begin FSM=2;valid=1'b1; end
   else if(i>0 && d[ds]<max-1)
   begin
    FSM=2;valid=1'b1;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=4;v=1'b1;
   end
  end
  else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0 && clk==1'b0)
  begin  // af
   FSM=0;
   if(valid==1'b1)
   begin
    i++;valid=1'b0;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
  end
  else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0 && clk==1'b1)
  begin  // af, clk
   if(valid==1'b0 && i==0 || d[ds]<max)
   begin
    valid=1'b1;FSM=0;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else if(valid==1'b1 && i==0 || d[ds]<max)
   begin
```

```
    i++;valid=1'b0;FSM=0;
    d[(df+1) % max]=0;
    df=(df+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
end
else if(as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b0)
begin   // bs
  if(i>0 && d[ds]>=min && d[ds]<=max)
  begin
    i--; FSM=3;
    ds=(ds+1) % max;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
end
else if(as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0 && clk==1'b1)
begin // bs, clk
  if(i>0 && d[ds]>=min-1 && d[ds]<max)
  begin
    i--;valid=1'b1;FSM=3;
    ds=(ds+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
end
else if(as==1'b0 && af==1'b1 && bs==1'b1 && bf==1'b0 && clk==1'b0)
 FSM=1;   // af, bs
else if(as==1'b0 && af==1'b1 && bs==1'b1 && bf==1'b0 && clk==1'b1)
begin          // af, bs, clk
  if(i==0 || d[ds]<max)
  begin
    valid=1'b1;FSM=1;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
```

```
   FSM=4;v=1'b1;
  end
 end
 else
 begin
  FSM=4;v=1'b1;
 end
// ------------------------------------------------------------
3: // State AB
 if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b0)
  FSM=3;   // empty
 else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0 && clk==1'b1)
 begin     // clk
  if(i==0) begin FSM=3;valid=1'b1; end
  else if(i>0 && d[ds]<max-1)
  begin
   FSM=3;valid=1'b1;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
 end
 else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0 && clk==1'b0)
 begin    // af
  FSM=1;
  if(valid==1'b1)
  begin
   i++;valid=1'b0;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
 end
 else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0 && clk==1'b1)
 begin    // af, clk
  if(valid==1'b0 && i==0 || d[ds]<max)
  begin
   valid=1'b1;FSM=1;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else if(valid==1'b1 && i==0 || d[ds]<max)
  begin
   i++;valid=1'b0;FSM=1;
   d[(df+1) % max]=0;
   df=(df+1) % max;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
end
else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b0)
 FSM=2;   // bf
else if(as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1 && clk==1'b1)
begin          // bf, clk
  if(i==0 || d[ds]<max)
  begin
   FSM=2;valid=1'b1;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
  end
end
else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1 && clk==1'b0)
begin          // af, bf
 FSM=0;
  if(valid==1'b1)
  begin
   i++;valid=1'b0;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
end
else if(as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1 && clk==1'b1)
begin          // af, bf, clk
  if(valid==1'b0 && i==0 || d[ds]<max)
  begin
   valid=1'b1;FSM=0;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else if(valid==1'b1 && i==0 || d[ds]<max)
  begin
   i++;valid=1'b0;FSM=0;
   d[(df+1) % max]=0;
   df=(df+1) % max;
   for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=4;v=1'b1;
```

```
        end
      end
      else
      begin
        FSM=4;v=1'b1;
      end
    endcase
  end

  assign violation = v;

  initial
  begin
    valid=1'b1;
    ds=0;
    df=max-1;
    d[max-1]=0;
    i=0;
    FSM = 0;
    v = 0;
  end
endmodule
```

2.1.2 Starts Relation

Mathematical Notation

The relation *'A starts B'* means the state B starts with the state A. It includes a delay/deadline clause to explicitly specify a delayed start of state B. This delay specifies the duration between the start of state A and the start of state B. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), a constant n and a clock clk, the equation

$$A \boxed{\vdash} B \quad after\ [m, n]\ on\ clk$$

means b_s occurs within the duration $a_s + \Delta$, where Δ is between m and n ticks of event *clk*, as shown in Figure 2.4.

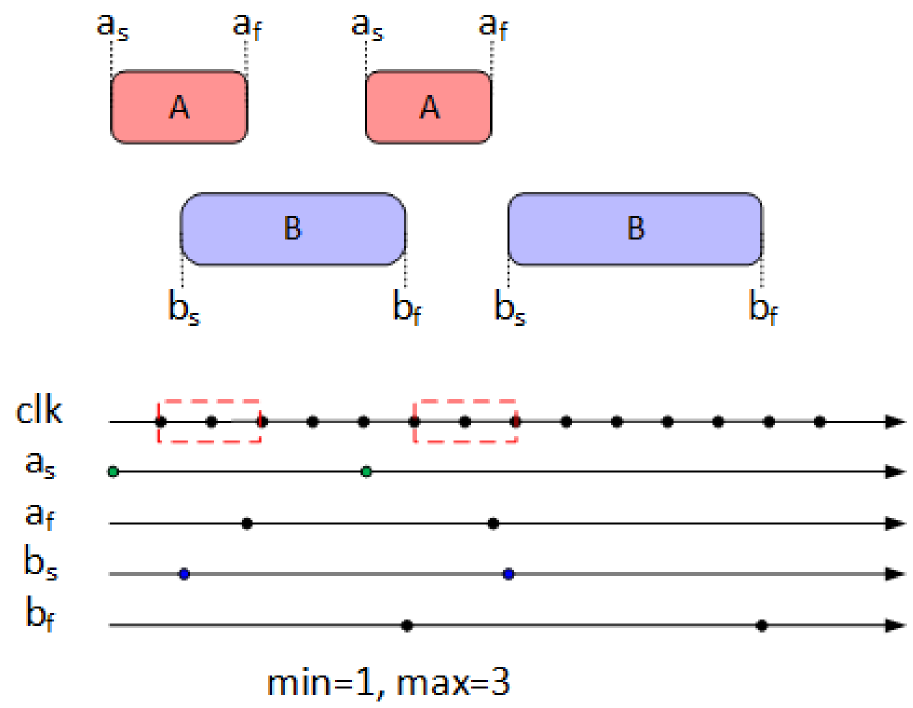


Figure 2.4: Depiction of A starts B after [1,3] on clk

Graphical Representation

Graphically, starts is shown as in Figure 2.5 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. If we need to specify duration, *TimedStatePattern* stereotype is applied. Time interval is specified using min and max attributes along with the base clock for the delay (using 'on'). Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

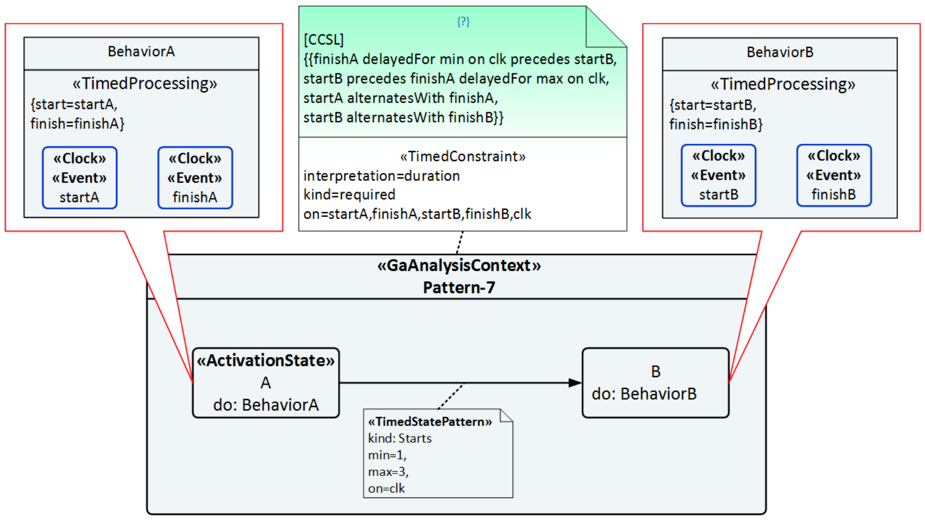


Figure 2.5: MARTE Model of A starts B after [1,3] on clk

Representation in CCSL

$$as(min) \rightsquigarrow clk \boxed{\preccurlyeq} bs \boxed{\preccurlyeq} as(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

as delayedFor min on clk precedes bs
bs precedes as delayedFor max on clk
as alternatesWith af
bs alternatesWith bf

State Automaton

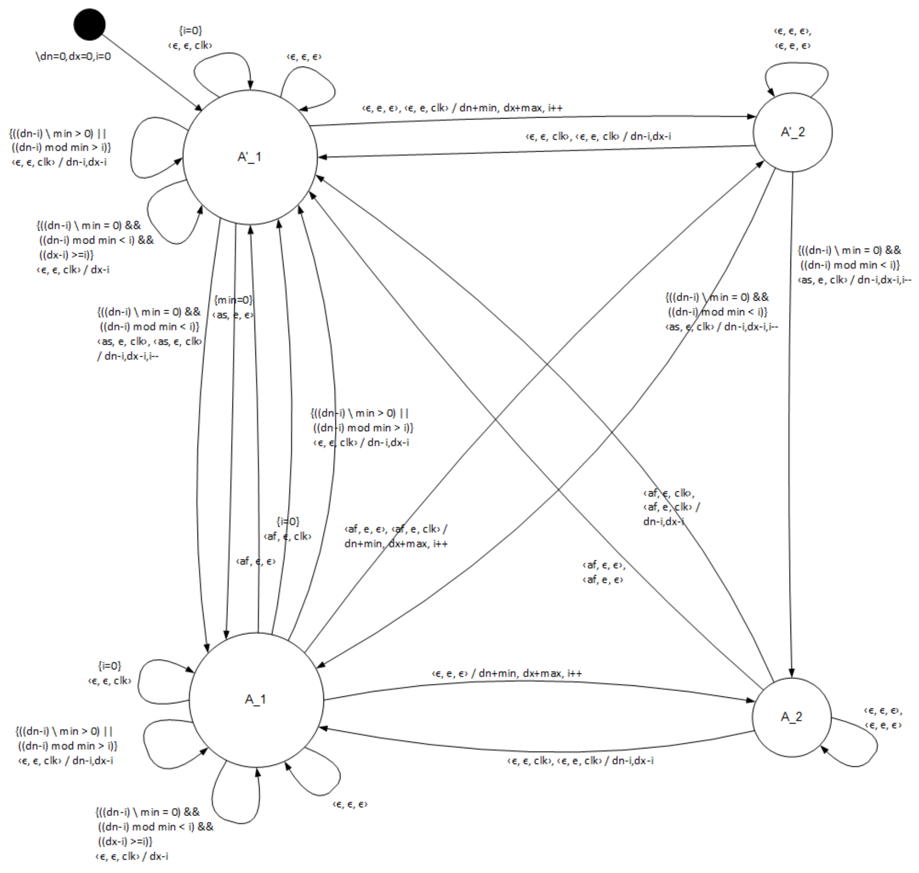


Figure 2.6: Automaton of A starts B after [m,n] on clk

SystemVerilog Observer

Starts timed pattern is similar to precedes timed pattern in implementation. To save the space, observer code is not shown here.

2.1.3 Finishes Relation

Mathematical Notation

The relation *'A finishes B'* means the state B finishes with the state A. It includes a delay/deadline clause to explicitly specify a delayed termination of state B. This delay specifies the duration between the termination of state A and the termination of state B. Mathematically, given a strict partial ordering

$\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), a constant n and a clock clk, the equation

$$A \boxed{\dashv} B \quad after\ [m, n]\ on\ clk$$

means b_f occurs within the duration $a_f + \Delta$, where Δ is between m and n ticks of event *clk*, as shown in Figure 2.7.

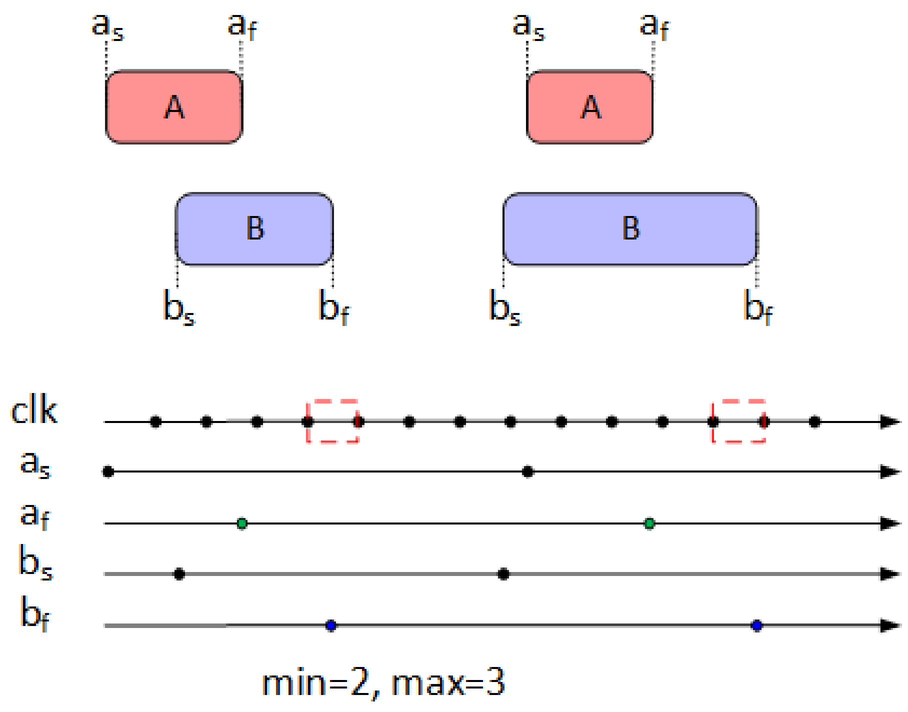


Figure 2.7: Depiction of A finishes B after [2,3] on clk

Graphical Representation

Graphically, finishes is shown as in Figure 2.8 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. If we need to specify duration, *TimedStatePattern* stereotype is applied. Time interval is specified using min and max attributes along with the base clock for the delay (using 'on'). Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

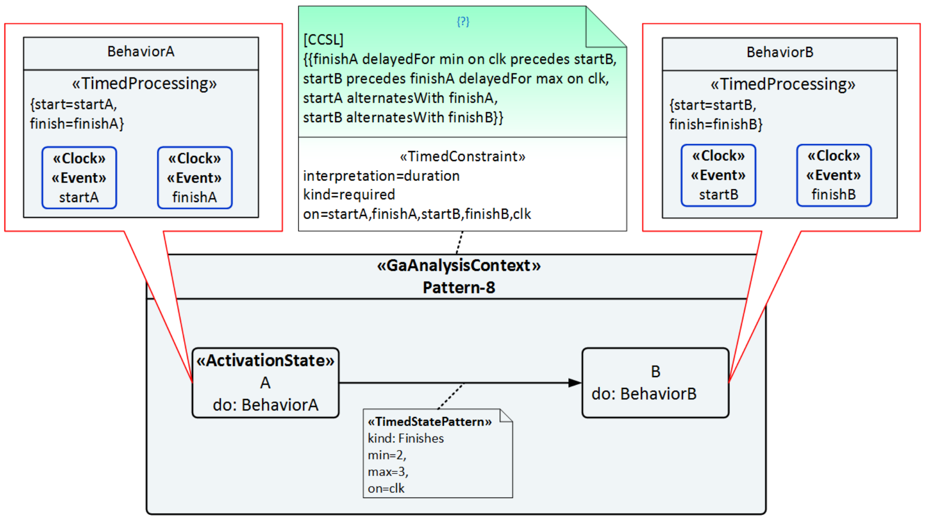


Figure 2.8: MARTE Model of A finishes B after [2,3] on clk

Representation in CCSL

$$af(min) \rightsquigarrow clk \boxed{\preccurlyeq} bf \boxed{\preccurlyeq} af(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

af delayedFor min on clk precedes bf
bf precedes af delayedFor max on clk
as alternatesWith af
bs alternatesWith bf

State Automaton

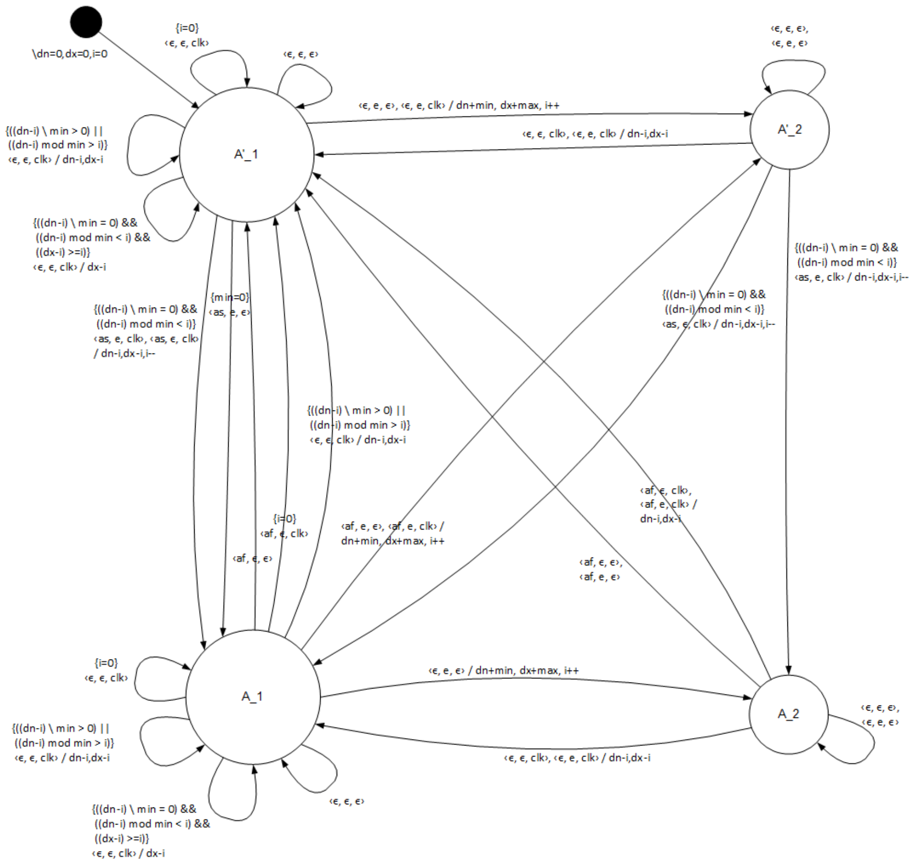

\dn=0,dx=0,i=0
{i=0}
‹ϵ, ϵ, clk›
‹ϵ, ϵ, ϵ›
‹ϵ, ϵ, ϵ›, ‹ϵ, ϵ, clk› / dn+min, dx+max, i++
‹ϵ, ϵ, ϵ›,
‹ϵ, ϵ, ϵ›
A'_1
A'_2
‹ϵ, ϵ, clk›, ‹ϵ, ϵ, clk› / dn-i,dx-i
{((dn-i) \ min > 0) || ((dn-i) mod min > i)}
‹ϵ, ϵ, clk› / dn-i,dx-i
{((dn-i) \ min = 0) && ((dn-i) mod min < i) && ((dx-i) >=i)}
‹ϵ, ϵ, clk› / dx-i
{((dn-i) \ min = 0) && ((dn-i) mod min < i)}
‹as, ϵ, clk› / dn-i,dx-i,i--
{min=0}
‹as, ϵ, ϵ›
{((dn-i) \ min = 0) && ((dn-i) mod min < i)}
‹as, ϵ, clk›, ‹as, ϵ, clk› / dn-i,dx-i,i--
{((dn-i) \ min = 0) && ((dn-i) mod min < i)}
‹as, ϵ, clk› / dn-i,dx-i,i--
{((dn-i) \ min > 0) || ((dn-i) mod min > i)}
‹ϵ, ϵ, clk› / dn-i,dx-i
{i=0}
‹af, ϵ, clk›
‹af, ϵ, ϵ›
‹af, ϵ, ϵ›, ‹af, ϵ, clk› / dn+min, dx+max, i++
‹af, ϵ, clk›, ‹af, ϵ, clk› / dn-i,dx-i
‹af, ϵ, ϵ›, ‹af, ϵ, ϵ›
{i=0}
‹ϵ, ϵ, clk›
A_1
A_2
‹ϵ, ϵ, ϵ› / dn+min, dx+max, i++
‹ϵ, ϵ, ϵ›, ‹ϵ, ϵ, ϵ›
{((dn-i) \ min > 0) || ((dn-i) mod min > i)}
‹ϵ, ϵ, clk› / dn-i,dx-i
‹ϵ, ϵ, clk›, ‹ϵ, ϵ, clk› / dn-i,dx-i
‹ϵ, ϵ, ϵ›
{((dn-i) \ min = 0) && ((dn-i) mod min < i) && ((dx-i) >=i)}
‹ϵ, ϵ, clk› / dx-i

Figure 2.9: Automaton of A finishes B after [min,max] on clk

SystemVerilog Observer

Finishes timed pattern is similar to precedes timed pattern in implementation. To save the space, observer code is not shown here.

2.2 Logical/Untimed State-State Patterns

Logical patterns are untimed and relate two states directly. All the temporal patterns mentioned earlier can also be logical in nature. Logical patterns explained here are precedes, starts, finishes, causes, contains, implies, forbids, and excludes.

2.2.1 Precedes Relation

Mathematical Notation

The relation *'A precedes B'* means the state B occurs after the state A terminates. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\prec} B$ means $a_f \leqslant b_s$ and implies that whenever state A occurs, state B follows afterwards, as shown in Figure 2.10.

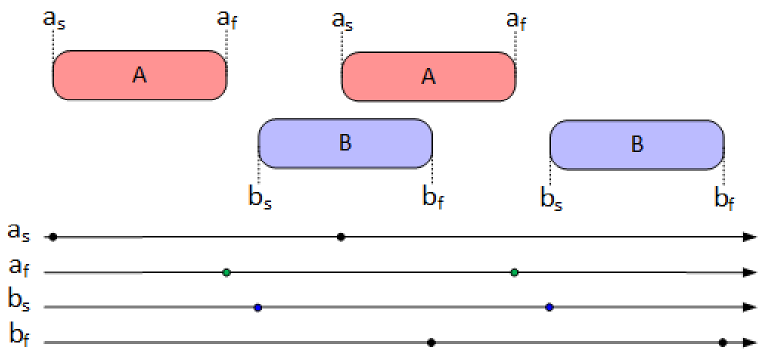


Figure 2.10: Depiction of A precedes B

Graphical Representation

Graphically, contains relation is shown in Figure 2.11 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

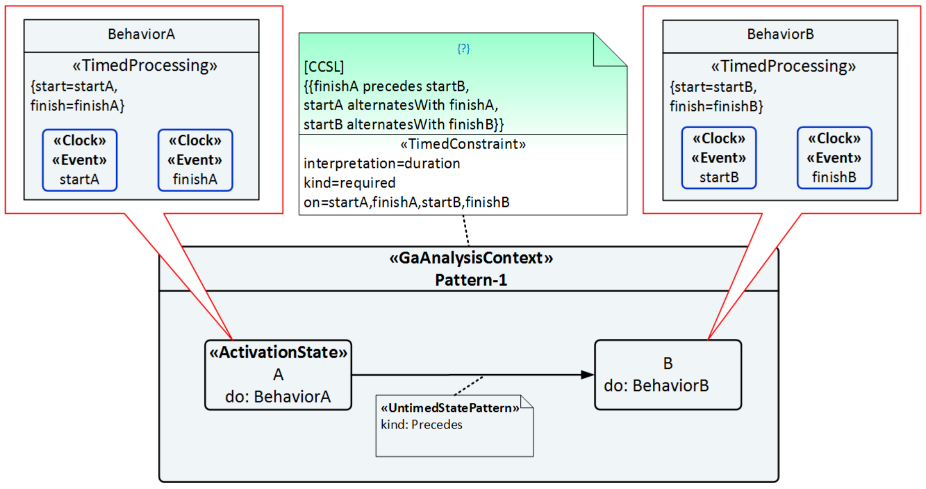


Figure 2.11: MARTE Model of A causes B

Representation in CCSL

$$
\begin{aligned}
af \; &\boxed{<} \; bs, \\
as \; &\boxed{\sim} \; af, \\
bs \; &\boxed{\sim} \; bf
\end{aligned}
$$

Representation in CCSL (alternate form)

af precedes bs,
as alternatesWith af,
bs alternatesWith bf

State Automaton

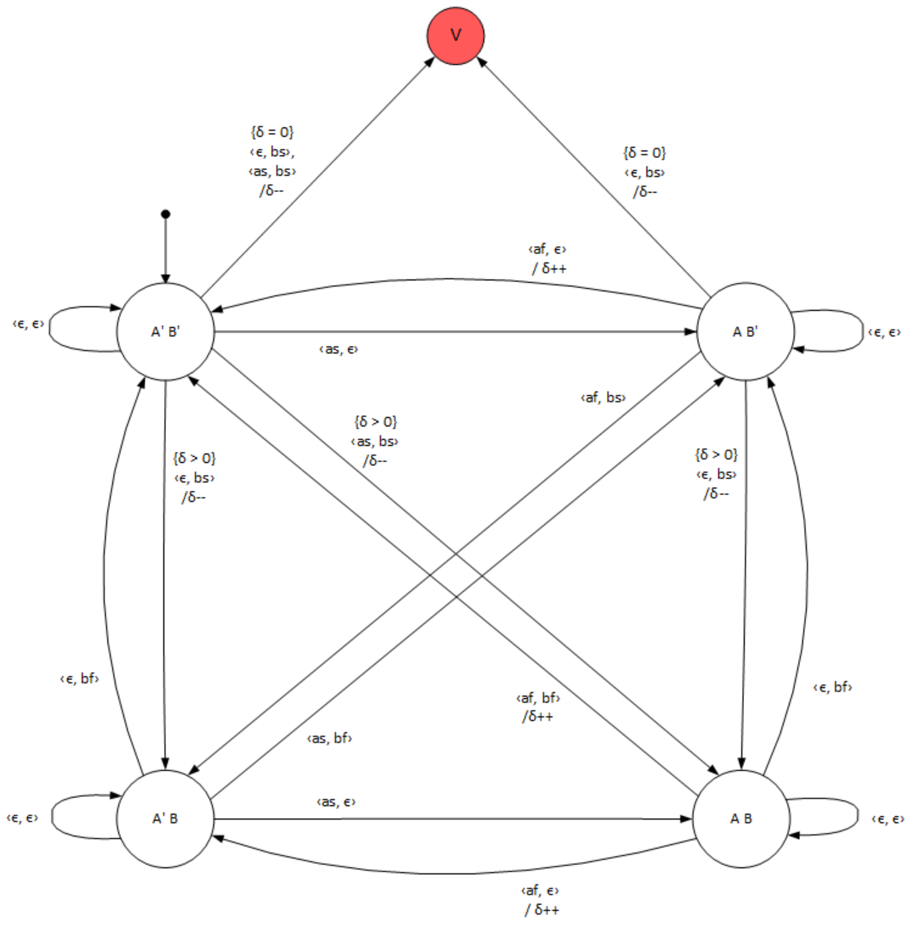


Figure 2.12: Automaton of A precedes B

SystemVerilog Observer

```
// A precedes B

module Precedes (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);
```

```
int unsigned delta;
int unsigned FSM;
reg v;

always @ (as or af or bs or bf)
begin
 case (FSM)
  //-------------------------------------------------------------
  0 : // State A'B'
  begin
   if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
   else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // as
   else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) // bs
   begin
    if (delta>0)
    begin
     delta--;
     FSM=1;
    end
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) // as,bs
   begin
    if (delta>0)
    begin
     delta--;
     FSM=3;
    end
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   else
   begin
    FSM=4;
    v=1'b1;
   end
  end
  //-------------------------------------------------------------
```

```
1 : // State A'B
begin
 if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
 else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // as
 else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=0; // bf
 else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // as,bf
 else
 begin
  FSM=4;
  v=1'b1;
 end
end
//------------------------------------------------------------
2 : // State AB'
begin
 if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
 else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) // af
 begin
  delta++;
  FSM=0;
 end
 else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) // bs
 begin
  if (delta>0)
  begin
   delta--;
   FSM=3;
  end
  else
  begin
   FSM=4;
   v=1'b1;
  end
 end
 else if (as==1'b0 && af==1'b1 && bs==1'b1 && bf==1'b0) FSM=1; // af,bs
 else
 begin
  FSM=4;
  v=1'b1;
 end
end
//------------------------------------------------------------
3 : // State AB
begin
 if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
 else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) // af
```

```
        begin
         delta++;
         FSM=1;
        end
        else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // bf
        else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1) // af, bf
        begin
         delta++;
         FSM=0;
        end
        else
        begin
         FSM=4;
         v=1'b1;
        end
      end
      // ------------------------------------------------------------
      default : // State 4: Violation
      begin
       FSM=4;
       v=1'b1;
      end
    endcase
  end

  assign violation = v;
  initial
  begin
   FSM = 0;
   v = 0;
  end
endmodule
```

2.2.2 Starts Relation

Mathematical Notation

The relation *'A starts B'* means the state A and state B trigger at the same instant. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\vdash} B$ means $b_s \equiv a_s$ and it implies that whenever state A occurs, the state B also has to trigger, as shown in Figure 2.13.

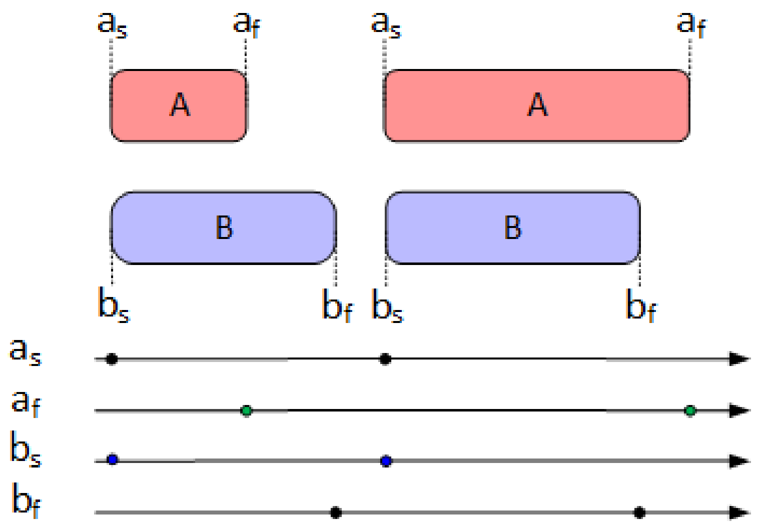


Figure 2.13: Depiction of A starts B

Graphical Representation

Graphically, contains relation is shown in Figure 2.14 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

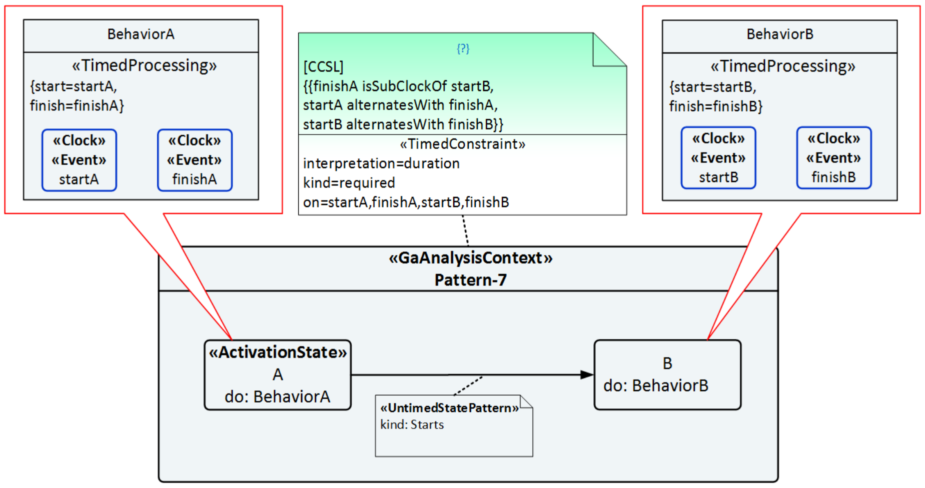


Figure 2.14: MARTE Model of A Starts B

Representation in CCSL

$$\begin{aligned} bs &\boxed{\subset} as, \\ as &\boxed{\sim} af, \\ bs &\boxed{\sim} bf \end{aligned}$$

Representation in CCSL (alternate form)

bs isSubclockOf as,
as alternatesWith af,
bs alternatesWith bf

State Automaton

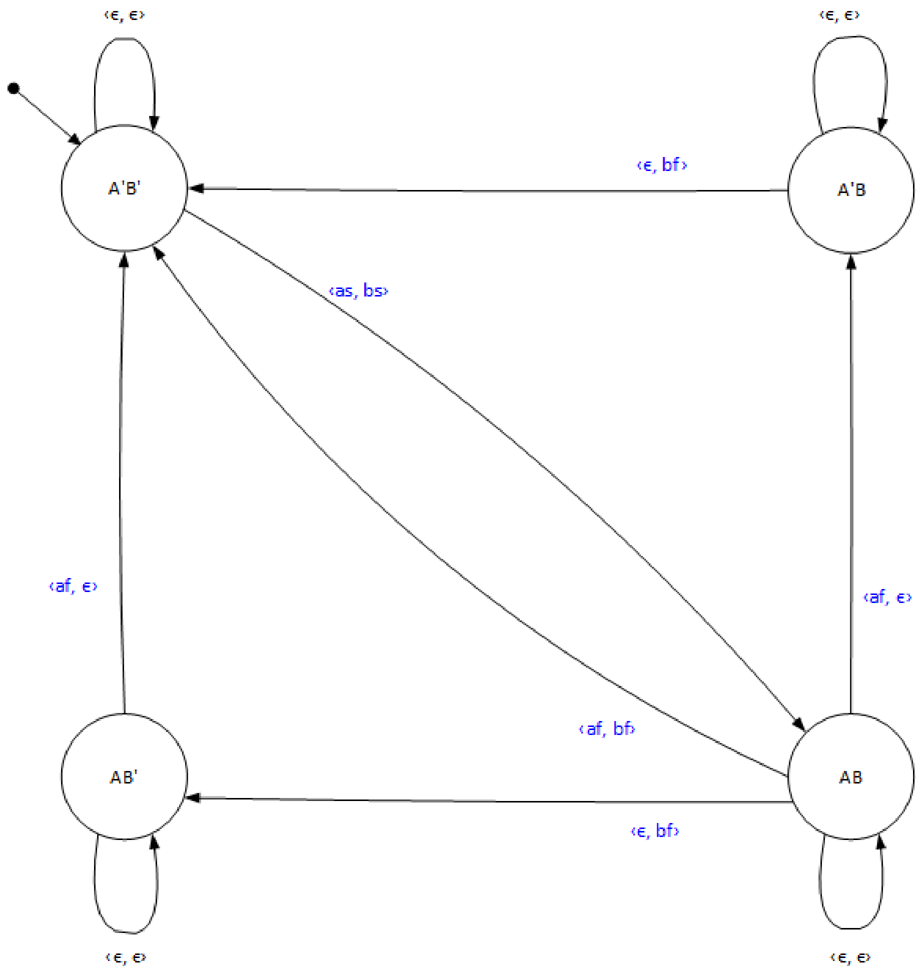


Figure 2.15: Automaton of A Starts B

SystemVerilog Observer

```
// A starts B

module Starts (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   //―――――――――――――――――――――――――――――――――――――――――
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // as,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //―――――――――――――――――――――――――――――――――――――――――
   1 : // State A'B
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
    else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=0; // bf
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //―――――――――――――――――――――――――――――――――――――――――
   2 : // State AB'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
    else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) FSM=0; // af
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //―――――――――――――――――――――――――――――――――――――――――
   3 : // State AB
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
```

```
      else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) FSM=1; // af
      else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // bf
      else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1) FSM=0; // af,bf
      else
      begin
       FSM=4;
       v=1'b1;
      end
    end
    //------------------------------------------------------------------
    default : // State 4: Violation
    begin
     FSM=4;
     v=1'b1;
    end
  endcase
 end

 assign violation = v;
 initial
 begin
  FSM = 0;
  v = 0;
 end
endmodule
```

2.2.3 Finishes Relation

Mathematical Notation

The relation *'A finishes B'* means the state B terminates when the state A gets inactive. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\dashv} B$ means $b_f \equiv a_f$ and it implies that b_f is subset of a_f, as shown in Figure 2.16.

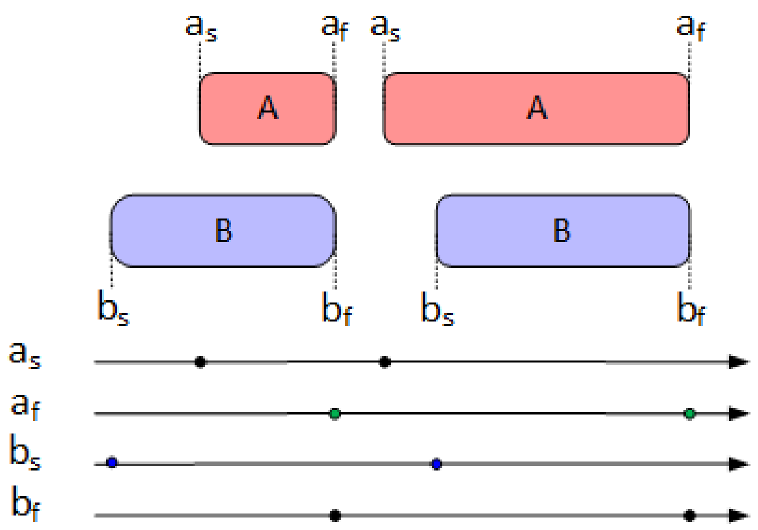


Figure 2.16: Depiction of A finishes B

Graphical Representation

Graphically, contains relation is shown in Figure 2.17 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

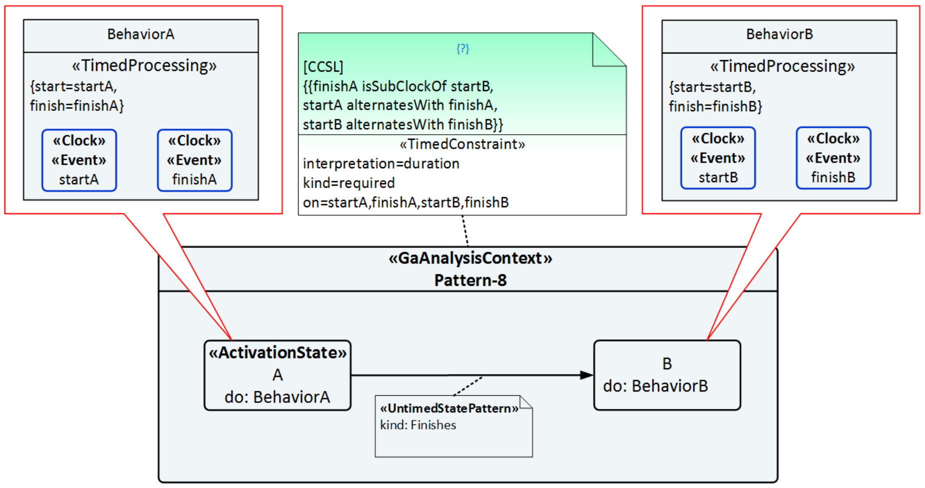


Figure 2.17: MARTE Model of A finishes B

Representation in CCSL

$$\begin{aligned} bf \; \boxed{\subset} \; af, \\ as \; \boxed{\sim} \; af, \\ bs \; \boxed{\sim} \; bf \end{aligned}$$

Representation in CCSL (alternate form)

bf isSubclockOf af,
as alternatesWith af,
bs alternatesWith bf

State Automaton

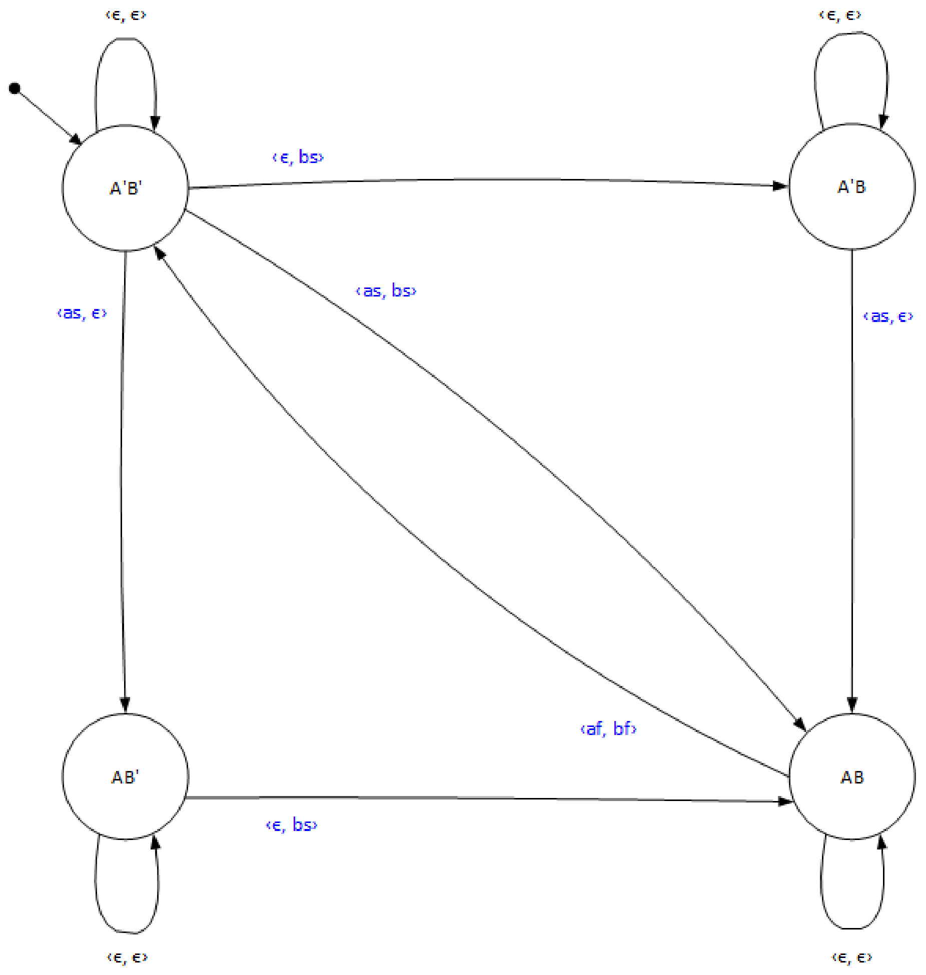


Figure 2.18: Automaton of A finishes B

SystemVerilog Observer

```
// A finishes B

module Finishes (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   // ----------------------------------------------------------------------
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // as
    else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=1; // bs
    else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // as,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ----------------------------------------------------------------------
   1 : // State A'B
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // as
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ----------------------------------------------------------------------
   2 : // State AB'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
    else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ----------------------------------------------------------------------
   3 : // State AB
```

```
    begin
     if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
     else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1) FSM=0; // af, bf
     else
     begin
      FSM=4;
      v=1'b1;
     end
    end
    // ------------------------------------------------------------
    default : // State 4: Violation
    begin
     FSM=4;
     v=1'b1;
    end
  endcase
 end

 assign violation = v;
 initial
 begin
  FSM = 0;
  v = 0;
 end
endmodule
```

2.2.4 Causes Relation

Mathematical Notation

The relation *'A causes B'* means the state B occurs when the state A terminates. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\prec} B$ means $b_s \equiv a_f$ and implies that whenever state A occurs it leads to the state B, as shown in Figure 2.19.

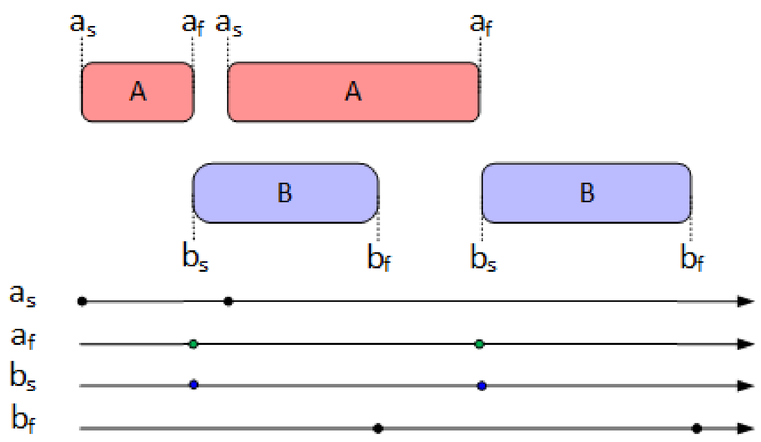


Figure 2.19: Depiction of A causes B

Graphical Representation

Graphically, contains relation is shown in Figure 2.20 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

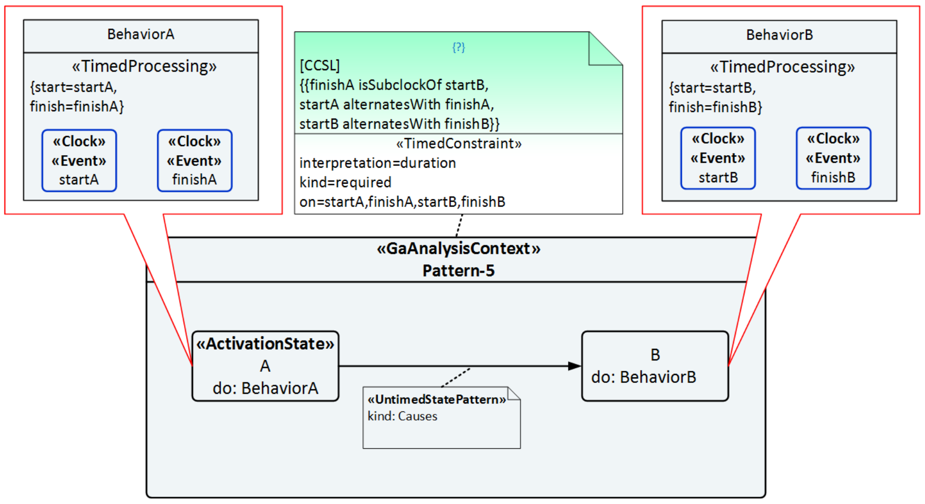


Figure 2.20: MARTE Model of A causes B

Representation in CCSL

$$bs \boxed{\subset} af,$$
$$as \boxed{\sim} af,$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

bs isSubclockOf af,
as alternatesWith af,
bs alternatesWith bf

State Automaton

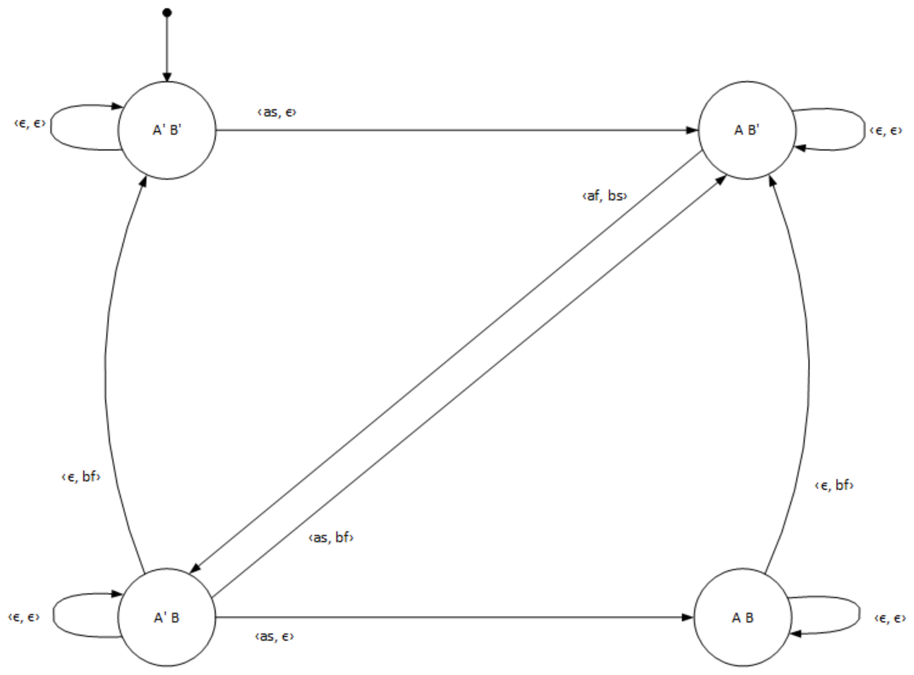


Figure 2.21: Automaton of A causes B

SystemVerilog Observer

```
// A causes B

module Causes (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   // ------------------------------------------------------------
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // as
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ------------------------------------------------------------
   1 : // State A'B
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // as
    else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=0; // bf
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // as,bf
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ------------------------------------------------------------
   2 : // State AB'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
    else if (as==1'b0 && af==1'b1 && bs==1'b1 && bf==1'b0) FSM=1; // af,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
```

```
    //────────────────────────────────────────────────
    3 : // State AB
    begin
      if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
      else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // bf
      else
      begin
       FSM=4;
       v=1'b1;
      end
    end
    //────────────────────────────────────────────────
    default : // State 4: Violation
    begin
     FSM=4;
     v=1'b1;
    end
  endcase
 end

 assign violation = v;
 initial
 begin
  FSM = 0;
  v = 0;
 end
endmodule
```

2.2.5 Contains Relation

Mathematical Notation

The relation *'A contains B'* means the state B (if it occurs) exists while state A is active. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\supseteq} B$ means $b_s, b_f \in [a_s, a_f]$ and it implies that whenever state B occurs, the relation $a_s \leqslant b_s$ and $b_f \leqslant a_f$ holds valid, as shown in Figure 2.22.

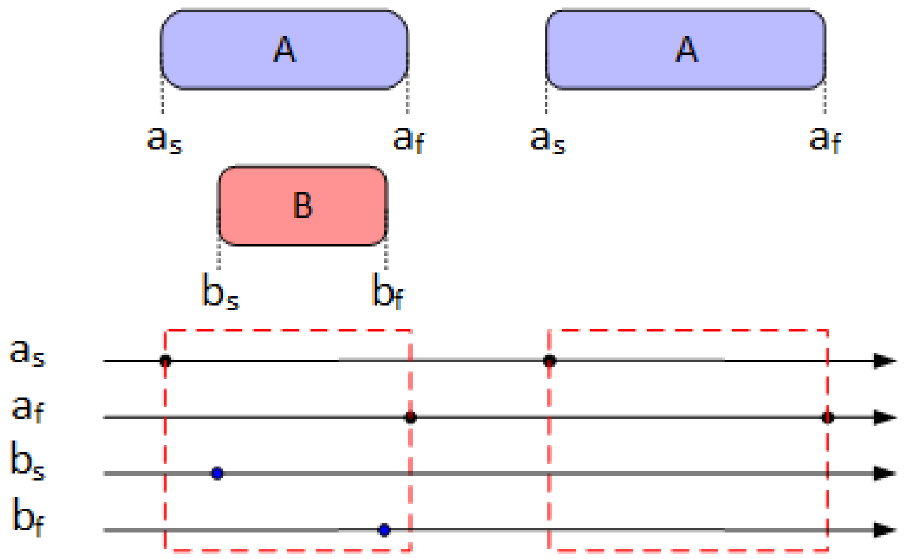


Figure 2.22: Depiction of A contains B

Graphical Representation

Graphically, contains relation is shown in Figure 2.23 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

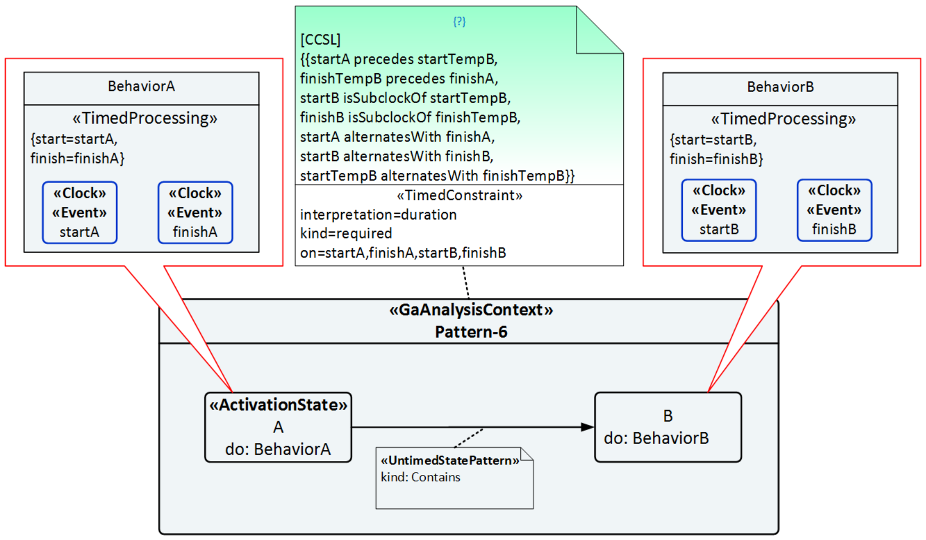


Figure 2.23: MARTE Model of A contains B

Representation in CCSL

$$\begin{aligned} as \; &\boxed{\preccurlyeq}\; bsTemp \\ bfTemp \; &\boxed{\preccurlyeq}\; af \\ bs \; &\boxed{\subset}\; bsTemp, \\ bf \; &\boxed{\subset}\; bfTemp, \\ as \; &\boxed{\sim}\; af \\ bs \; &\boxed{\sim}\; bf \\ bsTemp \; &\boxed{\sim}\; bfTemp \end{aligned}$$

Representation in CCSL (alternate form)

as precedes bsTemp,
bfTemp precedes af,
bs isSubclockOf bsTemp,
bf isSubclockOf bfTemp,
as alternatesWith af,
bs alternatesWith bf,
bsTemp alternatesWith bfTemp

State Automaton

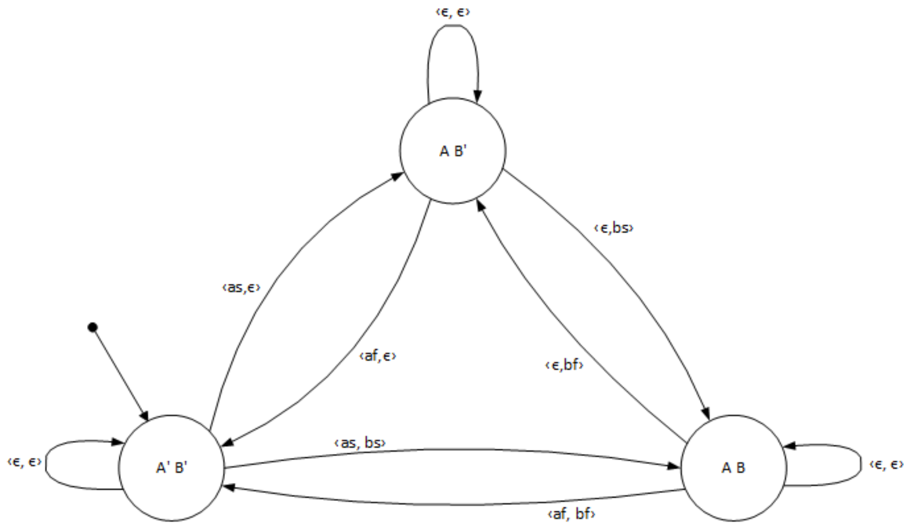


Figure 2.24: Automaton of A contains B

SystemVerilog Observer

```
// A contains B

module Contains (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   // ─────────────────────────────────────────────
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // as
    else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // as,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ------------------------------------------------------------
   2 : // State AB'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
    else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) FSM=0; // af
    else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ------------------------------------------------------------
   3 : // State AB
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
    else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // bf
    else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1) FSM=0; // af,bf
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ------------------------------------------------------------
   default : // State 4: Violation
   begin
    FSM=4;
    v=1'b1;
   end
  endcase
end

assign violation = v;
initial
begin
 FSM = 0;
 v = 0;
end
```

endmodule

2.2.6 Implies Relation

Mathematical Notation

The relation *'A implies B'* means the state B activates when the state A gets active. Mathematically, given a strict partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the equation $A \boxed{\Rightarrow} B$ means a_s coincides with b_s and a_f coincides with b_f, as shown in Figure 2.25.

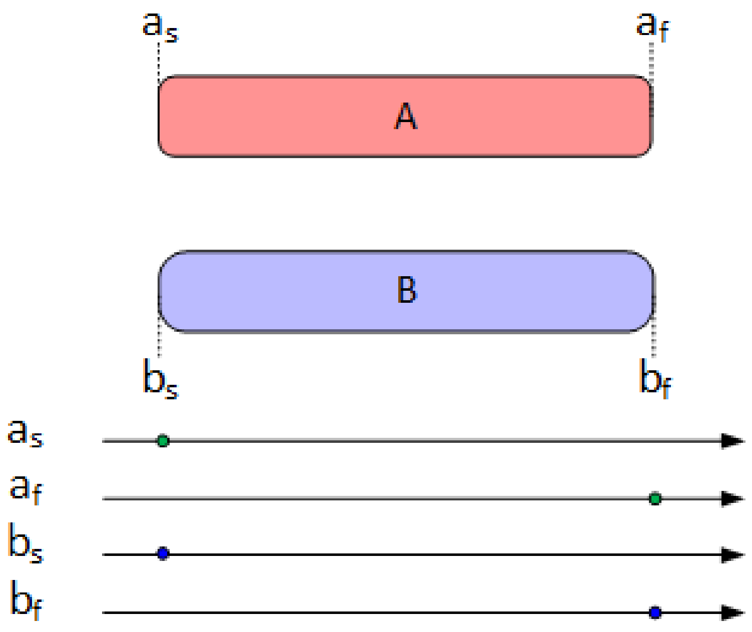


Figure 2.25: Depiction of A implies B

Graphical Representation

Graphically, contains relation is shown in Figure 2.26 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

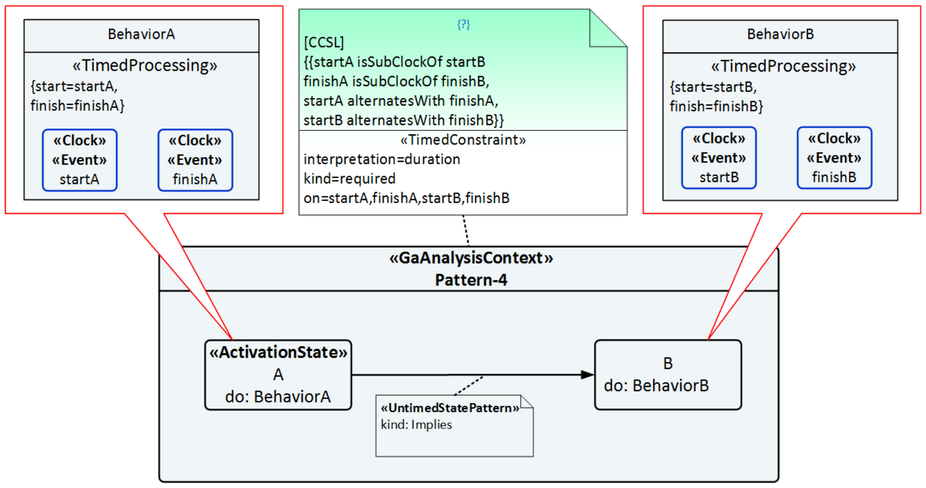


Figure 2.26: MARTE Model of A implies B

Representation in CCSL

$$
\begin{aligned}
bs &\boxed{\subset} as,\\
bf &\boxed{\subset} af,\\
as &\boxed{\sim} af,\\
bs &\boxed{\sim} bf
\end{aligned}
$$

Representation in CCSL (alternate form)

bs isSubclockOf as,
bf isSubclockOf af,
as alternatesWith af,
bs alternatesWith bf

State Automaton

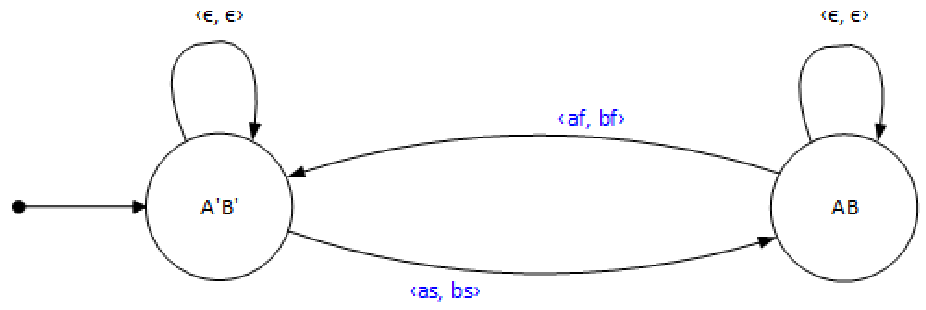


Figure 2.27: Automaton of A implies B

SystemVerilog Observer

```
// A implies B

module Implies (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   //------------------------------------------------------------
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // as,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //------------------------------------------------------------
```

```
    3 : // State AB
    begin
     if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
     else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b1) FSM=0; // af,bf
     else
     begin
      FSM=4;
      v=1'b1;
     end
    end
    //---------------------------------------------------------------------
    default : // State 4: Violation
    begin
     FSM=4;
     v=1'b1;
    end
   endcase
  end

  assign violation = v;
  initial
  begin
   FSM = 0;
   v = 0;
  end
endmodule
```

2.2.7 Forbids Relation

Mathematical Notation

Forbiddance is a negation property. Relation *'A forbids B'* bars B to occur after state A. Mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the relation $A \boxed{\neg} B$ means a_f is exclusive with the interval $[b_s, b_f]$, as shown in Figure 2.28.

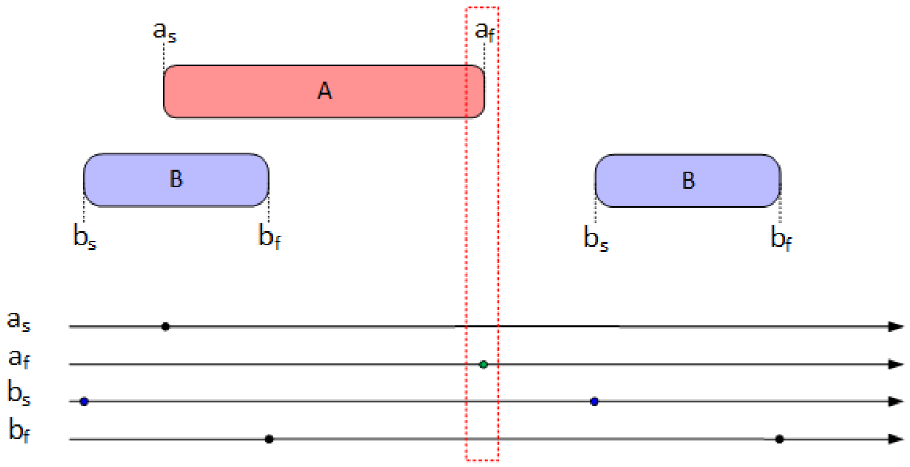


Figure 2.28: Depiction of A forbids B

Graphical Representation

Graphically, forbids relation logical/causal pattern is shown in Figure 2.29 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype.

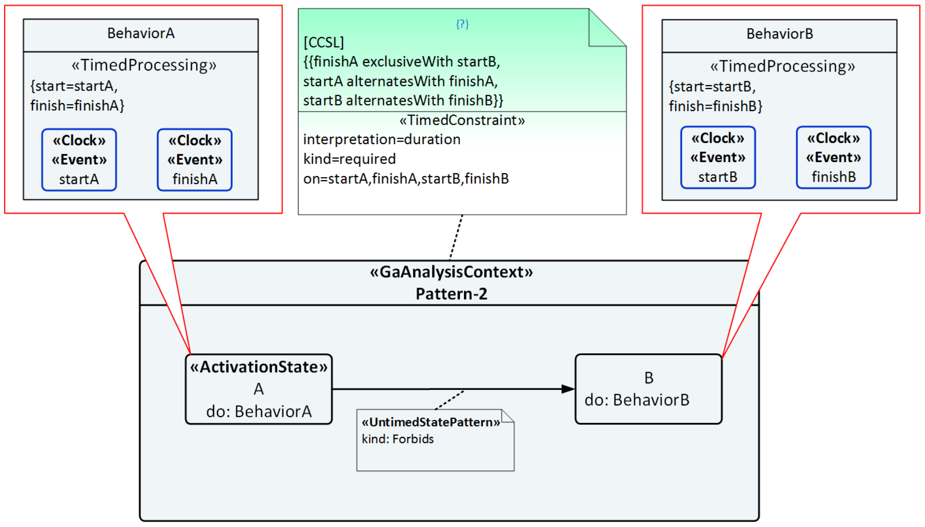


Figure 2.29: MARTE Model of A forbids B

Representation in CCSL

$$af \searrow bs \boxed{\prec} bf,$$
$$as \boxed{\sim} af,$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

af sampledOn bs precedes bf,
as alternatesWith af,
bs alternatesWith bf

State Automaton

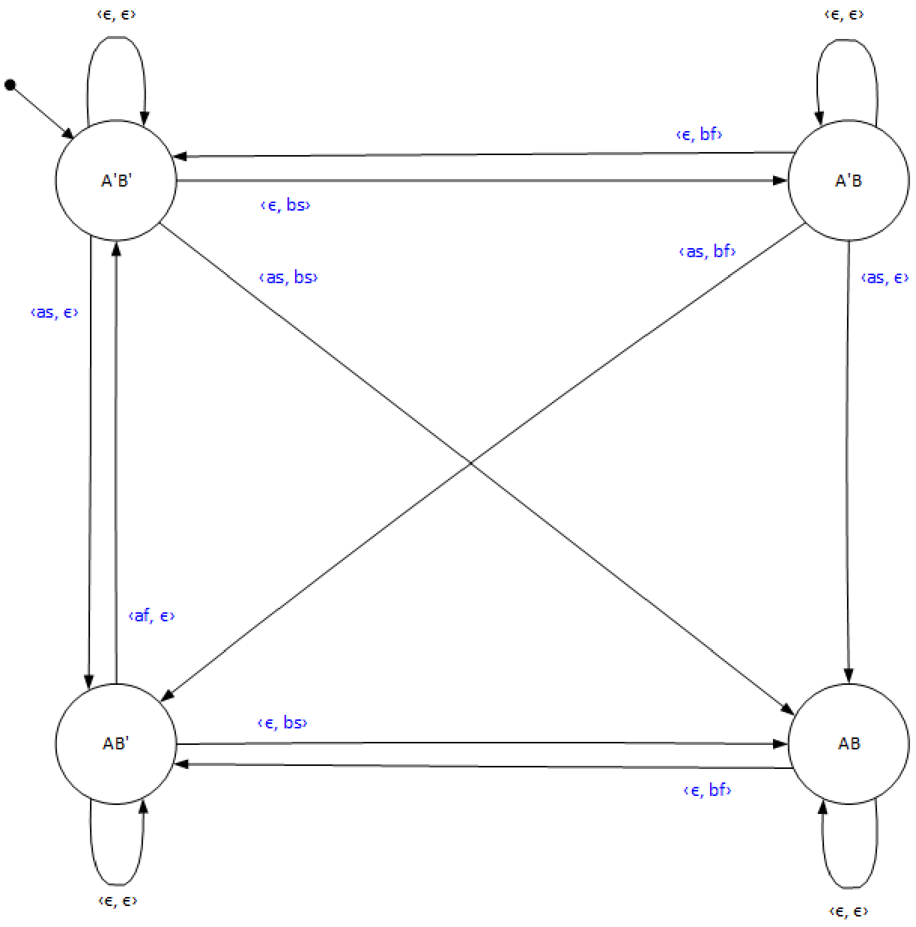


Figure 2.30: Automaton of A forbids B

SystemVerilog Observer

```
// A forbids B

module Forbids (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;

 always @ (as or af or bs or bf)
 begin
  case (FSM)
   // ----------------------------------------------------------------
   0 : // State A'B'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // as
    else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=1; // bs
    else if (as==1'b1 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // as,bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ----------------------------------------------------------------
   1 : // State A'B
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // as
    else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=0; // bf
    else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // as,bf
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   // ----------------------------------------------------------------
   2 : // State AB'
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
    else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) FSM=0; // af
    else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=3; // bs
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //------------------------------------------------------------------
   3 : // State AB
   begin
    if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=3; // empty
    else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=2; // bf
    else
    begin
     FSM=4;
     v=1'b1;
    end
   end
   //------------------------------------------------------------------
   default : // State 4: Violation
   begin
    FSM=4;
    v=1'b1;
   end
  endcase
 end

 assign violation = v;
 initial
 begin
  FSM = 0;
  v = 0;
 end
endmodule
```

2.2.8 Excludes Relation

Mathematical Notation

Exclusion relation between two states (*'A excludes B'*) restricts them to occur at the same time for any instant. Mathematically, given a partial ordering $\mathbb{S}$ having the states A ($[a_s, a_f]$) and B ($[b_s, b_f]$), the relation $A \# B$ means that $b_f < a_s$ or $a_f < b_s$ for all instances of states A and B, as shown in Figure 2.31.

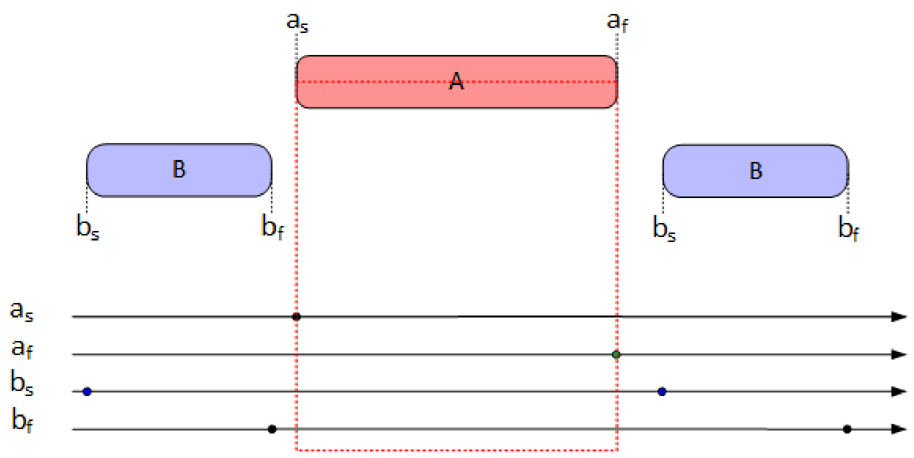
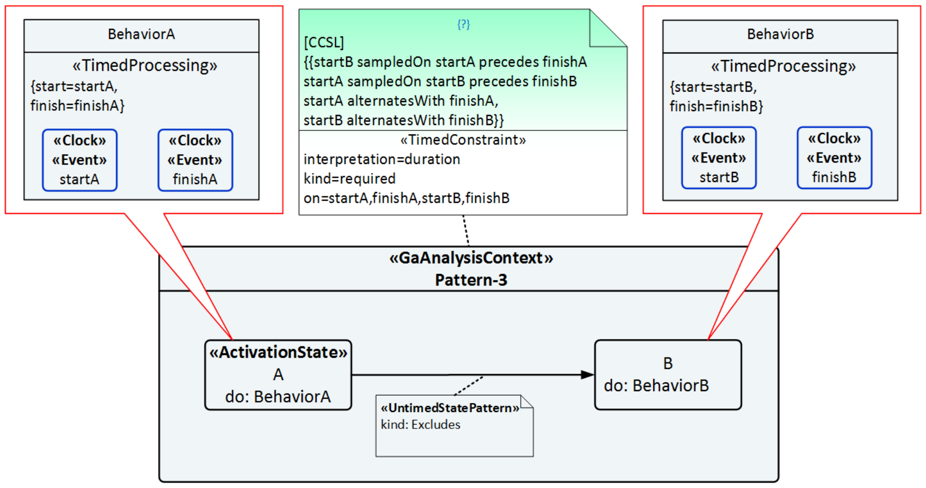


Figure 2.31: Depiction of A excludes B

Graphical Representation

Graphically, excludes relation logical/causal/untimed pattern is shown in Figure 2.32 as an annotated MARTE model. The first state (A) is an activation state as shown by the stereotype. Semantics of the states are defined using *TimedProcessing* stereotype applied to the behavior of the states. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint bars state B to exist concurrently with state A.



```
[CCSL]
{{startB sampledOn startA precedes finishA
startA sampledOn startB precedes finishB
startA alternatesWith finishA,
startB alternatesWith finishB}}
«TimedConstraint»
interpretation=duration
kind=required
on=startA,finishA,startB,finishB
```

Figure 2.32: MARTE Model of A excludes B

Representation in CCSL

$$bs \searrow as \boxed{\prec} af,$$
$$as \searrow bs \boxed{\prec} bf,$$
$$as \boxed{\sim} af,$$
$$bs \boxed{\sim} bf$$

Representation in CCSL (alternate form)

bs sampledOn as precedes af,
as sampledOn bs precedes bf,
as alternatesWith af,
bs alternatesWith bf

State Automaton

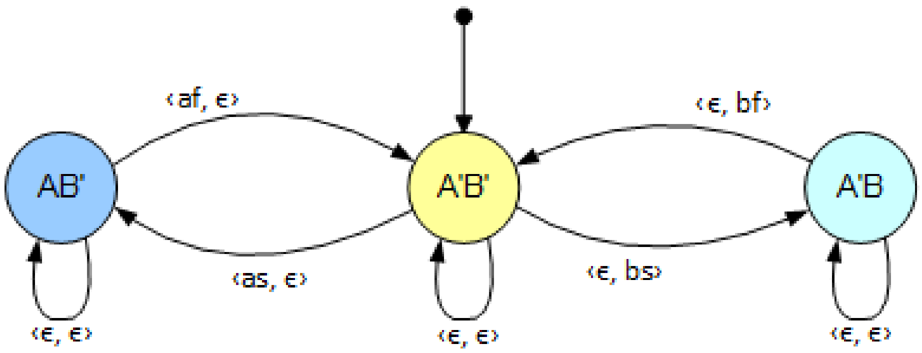


Figure 2.33: Automaton of A excludes B

SystemVerilog Observer

```
// A excludes B

module Excludes (
 input as,
 input af,
 input bs,
 input bf,
 output violation
);

 int unsigned FSM;
 reg v;
```

```
always @ (as or af or bs or bf)
begin
 case (FSM)
  //----------------------------------------------------------------------
  0 : // State A'B'
  begin
   if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=0; // empty
   else if (as==1'b1 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // as
   else if (as==1'b0 && af==1'b0 && bs==1'b1 && bf==1'b0) FSM=2; // bs
   else
   begin
    FSM=4;
    v=1'b1;
   end
  end
  //----------------------------------------------------------------------
  1 : // State A'B
  begin
   if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=1; // empty
   else if (as==1'b0 && af==1'b1 && bs==1'b0 && bf==1'b0) FSM=0; // af
   else
   begin
    FSM=4;
    v=1'b1;
   end
  end
  //----------------------------------------------------------------------
  2 : // State AB'
  begin
   if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b0) FSM=2; // empty
   else if (as==1'b0 && af==1'b0 && bs==1'b0 && bf==1'b1) FSM=0; // bf
   else
   begin
    FSM=4;
    v=1'b1;
   end
  end
  //----------------------------------------------------------------------
  default : // State 4: Violation
  begin
   FSM=4;
   v=1'b1;
  end
 endcase
end
```

```
  assign violation = v;
  initial
  begin
   FSM = 0;
   v = 0;
  end
endmodule
```

Chapter 3

State-Event Graphical Patterns

The relations between the system states and events can only be modeled using the UML sequence diagrams which suits modeling flow of events. The concept of behavior execution specification is used to represent the system states. Moreover the sequence diagrams have the consider/ignore combined fragments which allow us to maintain the list of events that are relevant to the current pattern scenario.

3.1 Temporal/Timed State-Event Patterns

Temporal patterns are the ones associated with notion of time or clock events. State-event temporal patterns identified in this work are triggers and terminates.

3.1.1 Triggers Relation

Mathematical Notation

The *triggers* relation is similar to starts relation for states. The temporal relation *e triggers A*, given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$), an event e, constants m, n and a clock clk, can be expressed mathematically as

$$e \models A \quad after\ [m, n]\ on\ clk$$

It means a_s occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk, as shown in Figure 3.1.

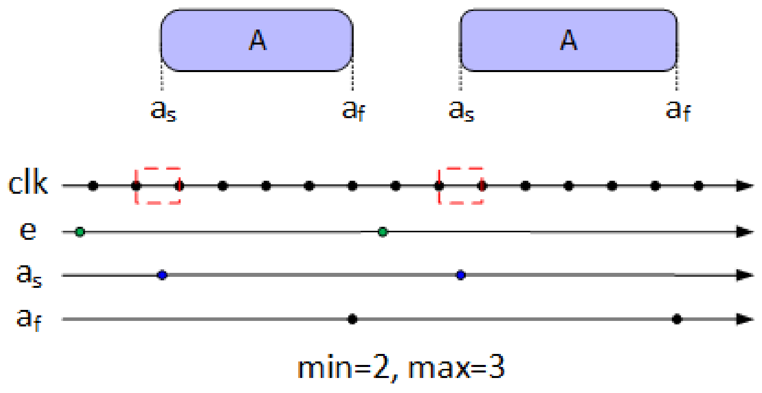


Figure 3.1: Depiction of e triggers A after [2,3] on clk

Graphical Representation

Graphically, the timed triggers pattern is shown in Figure 3.2 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. *Duration constraint* element is used to specify the required delay limits. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification.* Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e followed by event a_s within the given time limits.

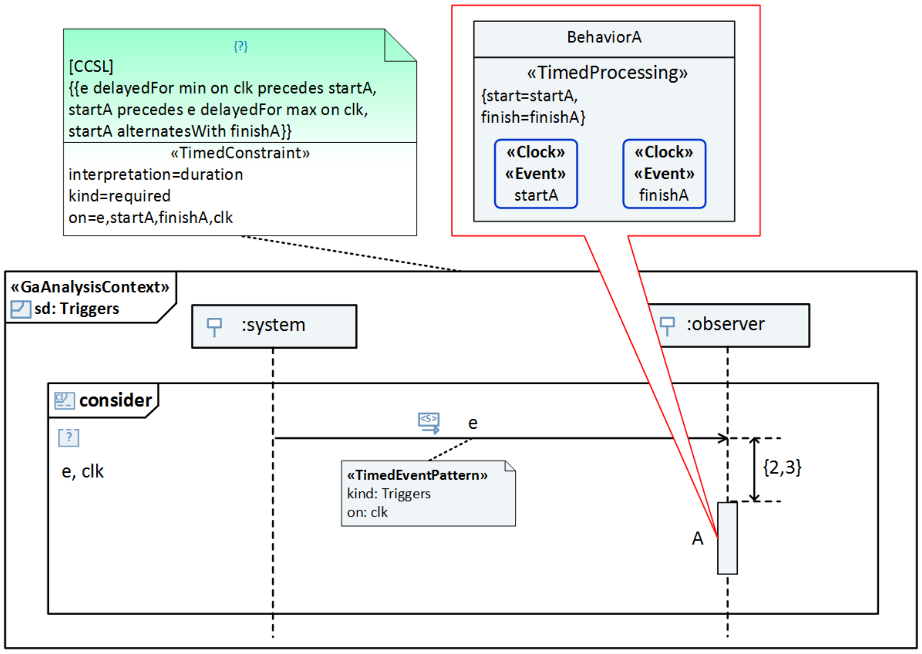


Figure 3.2: MARTE Model of e triggers A after [2,3] on clk

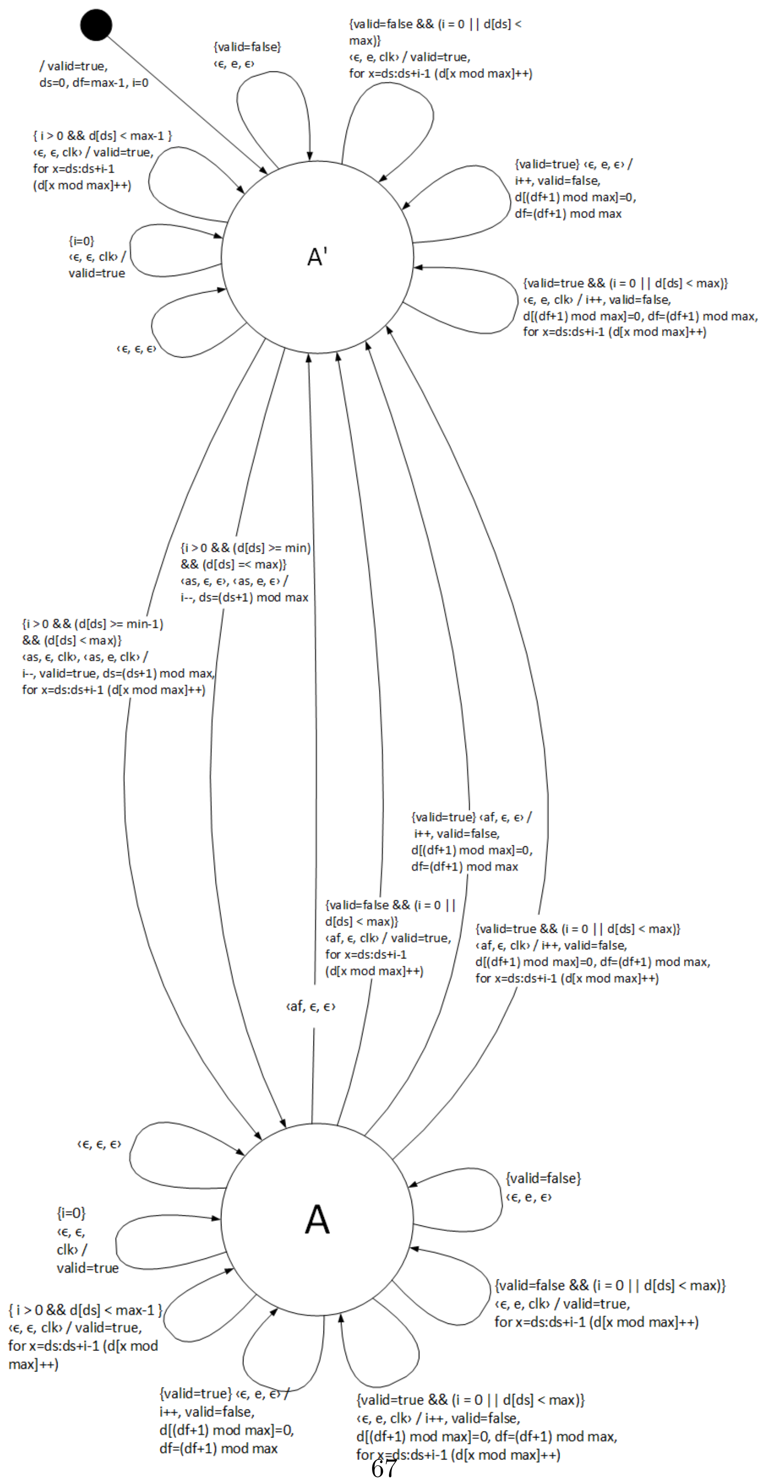
Representation in CCSL

$$e(min) \rightsquigarrow clk \boxed{\preccurlyeq} as \boxed{\preccurlyeq} e(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$

Representation in CCSL (alternate form)

e delayedFor min on clk precedes as
as precedes e delayedFor max on clk
as alternatesWith af

State Automaton



/ valid=true, ds=0, df=max-1, i=0
{valid=false} ‹ϵ, e, ϵ›
{valid=false && (i = 0 || d[ds] < max)} ‹ϵ, e, clk› / valid=true, for x=ds:ds+i-1 (d[x mod max]++)
{ i > 0 && d[ds] < max-1 } ‹ϵ, ϵ, clk› / valid=true, for x=ds:ds+i-1 (d[x mod max]++)
{valid=true} ‹ϵ, e, ϵ› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max
A'
{i=0} ‹ϵ, ϵ, clk› / valid=true
{valid=true && (i = 0 || d[ds] < max)} ‹ϵ, e, clk› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max, for x=ds:ds+i-1 (d[x mod max]++)
‹ϵ, ϵ, ϵ›
{i > 0 && (d[ds] >= min) && (d[ds] =< max)} ‹as, ϵ, ϵ›, ‹as, e, ϵ› / i--, ds=(ds+1) mod max
{i > 0 && (d[ds] >= min-1) && (d[ds] < max)} ‹as, ϵ, clk›, ‹as, e, clk› / i--, valid=true, ds=(ds+1) mod max, for x=ds:ds+i-1 (d[x mod max]++)
{valid=true} ‹af, ϵ, ϵ› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max
{valid=false && (i = 0 || d[ds] < max)} ‹af, ϵ, clk› / valid=true, for x=ds:ds+i-1 (d[x mod max]++)
{valid=true && (i = 0 || d[ds] < max)} ‹af, ϵ, clk› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max, for x=ds:ds+i-1 (d[x mod max]++)
‹af, ϵ, ϵ›
‹ϵ, ϵ, ϵ›
A
{valid=false} ‹ϵ, e, ϵ›
{i=0} ‹ϵ, ϵ, clk› / valid=true
{valid=false && (i = 0 || d[ds] < max)} ‹ϵ, e, clk› / valid=true, for x=ds:ds+i-1 (d[x mod max]++)
{ i > 0 && d[ds] < max-1 } ‹ϵ, ϵ, clk› / valid=true, for x=ds:ds+i-1 (d[x mod max]++)
{valid=true} ‹ϵ, e, ϵ› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max
{valid=true && (i = 0 || d[ds] < max)} ‹ϵ, e, clk› / i++, valid=false, d[(df+1) mod max]=0, df=(df+1) mod max, for x=ds:ds+i-1 (d[x mod max]++)

Figure 3.3: Automaton of e triggers A after [m,n] on clk

SystemVerilog Observer

```
// e triggers A after [m,n] on clk

module Triggers (
 input as,
 input af,
 input e,
 input clk,
 output violation
 );
 parameter min=2,max=4;

 reg valid;
 int unsigned d[max-1];
 int unsigned ds;
 int unsigned df;
 int unsigned i;

 int unsigned FSM;
 reg v;

 always @ (as or af or e or clk)
 begin
  case(FSM)
   // ------------------------------------------------------------------
   0: // State A'
    if(as==1'b0 && af==1'b0 && e==1'b0 && clk==1'b0) FSM=0; // empty
    else if(as==1'b0 && af==1'b0 && e==1'b0 && clk==1'b1) // clk
    begin
     if(i==0) begin FSM=0;valid=1'b1; end
     else if(i>0 && d[ds]<(max-1))
     begin
      FSM=0;valid=1'b1;
      for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
     end
     else
     begin
      FSM=2;v=1'b1;
     end
    end
    else if(as==1'b0 && af==1'b0 && e==1'b1 && clk==1'b0) // e
    begin
     FSM=0;
     if(valid==1'b1)
     begin
    valid=1'b0; i++;
    d[(df+1) % max]=0;
    df=(df+1) % max;
  end
end
else if(as==1'b0 && af==1'b0 && e==1'b1 && clk==1'b1) // e,clk
begin
 FSM=0;
  if(valid==1'b0 && i==0 || d[ds]<max)
  begin
    valid=1'b1;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else if(valid==1'b1 && i==0 || d[ds]<max)
  begin
    valid=1'b0; i++;
    d[(df+1) % max]=0;
    df=(df+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
  begin
   FSM=2;v=1'b1;
  end
end
else if(as==1'b1 && af==1'b0 && clk==1'b0) // as
begin
  if(i>0 && d[ds]>=min && d[ds]<=max)
  begin
    i--;FSM=1;
    ds=(ds+1) % max;
  end
  else
  begin
   FSM=2;v=1'b1;
  end
end
else if(as==1'b1 && af==1'b0 && clk==1'b1) // as,clk
begin
  if(i>0 && d[ds]>=min-1 && d[ds]<max)
  begin
    i--;valid=1'b1;FSM=1;
    ds=(ds+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
  end
  else
```

```
    begin
     FSM=2;v=1'b1;
    end
  end
  else
  begin
   FSM=2;v=1'b1;
  end
//----------------------------------------------------------------
1: // State A
  if(as==1'b0 && af==1'b0 && e==1'b0 && clk==1'b0) FSM=1; // empty
  else if(as==1'b0 && af==1'b0 && e==1'b0 && clk==1'b1) // clk
  begin
    if(i==0) begin FSM=1;valid=1'b1; end
    else if(i>0 && d[ds]<(max-1))
    begin
     FSM=1;valid=1'b1;
     for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
    end
    else
    begin
     FSM=2;v=1'b1;
    end
  end
  else if(as==1'b0 && af==1'b0 && e==1'b1 && clk==1'b0) // e
  begin
   FSM=1;
    if(valid==1'b1)
    begin
      valid=1'b0; i++;
      d[(df+1) % max]=0;
      df=(df+1) % max;
    end
  end
  else if(as==1'b0 && af==1'b0 && e==1'b1 && clk==1'b1) // e,clk
  begin
   FSM=1;
    if(valid==1'b0 && i==0 || d[ds]<max)
    begin
      valid=1'b1;
      for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
    end
    else if(valid==1'b1 && i==0 || d[ds]<max)
    begin
      valid=1'b0; i++;
      d[(df+1) % max]=0;
    df=(df+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=2;v=1'b1;
   end
  end
  else if(as==1'b0 && af==1'b1 && clk==1'b0) // af
  begin
   FSM=0;
   if(valid==1'b1)
   begin
    valid=1'b0; i++;
    d[(df+1) % max]=0;
    df=(df+1) % max;
   end
  end
  else if(as==1'b0 && af==1'b1 && clk==1'b1) // af,clk
  begin
   FSM=0;
   if(valid==1'b0 && i==0 || d[ds]<max)
   begin
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else if(valid==1'b1 && i==0 || d[ds]<max)
   begin
    valid=1'b0; i++;
    d[(df+1) % max]=0;
    df=(df+1) % max;
    for(int unsigned x=ds;x<=ds+i-1;x++) d[x % max]++;
   end
   else
   begin
    FSM=2;v=1'b1;
   end
  end
  else
  begin
   FSM=2;v=1'b1;
  end
//------------------------------------------------------------
default: // Violation
begin
 FSM=2;v=1'b1;
end
```

```
  endcase
 end

 assign  violation = v;

 initial
 begin
  valid=1'b1;
  ds=0;
  df=max-1;
  d[max-1]=0;
  i=0;
  FSM = 0;
  v = 0;
 end
endmodule
```

3.1.2 Terminates Relation

Mathematical Notation

The *terminates* relation is similar to finishes relation for states. The temporal relation e *terminates* A can be expressed mathematically, given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$), an event e, constants m, n and a clock clk, as

$$e \dashv A \quad after\ [m, n]\ on\ clk$$

It means a_f occurs within the duration $e + \Delta$, where Δ is between m and n ticks of event clk, as shown in Figure 3.4.

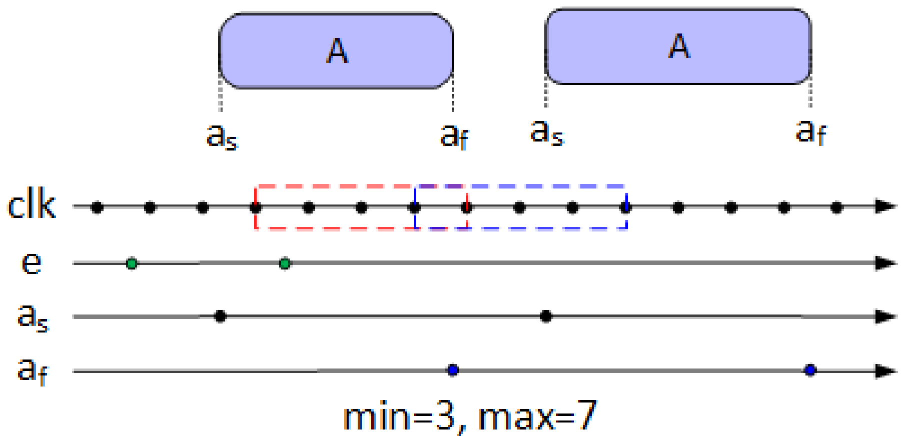


Figure 3.4: Depiction of e terminates A after [3,7] on clk

Graphical Representation

Graphically, the timed terminates pattern is shown in Figure 3.5 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. *Duration constraint* element is used to specify the required delay limits. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification*. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e followed by event a_s within the given time limits.

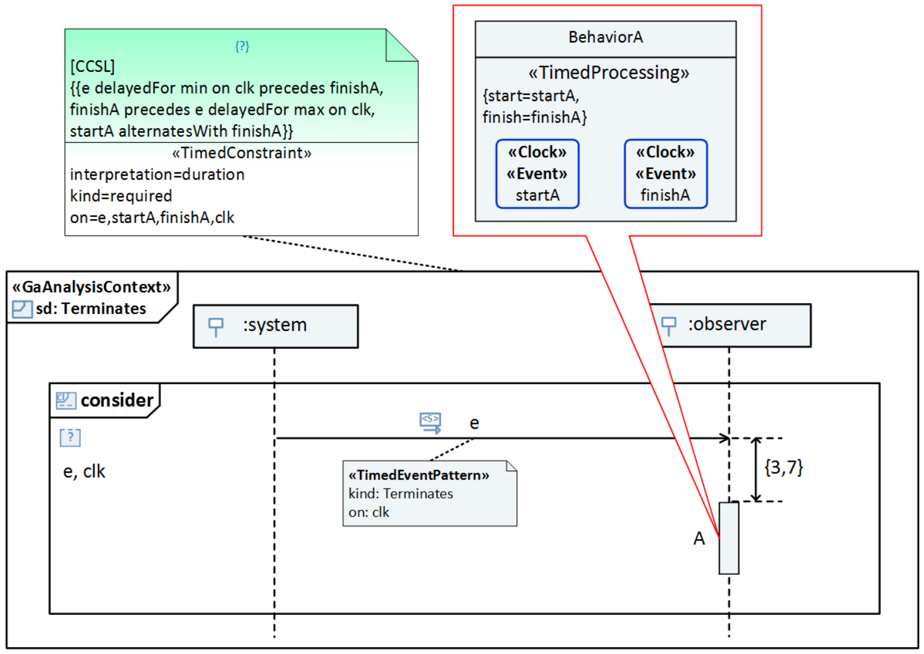


Figure 3.5: MARTE Model of e terminates A after [3,7] on clk

Representation in CCSL

$$e(min) \rightsquigarrow clk \boxed{\preccurlyeq} af \boxed{\preccurlyeq} e(max) \rightsquigarrow clk$$
$$as \boxed{\sim} af$$

Representation in CCSL (alternate form)

e delayedFor min on clk precedes af
af precedes e delayedFor max on clk
as alternatesWith af

State Automaton

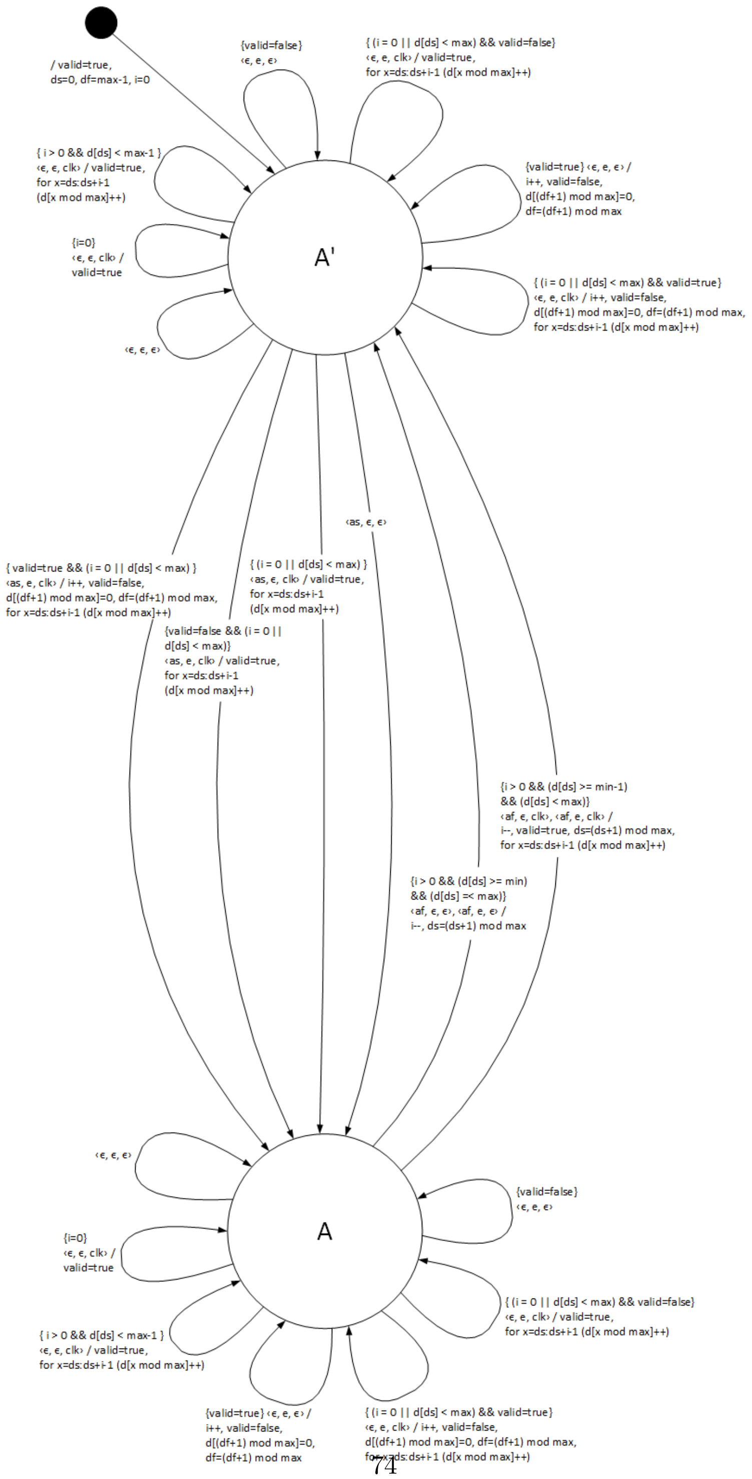

/ valid=true,
ds=0, df=max-1, i=0
{valid=false}
‹ϵ, e, ϵ›
{ (i = 0 || d[ds] < max) && valid=false}
‹ϵ, e, clk› / valid=true,
for x=ds:ds+i-1 (d[x mod max]++)
{ i > 0 && d[ds] < max-1 }
‹ϵ, ϵ, clk› / valid=true,
for x=ds:ds+i-1
(d[x mod max]++)
{valid=true} ‹ϵ, e, ϵ› /
i++, valid=false,
d[(df+1) mod max]=0,
df=(df+1) mod max
A'
{i=0}
‹ϵ, ϵ, clk› /
valid=true
{ (i = 0 || d[ds] < max) && valid=true}
‹ϵ, e, clk› / i++, valid=false,
d[(df+1) mod max]=0, df=(df+1) mod max,
for x=ds:ds+i-1 (d[x mod max]++)
‹ϵ, ϵ, ϵ›
‹as, ϵ, ϵ›
{ valid=true && (i = 0 || d[ds] < max) }
‹as, e, clk› / i++, valid=false,
d[(df+1) mod max]=0, df=(df+1) mod max,
for x=ds:ds+i-1 (d[x mod max]++)
{ (i = 0 || d[ds] < max) }
‹as, ϵ, clk› / valid=true,
for x=ds:ds+i-1
(d[x mod max]++)
{valid=false && (i = 0 ||
d[ds] < max)}
‹as, e, clk› / valid=true,
for x=ds:ds+i-1
(d[x mod max]++)
{i > 0 && (d[ds] >= min-1)
&& (d[ds] < max)}
‹af, ϵ, clk›, ‹af, e, clk› /
i--, valid=true, ds=(ds+1) mod max,
for x=ds:ds+i-1 (d[x mod max]++)
{i > 0 && (d[ds] >= min)
&& (d[ds] =< max)}
‹af, ϵ, ϵ›, ‹af, e, ϵ› /
i--, ds=(ds+1) mod max
‹ϵ, ϵ, ϵ›
A
{valid=false}
‹ϵ, e, ϵ›
{i=0}
‹ϵ, ϵ, clk› /
valid=true
{ (i = 0 || d[ds] < max) && valid=false}
‹ϵ, e, clk› / valid=true,
for x=ds:ds+i-1 (d[x mod max]++)
{ i > 0 && d[ds] < max-1 }
‹ϵ, ϵ, clk› / valid=true,
for x=ds:ds+i-1 (d[x mod max]++)
{valid=true} ‹ϵ, e, ϵ› /
i++, valid=false,
d[(df+1) mod max]=0,
df=(df+1) mod max
{ (i = 0 || d[ds] < max) && valid=true}
‹ϵ, e, clk› / i++, valid=false,
d[(df+1) mod max]=0, df=(df+1) mod max,
for x=ds:ds+i-1 (d[x mod max]++)

Figure 3.6: Automaton of e terminates A after [m,n] on clk

SystemVerilog Observer

Terminates timed pattern is similar to triggers timed pattern in implementation. To save the space, observer code is not shown here.

3.2 Logical/Untimed State-Event Patterns

Logical or causal patterns are untimed and relate a state and an event directly. All the temporal state-event patterns mentioned earlier can also be logical in nature. Logical patterns explained next are triggers, terminates, forbids, and contains.

3.2.1 Triggers Relation

Mathematical Notation

Triggering relation (*'e triggers A'*) requires that event e starts the state A. Mathematically, given a partial ordering $\mathbb{S}$ having the event e and state A ($[a_s, a_f]$), the relation $e \models A$ means that $e \equiv a_s$ for all instances of state A and event e, as shown in Figure 3.7.

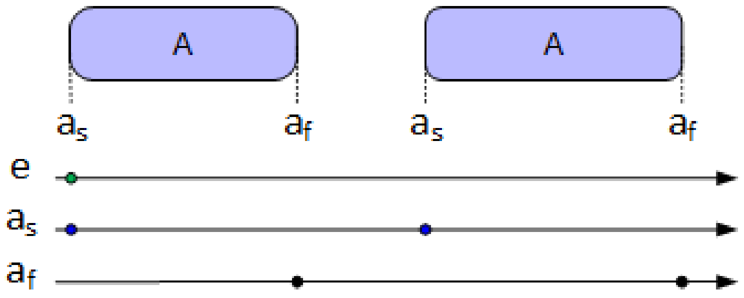


Figure 3.7: Depiction of e triggers A

Graphical Representation

Graphically, the triggers logical/causal/untimed pattern is shown in Figure 3.8 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification*. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e to be sub-clock of a_s.

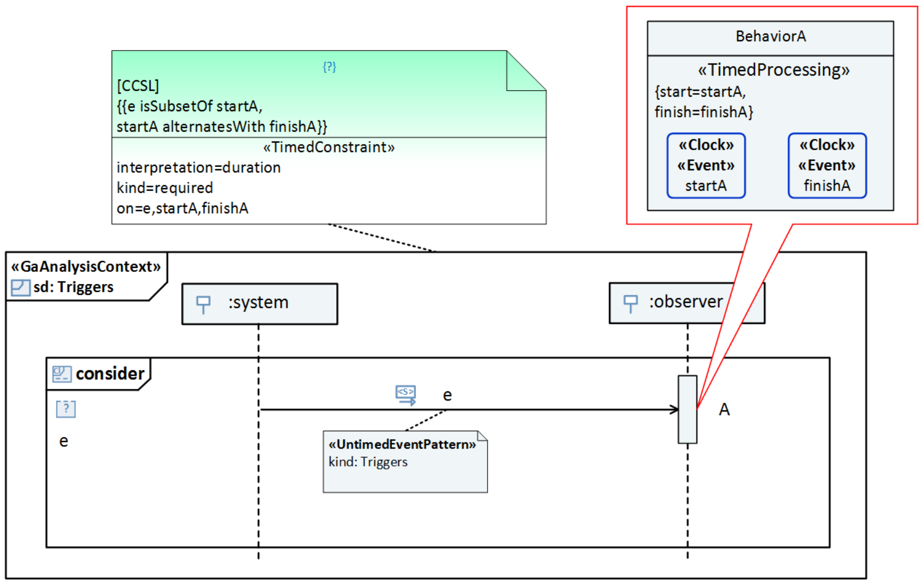


Figure 3.8: MARTE Model of e triggers A

Representation in CCSL

$$
\begin{aligned}
e &\boxed{\subset} as, \\
as &\boxed{\sim} af
\end{aligned}
$$

Representation in CCSL (alternate form)

e isSubclockOf as,
as alternatesWith af

State Automaton

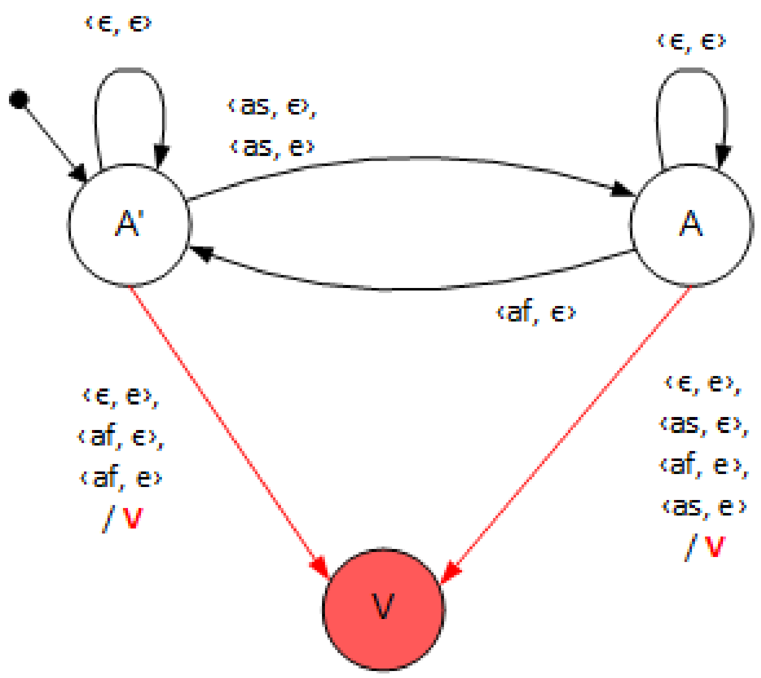


Figure 3.9: Automaton of e triggers A

SystemVerilog Observer

```
// e triggers A

module Triggers (
 input as,
 input af,
 input e,
 output violation
 );

 int unsigned FSM;
 reg v;

 always @ (as or af or e)
 begin
  case (FSM)
   // ----------------------------------------------------------------
   0 : // State A'
```

```
    begin
     if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;        // empty
     else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;   // as
     else if (as==1'b1 && af==1'b0 && e==1'b1) FSM=1;   // as, e
     else
     begin
      FSM=2;
      v=1'b1;
     end
    end
    // ----------------------------------------------------------------
    1 : // State A
    begin
     if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;        // empty
     else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;   // af
     else
     begin
      FSM=2;
      v=1'b1;
     end
    end
    // ----------------------------------------------------------------
    default : // Violation
    begin
     FSM=2;
     v=1'b1;
    end
   endcase
  end

  assign violation = v;

  initial
  begin
   FSM = 0;
   v = 0;
  end
endmodule
```

3.2.2 Terminates Relation

Mathematical Notation

Termination relation (*'e terminates A'*) requires that event e ends the state A. Mathematically, given a partial ordering $\mathbb{S}$ having the event e and state A ($[a_s, a_f]$), the relation $e \dashv A$ means that $e \equiv a_f$ for all instances of state A and event e, as shown in Figure 3.10.

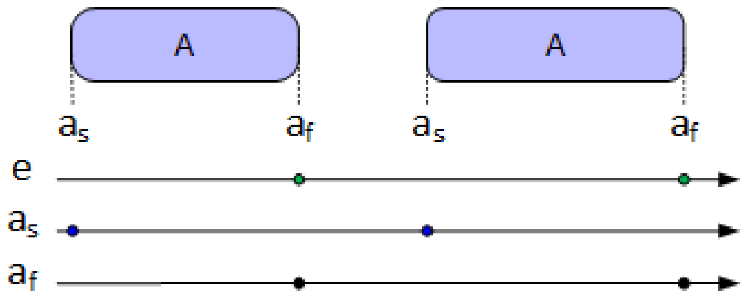


Figure 3.10: Depiction of e terminates A

Graphical Representation

Graphically, the terminates logical/causal/untimed pattern is shown in Figure 3.11 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification*. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e to be sub-clock of a_f.

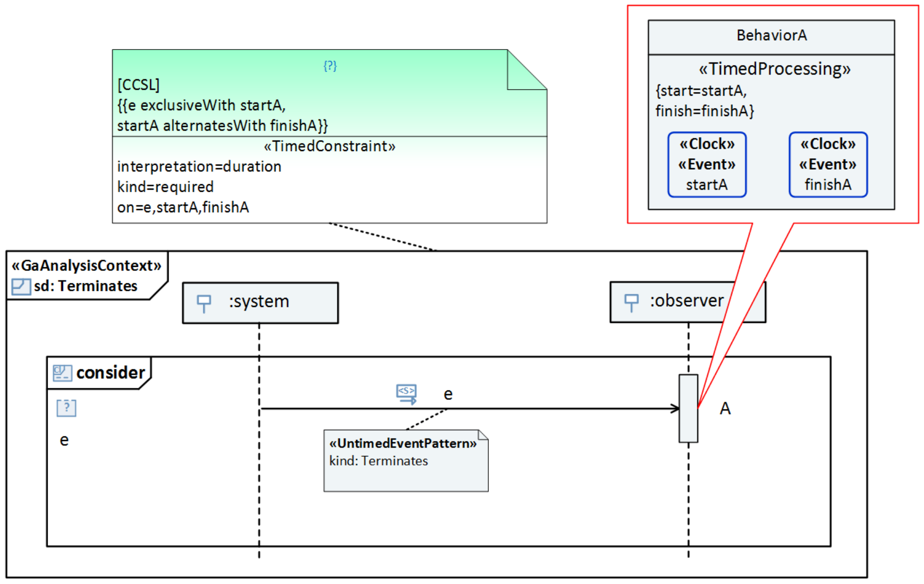


Figure 3.11: MARTE Model of e terminates A

Representation in CCSL

$$e \boxed{\subset} af,$$
$$as \boxed{\sim} af$$

Representation in CCSL (alternate form)

e isSubclockOf af,
as alternatesWith af

State Automaton

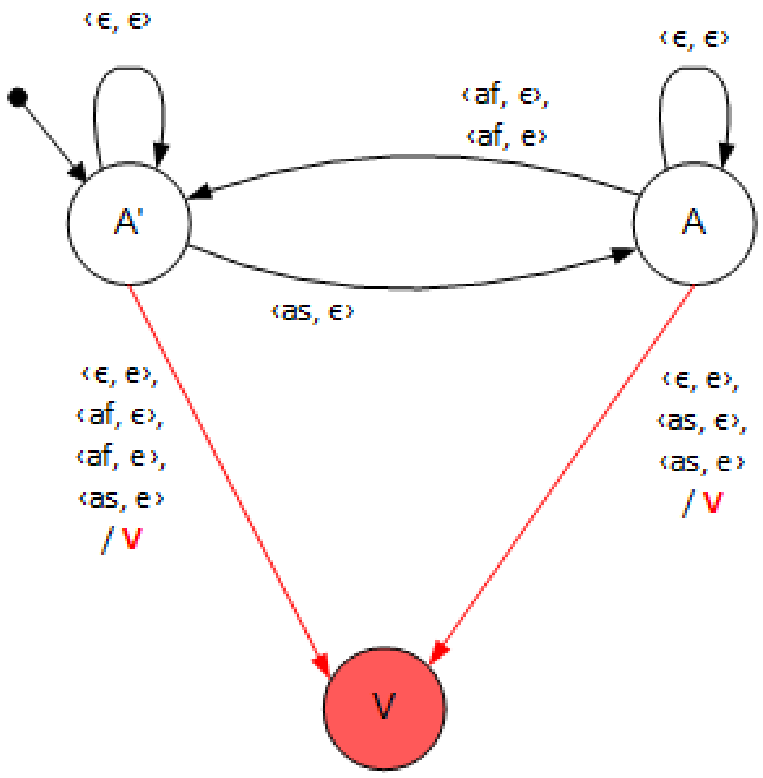


Figure 3.12: Automaton of e terminates A

SystemVerilog Observer

```
// e terminates A

module Terminates (
input as,
input af,
input e,
output violation
);

int unsigned FSM;
reg v;

always @ (as or af or e)
begin
 case (FSM)
  //--------------------------------------------------------------
  0 : // State A'
  begin
   if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;         // empty
   else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;   // as
   else
   begin
    FSM=2;
    v=1'b1;
   end
  end
  //--------------------------------------------------------------
  1 : // State A
  begin
   if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;         // empty
   else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;   // af
   else if (as==1'b0 && af==1'b1 && e==1'b1) FSM=0;   // af,e
   else
   begin
    FSM=2;
    v=1'b1;
   end
  end
  //--------------------------------------------------------------
  default : // Violation
  begin
   FSM=2;
   v=1'b1;
  end
 endcase
end

assign violation = v;
```

```
 initial
 begin
  FSM = 0;
   v = 0;
 end
endmodule
```

3.2.3 Forbids Relation

Mathematical Notation

The *forbids* relation is similar to forbids relation for states. Relation *e forbids A* implies A must not occur (or is in occurrence) when the event e triggers. Hence given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$) and an event e, the relation is expressed mathematically as $e \neg A$ which means $e < a_s$ or $a_f < e$, as shown in Figure 3.13.

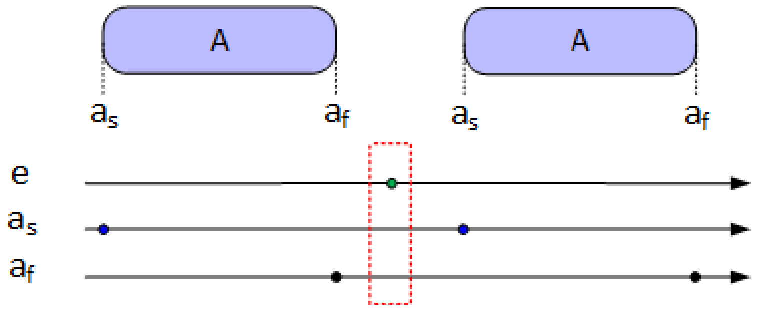


Figure 3.13: Depiction of e forbids A

Graphical Representation

Graphically, the forbids logical/causal/untimed pattern is shown in Figure 3.14 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification*. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e to be exclusive with interval $[a_s, a_f]$.

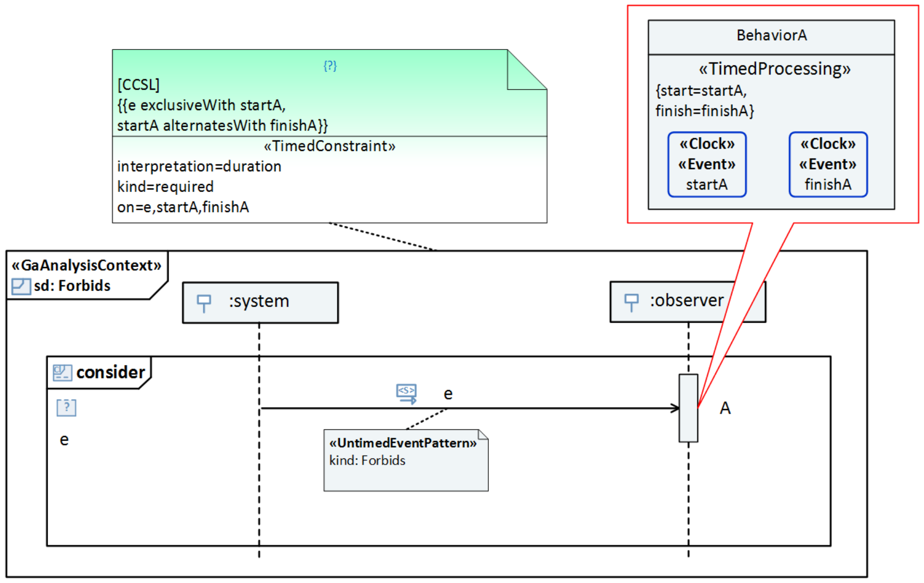


Figure 3.14: MARTE Model of e forbids A

Representation in CCSL

$$\begin{aligned} e \searrow \; as \boxed{\prec} af, \\ as \boxed{\sim} af \end{aligned}$$

Representation in CCSL (alternate form)

e sampledOn as precedes af,
as alternatesWith af

State Automaton

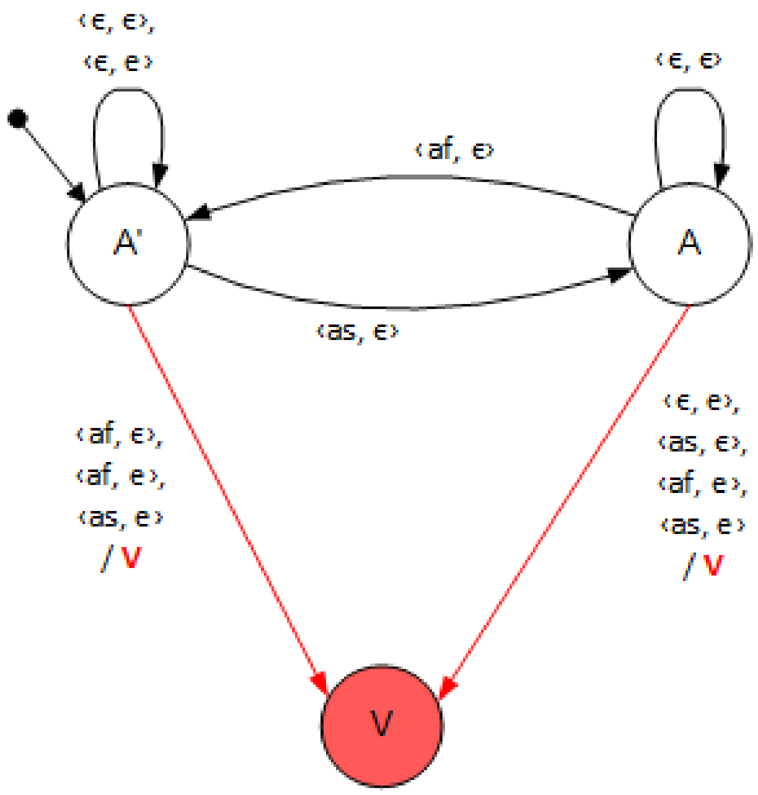


Figure 3.15: Automaton of e forbids A

SystemVerilog Observer

```
// e forbids A

module Forbids (
 input as,
 input af,
 input e,
 output violation
 );

 int unsigned FSM;
 reg v;

 always @ (as or af or e)
 begin
  case (FSM)
   //------------------------------------------------------------
   0 : // State A'
   begin
    if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;        // empty
    else if (as==1'b0 && af==1'b0 && e==1'b1) FSM=0;   // e
    else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;   // as
    else
    begin
     FSM=2;
     v=1'b1;
    end
   end
   //------------------------------------------------------------
   1 : // State A
   begin
    if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;        // empty
    else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;   // af
    else
    begin
     FSM=2;
     v=1'b1;
    end
   end
   //------------------------------------------------------------
   default : // Violation
   begin
    FSM=2;
    v=1'b1;
   end
  endcase
 end

 assign violation = v;

 initial
 begin
  FSM = 0;
  v = 0;
 end
endmodule
```

3.2.4 Contains Relation

Mathematical Notation

The *contains* relation is exactly opposite to the forbids state-event relation explained earlier. The relation *A contains e* implies e must occur while the state A is active. Hence given a partial ordering $\mathbb{S}$ having the state A ($[a_s, a_f]$) and an event e, the relation is expressed mathematically as $A \boxed{\supset} e$ which means $a_s > e$ and $e < a_f$, as shown in Figure 3.16.

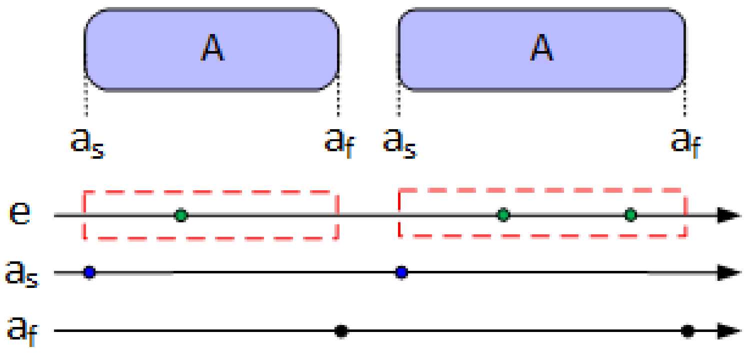


Figure 3.16: Depiction of A contains e

Graphical Representation

Graphically, the forbids logical/causal/untimed pattern is shown in Figure 3.17 as an annotated MARTE model. Sequence diagrams are used to represent state-event relations. Event e is shown by asynchronous message call between lifelines. Semantics of the state A is defined using *TimedProcessing* stereotype applied to the behavior of the *Behavior Execution Specification*. Semantics of the pattern are given in the constraint using CCSL which works with the logical clocks provided by the MARTE *TimedConstraint* stereotype. This behavior specified in the CCSL constraint requires event e to be coincident with interval $[a_s, a_f]$.

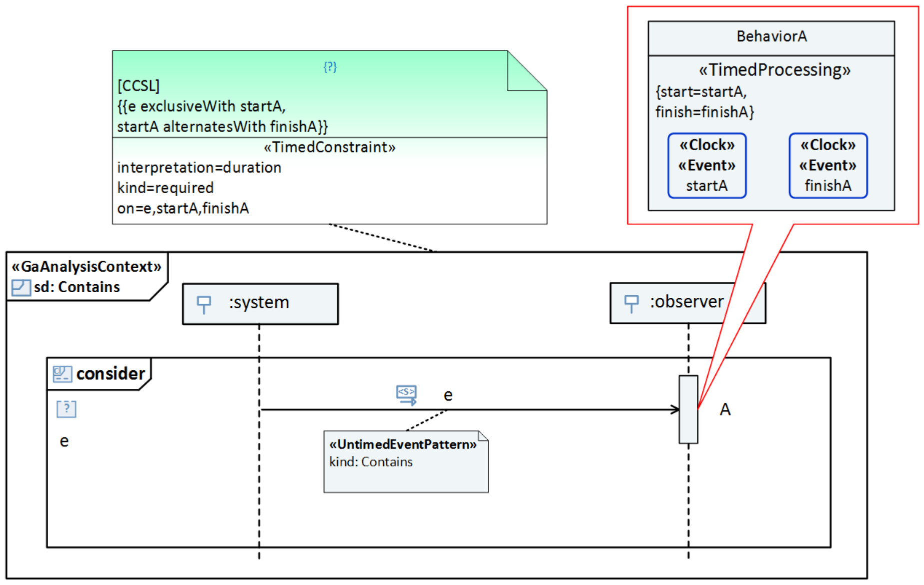


Figure 3.17: MARTE Model of A contains e

Representation in CCSL

$$\begin{aligned} as \searrow e \boxed{\prec} af, \\ as \boxed{\sim} af \end{aligned}$$

Representation in CCSL (alternate form)

as sampledOn e precedes af,
as alternatesWith af

State Automaton

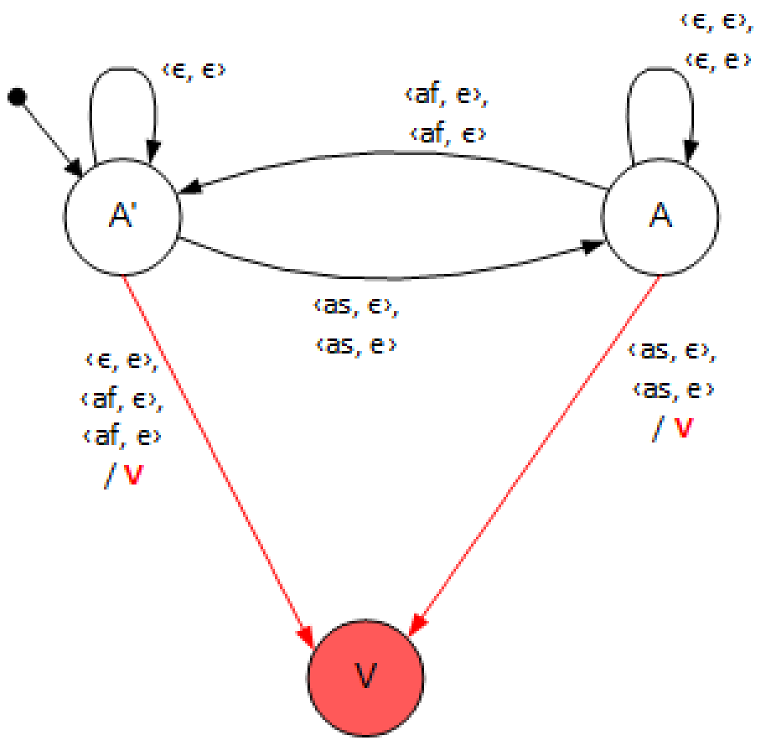


Figure 3.18: Automaton of A contains e

SystemVerilog Observer

```
// A contains e

module Contains (
 input as,
 input af,
 input e,
 output violation
 );

 int unsigned FSM;
 reg v;

 always @ (as or af or e)
 begin
  case (FSM)
```

```
    //-----------------------------------------------------------------
    0 : // State A'
    begin
     if (as==1'b0 && af==1'b0 && e==1'b0) FSM=0;        // empty
     else if (as==1'b1 && af==1'b0 && e==1'b0) FSM=1;  // as
     else if (as==1'b1 && af==1'b0 && e==1'b1) FSM=1;  // as, e
     else
     begin
      FSM=2;
      v=1'b1;
     end
    end
    //-----------------------------------------------------------------
    1 : // State A
    begin
     if (as==1'b0 && af==1'b0 && e==1'b0) FSM=1;        // empty
     else if (as==1'b0 && af==1'b0 && e==1'b1) FSM=1;  // e
     else if (as==1'b0 && af==1'b1 && e==1'b0) FSM=0;  // af
     else if (as==1'b0 && af==1'b1 && e==1'b1) FSM=0;  // af, e
     else
     begin
      FSM=2;
      v=1'b1;
     end
    end
    //-----------------------------------------------------------------
    default : // Violation
    begin
     FSM=2;
     v=1'b1;
    end
   endcase
  end

  assign violation = v;

  initial
  begin
   FSM = 0;
   v = 0;
  end
endmodule
```

References

[1] Mentor Graphics. Questa Advanced Simulator. [Online]. Available: https://www.mentor.com/products/fv/questa/

[2] J. Deantoni. TimeSquare: Logical Time Matters. [Online]. Available: http://timesquare.inria.fr/

[3] C. André, F. Mallet, and J. DeAntoni, "VHDL Observers for Clock Constraint Checking," in *Industrial Embedded Systems (SIES), 2010 International Symposium on*, July 2010, pp. 98–107.

[4] A. M. Khan, "TemLoPAC: Temporal and Logical Pattern Analyzer and Code-generator EMF Plugin." [Online]. Available: http://www.modeves.com/temlopac.html

[5] J.-R. Abrial, E. Börger, and H. Langmaack, "The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods," in *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the Book Grow out of a Dagstuhl Seminar, June 1995).* London, UK, UK: Springer-Verlag, 1996, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=647370.723887

[6] M. Al-Lail, W. Sun, and R. B. France, "Analyzing Behavioral Aspects of UML Design Class Models against Temporal Properties," in *Quality Software (QSIC), 2014 14th International Conference on*, Oct 2014, pp. 196–201.

[7] A. E. Haxthausen, "Automated Generation of Formal Safety Conditions from Railway Interlocking Tables," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 16, no. 6, pp. 713–726, Nov. 2014. [Online]. Available: http://dx.doi.org/10.1007/s10009-013-0295-9

[8] B. Cohen, S. Venkataramanan, A. Kumari, and L. Piper, *SystemVerilog Assertions Handbook: for Dynamic and Formal Verification*, 2nd ed. Palos Verdes Peninsula, CA, USA: VhdlCohen Publishing, 2010.

[9] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-state Verification," in *Proceedings of*

the 21st International Conference on Software Engineering, ser. ICSE '99. New York, NY, USA: ACM, 1999, pp. 411–420. [Online]. Available: http://doi.acm.org/10.1145/302405.302672

[10] J. F. Allen, "Maintaining Knowledge About Temporal Intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983. [Online]. Available: http://doi.acm.org/10.1145/182.358434

[11] C. André, F. Mallet, and R. de Simone, "Modeling Time(s)," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Springer Berlin Heidelberg, 2007, vol. 4735, pp. 559–573. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-75209-7_38

[12] J. Suryadevara, "Model Based Development of Embedded Systems using Logical Clock Constraints and Timed Automata," Ph.D. dissertation, Malardalen University, Sweden, 2013.

[13] F. Mallet, "Clock Constraint Specification Language: Specifying Clock Constraints with UML/MARTE," *Innovations in Systems and Software Engineering*, vol. 4, no. 3, pp. 309–314, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11334-008-0055-2

[14] ——, *Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015.* Wiesbaden: Springer Fachmedien Wiesbaden, 2015, ch. MARTE/CCSL for Modeling Cyber-Physical Systems, pp. 26–49. [Online]. Available: http://dx.doi.org/10.1007/978-3-658-09994-7_2

[15] F. Mallet and C. André, "UML/MARTE CCSL, Signal and Petri nets," INRIA, Research Report RR-6545, 2008. [Online]. Available: https://hal.inria.fr/inria-00283077v4