Shield Synthesis for Cyber Physical Systems

A B.Tech Project Report Submitted in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Technology

by

Samay Varshney
(180101097)

Siddhartha Jain
(180101078)

under the guidance of

Dr. Purandar Bhaduri

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Shield Synthesis for Cyber Physical Systems**" is a bonafide work of (**Roll No. 180101097 and 180101078**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Purandar Bhaduri**

Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam

Acknowledgements

We would like to express our gratitude to our supervisor **Dr. Purandar Bhaduri** for giving us the opportunity to explore this new field of Shield Synthesis for Cyber Physical Systems and the much needed zest to delve into some of the state-of-the-art works.

Contents

Chapter 1

Abstract & Introduction

1.1 Abstract

Cyber-Physical Systems are safety-critical real-time systems in which erroneous behaviour may inflict serious consequences. This motivates an additional layer of safety in the form of shield synthesis. Here, we explore the shield for timed properties, which is implemented after the controller and enforces the running system to be safe by interfering with system as low as possible, i.e. corrects the output of the system only when necessary. This timed shield construction can be extended to provide solutions to the recovery phase, i.e., the time between the violation of the specification and the time at which the system could get the control back.

Here, we also explore the bidirectional run-time enforcement in which the enforced policies depends not only on the status of the controller but also on the status of the plant.

In our work, we used timed shields to ensure safety in different types of safety-critical systems like controlling a platoon of cars, heart-pacemaker systems, which are very useful in day to day lives. We also used bidirectional run-time enforcement using enforcer synthesis for enforcing pacemaker policies.

1.2 Introduction

A run-time verification is an approach to detect violations of certain critical specifications while a system is executing. A run-time enforcement of certain specifications is the one where violations are not only detected but also overwritten such that the properties are always maintained.
We explored the **Bidirectional Run-time Enforcement** (bidirectional means monitoring both inputs and outputs of the controller and correcting thenm when necessary) of the heart-pacemaker cyber physical system which is bidirectional and closed-loop, in which interaction with an enforcer is made and safety properties are only enforced when the controller (pacemaker) fails to satisfy them.
The run-time enforcement for reactive systems is currently being explored. Enforcers for these systems must operate in the same reactive time step, unlike existing enforcers, which can cause buffering of events leading in delay of events. The shield synthesis in context to the pacemaker, is unidirectional in nature. While enforcing specifications/properties of cyber physical system, bi-directional enforcement is important.
There are several advantages that we would get because of using bidirectional run-time enforcement. Used in enforcing security policies. Enforcer suppresses malicious inputs from an attacker by modifying the inputs. For eg. in **Unmanned Aerial Systems**, to prevent an attacker feed-in bad inputs to take system control.

Shield synthesis is a general approach for implementing run-time enforcement in reactive systems. A timed shield is a component which ensures both correctness i.e. it corrects the output of the system with respect to the certain critical properties, and no unnecessary deviation i.e. the output of the shield should not deviate from those of system unnecessarily.
Games played on graphs with finite vertices have been a topic of study for various years with many applications like shield and controller synthesis. Considering, controller as a black-box, shield synthesis aims at producing a winning strategy that when executed with

an environment strategy always result in satisfaction of desired properties. An approach to solve this shield synthesis problem involves graph games.

The **shield synthesis** problem for reactive systems [KAB$^+$17] can be modelled into an interactive game of strategy between two players, *Player 0* is called the system and *Player 1* being the environment. These so-called **Graph Games** are played infinitely on a finite graph since there are no dead ends. Vertices are partitioned between players and the player owning the vertex decide the next move from that vertex. The moves in control of the system are controlled by the shield and are called controllable actions and the moves in control of the environment are called uncontrollable actions which can take any values irrespective of the shield.

There are different winning conditions depending on the required specification of the controller. It could be reachability, safety, etc. A strategy followed by a player is the method over the states to the collection of controllable actions or empty if no actions permitted. If a strategy allows the *Player 0 (the system)* to satisfy the required specification or achieve his goals, no matter what strategy the opponent *Player 1 (the environment)* follows, then it is called a *winning strategy* for *Player 0*.

Safety Games are a class of graph games which involves safety objectives. The condition in safety games is that *Player 0* is not allowed to enter a set of unsafe/bad vertices, i.e., the whole game play should be confined to a set of safe vertices. Shield follows safety game for implementing the winning strategy for the system.

Using the timed shield, we also explored the shields that provides the Guaranteed-Recovery (where within a finite time, the recovery phase ends) and Guaranteed-Time-Bounded Recovery (where after a given bounded time, the recovery phase ends) properties, which gives the control back to the system after the recovery phase.

There are several advantages that we would get because of using timed shield. Controllers are complex in nature so verification of them as a unit is tricky and takes more time but shield handles only the subset of properties and consider controller as a black-box so verification of them is easy. Shield provides an additional layer of safety to those safety critical properties, violations of them can lead to life or death problems.

1.3 Organization of The Report

This chapter introduces the problem statement covered in this report. We provided a brief overview of the topic and talked about the importance of bidirectional run-time enforcement and timed shield synthesis in real life. The rest of the chapters are arranged as follows: next chapter we discuss all the preliminaries that would help brush up the keywords required. In **Chapter 3**, we describe the methodology that we are following, we would briefly discuss the algorithms used and the idea that we worked on. In **Chapter 4** and **Chapter 5**, we presented implementation and the results as found in various examples and look at some real life applications where our work can be used. And finally in **Chapter 6**, we will conclude with the final remark and the future works.

Chapter 2

Preliminaries

The following section contains a background of the different terminologies that will be used in the later sections.

2.1 Specifications/Properties

A finite (resp. infinite) trace/word over a finite alphabet $\sum$ is a finite sequence $\sigma = a_1 \cdot a_2 a_n \cdots a_n$ (resp. infinite sequence $\Sigma = a_1 \cdot a_2 \cdot a_3 \cdots$) of elements of Σ. $\sum^{\infty}$ is defined as disjunction of both Σ^* and Σ^{ω}, where Σ^* represent the set of finite words and Σ^{ω} represent the set of infinite words.

A specification/property φ is a set $L(\varphi) \subseteq \Sigma^{\infty}$ of words/traces that satisfy specification. A program P $\models \varphi$ iff $L(P) \subseteq L(\varphi)$. A property φ represents a safety property if finite words that do not satisfy φ cannot be extended to words that satisfy φ. Specifications can be represented using **Timed Input/Output Automata (TIOA)**.

2.2 Discrete Timed Automata (DTA)

Discrete Timed Automata (DTA) are timed automata with integer valued clocks.

DTA is defined as a tuple:

$$A = (L, l_0, l_v, \Sigma, V, \triangle, F)$$

Here L represents the set of locations, l_0 represents the initial location, Σ denotes the alphabet, V represent the set of integer clocks, F $\subseteq$ L represents the set of accepting locations, and l_v is a non-accepting trap location, $\triangle$ is the transition relation.
Given specification defined as a DTA, **input DTA** is obtained by ignoring the outputs on the transitions in the DTA. All the clocks, transitions, locations etc are same in both.

2.3 Semantics of DTA

The semantics of a DTA, a transition system defined as a tuple:

$$[[A]] = (Q, q_0, \Sigma, \rightarrow, Q_F, q_v)$$

Here Q = L $\times$ N^V represents the set of states, q_0 = (l_0, X_0) represents the initial state where X_0 represent clock variable in V assigned to 0, Q_F = F $\times$ N^V denotes the set of accepting states, and q_v = l_v $\times$ N^V denotes the set of trap states.

2.4 Edit Functions

Given property φ, described as Discrete Timed Automata A_φ with semantics $[[A_\varphi]]$.
$editI_{\varphi I}$ is the edit function, used by the enforcer for editing inputs from the environment based on input property obtained from the property φ that enforcer want to satisfy.
$editO_\varphi$ is the function to edit the outputs from the controller based on the property φ.

- $editI_{\varphi I}(\sigma_I)$ is the set consisting of input events x in Σ_I such that the trace obtained by extending σ_I with x can be extended to some sequence which satisfies φ.

- $editO_\varphi(\sigma, x)$ is the set of output events y in Σ_O such that the input-output trace obtained by extending σ with (x,y) can be extended to some sequence which satisfies the property φ.

- $rand - editI_{\varphi I}(\sigma_I)$ returns a random element from $editI_{\varphi I}(\sigma_I)$, and if $editI_{\varphi I}(\sigma_I)$ is empty then it is undefined.
- $rand - editO_{\varphi}(\sigma, x)$ returns a random element from $editO_{\varphi}(\sigma, x)$, and if $editO_{\varphi}(\sigma, x)$ is empty then it is undefined.

2.5 Timed Input/Output Automata (TIOA)

Timed Input/Output Automata are timed automata with real value clocks.
TIOA is defined as a tuple:

$$A = (L, l_0, \Sigma^?, \Sigma^!, X, E, I).$$

Here, L is a collection of locations, l_0 represent the initial location, $\Sigma^?$ denotes the finite set of input actions, $\Sigma^!$ denotes the finite set of output actions, X represents a set of clocks, E is a set of transitions and I denotes the set of invariants assigned to each location.
States of a TIOA are pairs of discrete location and clock values which are real in nature. Two kinds of transitions are allowed: **discrete transitions**, in which there is an instant change in locations with some resetting of clocks and **delay transitions**, in which we remain in same location with increase in clock values.

2.6 Timed Labeled Transitions System (TLTS)

Timed Labeled Transitions System (TLTS) defines the semantics of Timed Input/Output Automata.
It is defined as a tuple:

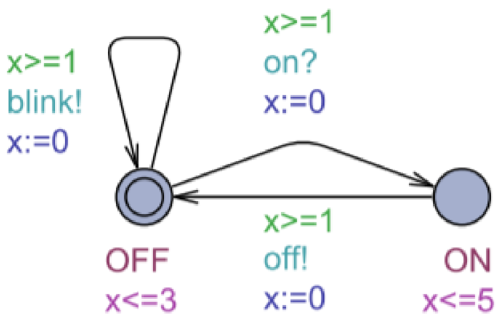
$$[[A]] = (Q, q_0, \rightarrow)$$

Here, Q is the collection of all states, the initial state is denoted by $q_0 = (l_0, 0)$ and transition relation is denoted by $\rightarrow$.

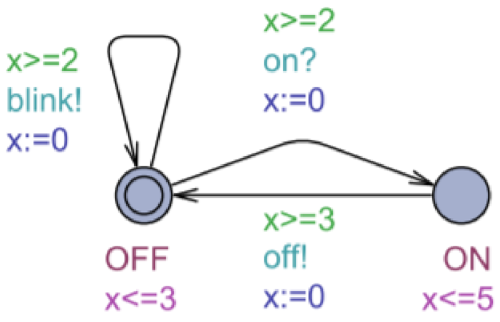
2.7 Refinement of Timed Automata

Refinement (denoted by $\leq$) of the Timed Automata is represented as the containment of timed behaviour. i.e. here, behaviour of shielded system is a subset of behaviour of specification.

For e.g, Timed Automata B refines Timed Automata A (B $\leq$ A) means L(B) $\subseteq L(A)$.



(a) A



(b) B

Fig. 2.1: Refinement of A by B (B $\leq$ A)

2.8 Timed Game Automata (TGA)

Timed shield is synthesized by solving a timed game, played by two players: System and Environment, represented using **Timed Game Automata (TGA)**.

It is a Timed Automata defined as a tuple:

$$G = ((L, l_0, \Sigma, X, E, I), i, \Sigma_{\cup}, \Sigma_C).$$

In this, output actions are divided into set of **controllable** and **uncontrollable actions**. The former is controlled by System and later by environment.

2.9 Safety Games

A safety game is a game with safety acceptance condition defined as follows:

$$SAFETY(S) := \{\rho \in Q^\omega \mid Occ(\rho) \subseteq S\}$$

Here, S is a collection of safe states such that $S \subseteq Q$m where Q refers to the total states in the TGA. Q^ω denotes various plays or traces possible in the TGA. $Occ(\rho)$ denotes the states reached at least once in the trace. The aim of *System/Player 0* is to remain confined in S always.

2.10 Strategy

In this paper, we will be focusing on 2-player games. So, $P = \{0, 1\}$. We will be looking at the games from perspective of *Player 0 (system).*

A strategy for the *System/Environment* in a timed game G is a function σ, over the states of TGA to the set of controllable (in case of System)/uncontrollable (in case of environment) actions or a nothing symbol (in case no action possible).

Depending on the safety objective of the game, the winning condition could be a reachability condition, safety condition, etc but here we will talk about safety condition only since it is used by the shield.

2.11 Winning Strategy

Given, a timed game G, a strategy σ is called a winning strategy for the *Player i* from a state $q \in Q$ if every trace starting from state q following the strategy σ satisfies the winning condition for that player, irrespective of what strategy is followed by the opponent. In case of system, the winning condition is to satisfy the specification to be followed.

For safety games, a winning strategy σ for the system from state q makes sure that if a trace starts from state q following that strategy, no unsafe state is reached during the play irrespective of what moves are chosen by environment.

Chapter 3

Methodology

So far, we have looked at the basic terminologies that would be used in the rest of the paper. We now turn our attention towards our main idea. We start by briefly discussing the algorithms used in the implementation. This includes algorithm for implementing the bidirectional synchronous enforcer for properties expressed as DTA. Algorithm for implementing timed shields for properties expressed as TIOA and at last we discuss algorithm for implementing timed shield with the ability to recover from faults.

3.1 Algorithms Used

In this section, we briefly discuss about the algorithms [PRS$^+$17] that were employed in our implementation.

3.1.1 Bidirectional run-time enforcement

We discuss the algorithm 1 in this section which will be used to implement the bidirectional run-time enforcement for the systems which are reactive in nature given the property express as DTA.

In the algorithm, q denotes current state, t denotes current timestamp, *read_in_chan* and *read_out_chan* are methods which will read input and output channels respectively.

Algorithm 1: Bidirectional run-time enforcement for reactive systems

```
Input: Automata $A_\varphi$ and Input Automata $A_{\varphi \mathrm{I}}$
Output: Edited input-output sequence which satisfies specification $\varphi$
$q \leftarrow q_0$;
$t \leftarrow 0$;
while True do
    $x_t = read\_in\_chan()$;
    if $\exists {\sigma_I}' \in {\Sigma_I}^* : q \xrightarrow{x_t \cdot {\sigma_I}'} q' \wedge q' \in Q_F$ then
        ${x_t}' = x_t$;
    end
    else
        ${x_t}' = rand\text{-}editI_{A_I}(q)$;
    end
    $ptick({x_t}')$;
    $y_t = read\_out\_chan()$;
    if $\exists {\sigma_I}' \in \Sigma^* : q \xrightarrow{({x_t}', y_t) \cdot \sigma'} q' \wedge q' \in Q_F$ then
        ${y_t}' = y_t$;
    end
    else
        ${y_t}' = rand\text{-}editO_{A_I}(q, {x_t}')$;
    end
    $release(({x_t}', {y_t}'))$;
    $q \leftarrow q''$ where $q \xrightarrow{({x_t}', {y_t}')} q'' \wedge q'' \notin q_v$
    $t \leftarrow t + 1$;
    end
```

We discuss an algorithm which takes inputs as automata A_φ and Input automata $A_{\varphi I}$. Algorithm is an infinite loop which is executed at every tick. t maintains the time step which is set to 0 initially, the present state of both the automata is denoted by q, initialized to q_0. As both automata differ only in that outputs are ignored in the input automata so both have same initial states.

Function **read_in_chan** is a function to read inputs from the plant and assigned its value to x_t. The algorithm then check if an accepting state can be achieved or not from the present state of automata upon extending with any trace $\sigma \in \Sigma_I$, In case if it is possible, then there is no need to edit the inputs from the plant and we assign $x_t^{'}$ to x_t. If not then we modify the input event with **rand-editI**$_{A_I}$**(q)**.

Function **ptick** represent the synchronous controller as a black-box. Algorithm pass the modified input to the controller through ptick function and then read the output from the controller through the function **read_out_chan** (a method to read outputs from the controller) and assign to y_t. The algorithm then checks if or not an accepting state can be achieved from the present state upon extending with an trace $\sigma^{'}$. In case if it possible, then there is no need to edit the outputs from the controller and we assign $y_t^{'}$ to y_t. If not then we modify the output event with **rand-editO**$_{A_I}$**(q,** x_t**')**.

Function release gives the output of the enforcer. Current state is updated before going through next iteration, which is the state achieved on receiving ($x_t^{'}$,$y_t^{'}$). At last, time t is incremented.

3.1.2 Construction of Timed Shields

Here, we will discuss about the timed shield [BJK$^+$20] where we create a shield using some strategy using safety games for a given reactive system. Consider specification as Spec, timed system as Sys, and Spec a monitor for Spec. We construct a timed shield as follows:

1. **Construct monitor mSpec':** mSpec' is created to make the distinction between the system and the shield outputs and is used to check that the shield outputs are correct.

It is constructed by making a duplicate of mSpec with all outputs being primed.

2. **Construct Ctr automaton:** The automata Ctr produces the primed outputs i.e. component to handle the control options of the shield. Ctr receive information from mSpec about whether the output of the system is wrong or the mSpec winning region is left. It is a component that handle the no unnecessary deviation property of the shield. Ctr perform mainly three actions:

    - **pre fault actions:** Before fault detection, Ctr should not modify the output of the system and pass as it is.
    - **post fault actions:** After fault detection, Ctr takes the control and can choose any output to prevent the system to go to error state.
    - **last-chance actions:** Whenever the system is in such a state that any wait would make the system go out of the winning region (leading to the dissatisfaction of the invariant of the location), Ctr can choose a suitable action such that mSpec' does not go into the error state.

3. **Construction of the timed safety game (G):** The timed safety game G is constructed by the composition as follows:

$$G = mSpec|mSpec'|Ctr$$

    mSpec observes whether the outputs of the System are correct w.r.t. the specification, mSpec' observes the correctness of the shield's outputs, and Ctr produces the primed outputs by enforcing that the shield does not unnecessarily deviate.

4. **Compute a winning strategy ω_s of Timed Game Automata (G):** Shields ensure correctness i.e. that shielded system never violates the Spec. Solving the safety game constructed in previous step with respect to the query construct a winning strategy (ω_s), which is used to construct a timed shield.

5. **Timed shield Sh_s construction:** A shield Sh_s is the timed automata which is a composition of TGA (G) with the strategy (ω_s), computed in previous step, denoted by $G||\omega_s$, leading to restriction of all unsafe transitions.

3.1.3 Timed Post-Shield construction with Recovery Guarantees

Now we discuss the synthesis procedure of timed post-shields with guaranteed recovery and time-bounded recovery. Consider specification as mSpec, specification monitor as mSpec, and $Spec^f = Spec^{f_1} \cdot Spec^{f_2} \cdots Spec^{f_n}$ as the collection of error models. Timed shield is constructed with guaranteed recovery or with time guaranteed recovery as follows:

1. **Construction of mSpec' and Ctr:** Same as step 1 and 2 of the previous algorithm.

2. **Construction of the monitors mSpec^{f_i}:** For all the fault models Spec^{f_i}, compute monitors mSpec^{f_i}.

3. **Construction of the timed Game G:** It is created as follows:

$$G = mSpec^{f_1}|\cdots|mSpec^{f_i}|mSpec|mSpec'|Ctr$$

   In timed game, we explore the system recovery w.r.t. the fault models mSpec^{f_i} along with mSpec to ensure correctness of the original property, mSpec' to ensure the correctness of the shield and Ctr to ensure there is no unnecessary deviations by the shield.

4. **Compute ω_g strategy of G for guaranteed recovery:** For recovery, we have to finally reach a state where all the error models together with the Spec' are in the same states. This can be attained with a property, stating that when we encounter a error, it will result in recovery eventually.

$$control : A\Box\ mSpec.ERR \wedge \neg mSpec'.ERR\ leadsto$$
$$(\forall i.[mSpec^{f_i}.ERR\ \vee\ (mSpec'.i == mSpec^{f_i}.l\ \wedge\ mSpec'.v == mSpec^{f_i}.v)])$$

5. **Compute ω_{gt} strategy of G for time guaranteed recovery:**

   Query to compute the strategy is as follows:

$$control : A\square\ mSpec.ERR \wedge \neg mSpec'.ERR\ \ leadsto_{\leq T}$$
$$(\forall i.[mSpec^{f_i}.ERR\ \ \vee\ \ (mSpec'.i == mSpec^{f_i}.l\ \ \wedge\ \ mSpec'.v == mSpec^{f_i}.v)])$$

6. **Construction of the timed shield:** We construct the timed shield with guaranteed recovery by composition of timed game with the strategy ω_g i.e. $G||\omega_g$ and a shield with time guaranteed recovery by composition of timed game with the strategy ω_{gt} i.e. $G||\omega_{gt}$.

Chapter 4

Implementation

In this chapter, we shall look at the implementation of few real life applications of the enforcer and the timed shield. We start by the example of the enforcer for the safety properties of the heart-pacemaker cyber-physical system and specifications can be found here [PJL$^+$12]. Next, we discuss about the timed shield for safety properties of the heart-pacemaker system, platooning of cars, and the lamp. We represent the shield using the two player timed game and compute the winning strategy using the **UPPAAL STRATEGO** tool to enforce the safety properties. These enforcer and timed shield are implemented using the algorithms mentioned in the chapter 3. We shall look at the enforcer for the heart-pacemaker system now.

4.1 Enforcer for Heart-Pacemaker system

We used a tool called easy-rte for implementation of bidirectional run-time enforcement and for checking the validity of the specification and automata created by us. Here we are only showing 2 properties which we have implemented.

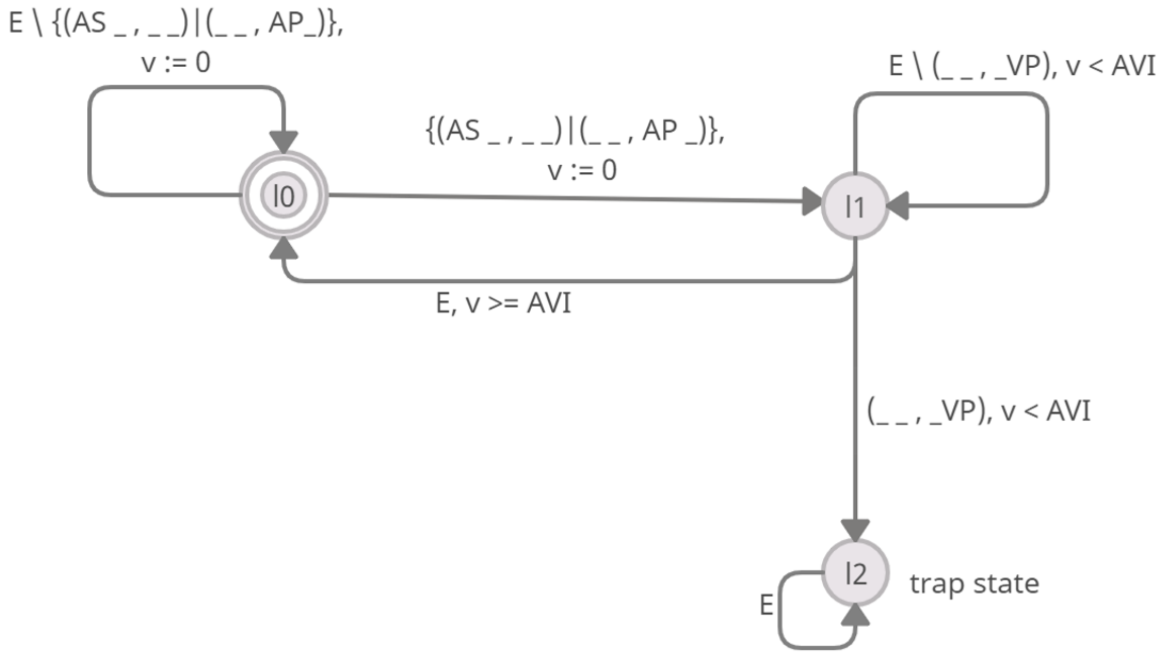
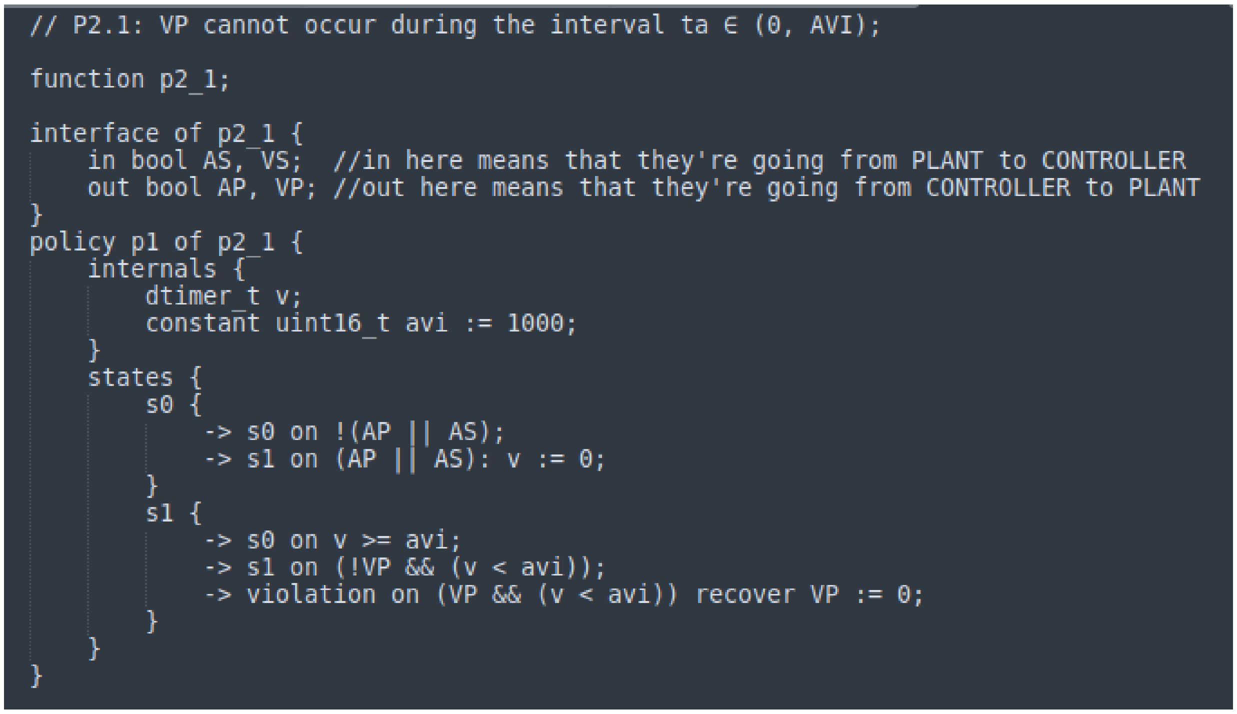


Fig. 4.1: Automata: Property 1

4.1.1 Property 1: Ventricular Pace should not happen within the interval $t_a \in$ (0,AVI)

t_a measures the delay since previous atrial event happens. AVI stands for Atrioventricular Interval. We express this property using the Discrete Time Automata (DTA). Input = {AS,VS} and output = {AP,VP} where AS stands for Atrial Sense i.e. the sensor for sensing the electrical pulse which contracts the atria walls, and VS stands for Ventricular Sense i.e. the sensor for sensing the electrical pulse which contracts the ventricle walls. AP and VP are the signals produced by the pacemaker for pacing the atrium and ventricle respectively. Location l_0 is the initial location as well as the only accepting location, location l_2 is the non-accepting trap location. v is the integer valued clock. Whenever a atrial event (AS or AP) happens, we take the transition from the location l_0 to location l_1 with resetting of clock v to 0. On encountering the VP output event within the AVI limit lead to violation of safety property and we move from location l_1 to location l_2 and on rest of the input/output events we remain in the same location increasing the clock value. When the clock value reaches the AVI limit, we move back to location l_0 as property is satisfied completely and wait for next atrial event to happen to repeat the process.



```
// P2.1: VP cannot occur during the interval ta ∈ (0, AVI);

function p2_1;

interface of p2_1 {
    in bool AS, VS;  //in here means that they're going from PLANT to CONTROLLER
    out bool AP, VP; //out here means that they're going from CONTROLLER to PLANT
}
policy p1 of p2_1 {
    internals {
        dtimer_t v;
        constant uint16_t avi := 1000;
    }
    states {
        s0 {
            -> s0 on !(AP || AS);
            -> s1 on (AP || AS): v := 0;
        }
        s1 {
            -> s0 on v >= avi;
            -> s1 on (!VP && (v < avi));
            -> violation on (VP && (v < avi)) recover VP := 0;
        }
    }
}
```

Fig. 4.2: Code: Property 1

Fig 4.2 is the descriptive version of the automata accepted by the tool. First, we declare the input (AS,VS) and output (AP,VP) variables. Then we declare the clock v and AVI as constant. We define the locations as s0 and s1 and the violation location and the detail of all the transitions as shown in the automata with the recover statement to handle the violations.

4.1.2 Property 2: Atrial Pace should not happen within the interval $t_v \in$ (0, LRI - AVI), Atrial Sense should not occur during interval $t_v \in$ (0, LRI - AVI), and Atrial Pace should happen at tv = LRI - AVI

t_v measures the delay since previous ventricular event happens. LRI stands for Lower Rate Interval. AVI stands for Atrioventricular Interval. We express this property using the

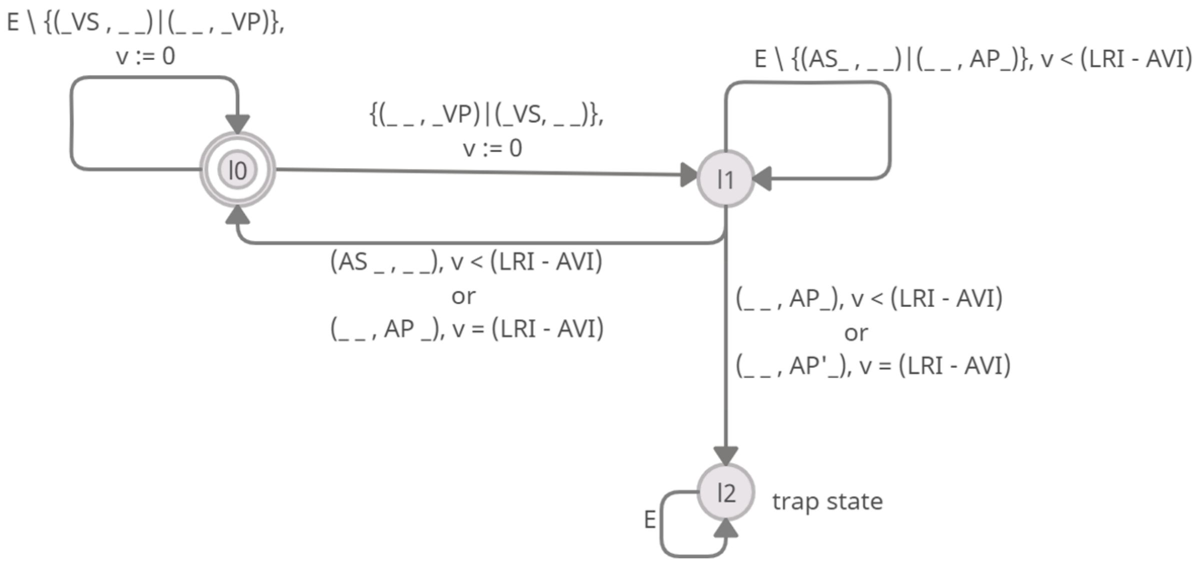
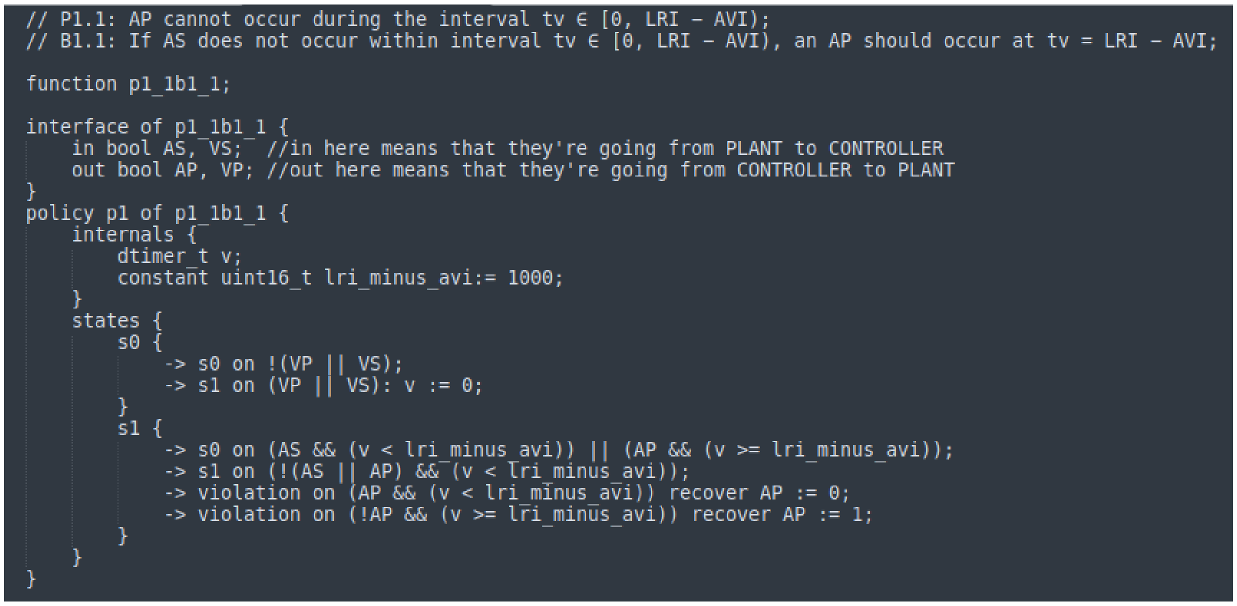

E \ {(_VS , _ _) | (_ _ , _VP)},
v := 0
{(_ _ , _VP) | (_VS, _ _)},
v := 0
E \ {(AS_ , _ _) | (_ _ , AP_)}, v < (LRI - AVI)
l0
l1
(AS _ , _ _), v < (LRI - AVI)
or
(_ _ , AP _), v = (LRI - AVI)
(_ _ , AP_), v < (LRI - AVI)
or
(_ _ , AP'_), v = (LRI - AVI)
l2
trap state
E

Fig. 4.3: Automata: Property 2

Discrete Time Automata (DTA). Input = {AS,VS} and output = {AP,VP}.

Location l_0 is the initial location as well as the only accepting location, location l_2 is the non-accepting trap location. v is the integer valued clock. Whenever a ventricular event (VS or VP) happens, we take the transition from the location l_0 to location l_1 with resetting of clock v to 0. On encountering the AP output event, within the LRI-AVI ticks, or not occurring of AP event at LRI-AVI if AS doesn't occur within LRI-AVI, lead to violation of safety property and we move from location l_1 to location l_2 and on rest of the input/output events we remain in the same location increasing the clock value. When the clock value is within LRI-AVI ticks, and AS is encountered, or if clock is LRI-AVI ticks and AP is encountered, we move back to location l_0 as property is satisfied completely and wait for next ventricular event to happen to repeat the process.

Fig 4.4 is the descriptive version of the automata accepted by the tool. First, we declare the input (AS,VS) and output (AP,VP) variables. Then we declare the clock v and AVI as constant. We define the locations as s0 and s1 and the violation location and the detail of all the transitions as shown in the automata with the recover statement to handle the violations.



```
// P1.1: AP cannot occur during the interval tv ∈ [0, LRI − AVI);
// B1.1: If AS does not occur within interval tv ∈ [0, LRI − AVI), an AP should occur at tv = LRI − AVI;

function p1_1b1_1;

interface of p1_1b1_1 {
    in bool AS, VS;  //in here means that they're going from PLANT to CONTROLLER
    out bool AP, VP; //out here means that they're going from CONTROLLER to PLANT
}
policy p1 of p1_1b1_1 {
    internals {
        dtimer_t v;
        constant uint16_t lri_minus_avi:= 1000;
    }
    states {
        s0 {
            -> s0 on !(VP || VS);
            -> s1 on (VP || VS): v := 0;
        }
        s1 {
            -> s0 on (AS && (v < lri_minus_avi)) || (AP && (v >= lri_minus_avi));
            -> s1 on (!(AS || AP) && (v < lri_minus_avi));
            -> violation on (AP && (v < lri_minus_avi)) recover AP := 0;
            -> violation on (!AP && (v >= lri_minus_avi)) recover AP := 1;
        }
    }
}
```

Fig. 4.4: Code: Property 2

4.2 UPPAAL

We have used UPPAAL STRATEGO which implements algorithms for solving games with reachability and safety properties based on timed game automata, producing non-deterministic safe strategies, as a tool for implementing the timed shields in pacemaker, platooning of car models etc. It uses and performs the model checking of the models mSpec, mSpec', Shield for simulation and uses the query in the verifier for creating the winning strategy which in composition with the timed game automata make sure that the safety property is never violated.

4.3 Timed Shield

4.3.1 Timed shield for Platooning of Cars

It is a real life application where one car is following other car while maintaining some range of distance between the two.

Fig 4.5 represent the system which is the ego car behind the front car which is the environment. As we regard the environment/system as a **black-box** so all its actions are uncontrollable in nature indicated by dotted edges in the fig 4.5.

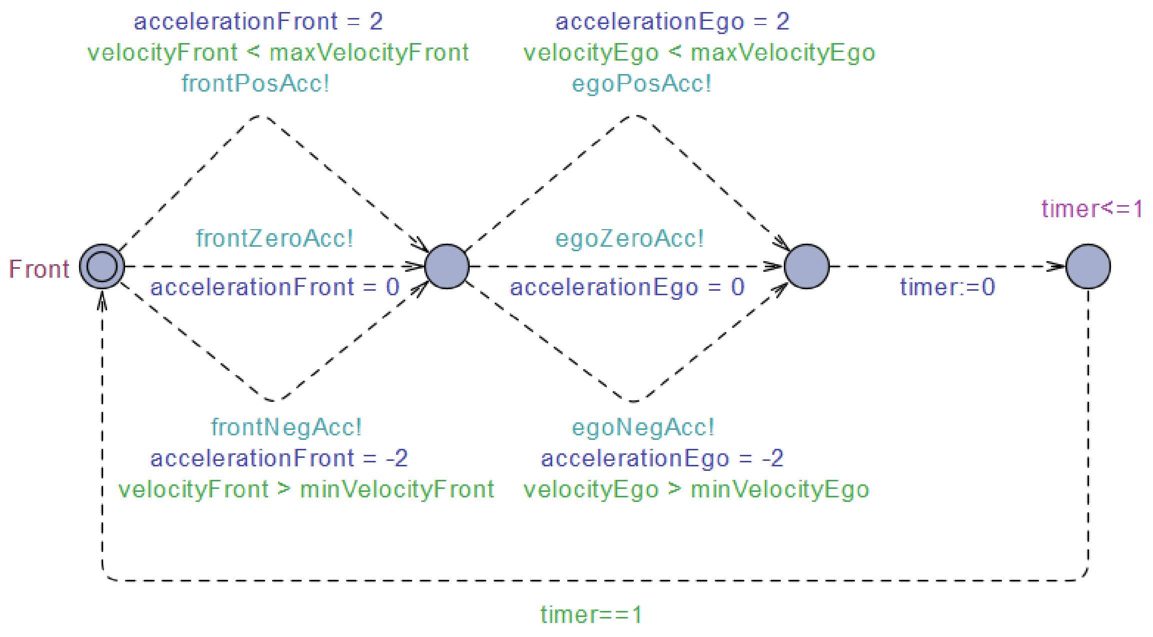


Fig. 4.5: System & Environment

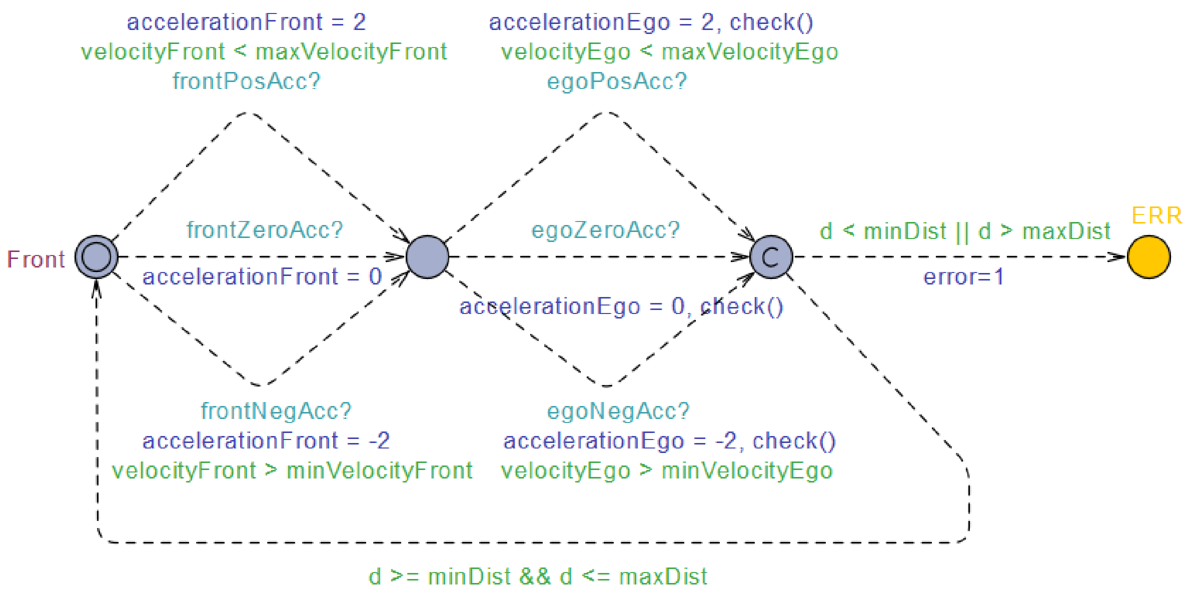

accelerationFront = 2
velocityFront < maxVelocityFront
frontPosAcc?
accelerationEgo = 2, check()
velocityEgo < maxVelocityEgo
egoPosAcc?
Front
frontZeroAcc?
accelerationFront = 0
egoZeroAcc?
accelerationEgo = 0, check()
d < minDist || d > maxDist
error=1
ERR
frontNegAcc?
accelerationFront = -2
velocityFront > minVelocityFront
egoNegAcc?
accelerationEgo = -2, check()
velocityEgo > minVelocityEgo
d >= minDist && d <= maxDist

Fig. 4.6: Specification

First front vehicle (environment) chooses a particular acceleration of its choice and then the system along with the shield chooses the acceleration and then we wait for 1 second and repeat the process.

Fig 4.6 represent the specification automata of the property that distance between the cars remain within the safe distance i.e. it is never more than maxDis and less than minDis. First, front vehicle chooses the acceleration (-2,0,2 m/s^2) then ego vehicle chooses the particular acceleration (-2,0,2 m/s^2). Then check for violation is performed and cycle repeats. Function **check()** determine the distance between the cars on selecting the particular acceleration to check whether it will lead to the violation of the safety property.

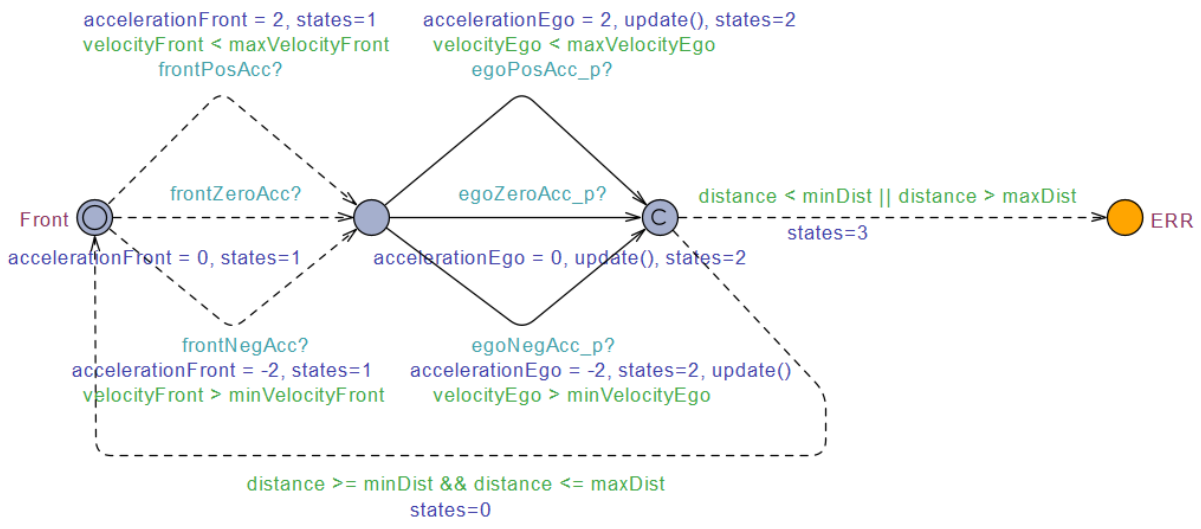

accelerationFront = 2, states=1
velocityFront < maxVelocityFront
frontPosAcc?
accelerationEgo = 2, update(), states=2
velocityEgo < maxVelocityEgo
egoPosAcc_p?
frontZeroAcc?
egoZeroAcc_p?
distance < minDist || distance > maxDist
states=3
Front
ERR
C
accelerationFront = 0, states=1
accelerationEgo = 0, update(), states=2
frontNegAcc?
accelerationFront = -2, states=1
velocityFront > minVelocityFront
egoNegAcc_p?
accelerationEgo = -2, states=2, update()
velocityEgo > minVelocityEgo
distance >= minDist && distance <= maxDist
states=0

Fig. 4.7: Primed Specification used by the Shield

Fig 4.7 represent the specification monitor for the primed output to be used by the shield to check if it produces the correct output or not. It is same as mSpec but with the primed output which is controlled by the shield to make sure property never gets violated. Function **update()** calculates the distance between the cars and their respective velocities for the selected acceleration of the cars.

Fig 4.8 is the Ctr automata that we discuss in chapter 3. It is the actual shield that make sure that there is **no-unnecessary-deviation** by the shield. First there is a pre-fault actions, which passes the output of the shield as it is (left most arrows), second there is a post-fault actions, in this shield can choose any possible outputs possible at any time, third there is a last-chance actions, which result when there is going to be violation of the invariants of the locations and shield can choose a particular action to avoid it.

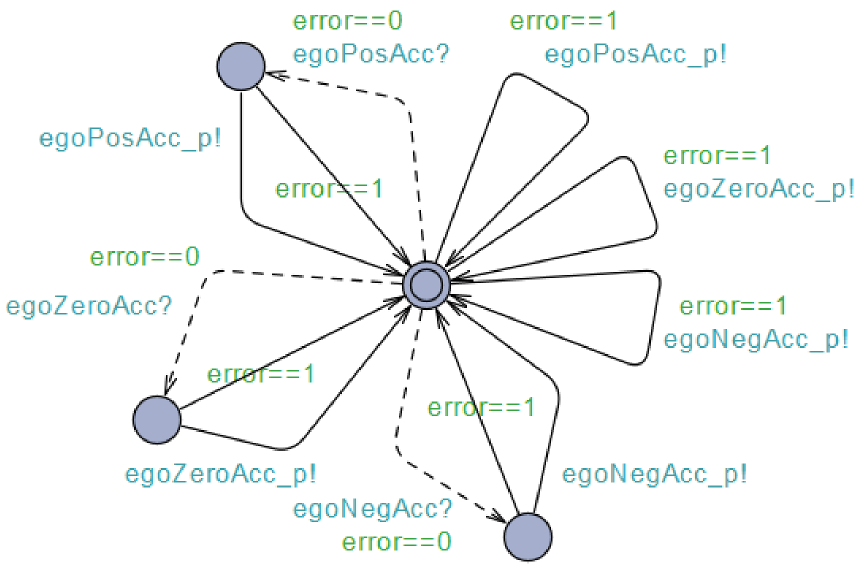


Fig. 4.8: Shield for correcting the outputs

We implemented all these automata in the UPPAAL STRATEGO tool, and using the verifier available in the tool, we compute the **winning strategy** which with composition of all these automata result in the automata which will satisfy he property no what what strategy environment chooses.

4.4 Timed Shield with Recovery Mechanism for Pacemaker

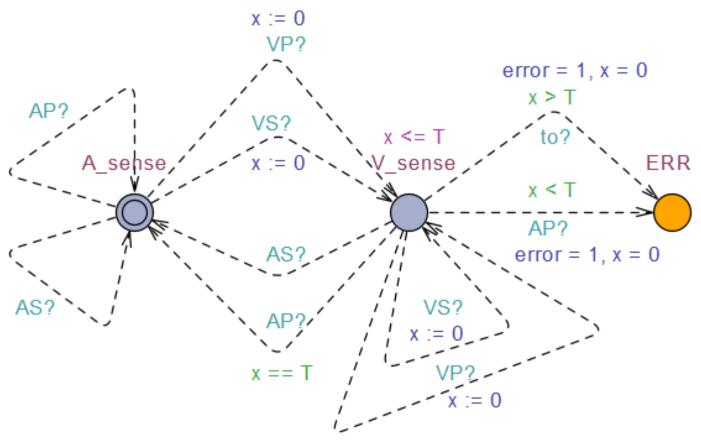


Fig. 4.9: Specification

Fig 4.9 is the automata to represent the property: AP should not happen within the interval $t_a \in$ (0, LRI - AVI). First, we move from location A_sense to V_sense on encountering the ventricular event. If AP occur within T (LRI-AVI) times after the transition then it leads to violation and we move to error state otherwise we move back to location A_sense and cycle repeats. This is uncontrollable in nature.

Fig 4.10 represent the specification monitor for the primed output to be used by the shield to check if it produces the correct output or not. It is same as mSpec but with the primed output which is controlled by the shield to make sure property never gets violated.

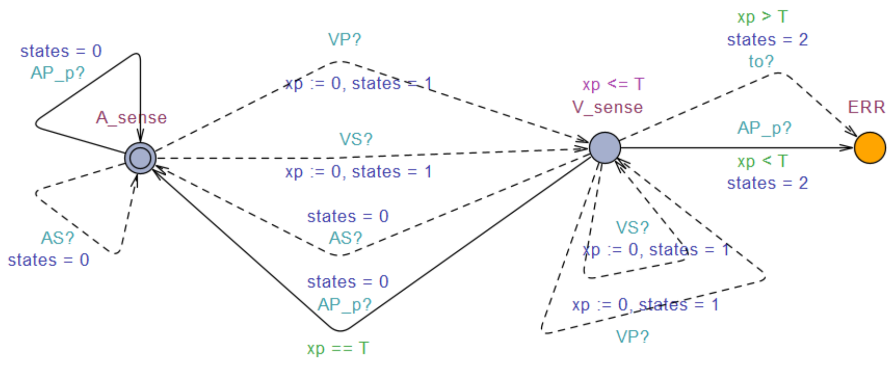


Fig. 4.10: Primed Specification

Fig 4.11 is the Ctr automata that we discuss in chapter 3. It is the actual shield that make sure that there is no-unnecessary-deviation by the shield. First there is a pre-fault actions, which passes the output of the shield as it is (left most arrows), second there is a post-fault actions, in this shield can choose any possible outputs possible at any time, third there is a last-chance actions, which result when there is going to be violation of the invariants of the locations and shield can choose a particular action to avoid it.

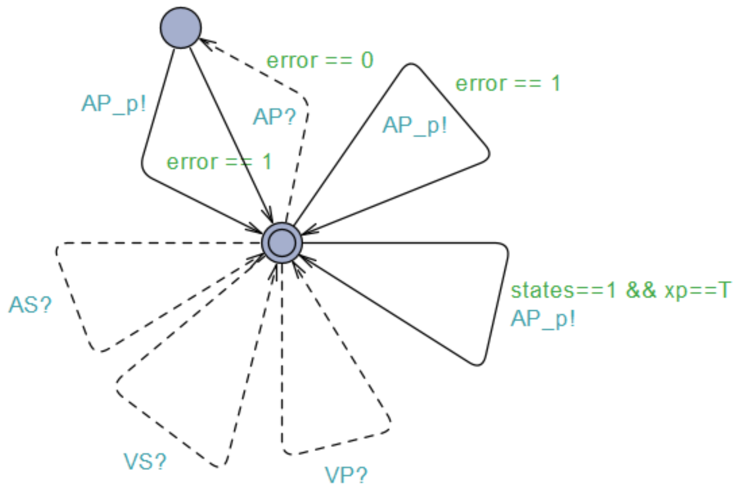


Fig. 4.11: Shield

Figures 4.12 and 4.13 represent the error/fault models to capture the errors that the system encounter during the execution so that the system and the shield can synchronize with each other after some time if the fault is captured by the fault models.

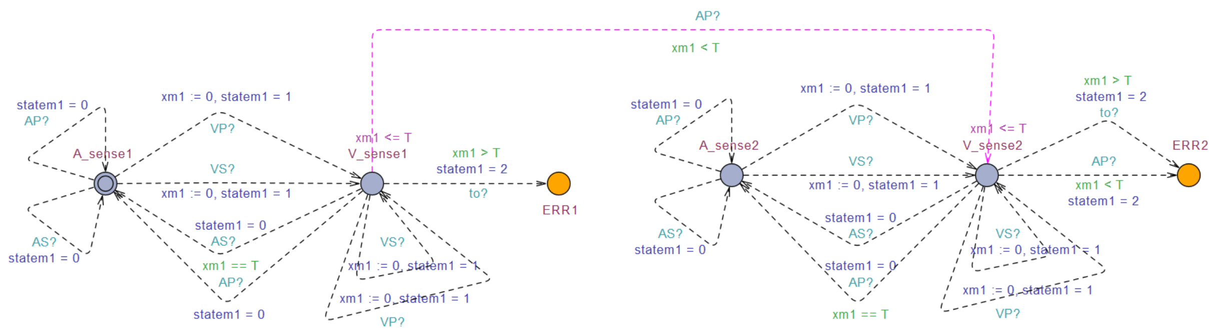


Fig. 4.12: Fault Model 1

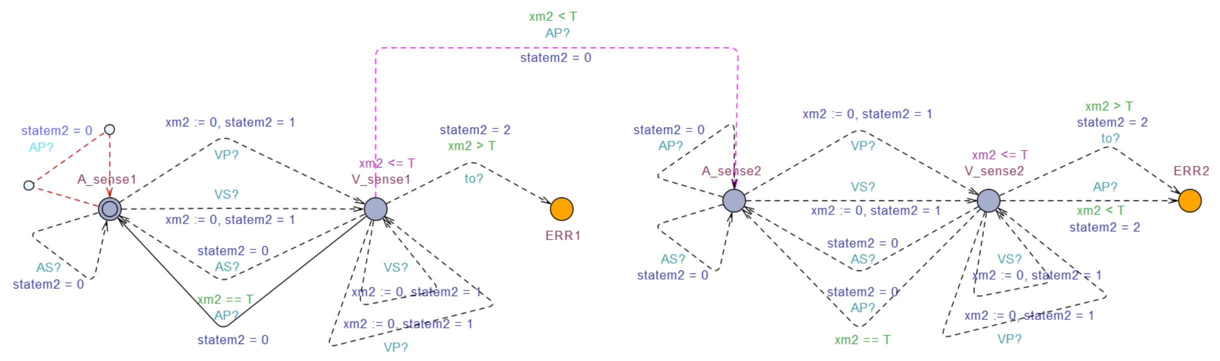


Fig. 4.13: Fault Model 2

Fig 4.14 represent the recovery mechanism automata so that the system and the shield can go back to sync with each other, with help of boolean variables recovered and successful which are made true after recovery is made.

We create the query to run in the UPPAAL STRATEGO tool as follows:

$$strategy\ \ REC = control : A[]\ \ not\ \ mSpec\_p.ERR\ \ and$$
$$((\text{not mSpec.ERR})\quad \text{or}\quad \text{x} \leq T1\ \ or\ \ recovered)$$

This query creates the winning strategy which in composition with all the automata create the shielded system.

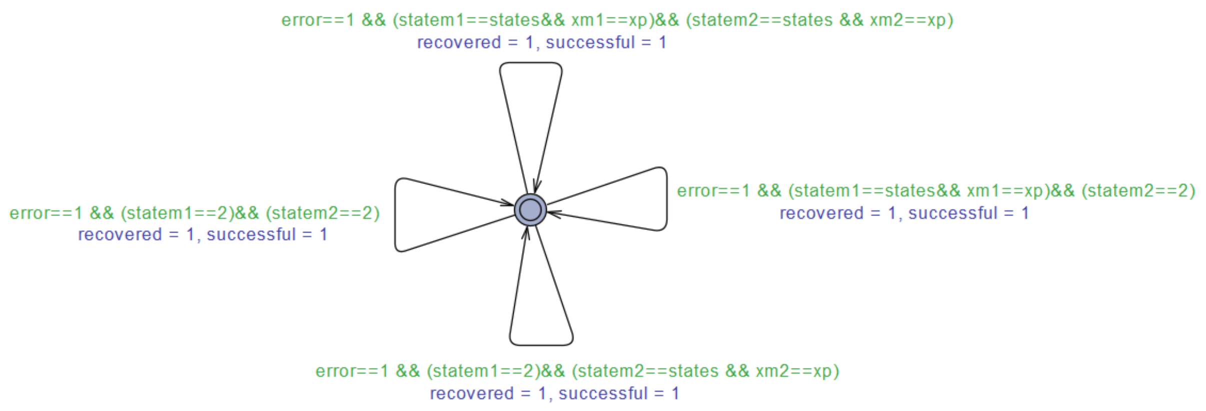

error==1 && (statem1==states&& xm1==xp)&& (statem2==states && xm2==xp)
recovered = 1, successful = 1
error==1 && (statem1==2)&& (statem2==2)
recovered = 1, successful = 1
error==1 && (statem1==states&& xm1==xp)&& (statem2==2)
recovered = 1, successful = 1
error==1 && (statem1==2)&& (statem2==states && xm2==xp)
recovered = 1, successful = 1

Fig. 4.14: Recovery Model

All the examples of the enforcer and timed shield that we implemented can be found here.

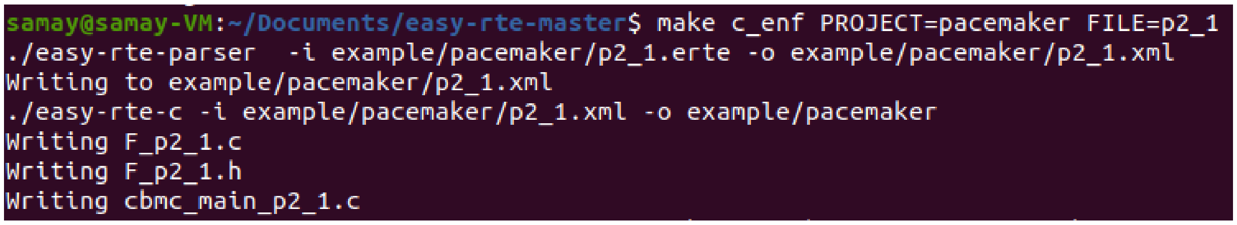
Chapter 5

Results

In this chapter, we explore the results that was observed when we run the bidirectional run-time enforcement algorithm and timed shield algorithm with recovery guarantee on different types of examples used in daily lives.

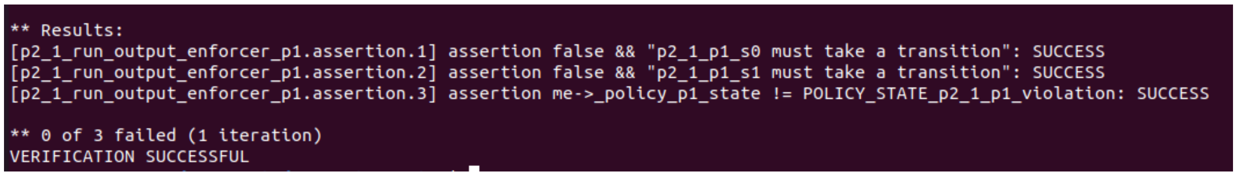
5.1 Enforcer for Heart-Pacemaker System

5.1.1 Property 1: Ventricular Pace should not happen within the interval $t_a \in$ (0,AVI)



```
samay@samay-VM:~/Documents/easy-rte-master$ make c_enf PROJECT=pacemaker FILE=p2_1
./easy-rte-parser  -i example/pacemaker/p2_1.erte -o example/pacemaker/p2_1.xml
Writing to example/pacemaker/p2_1.xml
./easy-rte-c -i example/pacemaker/p2_1.xml -o example/pacemaker
Writing F_p2_1.c
Writing F_p2_1.h
Writing cbmc_main_p2_1.c
```

Fig. 5.1: Property 1



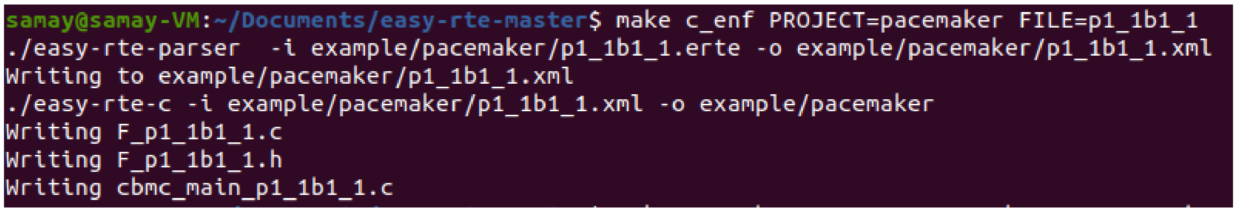
```
** Results:
[p2_1_run_output_enforcer_p1.assertion.1] assertion false && "p2_1_p1_s0 must take a transition": SUCCESS
[p2_1_run_output_enforcer_p1.assertion.2] assertion false && "p2_1_p1_s1 must take a transition": SUCCESS
[p2_1_run_output_enforcer_p1.assertion.3] assertion me->_policy_p1_state != POLICY_STATE_p2_1_p1_violation: SUCCESS

** 0 of 3 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Fig. 5.2: Verification: Property 1

In this example, the tool uses the code in fig 4.1 for compilation as seen in the fig 5.1 and creates executable and some C files which upon running, gives verification output as seen in fig 5.2.

On compiling the code, a file with a prefix **F** is created which is the actual enforcer for the safety property taking the controller as a **black-box** and a file with a prefix **cbmc** is created which is a **model checker**, it checks the outputs of the enforcer on all possible input scenario of the controller and return successful message if enforcer is able to produce correct the outputs of the controller else it gives a counter example of the input-output event which will lead the shielded system to violation.

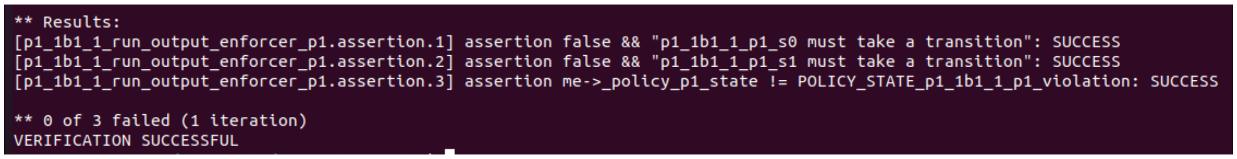
5.1.2 Property 2: Atrial Pace should not happen within the interval $t_v \in$ (0, LRI - AVI), Atrial Sense should not happen during interval $t_v \in$ (0, LRI - AVI), and Atrial Pace will happen at t_v = LRI - AVI



```
samay@samay-VM:~/Documents/easy-rte-master$ make c_enf PROJECT=pacemaker FILE=p1_1b1_1
./easy-rte-parser  -i example/pacemaker/p1_1b1_1.erte -o example/pacemaker/p1_1b1_1.xml
Writing to example/pacemaker/p1_1b1_1.xml
./easy-rte-c -i example/pacemaker/p1_1b1_1.xml -o example/pacemaker
Writing F_p1_1b1_1.c
Writing F_p1_1b1_1.h
Writing cbmc_main_p1_1b1_1.c
```

Fig. 5.3: Property 2

In this example, the tool uses the code in fig 4.3 for compilation as seen in the fig 5.3 and creates executable and some C files which upon running, gives verification output as seen in fig 5.4.

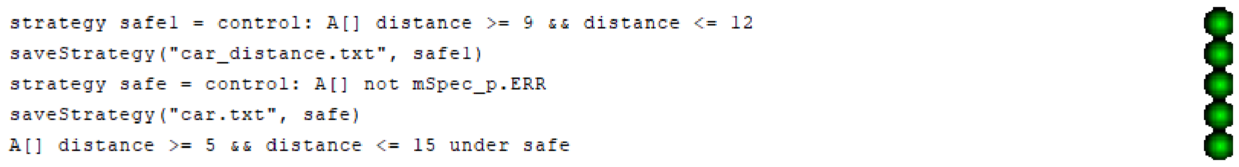


```
** Results:
[p1_1b1_1_run_output_enforcer_p1.assertion.1] assertion false && "p1_1b1_1_p1_s0 must take a transition": SUCCESS
[p1_1b1_1_run_output_enforcer_p1.assertion.2] assertion false && "p1_1b1_1_p1_s1 must take a transition": SUCCESS
[p1_1b1_1_run_output_enforcer_p1.assertion.3] assertion me->_policy_p1_state != POLICY_STATE_p1_1b1_1_p1_violation: SUCCESS

** 0 of 3 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

Fig. 5.4: Verification: Property 2

5.2 Timed Shield

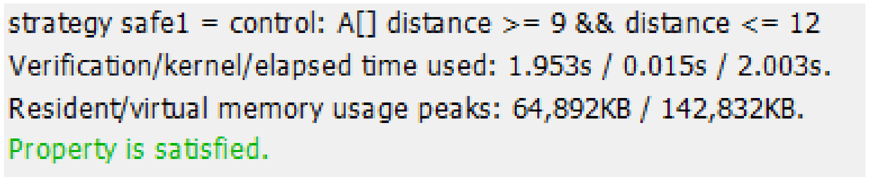
5.2.1 Timed shield for Platooning of Cars



```
strategy safe1 = control: A[] distance >= 9 && distance <= 12
saveStrategy("car_distance.txt", safe1)
strategy safe = control: A[] not mSpec_p.ERR
saveStrategy("car.txt", safe)
A[] distance >= 5 && distance <= 15 under safe
```

Fig. 5.5: Verification: Car Platooning

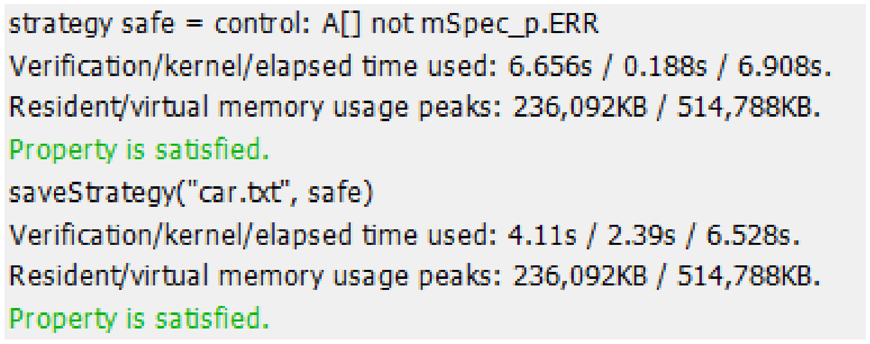
Here, we use the UPPAAL STRATEGO tool to compute the winning strategy for the automata we implemented in the chapter 4. The query statements shown in 5.5 are used for verifying the model of Car Platooning. In fig 5.5, first we write the query to compute the winning strategy such that distance between the cars remain between 9 and 12, then we save the strategy as a text file for future use. At last we check if the shielded system (implemented in Chapter 4) on composition with the winning strategy (using **under** keyword) satisfy the desired safety property. We run these queries in the verifier of the tool and all this queries are satisfied by the shielded system as shown by green circle in front of them.



```
strategy safe1 = control: A[] distance >= 9 && distance <= 12
Verification/kernel/elapsed time used: 1.953s / 0.015s / 2.003s.
Resident/virtual memory usage peaks: 64,892KB / 142,832KB.
Property is satisfied.
```

Fig. 5.6: Verifying Car Distance Query

The query statement in 5.6 is checking whether distance between both the cars is between 9 and 12 or not, at all time of the simulation. Since property is satisfied, the condition follows.

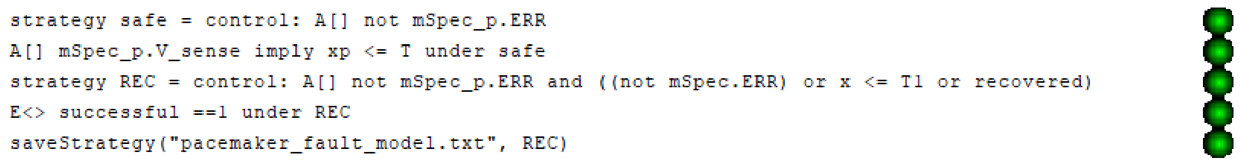


```
strategy safe = control: A[] not mSpec_p.ERR
Verification/kernel/elapsed time used: 6.656s / 0.188s / 6.908s.
Resident/virtual memory usage peaks: 236,092KB / 514,788KB.
Property is satisfied.
saveStrategy("car.txt", safe)
Verification/kernel/elapsed time used: 4.11s / 2.39s / 6.528s.
Resident/virtual memory usage peaks: 236,092KB / 514,788KB.
Property is satisfied.
```

Fig. 5.7: Creating and Saving the Shield Strategy

The query statement in 5.7 checks whether the mSpec' doesn't enter into ERROR state since it is controlled by the shield. Since it is showing Property is satisfied, it doesn't fall into ERROR state at any instance. After that the saveStrategy function is used for saving the strategy created for not going into ERROR state into a text file named car.txt.

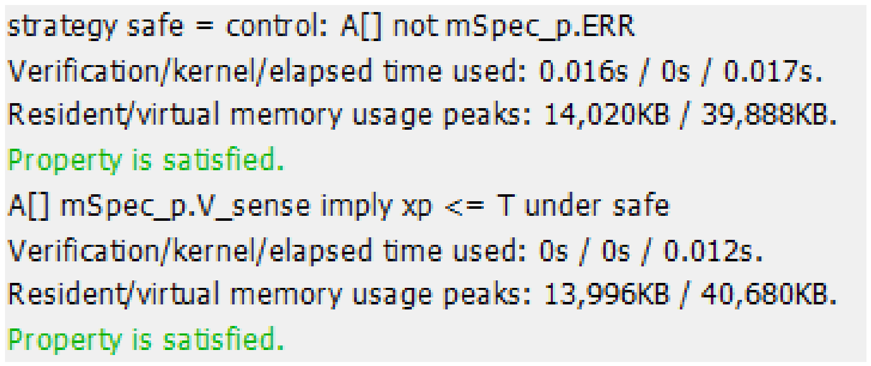
5.2.2 Timed shield with Recovery Mechanism for Pacemaker



```
strategy safe = control: A[] not mSpec_p.ERR
A[] mSpec_p.V_sense imply xp <= T under safe
strategy REC = control: A[] not mSpec_p.ERR and ((not mSpec.ERR) or x <= T1 or recovered)
E<> successful ==1 under REC
saveStrategy("pacemaker_fault_model.txt", REC)
```

Fig. 5.8: Verification: Heart-Pacemaker System

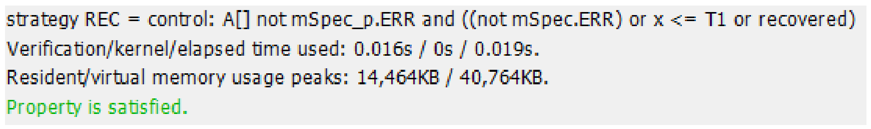
The query statements in 5.8 are used for verifying the model of Heart-Pacemaker.In fig 5.8, first we compute the winning strategy such the mSpec' does not enter into the error state which in turn indicates that the shield corrects the output of the controller always. For **fault recovery**, we compute the winning strategy, indicating that either there is no fault in system and if there is a fault, then it should be corrected within T1 time after fault occurs. We also save the strategy in the file for future use using saveStrategy function. The green mark at the right of each statement shows that each query statement is satisfied.



```
strategy safe = control: A[] not mSpec_p.ERR
Verification/kernel/elapsed time used: 0.016s / 0s / 0.017s.
Resident/virtual memory usage peaks: 14,020KB / 39,888KB.
Property is satisfied.
A[] mSpec_p.V_sense imply xp <= T under safe
Verification/kernel/elapsed time used: 0s / 0s / 0.012s.
Resident/virtual memory usage peaks: 13,996KB / 40,680KB.
Property is satisfied.
```

Fig. 5.9: Verifying Pacemaker Specification

The first query statement in 5.9 checks whether the mSpec' doesn't enter into ERROR state since it is controlled by the shield. Since it is showing Property is satisfied, it doesn't fall into ERROR state at any instance. The second statement is checking the actual specification which needs to be followed by the system. Since Property is satisfied, the Heart-Pacemaker model is satisfying the specification.



```
strategy REC = control: A[] not mSpec_p.ERR and ((not mSpec.ERR) or x <= T1 or recovered)
Verification/kernel/elapsed time used: 0.016s / 0s / 0.019s.
Resident/virtual memory usage peaks: 14,464KB / 40,764KB.
Property is satisfied.
```

Fig. 5.10: Verifying Pacemaker with Fault Models

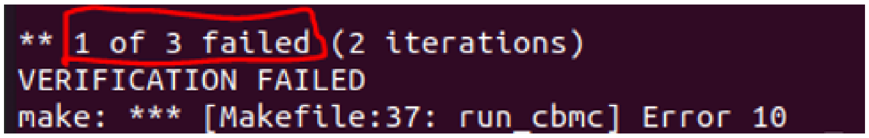
Fig 5.10 is the output of the tool indicating that the property is always satisfied by the shielded system no what strategy the environment choose.

5.3 Limitations of the Bidirectional Run-time Enforcement Algorithm

For the algorithm discussed in Chapter 4 to work correctly, the property should be **enforceable** in nature but enforceable property are only a small subset of all safety property that we want to enforce in real world.

A property is enforceable in nature, if and only if an **accepting state** can be reached from every state other than trap states.

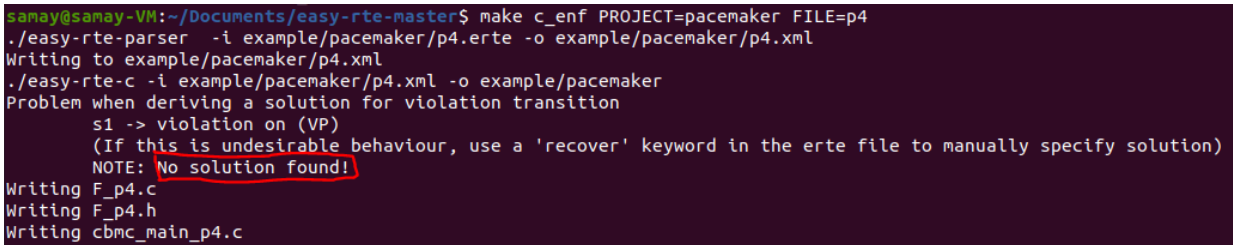
5.4 Limitations of easy-rte tool



```
** 1 of 3 failed (2 iterations)
VERIFICATION FAILED
make: *** [Makefile:37: run_cbmc] Error 10
```

Fig. 5.11: Limitation 1

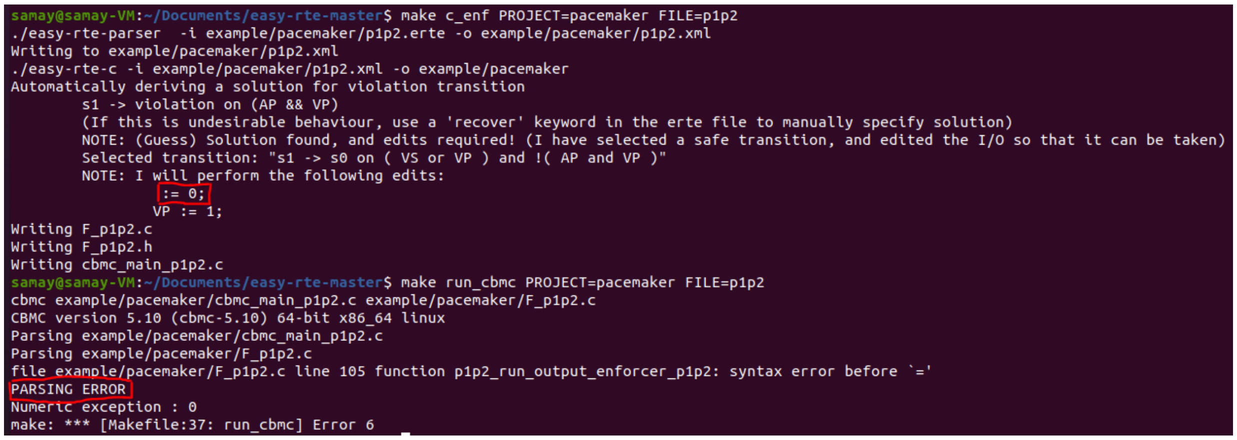
In 5.11, we removed some of the recover statements, then it is not working properly and showing some failed iterations. It is making the whole tool useless if we have to correct violation manually.



```
samay@samay-VM:~/Documents/easy-rte-master$ make c_enf PROJECT=pacemaker FILE=p4
./easy-rte-parser  -i example/pacemaker/p4.erte -o example/pacemaker/p4.xml
Writing to example/pacemaker/p4.xml
./easy-rte-c -i example/pacemaker/p4.xml -o example/pacemaker
Problem when deriving a solution for violation transition
        s1 -> violation on (VP)
        (If this is undesirable behaviour, use a 'recover' keyword in the erte file to manually specify solution)
        NOTE: No solution found!
Writing F_p4.c
Writing F_p4.h
Writing cbmc_main_p4.c
```

Fig. 5.12: Limitation 2

In our policies, we need to add manual recover statements, otherwise it is giving output like **"No solution is found"**, although solution exists, as seen in 5.12.



```
samay@samay-VM:~/Documents/easy-rte-master$ make c_enf PROJECT=pacemaker FILE=p1p2
./easy-rte-parser  -i example/pacemaker/p1p2.erte -o example/pacemaker/p1p2.xml
Writing to example/pacemaker/p1p2.xml
./easy-rte-c -i example/pacemaker/p1p2.xml -o example/pacemaker
Automatically deriving a solution for violation transition
        s1 -> violation on (AP && VP)
        (If this is undesirable behaviour, use a 'recover' keyword in the erte file to manually specify solution)
        NOTE: (Guess) Solution found, and edits required! (I have selected a safe transition, and edited the I/O so that it can be taken)
        Selected transition: "s1 -> s0 on ( VS or VP ) and !( AP and VP )"
        NOTE: I will perform the following edits:
                 := 0;
                VP := 1;
Writing F_p1p2.c
Writing F_p1p2.h
Writing cbmc_main_p1p2.c
samay@samay-VM:~/Documents/easy-rte-master$ make run_cbmc PROJECT=pacemaker FILE=p1p2
cbmc example/pacemaker/cbmc_main_p1p2.c example/pacemaker/F_p1p2.c
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 linux
Parsing example/pacemaker/cbmc_main_p1p2.c
Parsing example/pacemaker/F_p1p2.c
file example/pacemaker/F_p1p2.c line 105 function p1p2_run_output_enforcer_p1p2: syntax error before `='
PARSING ERROR
Numeric exception : 0
make: *** [Makefile:37: run_cbmc] Error 6
```

Fig. 5.13: Limitation 3

In some of the specifications, it is not compiling properly and it is adding unnecessary statements like **":=0"** which does not make sense as shown in 5.13.

5.5 Limitations of the Timed Shield with Recovery Mechanism Algorithm

The algorithm we discussed in Chapter 4 can only recover the controller in case of **transient fault** (error that happens once and has correct pre and post fault). In case another fault is encountered shield will not be able to recover i.e. cannot go in sync with the system. Second, to handle the transient faults for the system, a huge number of fault models are required which can be sometime exponential in nature. So, it is not practical to implement shields with recovery for systems with large property due to huge increase in fault models.

Chapter 6

Conclusion and Future Work

6.1 Results

We explored bidirectional run-time enforcement algorithm for the heart-pacemaker cyber physical system and run it using the easy-rte tool. We explore the limitations of the algorithm and tool. Algorithm is only able to enforce the enforceable property which is a subset of safety property that one want to enforce.

We explored the timed shield algorithm and implemented a few real life applications such as the Heart-Pacemaker Model and the Car Platooning System and some others also using the UPPAAL STRATEGO tool. We saw how we could model, simulate and verify these complex reactive systems using the tool and explore the shortcomings of the algorithm.

We also explored the time shield with time-guaranteed recovery with the help of fault models and analyze the scenarios where algorithm will get some problems. As number of faults models are directly proportional to the number of available transitions to violations, so it will increase exponentially as specification gets large. Algorithm can only handle systems with just one fault with correct pre and post fault behaviour.

6.2 Future Work

In case of bidirectional run-time enforcement, we have assumed that all the properties which we are implementing are enforceable in nature which may not be true always. So checking whether the specification is enforceable or not can be done separately.

We could apply the timed shield algorithm and the work to other real life applications as well. We could think of more such examples.

Timed shield with time-guaranteed recovery mechanism use the fault models but those fault models can only handle systems with transient fault (systems with just one fault), if we encounter another fault afterwards, shield may not be able to synchronize with the system again. We can explore this area and think of a approach that can handle multiple faults.

Currently we synthesize the timed shield for systems with boolean values, we can extend this for cyber physical systems with real values as input and output [WWDW19].

References

[BJK$^+$20] Roderick Bloem, Peter Gjøl Jensen, Bettina Könighofer, Kim G. Larsen, Florian Lorber, and Alexander Palmisano. It's time to play safe: Shield synthesis for timed systems. *ArXiv*, abs/2006.16688, 2020.

[KAB$^+$17] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51, 11 2017.

[PJL$^+$12] Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, pages 173–184, 2012.

[PRS$^+$17] Srinivas Pinisetty, Partha S. Roop, Steven Smyth, Nathan Allen, Stavros Tripakis, and Reinhard Von Hanxleden. Runtime enforcement of cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 16(5s), September 2017.

[WWDW19] Meng Wu, Jingbo Wang, Jyotirmoy V. Deshmukh, and Chao Wang. Shield synthesis for real: Enforcing safety in cyber-physical systems. *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 129–137, 2019.