REVERT: Runtime Verification for Real-Time Systems

INDRAPRASTHA INSTITUTE *of* INFORMATION TECHNOLOGY **DELHI**

Sangeeth Kochanthara

A Thesis Report submitted in partial fulfilment for the degree of

M.Tech in Computer Science

Under the Supervision of Dr. Rahul Purandare, Dr. David Pereira and Dr. Geoffrey Nelissen

November 1, 2016

Keywords: Run-time Verification, Real-Time Systems, Monitor, Timed Automata

Certificate

This is to certify that the thesis titled **"REVERT: Runtime Verification for Real-Time Systems"** submitted by **Sangeeth Kochanthara** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by him under our guidance and supervision in the Program Analysis group at Indraprastha Institute of Information Technology - Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Dr. Rahul Purandare
Indraprastha Institute of Information Technology - Delhi, India

Dr. David Pereira
CISTER research unit, Instituto Superior de Engenharia do Porto, Porto, Portugal

Dr. Geoffrey Nelissen
CISTER research unit, Instituto Superior de Engenharia do Porto, Porto, Portugal

Abstract

Real-time systems are becoming more complex and open, thus increasing their development and verification costs. Although several static verification tools have been proposed over the last decades, they suffer from scalability and precision problems. As a result, the tools fail to cover all the necessary safety properties for realistic real-time applications involving a large number of components and tasks. Runtime verification is a formal technique that verifies properties during system execution with the support of monitors. The monitors are generated from formal languages using correct-by-construction generation methods. Runtime verification can thus be used as a complement or replacement for static verification approaches. The current state-of-the-art tools either do not have notion of time, or suffer from the potential blowup of states at run-time. This thesis proposes Revert, a framework developed with a focus on the verification of functional and non-functional properties with timing constraints. The contribution of this work is threefold: (i) a domain-specific specification language allowing the definition of requirements for real-time applications; (ii) a novel mechanism to generate monitors, with state-space and time guarantees, capable of identifying and reacting to timing properties defined with the proposed specification language. (iii) a tool that automatically transforms specifications written in Revert to monitors specified as complete timed deterministic finite automata in xml format.

Acknowledgments

"And, when you want something, all the universe conspires in helping you to achieve it." - The Alchemist, Pauolo Coelho.

As I reach the end of my master's, I would like to thank all those who made the journey beautiful and whose thoughtfulness, care and support helped me reach here.

My deepest gratitude to my advisors Dr. Rahul Purandare, Dr. David Pereira and Dr. Geoffrey Nelissen for their guidance and support. The quality of this work would not have been nearly as high without their well-appreciated advice. Thanks to my thesis committee: Dr. P.B. Sujit and Dr. Subodh Sharma for their feedback.

I am always indebted to my parents who have always been with me with invaluable support and strong belief in me. I am grateful to my classmates, juniors, seniors, the PhD group at IIIT-Delhi, and my colleagues at CISTER for offering a fresh perspective of research and being with me in all my crazy deeds.

Sonia Dalal and Vedanshi Kataria from B.Tech 2016 batch did the succeeding section of my work: code generation from monitors. Thanks Sonia and Vedanshi for productive discussions.

I would like to thank CISTER, Porto, Portugal for providing infrastructure support and funding via project CONCERTO (JU grant nr. 333053 ARTEMIS/0003/2012) for the initial stage of this work. This work is supported in part by Infosys Foundation and Infosys Center for Artificial Intelligence at IIIT-Delhi.

Contents

1 Introduction **1**

2 Motivation and Related Work **3**

2.1 Motivation 3

2.1.1 Modeling System Properties 4

2.1.2 Monitor Placement 5

2.1.3 Monitor Synthesis 7

2.2 Related Work 7

3 Background **10**

3.1 Properties 10

3.2 Monitoring Architecture 11

3.3 Monitored Applications 11

3.4 Timed Automata 12

3.5 Timed Regular Expressions 13

3.6 Derivatives 13

4 The Revert Framework **15**

4.1 Revert Specification Language 15

4.1.1 Monitor Data 16

4.1.2 Monitor Environment 17

4.1.3 Nodes 18

4.1.4 Transitions and Transition Guard Expressions 19

4.2 Formal Monitor Specification 20

4.2.1 Modeling Monitors 20

4.2.2 Modeling Properties 21

4.3 Monitor Generation 22

4.3.1 Transition Automaton Generation 23

4.3.2 Node Automaton Generation 26
4.3.3 Monitor Automaton Generation 27

5 Implementation **28**
5.1 The Tool Chain 28

6 Example **30**

7 Conclusion and Future Work **32**
7.1 Conclusion and Future Work 32
7.2 Limitation 32

A Example Monitor Output **38**

List of Figures

1.1 General runtime verification structure 1

2.1 Inline monitoring and outline monitoring 6

3.1 Monitoring architecture. Adapted from [24] 11

4.1 FSM of the expression `failure`(`duration`(j_1)<10) 26
4.2 FSM of the expression `failure`(`jitter`(`time`(*startT complT*))≤ 3) 26

5.1 Revert implementation 29

6.1 Example of the specification of a monitor with Revert. 30

List of Algorithms

1 Algorithm to generate timed automaton from TRE expressed using `time` operator 25

Chapter 1

Introduction

Real-Time Systems (RTS) have become pervasive. Typical examples include Air Traffic Control Systems, Networked Multimedia Systems, and Command Control Systems [21]. In RTS, the correctness of an operation depends not only on its functional correctness, but also on the time in which it is performed. Depending on the application domain and on the level of criticality associated with the application, failing to meet some timing constraints can lead to drastic consequences for the system's environment and the agents involved in its operation. Due to the strong reliability, safety and predictability demanded from these systems, verification and validation are fundamental activities often required to be performed according to directives of legal certification entities [18].

Runtime Verification (RV) and runtime monitoring are techniques that uses monitors to observe system properties during execution time and therefore enables to trigger corrective actions on violation of system properties. Figure 1.1 outlines the general RV structure. Thus, RV improves safety, predictability and reliability of RTS.

RV can also help in accelerating the development of RTS whilst maintaining high degree of reliability demanded by such systems.

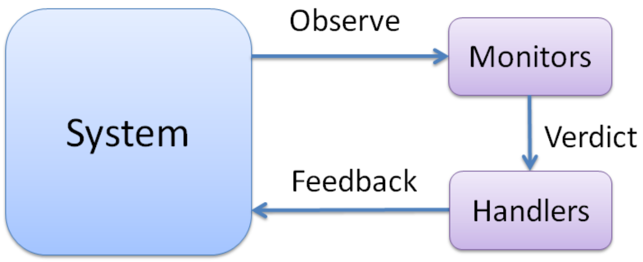


Figure 1.1: General runtime verification structure

We present Revert framework for RV of RTS. We address most of the limitations of state-of-the-art RV frameworks (refer Chapter 2) in Revert. The key contributions of this thesis are:

i Revert, a domain-specific specification language allowing the definition of requirements for real-time applications. Revert is designed to be simple and easy to learn. It supports timing constraints on top of events relative ordering. Revert is a combination of state machines, extended regular expressions, boolean expressions, and timing constraints.

ii A novel mechanism to generate monitors, with state-space and time guarantees, capable of identifying and reacting to timing properties defined with the proposed specification language. We argue that novel monitor generation mechanism guarantees that the generated monitor has to keep track of only one state at any point in run-time, thus avoiding the potential blowup in the number of states of the generated monitor has to keep track of at runtime. A problem that most RV frameworks suffer from today.

iii A new tool to generate monitors for RTS from specifications written in Revert using the novel generation mechanism. The tool takes specifications written in Revert and automatically generates monitors under the form of complete timed Deterministic Finite Automata (DFA). The generated timed DFA can later be used to generate code that can eventually be integrated within the monitored system.

Thesis Organization

This thesis is organized as follows: Chapter 2 discusses problem motivation from three different perspectives and talks about the state-of-the-art. Chapter 3 provides the necessary background. Chapter 4 and 5 explain the contributions in detail followed by a walk-through example in Chapter 6 and its output in Appendix A. Chapter 7 concludes the work with some future directions.

Chapter 2

Motivation and Related Work

2.1 Motivation

Software flaws and failures have an increasing role in many recent accidents [15]. The smallest flaw could have devastating consequences that can lead to significant damage, including loss of life [30]. Embedded software has a clear role in recent failures in commercial avionics, mid air software glitches reported for fighter jets, and recent space accidents [14, 30]. These incidents also shows that the systems built in compliance with the most stringent standards around the globe, namely, NASA Software Safety Standard, FAA System Safety Handbook, MIL-STD-882D (US Department of Defense), DEF-STAN 00-56 (UK Ministry of Defense), DO-178B (Commercial avionics), etc., are not always resilient to software flaws. Though the above mentioned examples are from the space and aviation domains, similar scenarios exist in other domains too [1]. These highlight the importance of validation and verification for real-time embedded software.

The complexity and openness of the current RTS makes verification and validation two of the most costly and time consuming steps during their development. The major traditional validation and verification techniques are testing [8, 22] and model checking [12]. Though testing is a wide field of diverse methods for finding bugs, it fails to show the absence of bugs or the total correctness of the system being tested. Correctness of the system can be shown using exhaustive testing, covering all possible program execution paths with all possible constraints. However, exhaustive testing is impossible for complex systems. Model checking is an automatic verification technique, mainly applicable to finite state systems. Model checking needs the entire system to be modeled as a finite state automaton, which often is impractical. So, static program analysis became next immediate candidate to address the strong reliability and predictability demands of RTS [13].

Static program analysis is the method of inspecting code without running the program. However, static program analysis experiences practical limitations such as undecidability of properties of the underlying formal model, or blowup of the potential states to track. Moreover, extra-functional properties like the time at which events occur are usually only available at run-time. This scenario make RV techniques the natural candidate to address the current limitations of static approaches [20].

RV is a technique used to verify the correctness of system properties during execution of the system. The first step towards RV is modeling system properties. The second step is the placement of monitors which specifies how the monitor is integrated in the system. The third step is how monitors are synthesized from specified properties.

2.1.1 Modeling System Properties

RV frameworks need an input that specifies the system properties to be monitored. Any system behavior that needs to be followed for correct system operation is termed as a system property. Duration of a job and event sequences to be followed (say a file read event should happen only after a file open event) are examples of system properties.

Early works [29] used automata as an input to the verification framework or manually weaved monitor code into the system. Creating an automaton to model system properties by hand is impractical for complex systems such as the current generation of avionic systems, space systems, or medical systems. Manually coded monitors are prone to errors same as coding errors in software development. Moreover, verifiability of such monitors is hard. Using manual methods for knowledge transfer of what the monitor does, among the entire team working on the design and development of the system, is even a bigger problem. The certification of such systems essentially requires certification of the monitor too, which again increases the development cost. To overcome these challenges, formal specification languages for monitor specification are proposed.

Specification languages allow system designers and developers to express properties in at a very high level of abstraction, decreasing the possibility of human errors and increasing their understandability. There are a lot of proven ways to express system behaviors using event sequences: Linear Temporal Logic (LTL), regular expressions, context free grammars, etc. We chose an extended version of regular expressions due to its universal understandability, ease to use and direct mapping to a complete deterministic automaton. Universal understandability means that regular expressions are easiest to understand and use among languages with same expressive power like LTL, and is used widely in software development. However, such languages do not have notion of absolute time (specifying bounds on sequence of events in time units).

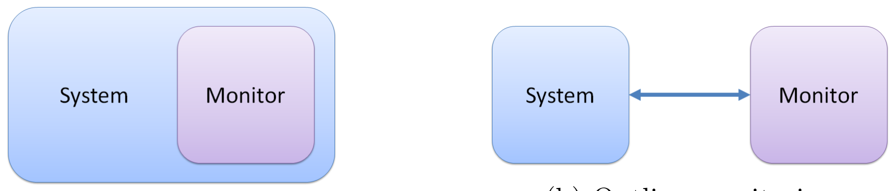
Expressing properties of RTS not only needs specifying event sequences to be followed, but also time bounds on these event sequences. The majority of the specification languages do not have explicit notion of time. The specification languages with notion of time are either too low-level or hard to comprehend (need domain expertise to comprehend) and error-prone. For instance, metric interval temporal logic [2], timed computation tree logic [2], and a few of RV frameworks like RuleR(refer to the next section) are a few examples. In this work, we introduce the three operators `time`, `duration`, and `jitter` (refer to Chapter 4) to specify time bounds on top of event sequences, which covers all the basic timing properties in RTS that need to be monitored.

Systems are changing with time so the properties that need to be monitored changes with time. For instance, the properties that need to be monitored in different modes during flight of a plane: taxi, take off, cruise, and landing, are different. The specification language should be able to express these mode changes, expressing properties needed to be monitored in each mode, separate from one another. Existing languages and frameworks do not provide this flexibility which is essential to specify behavior of complex RTS systems that exist today.

This led us to create a new domain specific specification language, suited for RTS. Revert is crafted with notion of time while being easy to understand and used by the designers and developers of RTS, and yet being expressive.

2.1.2 Monitor Placement

Monitor placement refers to how the monitor is integrated in the system. Broadly there are two ways of monitor integration: *inline* and *outline*. In



(a) Inline monitoring

(b) Outline monitoring

Figure 2.1: Inline monitoring and outline monitoring

the inline monitoring method, the monitor is a part of the system. Conversely in the outline monitoring context, the monitor and the system are two completely different entities. In outline monitoring, since the monitor is not integrated with the system, there will be an essential issue of synchronization between monitor and monitored application. If the system to be monitored is a distributed one, then the network delay also has to be taken care of while estimating the response time of the monitor. There would be shared memory problems for accessing events from the application being monitored since all the events have to be passed via shared memory between the system and the monitor. So inline monitoring seems to be a better candidate over outline monitoring to use with RTS. Figure 2.1 shows pictorial representation of inline and outline monitoring methods.

Inline monitoring avoids or reduces possible adverse impact of monitor on monitored system. A good inline monitoring system should provide time partitioning, space partitioning and independence between different monitors and between monitors and monitored system. Time partitioning ensures response time of one task is not influenced by other tasks, removing monitor impact on task execution times. In space partitioning, each monitor executes in its own memory space avoiding possible corruption of the monitor by a task and (or) or other monitors. Independence between monitors ensures that failure of a monitor will not cause failure of any other monitor or another task in the system being monitored. Monitor architecture should also ensure bounded responsiveness of monitors. Bounded responsiveness of monitors guarantee the feedback or judgement on violation of a property being monitored in bounded time. The monitors that the REVERT framework generates can operate standalone or can be integrated to a system using any of the above-mentioned methods. However, the monitors need to provide these guarantees to be fabricated with RTS especially on safety critical systems.

Nelissen *et. al* [24] proposed a monitor architecture (refer Chapter 3) with all these guarantees, which we use as reference architecture for REVERT.

Offline monitoring and *online* monitoring make another broad classification based on time at which monitoring occurs. In offline monitoring the traces of events produced during runtime (with or without timestamps depending on whether the monitoring system needs notion of time) are analyzed by the monitor aposteriori. The monitor analyzes log file or trace dump. This method can be used to analyze the reason for crash or malfunctioning of the system, to make it better in the future. However, this comes with a toll that the analysis happens after the damage. It cannot be used to prevent the damage or to steer the system from unsafe to safe state which is highly desirable in RTS. In the online monitoring context, events are analyzed at runtime. So the online monitoring method can be used to prevent damage and steer the system from unsafe to safe state on violation of system properties.

2.1.3 Monitor Synthesis

To automate the process of building monitor there must be a defined procedure to generate monitor from the given specification. The monitor should be correct-by-construction, so that monitor will be correct if the specification is correct. This reduces the cost and time needed for certification process, because now the monitor does not need to be verified, rather only the generation process needs to be verified.

The generated monitor should give memory usage and time guarantees for integration with a production level RTS. To the best of our knowledge none of state-of-the-art systems with explicit notion of time are able to provide both space and time guarantees. We argue that our method of monitor generation produces monitors with memory usage and runtime guarantees so that it can be integrated with a production level RTS.

2.2 Related Work

The earliest works in RV focused on *event-triggered* monitoring in which monitors are invoked on each event occurrence that is being monitored. RMOR [16] and MOP [10], are examples of event-triggered monitoring. RMOR and MOP use aspect-oriented programming to instrument the target application's source code from a specification specified used regular expression, linear temporal logic, context free grammar, etc.

Time of occurrence of events in runtime are not always predictable. The events does not occur in a linearly distributed fashion in runtime. This causes event-triggered methods to have unpredictable overheads [23] making them unsuitable for RTS. Moreover, aspect-oriented programming may impact the timing and correctness of the target system and may interfere with certifiability constraints.

In order to make RV suitable for RTS, Zhu et al. [31] proposed predictable monitoring which ensures temporal non-interference of the system being monitored while ensuring temporal correctness of monitor itself. It demands bound on detection latency for deviation from specified behaviors. More recently, Navabpour et al. introduced Rithm [23] for RV on many-core platforms using LTL 3-valued logic to specify properties. Rithm is based on a time-triggered framework. Time-triggered frameworks guarantees predictable overhead since the monitor invocations are predictable. Further, Rithm can use a GPU to improve the responsiveness of the monitors by parallel execution of monitors on accumulated traces. However, it may face a trade-off between responsiveness and efficiency. Execution of parallel monitors needs to have trace accumulated over a longer time to make it efficient, compromising response time. If the same method is used for each event occurrence, in real-time, for better response time, it will reduce efficiency of parallel monitors. Furthermore, there is a significant overhead incurred while transferring data between the host monitoring process on the CPU and the monitoring threads on the GPU since the events are captured using CPU. In comparison, self-monitoring [7], where monitoring code is directly inserted in the application code, has a better response time, but with the potential drawback of hampering the timing properties of the program being monitored, as well as linking the behavior of the monitors to the behavior of the monitored tasks. The main limitation of Rithm is its lack of notion of time. It relies on the relative ordering of events but cannot specify timing constraints on a sequence of events.

RuleR [5], RT-MaC [27], and Copilot [19] are examples of tools with notion of time. RT-MaC is built specifically for applications written in C language. RT-MaC is language specific and unsuitable for current generation commercial off the shelf systems where different parts of the same system may be built using different programming languages. RuleR has a highly expressive monitoring language which models constraints as rules. Yet, RTS properties may be difficult to model as rules, which makes RuleR hard to comprehend, error-prone, and better suited for domain experts rather than for industrial developers. RuleR suffers from unpredictable memory use in runtime since multiple rules may be active at the same time causing an exponential number of active rules. Notably, Copilot is one of the RV tools designed to handle ultra critical systems and uses Satisfiability Modulo Theories (SMT) based k-induction [19] to prove invariant properties of generated monitors. Copilot relies on sampling rates to model time rather than having notion of absolute time (specifying time bounds in time units). This may lead to complete re-writing of specifications written in Copilot in case of changes in sampling rate.

In general, due to the non-deterministic properties of timed automata models, the tools with notion of time may have to keep track of multiple possible states at each time instant. It was shown that, under such models, the number of states that need to be tracked by the monitor may grow exponentially [3]. Therefore the memory space and the computing time required by those tools are hard to predict.

To specify monitoring constraints for an RTS, a language should allow specification of event sequences and time constraints on sequence of events. RMOR and MOP despite being simple and efficient do not allow specification of time constraints as implicit costructs. The language should give enough level of abstraction to specify constraints at a high level so that it can also be used by usual software engineer in industry. The language should be simple yet expressive. RuleR is one of the highly expressive languages, but it is complex to understand and even more tedious to model a complex RTS in it with unpredictable memory usage in runtime. Furthermore, the system changes with time, and the properties that needs to be monitored in different stages can be completely different. There should be methods to express different properties that needs to be monitored according to such changes in the system.

Chapter 3

Background

3.1 Properties

System requirements or system behaviors need to be expressed in some form to check whether they are respected at runtime. System behaviors can be specified in the form of properties. RTS have two types of properties to be verified: functional and extra-functional. Functional properties are those which are related to the result produced or the order of execution of events. Examples are, a file read operation should be preceded by a file open operation, a file open operation should eventually be succeeded by a file close operation, a file close operation should not come between a file open operation and a file read operation, or a file open operation should return a positive value. All such properties can be encoded as regular expressions with the corresponding alphabet being the events of interest (events in the above examples are file read operation, file open operation, and file close operation).

RTS needs the notion of time. Extra-functional properties are used to bridge this gap. Extra-functional properties can be defined as everything that does not relate to the result produced or the order of execution of events. Examples are, a train gate should close in 10 milliseconds, RFID read should not execute for more than 1 second, door open should execute atleast 10 millisecond after RFID read, core temperature must remain under $60°C$, or power consumption of a sensor should be under 5 watts. In this work we limit ourselves to timing properties.

3.2 Monitoring Architecture

Nelissen *et. al* [24] proposed a runtime monitoring architecture for real-time (concurrent) applications that enforces space and time partitioning between the monitored application and the monitors checking its behavior. The architecture is similar to [11]. It limits the impact of the monitoring architecture on the application, avoids propagation of fault from the application to the monitors, (and vice-versa) and ensures bounded responsiveness.

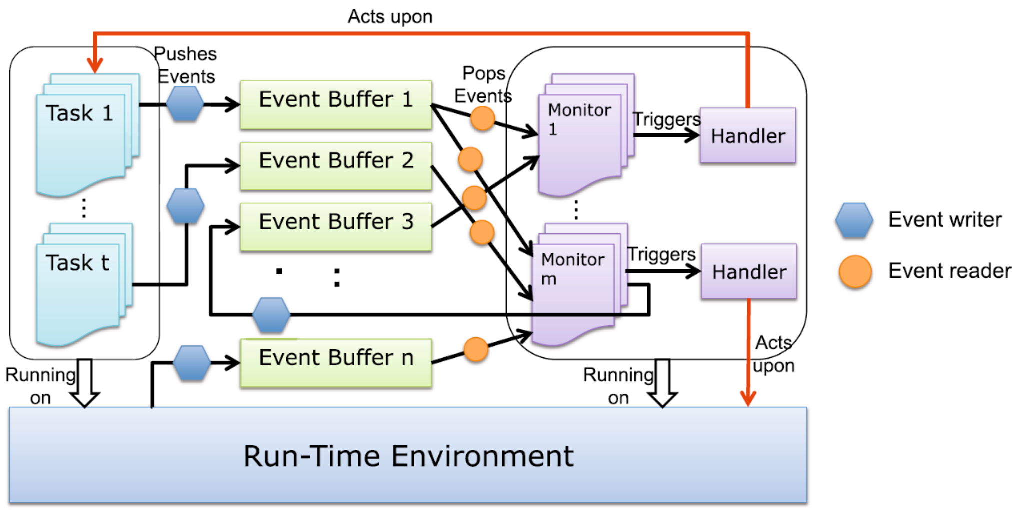


Figure 3.1: Monitoring architecture. Adapted from [24]

Figure 3.1 shows a pictorial representation of the monitoring architecture. Events from different tasks are written to different buffers. This ensures isolation between monitored application and monitors avoiding shared memory issues. Each monitor reads events from the buffers, allowing them to operate independently from other monitors and read only those events that are relevant to each monitor. The architecture is made for RTS and relies on the scheduling by real-time operating system for time partitioning. Memory partitioning is ensured by running each monitor on its own address space.

3.3 Monitored Applications

The kind of RTS that we target with REVERT are the class of periodic or sporadic task systems.

Definition 1 (Application) *An application is a set of periodic or sporadic tasks $T = \{\tau_1, \ldots, \tau_l\}$ where $\tau_i = (p_i, d_i)$ with $p_i, d_i \in \mathbb{T}$ ($\mathbb{T}$ is a time domain), such that p_i and d_i are the period (or minimum inter-arrival time) and the deadline of the task τ_i, respectively.*

Each task $\tau_i \in T$ generates events during their operation on the environment, and we denote such set by $\Sigma(\tau_i)$ and a particular event by ev_j. The set of all events that can occur during the application run-time is the union of the events of its constituent tasks, which we denote by Σ, defined as the union of all the events produced by its tasks, that is, $\Sigma = \Sigma(\tau_1) \cup \cdots \cup \Sigma(\tau_l)$.

Events in Σ denote the concrete events that occur in the monitored application. To capture real-time properties, just the event is not sufficient. For instance, if we are interested in verifying the duration of a job, we would need the time of its start and the time when it terminated. Hence, we refer to events as pairs $\xi = (ev, t)$ such that $ev \in \Sigma$ and $t \in \mathbb{T}$, and define its projections as $\mathsf{event}(\xi) = ev$ and $\mathsf{time}(\xi) = t$.

Definition 2 (Trace) *Let Σ be the alphabet of observable events, and let $\mathbb{T}$ be a time domain. A trace is a sequence $\rho = (\xi_1\xi_2\cdots)$ such that for all $i \geq 1$ we have $\mathsf{time}(\xi_i) < \mathsf{time}(\xi_{i+1})$.*

Although traces are defined as potentially being infinite, monitors will only analyze prefixes of these traces which are finite.

3.4 Timed Automata

Timed automata [3] is a finite state automaton with a set of asynchronous clocks and a set of clock constraints. The clocks are asynchronous in the sense that all the clocks active at any instant may not have the same value, but clock ticks (value increment of all clocks) happens synchronously. A clock valuation maps each clock in the set of clocks to a non-negative integer value. Initial value of a clock in timed automaton is zero if nothing is explicitly specified. Initial value of a clock can start from any value assigned from an integer constant, a variable, an expression, or value of another clock variable. A vertex in a timed automaton may be associated with a clock reset. An edge in a timed automaton is associated with a clock constraint apart from an event of the alphabet of the automaton. An edge of a timed automaton can contain computations and clock reset that are executed if all the clock constraints associated with that edge evaluates to true and current event encountered is same as the event in that edge.

Definition 3 (Timed Automaton) *A timed automaton A is a tuple $\langle Q, q_0, q_f, \Sigma, C, E, I\rangle$, where Q is the set of vertices or states, $q_0 \subseteq Q$ is the set of initial states, $q_f \subseteq Q$ is the set of final states, Σ is a nonempty finite alphabet of events, C is a finite set of non-negative integer valued clocks, $E \in Q \times \Phi(C) \times \Sigma \times Comp \times 2^C \times Q$ is the set of edges or transitions, $I : Q \rightarrow \Psi(C)$ is a mapping that maps states on clock resets, $\Phi(C)$ is the set of clock constraints of the form $c1 \odot x$ (where $c1 \in C$ and x can be another clock in C or any expression that evaluates to a positive integer and $\odot := < \mid > \mid <= \mid >= \mid =$), $\Psi(C)$ is the set of clock resets of the form $c1 := x$, and $Comp$ is the set of all computations.*

3.5 Timed Regular Expressions

Timed Regular Expressions (TRE) are regular expressions (RE) with the notion of time. The expressive power of TRE is strictly less than timed automata. TRE captures only a subclass of timed automata [4].

Definition 4 (Timed Regular Expression) *Timed Regular Expression, α is recursively defined as follows*

$$\alpha \ ::= \ 0 \mid 1 \mid ev \mid \alpha_1 \vee \alpha_2 \mid \alpha_1.\alpha_2 \mid \alpha^\star \mid \langle\alpha\rangle_I$$

where α_1 and α_2 are TRE, 0 denotes the empty language, 1 denotes the language containing only the null word, ev denotes the language containing the string with the single symbol ev, where $ev \in \Sigma$ where Σ is the nonempty finite alphabet of all events, I is a closed interval $[a, b]$ with $a, b \in \mathbb{N}^+$ and $a \leq b$, '$\vee$' is the logical or, '.' is the concatenation, '$\star$' is the Kleene's star operator, and $\langle\alpha\rangle_I$ is defined as the set of all strings that belong to the *language of α but where the duration of the string, that is, the total amount that takes to consume the string according to the timestamps of the events is a value that belongs to the interval I.*

3.6 Derivatives

The notion of derivative of a regular expression was introduced in the 60's by Brzozowski [9] and have received much attention in the last decade by the communities of program verification and formal language theory. Derivatives provide an alternative method to the classic finite automata construction methods. Pucella [26] extended derivatives for timed regular expressions.

Given a TRE α and a timestamped event $\xi = (ev, t)$, informally the derivation process will return a new TRE that removes the event ev from the head of all traces that are members of the language denoted by $L(\alpha)$, where $L(\alpha)$ denotes the language of timed traces. This method is used to check membership of a timed word in the language of a given TRE. We now provide the formal definition of derivative, but first we need to provide a syntactic function that is able to decide whether or not the empty trace belongs to the language of the expression given to be derived.

Definition 5 (Empty trace membership) *Let Σ be a non-empty finite set of events, and let α be a TRE defined over Σ. The syntactic emptiness function is inductively defined as follows:*

$$E(0) = \textbf{false} \qquad E(1) = \textbf{true}$$

$$E(ev) = \textbf{false}, ev \in \Sigma \qquad E(\alpha \vee \beta) = E(\alpha) \vee E(\beta)$$

$$E(\langle\alpha\rangle_I) = E(\alpha) \qquad E(\alpha^\star) = \textbf{true}$$

$$E(\alpha \cdot \beta) = E(\alpha) \wedge E(\beta)$$

Definition 6 (Derivative) *Let Σ be a non-empty finite set of events, let α be an TRE, and let $\xi = (ev, t)$ be a timed symbol with $ev \in \Sigma$ and $t \in \mathbb{T}$, where $\mathbb{T}$ is a time domain. The derivative of α with respect to ξ, denoted as $\mathcal{D}_\xi(\alpha)$, is inductively defined as follows:*

$$\mathcal{D}_\xi(\alpha_1 \vee \alpha_2) = \mathcal{D}_\xi(\alpha_1) \vee \mathcal{D}_\xi(\alpha_2) \qquad \mathcal{D}_\xi(1) = 0$$

$$\mathcal{D}_\xi(\alpha^\star) = \mathcal{D}_\xi(\alpha) \cdot \alpha^\star \qquad \mathcal{D}_\xi(0) = 0$$

$$\mathcal{D}_\xi(\langle\alpha\rangle_I) = \begin{cases} \langle\mathcal{D}_\xi(\alpha)\rangle_{I-t}, & \textit{if } I - t \neq \emptyset; \\ 0, & \textit{otherwise.} \end{cases} \qquad \mathcal{D}_\xi(\alpha) = \begin{cases} 1, & \textit{if } \alpha = ev; \\ 0, & \textit{otherwise.} \end{cases}$$

$$\mathcal{D}_\xi(\alpha_1 \cdot \alpha_2) = \mathcal{D}_\xi(\alpha_1) \cdot \alpha_2 \;\vee\; E(\alpha_1) \cdot \mathcal{D}_\xi(\alpha_2)$$

Chapter 4

The REVERT Framework

The REVERT framework proposes an end to end solution for RV from a new specification language to a novel monitor generation method. The framework is a combination of the REVERT specification language, monitor definition, and the generation method used to generate monitors from specifications.

4.1 REVERT Specification Language

We define the REVERT monitor specification language for building monitor-based safety nets for real-time applications. REVERT was designed with usability and easiness of specification in mind, with the industry practitioners of the real-time embedded computing sectors as special targets. Although the underlying mechanisms of the framework are based on rigorous formal models, the details of these models is reduced to the minimum in the REVERT syntax, so that it can be quickly adopted by users without deep knowledge in formal methods.

REVERT is a combination of state machine, extended regular expressions, boolean expressions, and timing constraints. REVERT relies on external events to reason about traces. Properties on execution patterns or execution order of events, that must be enforced during the application run-time, are specified using extended regular expressions. To express timing constraints on a sequence of events we use three high-level operators: `time`, `duration` and `jitter` (refer to section 4.2.2 for a formal definition). These operators are then automatically converted to finite timed automata. The syntactic structure of a monitor specification in REVERT is presented below:

```
use "$f_1.ev$"
...
use "$f_z.ev$"
monitor $m_i$ {
  observe { $ev_1$, ..., $ev_l$ }
  variables {$v_1$ : $type$, ..., $v_j$ : $type$ }
  jobs{
    $j_1${
     start: {$ev_h$,..., $ev_i$}
     suspend: {$ev_m$,..., $ev_n$}
     resume: {$ev_o$,..., $ev_p$}
     complete: {$ev_q$,..., $ev_r$}
    },
    ...,
    $j_p${...}
  }

  nodes { $n_1$, ..., $n_k$ }
  initial { $n_q$ }

  node $n_1$ { $init_1$ $prop_1$ $trans_1$ }
  ...
  node $n_k$ { $init_k$ $prop_k$ $trans_k$ }
}
```

Listing 4.1: Structure of a monitor specification in REVERT

4.1.1 Monitor Data

REVERT relies on external events to reason about traces. These external events are imported into the scope of a REVERT specification using the `use` statement. The name after the keyword `use` refers to external files that contain the real event identifiers. The `use` statement can also be used to specify the external procedures that can be invoked during the monitor execution to drive the monitored system from unsafe to safe state.

All the events in event files listed using the `use` statement will not be relevant to each monitor. The `observe` statement specifies the events that are monitored by the monitor m_i, out of the complete set of events. This helps in reducing the size of monitor automaton generated (refer to section 4.3) from the specification. Listing 4.2 shows an example `observe` statement.

```
observe {arrT, startT, suspT, blockedT,resumeT, unblockedT, complT }
```

Listing 4.2: Example observe statement

4.1.2 Monitor Environment

The sections `variables`, `jobs`, `nodes`, and `initial` outlines the environment of the monitor.

The `variables` statement defines variables local to the monitor m_i. v_1, ..., v_j are names of variables and *type* defines what kind of values a variable can take (either integer, boolean or real). Typically variables can be used for storing intermediate results that has to be carried among different nodes (defined later in this section), as flags for indicating current state of the monitor, or as arguments for external procedure calls.

The `jobs` statement declares the set of jobs associated with different tasks. Each job is characterized by a starting or release event and a completion event, between which its execution may be suspended (for instance, due to preemption, unavailability of a shared resource or a self-suspension) and resumed. Each job specification is defined by the set of events associated with its lifecycle from its release to its completion. REVERT defines a job using the following four sets of events; `start`, `suspend`, `resume` and `complete`, which contain events related to the release, suspension, resumption, and completion of the job, respectively. Listing 4.3 shows an example job specification

```
 jobs{
    job1{
      start: {startT}
      suspend: {suspT, blockedT}
      resume: {resumeT, unblockedT}
      complete: {complT}
      }
}
```

Listing 4.3: Example for a job in REVERT specification

At a high-level, the monitor is modeled as a finite state machine (FSM). We use the term node to denote the states of the state machine. The `nodes` statement declares identifiers of all the nodes of the monitor. They model the different states that can be reached by the finite state machine, which determine how the properties to be monitored evolve with the system state. These nodes can be seen as different specifications that must be monitored in different modes of execution of the system, for example, taxi, takeoff, cruise, and landing modes of a plane. The node in which the monitor starts its execution is declared using the `initial` statement.

4.1.3 Nodes

The structure of the specification language is built on the key observation that RTS may be dynamic, adapting to the changes in the environment, their workload, and the type of operations that must be performed at a given time or reacting to detected anomalies. Consequently, the properties that must be verified by the monitors may change over time, and it should be possible to specify different modes of operations that are activated depending on some constraints. In the monitor specification as presented in Listing 4.1, different nodes can be seen as different modes of execution. Transitions can be used to specify mode changes or the activation of corrective measures in case of a specification violation. The definition of a corrective action and the mechanism for its execution are not in the scope of this work.

The semantic model of a REVERT specification is an FSM with transitions guarded by regular expressions and logical constraints and the three special operators - `time`, `duration` and `jitter` (discussed in detail later in this section). Each state, which we call *node*, corresponds to one mode of operation of the monitor. A different set of constraints is associated to each node. The transitions between the nodes are guarded by expressions based on the success or failure of the constraints defined in the active node. Some of these transitions may have associated operations for modifying internal variables of the monitor, or calling external procedures.

The behavior of the monitor for each node n_i is specified in a `node` block. A node block n_i includes some initialization code $init_i$, the set of properties that need to be monitored $prop_i$, and the set of transitions $trans_i$ from the current node n_i to any other node defined in the monitor. The transitions between the nodes are guarded by guards based on the success or failure of the properties monitored in that node.

Computational code reacting to some specific properties can be added to the specification. The init block (as in Listing 4.4, where $resetAllSystemFlags();$

is an external procedure call) declares code executed when transitioning to a node, while trans provides means to declare code when transitioning out of the node (refer to subsection 4.1.4).

```
init {
        $resetAllSystemFlags();$
    }
```

Listing 4.4: Example init block in REVERT specification

4.1.4 Transitions and Transition Guard Expressions

As a main observation, we realized that the number of timing properties (extra-functional properties) that must be verifiable are rather limited and can all be expressed with a combination of the three operators `time`, `duration` and `jitter`, which return the time taken by a sequence of events, the execution time of a job, and the jitter on a timing property, respectively. The `time` operator may, for instance, be used to verify that a deadline, a period or a minimum separation time between two events is respected. The `duration` operator is useful to check that the execution time of a job does not exceed its budget or estimated worst-case execution time, or to ensure that the interference suffered by one task due to other tasks is bounded. Finally, the `jitter` operator may be used to bound the variation on any timing property. It can be argued that similar properties can be encoded in existing frameworks such as RULER and RT-MaC. However, they are not all intrinsic constructs of the language, which renders their specification difficult and error-prone to inexperienced users.

The properties (both functional and extra-functional) are expressed in the prop block syntactically specified as `constraints`, inside `node`. Listing 4.5 shows an example `constraints` section. $c1$, …, $c4$ are identifiers of properties. Functional properties are expressed in the form of extended regular expressions (discussed in detail in the next section) with an $[ERE]$ suffix after the identifier for that property. Extra-functional (timing) properties are expressed using the corresponding keyword after the identifier for that property, as specified in the example. These property identifiers are used later in the transition to specify guards.

```
constraints {
        c1: time(blockedT resumeT)) ≤ 2;
        c2: duration(Job1) ≤ 10;
        c3: jitter(time(startT complT)) ≤ 3
        c4[ERE]: startT _ complT;
    }
```
Listing 4.5: Example constraint block in REVERT specification

Transitions are defined node-wise. Transitions are declared in a block `transitions` { $\texttt{t}_1$ { $\texttt{b}_1$ } ... $\texttt{t}_y$ { $\texttt{b}_y$ } } where each $\texttt{t}_i$ is either of the form `success`$(c_i) \rightarrow \texttt{N}_i$ or `failure`$(c_i) \rightarrow \texttt{N}_i$, and $\texttt{b}_i$ is a block of executable code that is called if the transition $\texttt{t}_i$ is activated. The statement `success`(c_i) means that the transition from the current node to the node $\texttt{N}_i$ will be activated when the constraint expressed by c_i is valid, whereas `failure`(c_i) means that the transition to $\texttt{N}_i$ is activated when the constraint denoted by c_i fails. The identifier c_i, belongs to the properties associated with the current node, declared in `constraints` section.

The specification language does not explicitly impose but expects the guards on the transitions to be mutually exclusive. If some non-determinism exists in the specification due to non-mutually exclusive node transitions, it is resolved during the monitor generation using implicit priorities as in the order of their declaration. The guards on the transitions are evaluated in-order they appear in transitions and only the transition with the first guard to be true is activated, thereby ensuring that there is only one active node at any time.

4.2 Formal Monitor Specification

4.2.1 Modeling Monitors

Let Σ be the set of all observable events in the application. The monitoring model considers a finite set of monitors $\mathcal{M} = \{m_1, \ldots, m_k\}$, where each monitor $m_i \in \mathcal{M}$ is a tuple (P_i, E_i, A_i) such that $E_i \subseteq \Sigma$ specifies the subset of events of interest for the monitor m_i, P_i is a collection of properties over E_i, and A_i is a structure (N_i, ν_i) such that N_i is the set of states that the monitor m_i can reach, and $\nu_i : N_i \rightarrow \mathcal{G}_i \rightarrow N_i$ is a transition function dependent on a transition guard that is a member of the set of guarded expressions $\mathcal{G}_i$. Each guarded expression is expressed as the success or the failure of one property in P_i. Properties in P_i can be expressed as logical expressions and *extended* *regular expressions* (ERE) or *extended timed regular expressions* inductively defined over E_i.

4.2.2 Modeling Properties

Functional Properties

Functional properties are expressed using ERE which extends the traditional regular expression with an $\Box$ operator expressed as _ in the specification language.

Formally, *ERE* used in REVERT are defined as follows. Let Σ be a nonempty finite set of alphabets. The set of *Extended Regular Expressions* is inductively defined by the following BNF grammar:

$$\alpha \ ::= \ 0 \,|\, 1 \,|\, e \in \Sigma \,|\, \alpha \vee \alpha \,|\, \alpha.\alpha \,|\, \alpha^{\star} \,|\, | \,\Box\, \alpha.$$

where 0 is the empty set, 1 is the set containing the null string, '$\vee$' is the logical or, '.' is the concatenation, '$\star$' is the Kleene's star operator, and $\Box$ is a newly introduced operator. The introduction of the operator $\Box$ is based on the observation that regular expressions may become extremely complex when (i) the number of monitored event increases but (ii) some properties refer only to a small subset of those events. For instance, specifying that a monitor should shift from one mode to another after beginning and completion of a task with any sequence of events happening in between would require to express all possible sequence of events that do not comprise the completion event between the start and completion of the task. However, using the $\Box$ operator, the same property can simply be written as *start* $\Box$ *comp* (specified as *start* _ *comp* in the specification). $\Box$ operator is formally defined as follows: considering that $\mathcal{L}(\alpha) \subseteq \Sigma^{\star}$ is the language denoted by the event expression α, the language of $\mathcal{L}(\Box\, \alpha)$ is defined as the set of all words $w = w_1 w_2$ such that $w_2 \in \mathcal{L}(\alpha)$, and w_1 does not contain any word in the language denoted by α. In terms of regular expression this will translate to $\Box\, \alpha \ = \sim \alpha \ . \ \alpha$ Note that $\Box$ acts as a way more convenient and handy operator than complement operator in regular expression. So we decided to drop complement operator from the specification language.

Except for the newly introduced operator $\Box\, \alpha$, the syntax of ERE is the same as of classic regular expressions, as well as their semantic interpretation in the domain of regular languages.

Extra-Functional Properties

Extra-functional properties are expressed as logical expressions that extend the traditional propositional logic with the time-related predicates `time`$(\alpha) \odot$ `val`, `duration`$(j_i) \odot$ `val` and `jitter`$(\sigma) \odot$ `val`, where α is a Regular Expression, j_i is the identifier of a job (see section 4.1), σ is either a `time` or a `duration` predicate, $\odot \in \{<, \leq, =, \geq, >\}$, and `val` is a natural number. Assuming that the function Δ returns the timestamp associated with any event in Σ, the semantics of the previous three predicates are defined as follows: if $first$ and $last$ are the events that denote the start and the end of α, respectively, then `time`$(\alpha) \odot$ `val` holds iff $(\Delta(last)$ - $\Delta(first)) \odot$ `val`; similarly, let $start$, $susp_k$, res_k, and $comp$ be the events that denote the start, the k^{th} suspension, the k^{th} resumption, and the completion of the job j_i, then `duration`$(j_i) \odot$ `val` holds iff $(\Delta(comp)$ - $\Delta(start)$ - $\sum_k(\Delta(_{res_k})$ $-\Delta(_{susp_k})))$ $\odot$ `val`; finally for the case of `jitter`$(\sigma) \odot$ `val`, the predicate holds iff $(\max_t(\sigma) - \min_t(\sigma)) \odot$ `val` where $\max_t$ and $\min_t$ return the maximum and minimum value of σ until time t. Formally, TRE (refer to Chapter 3 for formal definition) is used in Revert to model these extra functional properties.

To demonstrate the ease and simplicity of Revert specification language, we compare a specification written in Revert with a specification written in one of the existing specification language: RuleR (refer to Chapter 2). For example "whenever property a occurs both now and in the immediate previous state then b must occur as a later observed property" [6] is expressed in RuleR as below.

$$\begin{array}{lll} r_0 : \multimap r_0, r_1, r_3 & r_1 : a \multimap r_2 & r_2 : \\ r_3 : a, r_2 \multimap b \mid \neg b, r_4 & r_4 : \multimap b \mid \neg b, r_4 & \end{array}$$

The same property is specified in Revert specification language as: $(a.a)\_b$

4.3 Monitor Generation

In order to generate the monitor that will be running beside the application, we transform the specification to a complete deterministic finite automaton with the notion of time. Note that the automaton generation occurs before run-time and the automaton has a maximum of one transition per event occurrence. This enables to generate a monitor with time and space guarantees by avoiding the potential blowup of states in run-time, irrespective of the size and complexity of autmaton from which the monitor is generated. The determinism and finiteness of the automaton ensures that the generated monitor will be tracking one single state at any time. To the best of our knowledge no other RV tool with the notion of time gives state-space and time guarantees.

The generation of monitors is achieved through the following steps:

1. Generating an automaton for each transition;

2. Generating an automaton for each node n_i by applying a product operation on the automata obtained for each transition from n_i to any other node. As mentioned in section 4.1, implicit priority is used to resolve potential conflicts on the final state;

3. Generating the monitor automaton by concatenating the automata of all nodes. The monitor automaton is then converted to XML format which can be used to produce code.

4.3.1 Transition Automaton Generation

Automaton Generation for Functional Properties

Functional properties are expressed in the form of ERE. The $\Box$ operator (_ in specification language) is converted to the equivalent regular expression with complement operator. Standard automaton construction algorithms are used to build complete deterministic automata from the regular expression. Non-deterministic finite automaton (NFA) is built from regular expression using Thompson's construction [28]. Subset construction algorithm [28] is employed to create deterministic finite automaton from NFA. Hopcroft's algorithm [17] is used for minimization of deterministic finite automaton. The final state or sink of the automaton is made the final state of property the transition based on success or failure specified in the transition, respectively.

Automaton Generation for Extra-Functional Properties

Extra-functional properties are expressed as a logical expression on the time taken for a property denoted as regular expression, the duration of a job, or the jitter of a time property. For transitions based on `time`, the traditional automaton construction methods were incapable of generating complete deterministic finite timed automaton from TRE that corresponds `time`. We extended the notion of derivative (refer to Chapter 3) with a notion of *pseudo-integral*.

Given an TRE α and a timestamped event $\xi_i = (ev_i, t_i)$, informally the derivation process will return a new TRE that removes the event ev_i from the head of all traces that are members of the language denoted by α; pseudo-integration process will give a TRE as result which will accept the language formed by appending event ev_i to the language of α. By applying these methods finitely many times with respect to all events of interest, the result will be a finite automaton that recognizes all the words of the original expression α.

Definition 7 (Pseudo Integral) *Let Σ be a non-empty finite set of events, let α be a TRE, and let $\xi = (ev, t)$ be a timed symbol with $ev \in \Sigma$ and $t \in \mathbb{T}$, where $\mathbb{T}$ is a time domain. The integral of α with respect to ξ, denoted as $\mathcal{I}_\xi(\alpha)$, is inductively defined as follows:*

$$\mathcal{I}_\xi(0) = 0 \qquad \mathcal{I}_\xi(1) = ev \qquad \mathcal{I}_\xi(\alpha_1 \cdot \alpha_2) = \alpha_1 \cdot \mathcal{I}_\xi(\alpha_2)$$

$$\mathcal{I}_\xi(\alpha) = \begin{cases} \alpha, & \text{if } \alpha = ev^\star; \\ \alpha \cdot ev, & \text{otherwise.} \end{cases} \qquad \mathcal{I}_\xi(\alpha_1 \vee \alpha_2) = \mathcal{I}_\xi(\alpha_1) \vee \mathcal{I}_\xi(\alpha_2)$$

$$\mathcal{I}_\xi(\langle\alpha\rangle_I) = \begin{cases} \langle\mathcal{I}_\xi(\alpha)\rangle_{I-t}, & \text{if } I - t \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases} \qquad \mathcal{I}_\xi(\alpha^\star) = \alpha^\star \cdot \mathcal{I}_\xi(\alpha)$$

We propose Algorithm 1 to build a complete deterministic finite timed automata from the logical expression `time`(α).

The transformation of a logical expression `duration` or `jitter` in a timed automaton, is implemented using predefined templates. For example, Figure 4.1 shows `failure`(`duration`(job_1) < 10), where job_1 is defined in Listing 4.3 and the list of observed events are defined in Listing 4.2. Figure 4.2 shows `failure`(`jitter`(`time`($startT\ compltT$)) ≤ 3) defined on the same set of events.

Algorithm 1: Algorithm to generate timed automaton from TRE expressed using `time` operator

```
every state $state_i$ is associated with two variables $REex_i$ and $REel_i$;
add start state ($state_0$) to the set $waiting\_states$;
reset clock variable $main\_clock$;
$REex_0 := \alpha$;
$REel_0 := 0$;
for all $state_i \in waiting\_states$ do
    for all $\xi \in \Sigma$ do
        if $\mathcal{D}_\xi(REex_i) \neq 0$ then
            if $\exists state_j \in waiting\_states$ s. t. $\mathcal{D}_\xi(REex_i) \in REex_j$ then
                $REel_j := REel_j \vee \mathcal{I}_\xi(REel_i)$;
            else if $\mathcal{D}_\xi(REex_i) = 1$ then
                create a new final state $state_j$;
                $REex_j := 1$;
                $REel_j := \mathcal{I}_\xi(REel_i)$;
            else
                add a new state $state_j$ to $waiting\_states$;
                $REex_j := \mathcal{D}_\xi(REex_i)$ ;
                $REel_j := \mathcal{I}_\xi(REel_i)$;
            end
            create a transition from $state_i$ to $state_j$;
        else
            $LSI$ := longest suffix of $\mathcal{I}_\xi(REel_i)$ matched with $REel_j$ for any state
            $state_j \in waiting\_states$;
            if $LSI$ is empty then
                create a transition from $state_j$ to $state_0$;
            else if length of $LSI = 1$ then
                create a self-loop on $state_i$ with $main\_clock$ reset;
            else
                add an auxiliary clock $aux\_clk_i$;
                $RE_{pre}$ := longest prefix of $\mathcal{I}_\xi(REel_i)$ before $LSI$;
                reset $aux\_clk_i$ at $state_k \in waiting\_states$ s. t. $RE_{pre} = REel_k$;
                create a transition from $state_i$ to $state_j$ with $main\_clock$ set to value of
                $aux\_clk_i$;
            end
        end
    end
end
```

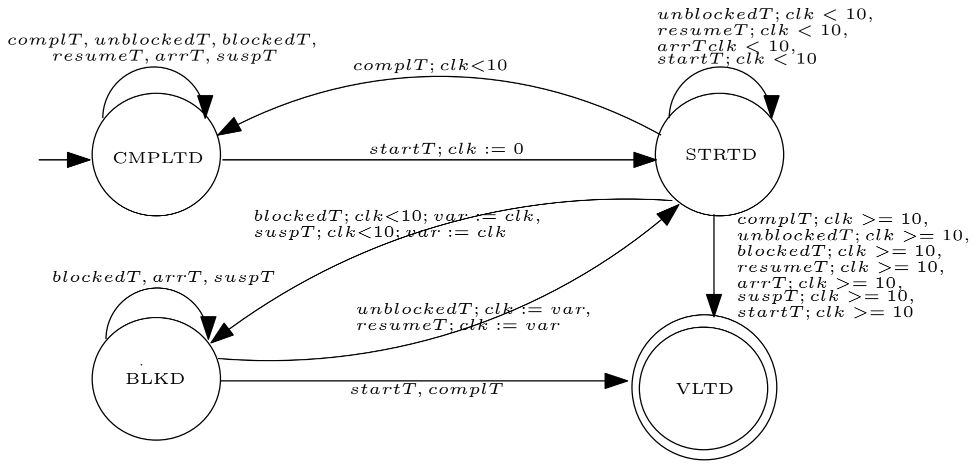

complT, unblockedT, blockedT, resumeT, arrT, suspT
complT; clk<10
unblockedT; clk < 10, resumeT; clk < 10, arrTclk < 10, startT; clk < 10
CMPLTD
startT; clk := 0
STRTD
blockedT; clk<10; var := clk, suspT; clk<10; var := clk
complT; clk >= 10, unblockedT; clk >= 10, blockedT; clk >= 10, resumeT; clk >= 10, arrT; clk >= 10, suspT; clk >= 10, startT; clk >= 10
blockedT, arrT, suspT
unblockedT; clk := var, resumeT; clk := var
BLKD
startT, complT
VLTD

Figure 4.1: FSM of the expression `failure`(`duration`(j_1)<10)

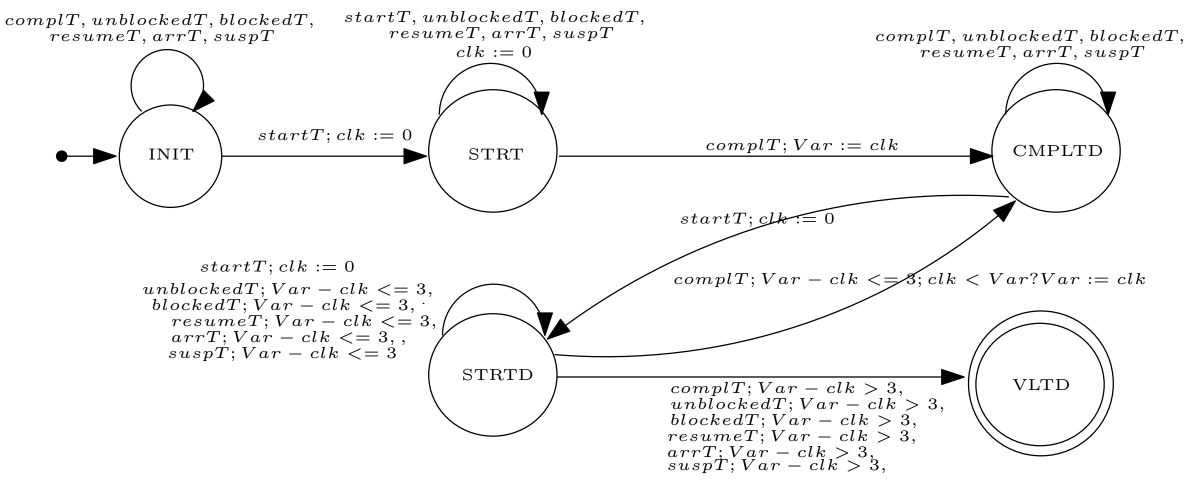

complT, unblockedT, blockedT, resumeT, arrT, suspT
startT, unblockedT, blockedT, resumeT, arrT, suspT clk := 0
complT, unblockedT, blockedT, resumeT, arrT, suspT
INIT
startT; clk := 0
STRT
complT; Var := clk
CMPLTD
startT; clk := 0
complT; Var − clk <= 3; clk < Var?Var := clk
startT; clk := 0
unblockedT; Var − clk <= 3, blockedT; Var − clk <= 3, resumeT; Var − clk <= 3, arrT; Var − clk <= 3, , suspT; Var − clk <= 3
STRTD
complT; Var − clk > 3, unblockedT; Var − clk > 3, blockedT; Var − clk > 3, resumeT; Var − clk > 3, arrT; Var − clk > 3, suspT; Var − clk > 3,
VLTD

Figure 4.2: FSM of the expression `failure`(`jitter`(`time`(*startT compltT*))≤ 3)

4.3.2 Node Automaton Generation

The node automaton is generated using product construction [28] for finite state automata among all the transition automata generated for a node. Since all the transition automata have a single event associated with every edge, the projection of them to finite state machine with only events (without notion of time), will give a complete deterministic finite automaton. We build the product automaton of them rather than actual transition automata.

Final states of product automaton are indicated with destination nodes of the corresponding transition. If the same state in product automaton is final state for more than one transition, then destination node for the transition which is first in-order in the specification, is considered. All the outgoing edges from all the final states of resulting product automaton is removed. The resulting unreachable portion of the node automaton is removed as the next step. The edges in product automaton are then replaced with the corresponding edges in the individual transition automata to get resultant node automaton.

4.3.3 Monitor Automaton Generation

Monitor automaton is built by concatenating all the node automata beginning from the node which is designated as `initial` in the specification. The standard concatenation procedure for concatenating finite state automata [28] is employed in this stage. The destination automaton is found from the destination node denotation done as part of the last step of node automaton generation. There will not be any conflict in this stage since, all the outgoing edges from final states of node automata are eliminated in node automata generation process.

Chapter 5

Implementation

The REVERT framework is built using java programming language. It takes REVERT specifications as input and automatically generates monitors under the form of complete timed deterministic finite automata. This automaton checks that the traces monitored during the system execution respect specifications. The monitor automaton generated is saved in an xml format. A graphical representation of the monitor automaton is also generated.

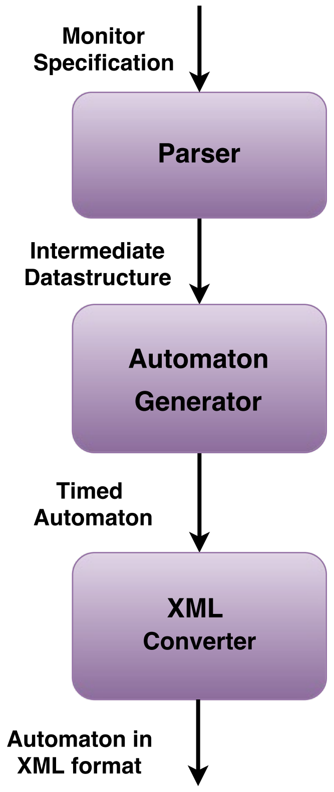
5.1 The Tool Chain

The REVERT framework is built as a tool chain as shown in Figure 5.1a. The parser is built using Antlr 4.0 [25]. The parser checks for syntax errors and builds an abstract syntax tree. A symbol table like structure, intermediate data structure, is built from the abstract syntax tree. Intermediate data structure provides a loose coupling between the parsing and the automaton generation phase.

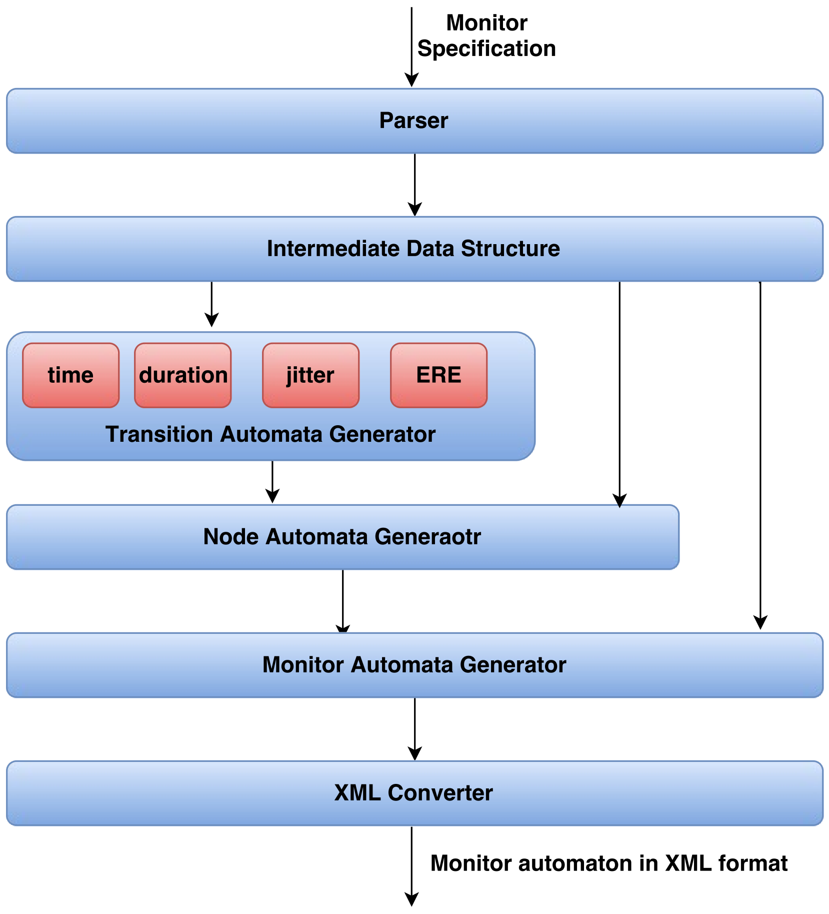
The input to the next phase in the tool chain, automaton generator, is the intermediate data structure. The automaton generation builds a complete deterministic timed automaton from the intermediate data structure. The dedicated packages that deal transformations in each step of the automaton generation are, transition automata generator, node automata generator, and monitor automata generator (refer to Figure 5.1b). Transition automata generator consists of independent methods to generate automaton for `time`, `duration` and `jitter` operators and ERE. Node automata generator takes these automata and names of nodes from the intermediate datastructure and builds node automata. Node automata acts as input to monitor automata generator along with monitor name and other details from intermediate data structure. The monitor automaton generator builds the final monitor automaton, which forms input to the XML converter. XML converter converts monitor to an XML format so that this architecture can be seamlessly integrated as a part of other monitor integration tools.

The xml monitor automaton can be used as is to verify the correctness of the traces. The trace or log can be fed to the monitor automaton and can check whether the log indicates a correct behavior according to the given specification. Also the xml format gives the flexibility to integrate the monitor with the system in the way desired by the implementer.

The xml automaton output generated by the tool chain from the example specification in Chapter 6 is listed in Appendix A.



(a) Tool chain


Monitor Specification
Parser
Intermediate Data Structure
time
duration
jitter
ERE
Transition Automata Generator
Node Automata Generaotr
Monitor Automata Generator
XML Converter
Monitor automaton in XML format

(b) Architecture diagram

Figure 5.1: Revert implementation

Chapter 6

Example

```
use "T_Events.ev";
use "Ext_Procs.h";

monitor MyMon {

   observe { arrT, startT, suspT, blockedT,
           resumeT, unblockedT, complT }

   variables { failureReason : integer; }

   jobs {
      Job1 {
        start: {startT}
        suspend: {suspT, blockedT}
        resume: {resumeT, unblockedT}
        complete: {complT}
      }
   }

   nodes { NormalMode, RecoveryMode }
   initial { NormalMode }

   node NormalMode {
      init{
        resetAllSystemFlags();
      }
      constraints {
         c1: time(blockedT resumeT)) ≤ 2;
         c2: duration(Job1) ≤ 10;
      }
      transitions {
         fail_blocked_time: failure(c1) →
RecoveryMode {
            failureReason := 1;
            recover_from_blocking();
         }
         fail_duration: failure(c2) → RecoveryMode {
            failureReason := 2;
            recover_from_duration();
         }
      }
   }
   node RecoveryMode {
      init{
         initializeSystemRecovery();
      }
      constraints {
         c1[ERE]:  _ complT;
      }
      transitions {
         job_completion: success(c1) →
NormalMode;
      }
   }
}
```

Figure 6.1: Example of the specification of a monitor with REVERT.

As an example, we present in Figure 6.1, a specification written with REVERT that declares a monitor verifying:

i Blocking time of a job is upper bounded ii The execution time of the job never exceeds its estimated worst-case execution time.

The monitor has two modes of operation that are declared as two different nodes, namely, the `NormalMode` which is also defined as the initial node, and the `RecoveryMode` which gets activated when an error is detected.

Monitor execution is started with a call to the external function specified in the node `NormalMode`. In the node `NormalMode`, the monitor verifies two different properties: `c1` and `c2`. The constraint `c1` bounds the maximum blocking time, and `c2` limits the maximum amount of time, `Job1` can execute on the processor until its completion. If either of these constraints fail, the monitor transits to the node `RecoveryMode`. Depending on the activated transition, a different external procedure is called to attempt recovery from the fault, after which a complete system recovery is attempted by execution of the external procedure, `initializeSystemRecovery`, specified in the `init` section of the node `RecoveryMode`. The monitor returns to the node `NormalMode` as soon as the task under analysis completes its execution, i.e., when the regular expression (`_ complT`) is detected.

Note that the monitor generated from this simple specification is a rather complex FSM with clocks which is listed in the Appendix A. During the monitor generation process, the nodes and transitions specified with Revert are expanded to build a final FSM that checks the specified properties.

Chapter 7

Conclusion and Future Work

7.1 Conclusion and Future Work

We presented REVERT, a specification language for performing RV on RTS. We proposed a novel method to generate complete deterministic timed automata from the specification. The proposed method avoids blowup in the number of states at run-time suffered by the other state-of-the-art tools. We implemented the REVERT framework as a tool-chain that generates monitors from given specifications.

A future direction of this work is formally proving the correctness of the presented algorithm, and extending it to support the $\square$ operator. Bounding the time and space complexity of the generated monitors would be another future work. An immediate next step on the implementation side is building a tool for automatic integration of generated monitors with monitored application.

7.2 Limitation

We limit our work to timing and functional properties. So REVERT specification language cannot express other extra-functional properties as intrinsic language constructs. For instance, REVERT specification language does not have intrinsic language constructs to express a constraint on power consumption or a constraint on temperature, similar to `time`, `duration`, and `jitter` operators for expressing time constraints. Due to inherent non-deterministic properties of the underlying model: timed automata, we do not allow complement operator inside `time` operator to ensure determinism. So expressing time bounds on top of functional properties that use $\Box$ operator will be difficult.

Bibliography

[1] Homa Alemzadeh, Ravishankar K Iyer, Zbigniew Kalbarczyk, and Jai Raman. Analysis of safety-critical computer failures in medical devices. *IEEE Security & Privacy*, 11(4):14–26, 2013.

[2] Rajeev Alur. *Techniques for automatic verification of real-time systems.* PhD thesis, stanford university, 1991.

[3] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[4] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[5] Howard Barringer, Klaus Havelund, David Rydeheard, and Alex Groce. Runtime verification. chapter Rule Systems for Runtime Verification: A Short Tutorial, pages 1–24. Springer, 2009.

[6] Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: from eagle to ruler. In *International Workshop on Runtime Verification*, pages 111–125. Springer, 2007.

[7] Borzoo Bonakdarpour, Johnson J Thomas, and Sebastian Fischmeister. Time-triggered program self-monitoring. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 260–269. IEEE, 2012.

[8] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-based testing of reactive systems: advanced lectures*, volume 3472. Springer, 2005.

[9] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[10] Feng Chen and Grigore Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA, pages 569–588, New York, NY, USA, 2007. ACM.

[11] Sarah E Chodrow, Farnam Jahanian, and Marc Donner. Run-time monitoring of real-time systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 74–83. IEEE, 1991.

[12] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[13] Alain Deutsch. Static verification of dynamic properties. *PolySpace White Paper*, page 45, 2003.

[14] Francesca M Favarò, David W Jackson, Joseph H Saleh, and Dimitri N Mavris. Software contributions to aircraft adverse events: Case studies and analyses of recurrent accident patterns and failure mechanisms. *Reliability Engineering & System Safety*, 113:131–142, 2013.

[15] Veronica L Foreman, Francesca M Favarò, and Joseph H Saleh. Analysis of software contributions to military aviation and drone mishaps. In *2014 Reliability and Maintainability Symposium*, pages 1–6. IEEE, 2014.

[16] Klaus Havelund. Runtime verification of c programs. In *Proceedings of the 20th IFIP TC 6/WG 6.1 International Conference on Testing of Software and Communicating Systems: 8th International Workshop*, TestCom '08 / FATES '08, pages 7–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] John Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, DTIC Document, 1971.

[18] Andrew Kornecki and Janusz Zalewski. Software certification for safety-critical systems: A status report. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 665–672. IEEE, 2008.

[19] Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In *Runtime Verification*, pages 87–101. Springer, 2015.

[20] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[21] Fan Liu, Ajit Narayanan, and Quan Bai. Real-time systems. 2000.

[22] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[23] Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: a tool for enabling time-triggered runtime verification for c programs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 603–606. ACM, 2013.

[24] Geoffrey Nelissen, David Pereira, and Luís Miguel Pinho. A novel runtime monitoring architecture for safe and efficient inline monitoring. In *Ada-Europe 2015*. Springer, 2015.

[25] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[26] Riccardo Pucella. On equivalences for a class of timed regular expressions. *Electronic Notes in Theoretical Computer Science*, 106:315–333, 2004.

[27] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *RTCSA 2005*, pages 147–153. IEEE Computer Society, 2005.

[28] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[29] Stavros Tripakis. Fault diagnosis for timed automata. In *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 205–221. Springer, 2002.

[30] W Eric Wong, Vidroha Debroy, and Andrew Restrepo. The role of software in recent catastrophic accidents. *IEEE Reliability Society 2009 Annual Technology Report*, 2009.

[31] Haitao Zhu, Matthew B Dwyer, and Steve Goddard. Predictable run-time monitoring. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 173–183. IEEE, 2009.

Appendix A

Example Monitor Output

Listing A.1: xml file generated by REVERT framework for the specification in Figure 6.1

```
<?xml version="1.0" encoding="UTF-8"?>
<automaton name= "MyMon">
   <event_files list_of_files= "T_Events.ev"/>  <!-- Set of states of
        the monitor -->
   <states>
      <state name="MyMonNormalMode2"/>
      <state name="MyMonNormalMode5"/>
      <state name="RecoveryModec1job_completion1"/>
      <state name="MyMonNormalMode6"/>
      <state name="MyMonNormalMode0"/>
      <state name="MyMonNormalMode1"/>
   </states>
   <!-- Set of  events  used in the monitor -->
   <events>
      <event name="complT"/>
      <event name="unblockedT"/>
      <event name="blockedT"/>
      <event name="resumeT"/>
      <event name="arrT"/>
      <event name="suspT"/>
      <event name="startT"/>
   </events>
   <!-- Set of clocks used by the monitor -->
   <clocks>
      <clock name="MyMonNormalModec2fail_durationFclk"/>
      <clock name="MyMonNormalModec1fail_blocked_timeclk0"/>
   </clocks>
   <!-- Set of initial computations -->
   <initial_computations>
      <function call="resetAllSystemFlags();" from="Ext_Procs.h"/>
   </initial_computations>
   <!-- Transitions -->
   <transitions>
   <transition src="MyMonNormalMode0" dst="MyMonNormalMode0">
```

```xml
      <guard>
         <event name="resumeT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode6" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="resumeT"/>
         <clocks>
            <clock name="MyMonNormalModec1fail_blocked_timeclk0">
               <condition_type>"gt"</condition_type>
               <value>"2"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <value>"MyMonNormalModec2fail_durationFvar"</value>
            </clock>
         </clocks>
         <computations>
            <function call="recover_from_jitter();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[1]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode2" dst="MyMonNormalMode0">
      <guard>
         <event name="unblockedT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="blockedT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"gt"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
```

```
    </actions>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode5">
    <guard>
      <event name="suspT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <variable name="MyMonNormalModec2fail_durationFvar">
          <expression>"[MyMonNormalModec2fail_durationFclk]"</expression>
        </variable>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode6">
    <guard>
      <event name="blockedT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <variable name="MyMonNormalModec2fail_durationFvar">
          <expression>"[MyMonNormalModec2fail_durationFclk]"</expression>
        </variable>
      </computations>
    </actions>
  </transition>
  <transition src="RecoveryModec1job_completion1" dst="RecoveryModec1job_completion1">
    <guard>
      <event name="unblockedT"/>
    </guard>
  </transition>
  <transition src="RecoveryModec1job_completion1" dst="RecoveryModec1job_completion1">
    <guard>
      <event name="blockedT"/>
    </guard>
  </transition>
  <transition src="RecoveryModec1job_completion1" dst="RecoveryModec1job_completion1">
    <guard>
      <event name="resumeT"/>
    </guard>
```

```
</transition>
<transition src="RecoveryModec1job_completion1" dst="
    RecoveryModec1job_completion1">
  <guard>
    <event name="arrT"/>
  </guard>
</transition>
<transition src="RecoveryModec1job_completion1" dst="
    RecoveryModec1job_completion1">
  <guard>
    <event name="suspT"/>
  </guard>
</transition>
<transition src="RecoveryModec1job_completion1" dst="
    RecoveryModec1job_completion1">
  <guard>
    <event name="startT"/>
  </guard>
</transition>
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
    ">
  <guard>
    <event name="resumeT"/>
    <clocks>
      <clock name="MyMonNormalModec2fail_durationFclk">
        <condition_type>"gt"</condition_type>
        <value>"10"</value>
      </clock>
    </clocks>
  </guard>
  <actions>
    <computations>
      <function call="recover_from_duration();" from="Ext_Procs.h"
          />
      <variable name="failureReason">
        <expression>"[2]"</expression>
      </variable>
      <function call="initializeSystemRecovery();" from="Ext_Procs
          .h"/>
    </computations>
  </actions>
</transition>
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
    ">
  <guard>
    <event name="arrT"/>
    <clocks>
      <clock name="MyMonNormalModec2fail_durationFclk">
        <condition_type>"gt"</condition_type>
        <value>"10"</value>
      </clock>
    </clocks>
  </guard>
  <actions>
    <computations>
      <function call="recover_from_duration();" from="Ext_Procs.h"
          />
```

```
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode2" dst="MyMonNormalMode2">
      <guard>
         <event name="blockedT"/>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec1fail_blocked_timeclk0">
               <value>"0"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="unblockedT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"gt"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode2" dst="MyMonNormalMode0">
      <guard>
         <event name="suspT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
      <guard>
         <event name="unblockedT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"leq"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
```

```
    </guard>
  </transition>
  <transition src="MyMonNormalMode5" dst="MyMonNormalMode1">
    <guard>
      <event name="resumeT"/>
    </guard>
    <actions>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <value>"MyMonNormalModec2fail_durationFvar"</value>
        </clock>
      </clocks>
    </actions>
  </transition>
  <transition src="RecoveryModec1job_completion1" dst="MyMonNormalMode0
     ">
    <guard>
      <event name="complT"/>
    </guard>
    <actions>
      <computations>
        <function call="resetAllSystemFlags();" from="Ext_Procs.h"/>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode0" dst="MyMonNormalMode0">
    <guard>
      <event name="arrT"/>
    </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
     ">
    <guard>
      <event name="arrT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"gt"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <function call="recover_from_duration();" from="Ext_Procs.h"
             />
        <variable name="failureReason">
          <expression>"[2]"</expression>
        </variable>
        <function call="initializeSystemRecovery();" from="Ext_Procs
             .h"/>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode5" dst="RecoveryModec1job_completion1
     ">
    <guard>
      <event name="complT"/>
```

```xml
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode6" dst="MyMonNormalMode6">
      <guard>
         <event name="blockedT"/>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec1fail_blocked_timeclk0">
               <value>"0"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode2" dst="MyMonNormalMode0">
      <guard>
         <event name="complT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode2" dst="MyMonNormalMode1">
      <guard>
         <event name="startT"/>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <value>"0"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode2" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="resumeT"/>
         <clocks>
            <clock name="MyMonNormalModec1fail_blocked_timeclk0">
               <condition_type>"gt"</condition_type>
               <value>"2"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_jitter();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[1]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode5" dst="MyMonNormalMode6">
      <guard>
         <event name="blockedT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
      <guard>
         <event name="arrT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"leq"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="complT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"gt"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode0" dst="MyMonNormalMode2">
      <guard>
         <event name="blockedT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="complT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
```

```
               <condition_type>"gt"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h" />
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode6" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="startT"/>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h" />
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode6" dst="MyMonNormalMode5">
      <guard>
         <event name="arrT"/>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
      <guard>
         <event name="startT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"leq"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="startT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"gt"</condition_type>
```

```xml
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h" />
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode0" dst="MyMonNormalMode1">
      <guard>
         <event name="startT"/>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <value>"0"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode6" dst="MyMonNormalMode1">
      <guard>
         <event name="resumeT"/>
         <clocks>
            <clock name="MyMonNormalModec1fail_blocked_timeclk0">
               <condition_type>"leq"</condition_type>
               <value>"2"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <value>"MyMonNormalModec2fail_durationFvar"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode1" dst="MyMonNormalMode6">
      <guard>
         <event name="blockedT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"leq"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
```

```
        <computations>
           <variable name="MyMonNormalModec2fail_durationFvar">
              <expression>"[MyMonNormalModec2fail_durationFclk]"</
                  expression>
           </variable>
        </computations>
     </actions>
  </transition>
  <transition src="MyMonNormalMode6" dst="MyMonNormalMode5">
     <guard>
        <event name="suspT"/>
     </guard>
  </transition>
  <transition src="MyMonNormalMode0" dst="MyMonNormalMode0">
     <guard>
        <event name="complT"/>
     </guard>
  </transition>
  <transition src="MyMonNormalMode2" dst="MyMonNormalMode0">
     <guard>
        <event name="resumeT"/>
        <clocks>
           <clock name="MyMonNormalModec1fail_blocked_timeclk0">
              <condition_type>"leq"</condition_type>
              <value>"2"</value>
           </clock>
        </clocks>
     </guard>
  </transition>
  <transition src="MyMonNormalMode5" dst="MyMonNormalMode1">
     <guard>
        <event name="unblockedT"/>
     </guard>
     <actions>
        <clocks>
           <clock name="MyMonNormalModec2fail_durationFclk">
              <value>"MyMonNormalModec2fail_durationFvar"</value>
           </clock>
        </clocks>
     </actions>
  </transition>
  <transition src="MyMonNormalMode2" dst="MyMonNormalMode0">
     <guard>
        <event name="arrT"/>
     </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
     <guard>
        <event name="resumeT"/>
        <clocks>
           <clock name="MyMonNormalModec2fail_durationFclk">
              <condition_type>"leq"</condition_type>
              <value>"10"</value>
           </clock>
        </clocks>
     </guard>
  </transition>
```

```
<transition src="MyMonNormalMode1" dst="MyMonNormalMode0">
   <guard>
      <event name="complT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"leq"</condition_type>
            <value>"10"</value>
         </clock>
      </clocks>
   </guard>
</transition>
<transition src="MyMonNormalMode0" dst="MyMonNormalMode0">
   <guard>
      <event name="suspT"/>
   </guard>
</transition>
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
   <guard>
      <event name="startT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"gt"</condition_type>
            <value>"10"</value>
         </clock>
      </clocks>
   </guard>
   <actions>
      <computations>
         <function call="recover_from_duration();" from="Ext_Procs.h"/>
         <variable name="failureReason">
            <expression>"[2]"</expression>
         </variable>
         <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
      </computations>
   </actions>
</transition>
<transition src="RecoveryModec1job_completion0" dst="MyMonNormalMode0">
   <guard>
      <event name="complT"/>
   </guard>
   <actions>
      <computations>
         <function call="resetAllSystemFlags();" from="Ext_Procs.h"/>
      </computations>
   </actions>
</transition>
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
   <guard>
      <event name="suspT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"gt"</condition_type>
```

```
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <function call="recover_from_duration();" from="Ext_Procs.h"
            />
        <variable name="failureReason">
          <expression>"[2]"</expression>
        </variable>
        <function call="initializeSystemRecovery();" from="Ext_Procs
            .h"/>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode5" dst="MyMonNormalMode5">
    <guard>
      <event name="suspT"/>
    </guard>
  </transition>
  <transition src="MyMonNormalMode5" dst="MyMonNormalMode5">
    <guard>
      <event name="arrT"/>
    </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode5">
    <guard>
      <event name="suspT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <variable name="MyMonNormalModec2fail_durationFvar">
          <expression>"[MyMonNormalModec2fail_durationFclk]"</
              expression>
        </variable>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
    <guard>
      <event name="arrT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
  </transition>
```

```
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
    ">
   <guard>
      <event name="unblockedT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"gt"</condition_type>
            <value>"10"</value>
         </clock>
      </clocks>
   </guard>
   <actions>
      <computations>
         <function call="recover_from_duration();" from="Ext_Procs.h"
             />
         <variable name="failureReason">
            <expression>"[2]"</expression>
         </variable>
         <function call="initializeSystemRecovery();" from="Ext_Procs
             .h"/>
      </computations>
   </actions>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="unblockedT"/>
   </guard>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="blockedT"/>
   </guard>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="resumeT"/>
   </guard>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="arrT"/>
   </guard>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="suspT"/>
   </guard>
</transition>
<transition src="RecoveryModec1job_completion0" dst="
    RecoveryModec1job_completion1">
   <guard>
      <event name="startT"/>
    </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode0">
    <guard>
      <event name="complT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
    <guard>
      <event name="resumeT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"leq"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
  </transition>
  <transition src="MyMonNormalMode0" dst="MyMonNormalMode0">
    <guard>
      <event name="unblockedT"/>
    </guard>
  </transition>
  <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
     ">
    <guard>
      <event name="suspT"/>
      <clocks>
        <clock name="MyMonNormalModec2fail_durationFclk">
          <condition_type>"gt"</condition_type>
          <value>"10"</value>
        </clock>
      </clocks>
    </guard>
    <actions>
      <computations>
        <function call="recover_from_duration();" from="Ext_Procs.h"
             />
        <variable name="failureReason">
          <expression>"[2]"</expression>
        </variable>
        <function call="initializeSystemRecovery();" from="Ext_Procs
             .h"/>
      </computations>
    </actions>
  </transition>
  <transition src="MyMonNormalMode6" dst="RecoveryModec1job_completion1
     ">
    <guard>
      <event name="complT"/>
    </guard>
```

```
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode6" dst="MyMonNormalMode1">
      <guard>
         <event name="unblockedT"/>
      </guard>
      <actions>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <value>"MyMonNormalModec2fail_durationFvar"</value>
            </clock>
         </clocks>
      </actions>
   </transition>
   <transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1">
      <guard>
         <event name="blockedT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"gt"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
      <actions>
         <computations>
            <function call="recover_from_duration();" from="Ext_Procs.h"/>
            <variable name="failureReason">
               <expression>"[2]"</expression>
            </variable>
            <function call="initializeSystemRecovery();" from="Ext_Procs.h"/>
         </computations>
      </actions>
   </transition>
   <transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
      <guard>
         <event name="startT"/>
         <clocks>
            <clock name="MyMonNormalModec2fail_durationFclk">
               <condition_type>"leq"</condition_type>
               <value>"10"</value>
            </clock>
         </clocks>
      </guard>
```

```xml
</transition>
<transition src="MyMonNormalMode1" dst="RecoveryModec1job_completion1
    ">
   <guard>
      <event name="resumeT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"gt"</condition_type>
            <value>"10"</value>
         </clock>
      </clocks>
   </guard>
   <actions>
      <computations>
         <function call="recover_from_duration();" from="Ext_Procs.h"
             />
         <variable name="failureReason">
            <expression>"[2]"</expression>
         </variable>
         <function call="initializeSystemRecovery();" from="Ext_Procs
             .h"/>
      </computations>
   </actions>
</transition>
<transition src="MyMonNormalMode5" dst="RecoveryModec1job_completion1
    ">
   <guard>
      <event name="startT"/>
   </guard>
   <actions>
      <computations>
         <function call="recover_from_duration();" from="Ext_Procs.h"
             />
         <variable name="failureReason">
            <expression>"[2]"</expression>
         </variable>
         <function call="initializeSystemRecovery();" from="Ext_Procs
             .h"/>
      </computations>
   </actions>
</transition>
<transition src="MyMonNormalMode1" dst="MyMonNormalMode1">
   <guard>
      <event name="unblockedT"/>
      <clocks>
         <clock name="MyMonNormalModec2fail_durationFclk">
            <condition_type>"leq"</condition_type>
            <value>"10"</value>
         </clock>
      </clocks>
   </guard>
</transition>
</transitions>
<!-- Initial state -->
<initial_state name="MyMonNormalMode0"/>
<!-- Final states -->
<final_states>
```

```
    </final_states>
</automaton>
```