A framework for timing analysis of event-driven applications

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN

Faculty of Engineering

BY

Sai Teja Kuchi

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

November, 2025

Declaration of Originality

I, **Sai Teja Kuchi**, with SR No. **04-04-00-10-22-22-1-21317** hereby declare that the material presented in the thesis titled

A framework for timing analysis of event-driven applications

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2022-2025**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Wednesday 30th July, 2025 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Vinod Ganapath Advisor Signature

Co-Advisor Name: Prof. Komondoor V. Raghavan Co-Advisor Signature

DEDICATED TO

my parents

for their unwavering love and support through this journey

Acknowledgements

First and foremost, I would like to express my deepest gratitude to Prof. Vinod Ganapathy and Prof. Raghavan Komondoor for their guidance throughout my research journey. Under their mentorship, I have gained a profound understanding of the research process, developed invaluable skills such as attention to detail and patience, and acquired some amount of knowledge in formal methods. I am continually inspired by their ability to break down complex problems into manageable, simpler components and by their unwavering support during even the most challenging moments. Their encouragement has been a constant source of motivation, and I can only hope to uphold the high standards they have instilled in me as I continue my professional journey.

I would also like to extend my sincere thanks to the LLVM community, particularly Quentin Colombet and Aaron Ballman. They took time out of their busy schedules to talk with me privately, offering invaluable guidance and helping me resolve my doubts. Their dedication to fostering a supportive learning environment for beginners is deeply appreciated.

My heartfelt thanks also go to my labmates and the various PhD students across different labs who have generously shared their insights on problems that I have been facing as part of my research. Rather than simply providing answers, they challenged me to think critically and arrive at solutions on my own. This approach has enriched my understanding and deepened my thinking in ways that I will carry forward in my work.

Finally, I am profoundly grateful to my parents, whose unconditional love, support, and encouragement have been the foundation of all my accomplishments. Their unwavering faith in me, especially during times of doubt and uncertainty, has been my greatest source of strength and inspiration.

Abstract

Event-driven applications, particularly those based on the publish–subscribe communication model, are widely adopted to build responsive and decoupled applications in domains such as robotics, the Internet of Things (IoT), and real-time control. While these architectures offer flexibility and scalability, they also pose significant challenges in meeting real-time timing requirements. These challenges often stem from factors such as long-running event-handler executions, misconfigured parameters, and execution orderings of event-handlers.

In this thesis, we address the problem: "Given code based on the publish–subscribe communication model, will it consistently deliver messages on time?". To answer this, we propose a framework for analyzing the timing behaviour of such systems. Our approach involves constructing a Timed Automata Model from the source code, capturing both timing and behavioural semantics. The constructed Timed Automata Model is then verified using a model checker to determine if the system could ever end up in a situation where messages are not delivered on time.

If such a situation arises, the framework provides feedback such as recommending adjustments to configuration parameters, reordering event-handler execution, or relaxing overly strict timing requirements. Through case studies on several real-world ROS packages, a widely adopted publish–subscribe system, we demonstrate the practical utility of our approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Event-Driven Programming (EDP) [2] is a programming paradigm in which the program's flow is dictated by events rather than a fixed sequence of instructions. Unlike procedural programming, EDP allows the program to listen for internal and external events and react using event-handling mechanisms, offering greater flexibility and responsiveness. This approach is particularly suitable for applications in IoT, real-time systems [55, 8], and graphical user interfaces. According to a survey [50] of 840 people in 9 countries across the globe, 85% of organizations have turned to event-driven architecture to meet their business needs. Financial services, telecoms, and media & technology companies are at the forefront of the adoption wave, with 27% of organizations having a central team promoting it within the organization. The lack of real-time data leading to decisions made on inconsistent or outdated information concerns 46% of transportation and logistics businesses and 44% of retail businesses. Although research highlights that the EDP paradigm results in more reliable software [10, 9], it also introduces challenges [13, 1], including overkill for periodic tasks, event loss, latency, event ordering issues, complexity, etc., that need to be addressed. Event-driven architecture encompasses various patterns, including publish-subscribe[1], request-reply, and point-to-point communication [51].

1.1 Publish-Subscribe Systems

A pub-sub system enables decoupled communication between event producers (publishers) and consumers (subscribers). As illustrated in Figure 1.1, publishers generate messages and associate them with specific topics (e.g., TopicA, TopicB), which

[1]From here on, we will be referring to publish-subscribe as pub-sub

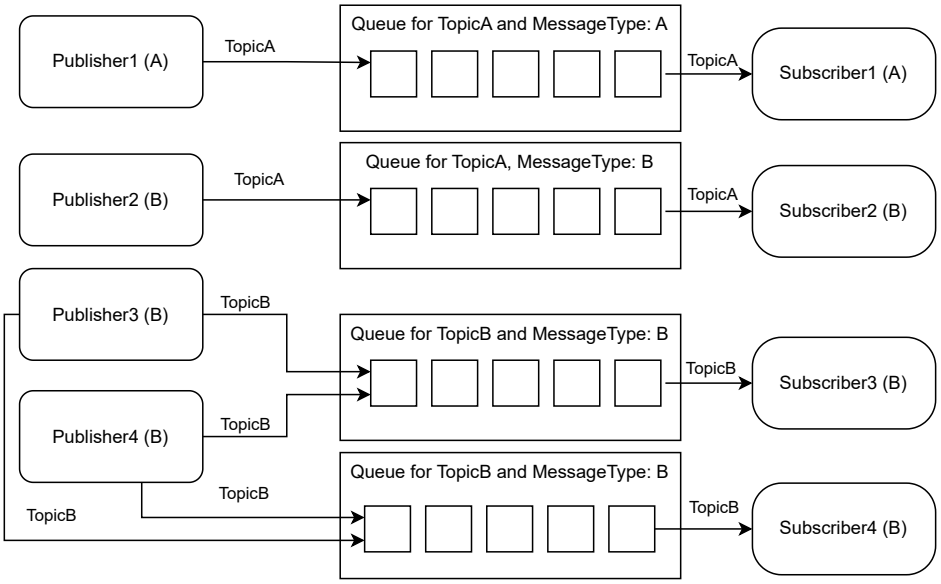


Figure 1.1: **Working model of publish-subscribe systems:** Incoming messages from the publishers are enqueued into the queue before getting processed by the event-handlers of the subscribers. The MessageType is indicated in parentheses to denote the structure of the message.

serve as the communication channels for message transmission. Each message conforms to a predefined data structure referred to as MessageType (e.g., A, B), which specifies the fields, names, and data types used in communication. The relationship between a subscriber and a topic is 1:1, as each subscriber registers to receive messages from a specific topic and MessageType. However, from the topic's perspective, the relationship is one-to-many since a single topic may have multiple subscribers. Similarly, each publisher is associated with a single topic (1:1), but a topic can have multiple publishers producing messages for it, thus forming a one-to-many relationship. Subscribers express interest in specific topics by registering for them and receive only the messages published to those topics, which are later processed through an event-handler. An event-handler is essentially a callback function used to process incoming messages of a specific MessageType. Each unique (Subscriber, Topic, MessageType) tuple has a dedicated queue and is associated with an event-handler. In some pub-sub systems, attempting to reuse the topic with different message types results in a runtime type-mismatch error. In such systems, the uniqueness of queues effectively reduces to the (Subscriber, Topic) pair, as MessageType does not matter when topic re-usability is not allowed. When a publisher

sends a message to a topic of a specific MessageType, the message is enqueued into the queues of all subscribers registered to that topic and MessageType. While event-handlers for the same MessageType can technically be reused and associated with different subscribers, this is seldom seen in practice. This is because each subscription typically has a unique purpose, requiring a specialized callback function to handle the message appropriately for that specific task.

At the core of the system is the queue, which plays a crucial role in maintaining system stability by buffering messages. Although implementation details may vary across different systems, a common design is to maintain a dedicated queue for each (Subscriber, Topic, MessageType) tuple, resulting in one queue per event-handler. The size of each queue is typically determined by a system-defined Quality of Service (QoS) parameter. These queues help prevent message loss during bursts of high-frequency publishing by temporarily storing incoming messages. However, it is important to note that these queues are of finite size. When a queue reaches its capacity, incoming messages may either overwrite older ones or be dropped, depending on the system's queue management policy. To process a message from the queue, the system relies on a dispatcher that is responsible for monitoring the queues and invoking the appropriate event-handler when a new message becomes available. We will explore the behavior of the dispatcher in detail in a later Section 2.3.1.1, particularly in the context of the underlying threading model. In the following section, we delve into the internal workings and limitations of pub-sub systems by examining various illustrative scenarios. We focus solely on the behavior of the functions explicitly shown by abstracting away other system components. For simplicity, we assume that all functions execute on a single machine unless explicitly stated otherwise in the examples, even though in practice they may be distributed across multiple systems. We further assume zero network delay, implying that messages[1] are delivered instantaneously. Additionally, we adopt a single-threaded execution model, meaning that only one event handler executes at a time.

[1]In a pub-sub system, the terms messages and events are often used interchangeably.

Publisher1:

```
1: while true do
2:     sleep(3)
3:     publish<TopicA>(msgA)
4: end while
```

Publisher2:

```
1: while true do
2:     sleep(5)
3:     publish<TopicA>(msgA)
4: end while
```

ASubscriber(msgA: A): // TopicA.queue_size ← 10

```
1: ... {takes 9 secs}
2: publish<TopicB>(msgB) {Max timing constraint for TopicB: 7 seconds}
```

BSubscriber(msgB: B): // TopicB.queue_size ← 10

```
1: ... {takes 20 secs}
```

Figure 1.2: Pseudocode of a pub-sub system with TopicB having a maximum timing constraint of 7 seconds.

1.2 Scenario-1 (Prolonged Event-handling Times)

In the pseudocode shown in Figure 1.2, two distinct publishers, Publisher1 and Publisher2, periodically publish messages to TopicA with the same MessageType A at intervals of 3 seconds and 5 seconds, respectively. In this setup, two queues are instantiated, each with a default size of 10, as messages are received by TopicA and TopicB, with their respective MessageTypes being A and B. At time intervals t = 3, 6, 9, 12, and 15, Publisher1 publishes messages that are enqueued into the queue associated with TopicA. Similarly, at t = 5, 10, and 15, Publisher2 publishes messages that are also added to the same queue. The ASubscriber event-handler processes messages from TopicA by dequeuing them from its corresponding queue. Each time a message from TopicA is dequeued and dispatched, the system triggers the ASubscriber event-handler. After processing the message for 9 seconds, it subsequently publishes a new message of MessageType B, which gets enqueued into the queue associated with TopicB. The BSubscriber event-handler is responsible for processing messages from TopicB that are enqueued into its corresponding queue.

To ensure timely communication in pub-sub systems, a maximum timing constraint[1] can be configured for each topic. While the previous statement does not explicitly specify who configures this value or how it is determined, it is typically defined by the package user i.e., the person who deploys and runs the package on the hardware. In such systems, the package user determines the maximum timing constraint (i.e., the maximum permissible gap between two consecutive messages) based on operational requirements, frequencies of the underlying devices, and insights gained from practical stress tests. In our work, we assume that this timing constraint is provided as an input to our tool, as elaborated in Section 3.1 with an example ROS package. In this example, TopicB is associated with a timing constraint of 7 seconds, meaning its event-handlers (in this case, BSubscriber) expect to receive a message at least once every 7 seconds. This timing constraint value is provided by the user, based on their domain knowledge and understanding of the expected message frequencies within the system. The responsibility of satisfying this constraint lies with the publishers of TopicB, in this case, the ASubscriber event-handler. This mechanism helps monitor and enforce that messages are published to topics within their specified time frames, ensuring the system meets real-time requirements while maintaining stability and efficiency. We assume the event-handlers are on different systems, so their execution order doesn't matter in this example.
Figure 1.3 illustrates the execution timeline corresponding to the pseudocode in

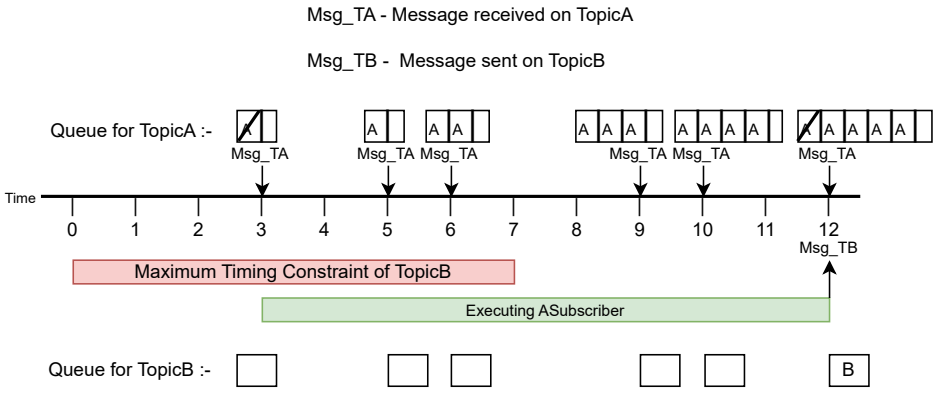


Figure 1.3: Execution timeline of the pub-sub system shown in Figure 1.2.

Figure 1.2. The boxes above the time axis represent the queue where messages for

[1]Timing constraint for a topic means we are expecting a message for that topic at least once within the specified time interval

TopicA are enqueued, while the boxes below the time axis represent the queue for TopicB. For simplicity, the queues are shown only at the points in time when a message is enqueued or dequeued from either queue. Publisher1 and Publisher2 enqueue messages into the queue corresponding to TopicA at periodic intervals of 3 seconds and 5 seconds, respectively. At time t = 3, a message from Publisher1 to TopicA is published and enqueued into its corresponding queue. Since there is no network delay, the message is immediately dequeued (represented with a strikethrough) and dispatched, triggering the execution of the ASubscriber event-handler (depicted in green). The event-handler takes 9 seconds to process the message before publishing a message to TopicB. However, as previously mentioned, TopicB has a maximum timing constraint of 7 seconds (illustrated in red), meaning that a message is expected to be published to TopicB at least once every 7 seconds. In this scenario, the message to TopicB is published at t = 12, whereas we are expecting it once before t $\leq$ 7, thus resulting in a timing violation. Such violations indicate that the system fails to meet its real-time requirements, potentially compromising reliability, especially in time-sensitive applications. To prevent timing violations and ensure the system remains in a safe operational state, the maximum timing constraint must be greater than 12 seconds. This value accounts for both the 9-second processing time and the 3-second delay before the message is initially published to TopicA. Only when the constraint exceeds 12 seconds can the system meet real-time guarantees, ensuring timely message delivery and maintaining operational stability.

Publisher1:

```
1: while true do
2:     sleep(3)
3:     publish<TopicA>(msgA)
4: end while
```

Publisher2:

```
1: while true do
2:     sleep(5)
3:     publish<TopicA>(msgA)
4: end while
```

ASubscriber(msgA: A): // TopicA.queue_size ← 10

```
1: if execute_long_path_ is true then
2:     ... {takes 10 secs}
3: else
4:     ... {takes 1 sec}
5: end if
6: publish<TopicB>(msgB) {Max timing constraint for TopicB: 7 seconds}
```

BSubscriber(msgB: B): // TopicB.queue_size ← 10

1: ... {takes 20 secs}

Figure 1.4: Pseudocode of a pub-sub system with a configuration parameter (execute_long_path_) and TopicB having a maximum timing constraint of 7 seconds.

1.3 Scenario-2 (Misconfiguration)

In the pseudocode shown in Figure 1.4, two distinct publishers, Publisher1 and Publisher2, periodically send messages to TopicA with the same MessageType A at intervals of 3 seconds and 5 seconds, respectively. In this setup, two queues are instantiated, each with a default size of 10, as messages are being received to TopicA and TopicB, with their respective MessageTypes being A and B. At time intervals t = 3, 6, 9, 12, and 15, Publisher1 publishes messages that are enqueued into the queue associated with TopicA. Similarly, at t = 5, 10, and 15, Publisher2 publishes messages that are also added to the same queue. The ASubscriber event-handler processes messages from TopicA by dequeuing them from its corresponding queue. Each time a message from TopicA is dequeued and dispatched, the system triggers the ASubscriber event-handler. After processing the message, it subsequently publishes a new message of MessageType B, which gets enqueued into the queue associated with TopicB. The execution time of the ASubscriber event-handler is determined by the value of the configuration parameter **execute_long_path_**. Configuration parameters are values loaded from external files. These parameters can dynamically influence the program's execution flow, enabling flexible testing and system adjustments without requiring changes to the source code. This design enhances the program's adaptability, supporting experimentation with different configurations and fine-tuning of system behavior as needed. If **execute_long_path_** value in the external file is true, its corresponding if-branch is taken and the execution time is 10 seconds. If the alternate branch is taken, the execution time reduces to 1 second. The BSubscriber event-handler is responsible for processing messages from TopicB that are enqueued into its corresponding queue. To ensure timely communication in pub-sub systems, a maximum timing constraint can be configured for each topic. In this example, TopicB is associated with a timing constraint of 7 seconds, meaning its event-handlers (in this case, BSubscriber) expect to receive a

message at least once every 7 seconds. This timing constraint value is provided by the user, based on their domain knowledge and understanding of the expected message frequencies within the system. The responsibility of satisfying this constraint lies with the publishers of TopicB, in this case, the ASubscriber event-handler. This mechanism helps monitor and enforce that messages are published to topics within their specified time frames, ensuring the system meets real-time requirements while maintaining stability and efficiency. We assume the event-handlers are on different systems, so their execution order doesn't matter in this example.
Figure 1.5 illustrates the execution timeline corresponding to the pseudocode in

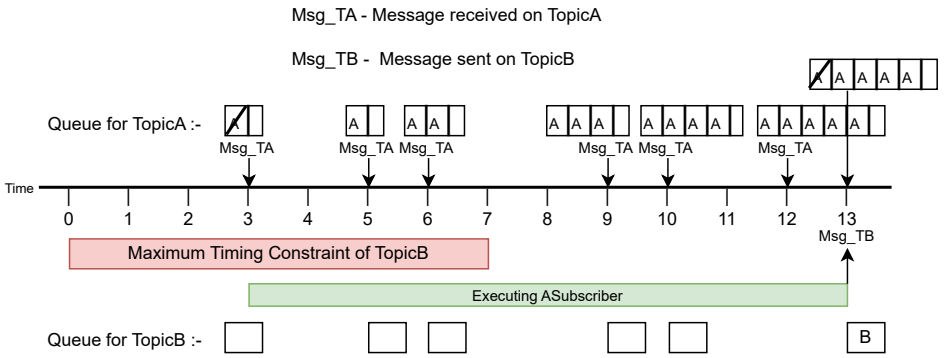


Figure 1.5: Execution timeline of the pub-sub system shown in Figure 1.4, following the if-branch.

Figure 1.4. The boxes above the time axis represent the queue where messages for TopicA are enqueued, while the boxes below the time axis represent the queue for TopicB. For simplicity, the queues are shown only at the points in time when a message is enqueued or dequeued from either queue. Publisher1 and Publisher2 enqueue messages into the queue corresponding to TopicA at periodic intervals of 3 seconds and 5 seconds, respectively. At time t = 3, a message from Publisher1 to TopicA is published and enqueued into its corresponding queue. Since there is no network delay, the message is immediately dequeued (represented with a strikethrough) and dispatched, triggering the execution of the ASubscriber event-handler (depicted in green). In a situation where the configuration parameter **execute_long_path_** is set to `true`, the time taken by the ASubscriber event-handler before publishing to TopicB (i.e., 10 seconds) exceeds the maximum timing constraint of 7 seconds (illustrated in red), resulting in a timing violation, as illustrated in Figure 1.5. To prevent timing violations and ensure the system remains in a safe operational state,

one solution is to set the **execute_long_path_** parameter to `false`, reducing the event-handler execution time to within acceptable limits. Alternatively, the timing constraint associated with TopicB could be relaxed and increased beyond 10 seconds to tolerate both the values of the configuration parameter. Either approach ensures the system adheres to its real-time constraints, enabling timely message delivery and preserving operational stability.

main:

1: register_subscriber(**BSubscriber**)
2: register_subscriber(**ASubscriber**)

Publisher1:

1: **while** true **do**
2: sleep(2)
3: publish<TopicA>(msgA)
4: **end while**

Publisher2:

1: **while** true **do**
2: sleep(2)
3: publish<TopicB>(msgB)
4: **end while**

ASubscriber(msgA: A): // TopicA.queue_size ← 10

1: ... {takes 2 secs}
2: publish<TopicC>(msgC) {Max timing constraint for TopicC: 4 seconds}

BSubscriber(msgB: B): // TopicB.queue_size ← 10

1: ... {takes 2 secs}
2: publish<TopicD>(msgD) {Max timing constraint for TopicD: 8 seconds}

CSubscriber(msgC: C): // TopicC.queue_size ← 10

1: ... {takes 15 secs}

DSubscriber(msgD: D): // TopicD.queue_size ← 10

1: ... {takes 20 secs}

Figure 1.6: Pseudocode of a pub-sub system with multiple incoming messages, TopicC and TopicD have maximum timing constraints of 4 seconds and 8 seconds, respectively.

1.4 Scenario-3 (Event-Handlers Ordering)

In the pseudocode shown in Figure 1.6, two distinct publishers, Publisher1 and Publisher2, periodically send messages to TopicA with MessageType A and to TopicB with MessageType B, respectively, at 2-second intervals. In this setup, four queues are instantiated, each with a default size of 10, as messages are being received to TopicA, TopicB, TopicC, and TopicD, with their respective MessageTypes being A, B, C, and D. At time intervals t = 2, 4, 6, 8, and 10, Publisher1 publishes messages that are enqueued into the queue associated with TopicA. Similarly, at the same intervals, Publisher2 publishes messages that are enqueued into the queue associated with TopicB. The ASubscriber event-handler processes messages from TopicA by dequeuing them from its corresponding queue. Each time a message from TopicA is dequeued and dispatched, the system triggers the ASubscriber event-handler. After processing the message for 2 seconds, it subsequently publishes a new message of MessageType C, which gets enqueued into the queue associated with TopicC. The CSubscriber event-handler is responsible for processing messages from TopicC that are enqueued into its corresponding queue. The BSubscriber event-handler processes messages from TopicB by dequeuing them from its corresponding queue. Each time a message from TopicB is dequeued and dispatched, the system triggers the BSubscriber event-handler. After processing the message for 2 seconds, it subsequently publishes a new message of MessageType D, which gets enqueued into the queue associated with TopicD. The DSubscriber event-handler is responsible for processing messages from TopicD that are enqueued into its corresponding queue. To ensure timely communication in pub-sub systems, a maximum timing constraint can be configured for each topic. In this example, TopicC and TopicD are configured with timing constraints of 4 seconds and 8 seconds, respectively. These constraints help monitor and enforce timely message publication, ensuring that the system meets real-time requirements while maintaining overall stability and efficiency. For this example, we assume that the event-handlers ASubscriber and BSubscriber execute on the same machine and same thread, while all others are distributed across different machines. When messages from multiple topics arrive simultaneously, the execution order of their respective event-handlers is determined

by subscription priority. Typically, priority is assigned based on the order of registration. When multiple messages arrive, the event-handler corresponding to the subscriber registered first is given priority to dequeue one message from its queue, followed by its execution. After this, event-handlers of other subscribers are considered in the registration order, each allowed to dequeue one message and dispatch its event-handler, provided messages are available in their respective queues. This ordering can be difficult to resolve statically when subscriptions are created conditionally, such as within if statements; therefore, the final order is determined at run time. In this example, BSubscriber is given priority over ASubscriber because BSubscriber was registered first, as shown in the **main** function in Figure 1.6. Figure 1.7

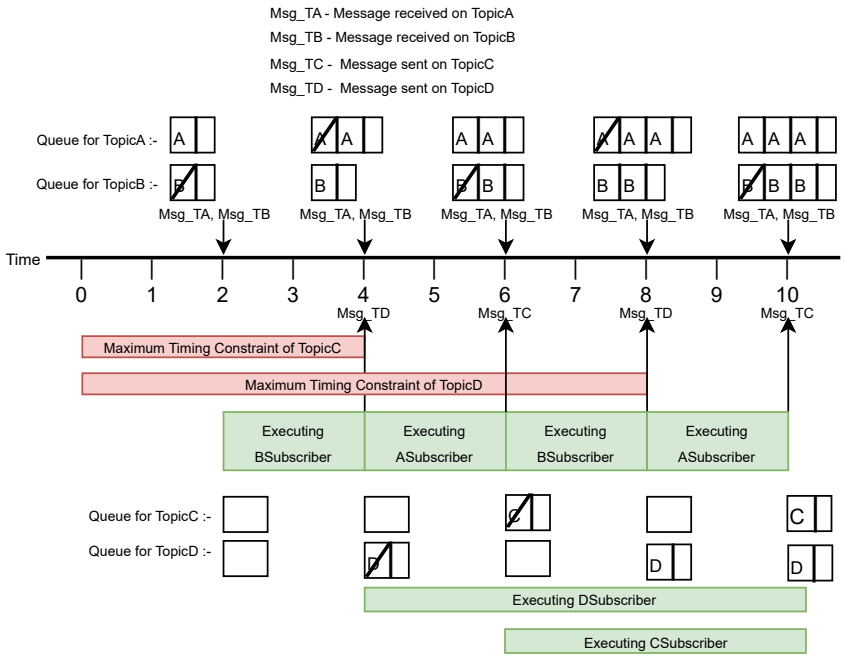


Figure 1.7: Execution timeline of the pub-sub system shown in Figure 1.6.

illustrates the execution timeline corresponding to the pseudocode in Figure 1.6. The boxes above the time axis represent the queues where messages for TopicA and TopicB are enqueued, while the boxes below the axis represent the queues for TopicC and TopicD, respectively. For simplicity, queues are shown only at the moments when messages are enqueued or dequeued. At time t = 2, messages from Publisher1 and Publisher2 are published to TopicA and TopicB, respectively, and enqueued

into their corresponding queues. Since these messages arrive simultaneously on the same machine and same thread, the event-handler for the topic with higher priority, BSubscriber in this case, is executed first. With zero network delay, the message is immediately dequeued (denoted with a strikethrough) and dispatched, triggering execution of the BSubscriber event-handler (shown in green). At time t = 4, BSubscriber finishes processing its message and publishes a new message to TopicD, satisfying its timing constraint. Once BSubscriber completes, the executor dequeues the message from the queue associated with TopicA, and the ASubscriber event-handler begins execution at t = 4. Simultaneously, on a different machine, the DSubscriber event-handler also begins execution at t = 4, triggered by the message published to TopicD. ASubscriber takes 2 seconds to process its message and publishes a new message to TopicC at t = 6. However, because the timing constraint for TopicC is 4 seconds (indicated in red from t = 0 to t = 4), while the TopicC message got published at t = 6, this results in a timing violation. Such violations indicate a failure to meet real-time constraints, which can compromise reliability in time-sensitive systems. To prevent this, the subscription registration order should be swapped, giving priority to ASubscriber over BSubscriber. This adjustment ensures that messages are processed in an order that respects timing guarantees, where TopicC, with a timing constraint of 4 seconds, receives a published message at least once every 4 seconds, and TopicD, with a constraint of 8 seconds, also meets its requirement, thereby enabling the system to satisfy its real-time constraints and maintain operational stability.

1.5 Applications of Publish-Subscribe Systems

The pub-sub systems are widely used in various domains where decoupled, scalable, and event-driven communication is essential. These systems enable efficient data dissemination by allowing publishers to send messages to specific topics, which subscribers can listen to without direct dependencies on the publishers. In distributed systems, pub-sub architectures are used for real-time messaging, such as in financial trading platforms [48, 20], where stock market updates are broadcast to multiple subscribers instantaneously. Similarly, in IoT (Internet of Things) applications [18, 43], pub-sub mechanisms facilitate communication between sensors, edge devices, and cloud services, enabling real-time monitoring of smart homes [54, 3], industrial automation [12], and connected vehicles [52, 22, 4]. In the field of au-

tonomous robotics, frameworks like ROS [36] utilize pub-sub models [37, 41, 31, 45] to handle communication between different robotic components, such as sensors [23] and controllers [35]. In a Graphical User Interface (GUI) [58], when a user clicks a button, the event (e.g., "click") is published to an event system, and multiple subscribers (event listeners) can respond accordingly, such as updating the User Interface or triggering an action. Cloud platforms like AWS [5], IBM [21], Google Cloud [17], and Azure [6] provide managed pub-sub services for event-driven architectures, making them integral to modern software ecosystems.

1.6 Timing Issues in Real Software

While the scenarios in Sections 1.2–1.4 are intentionally constructed to illustrate timing issues, similar timing challenges arise in real-world pub-sub systems. Factors such as event-handlers execution order and misconfigurations can cause delays, priority inversions, or missed deadlines. The Robot Operating System (ROS)[1] is a widely used open-source pub-sub framework for building distributed robotic systems. Despite its name, ROS is not an operating system but rather a collection of libraries that run on top of an existing OS to facilitate robotic development. Prior studies [56, 60, 19, 29, 30] show that long event-handler latencies can cause buffer overflows and message loss, which in real-time systems may lead to unpredictable behavior or crashes. Gog et al. [16] found that misconfigured message arrival timings in autonomous vehicles led to increased collision risk. As detailed on Page 13 in their paper, their findings clearly demonstrate that "configurations with higher response times collide with the person at collision speeds that increase with the response time." This highlights that the longer it takes for the system to process information and react, the greater the risk of a collision. While the paper specifically discusses response times and deadline timings, the underlying cause of such delays can often be attributed to the late arrival of crucial sensor messages into the system. If critical information, such as the detection of a pedestrian or another vehicle, arrives after its effective deadline for processing, the system's ability to compute a safe trajectory and execute a timely maneuver is severely compromised, directly leading to an increased likelihood of collisions and, consequently, jeopardizing overall system safety. Similarly, Li et al. [28] demonstrated how delayed messages in ROS applications introduced deliberately as part of an attack caused deadline misses and

[1] We refer to ROS2, as ROS1 is no longer supported.

system crashes. One example in the path planning module showed a robot using stale map data due to a delayed message, causing it to crash into an obstacle it failed to detect.

While these studies primarily focus on the runtime behavior of the ROS scheduler or application-level execution, we observed that in certain ROS applications, messages are published from the event-handlers. While this approach may be necessary for specific functional requirements, it introduces complex timing and synchronization challenges. It also impacts the predictability of system behavior, as message propagation timing becomes heavily dependent on event-handlers execution order and latency. Additionally, developers configure several tunable parameters, including message arrival timings, queue sizes, and Quality-of-Service (QoS) attributes such as security, etc. These parameters are typically set based on domain knowledge and experience. However, overlooking critical architectural aspects such as underlying scheduling mechanisms and concurrency behavior when defining these values can lead to unintended timing violations and failure to meet real-time constraints. If not carefully managed, these issues can result in unpredictable system behavior, potentially leading to system failures or crashes.

1.7 Contributions

Contributions made in this thesis are summarized below:

- We highlight the potential timing issues in event-driven applications and propose a framework to identify them.
- An implementation of our approach.
- A case study-based discussion on potential timing issues in selected packages.

1.8 Outline

The rest of the thesis is organised as follows:

- **Chapter 2** presents the necessary background on Timed Automata, UPPAAL, and the Robot Operating System (ROS), which are essential for understanding the concepts and methodologies discussed in this thesis.

- **Chapter 3** defines the problem statement formally, and the algorithms used to construct the Timed Automata model.
- **Chapter 4** explains the various implementation aspects of the framework, focusing on the algorithms and heuristics applied.
- **Chapter 5** evaluates the proposed framework on a variety of ROS packages and presents a case study to illustrate its effectiveness in identifying potential timing-related issues.
- **Chapter 6** explores potential directions for extending the current work, discussing improvements and new features that could enhance the framework's capabilities.
- **Chapters 7** and **8** summarize related research efforts in this area and conclude the thesis with a discussion on key takeaways and future possibilities.

Chapter 2

Background

This chapter presents the essential background required to understand the remainder of the thesis. We begin by introducing Timed Automata [42] and illustrate their behavior through examples, demonstrating how these models can be verified using a tool called UPPAAL [26]. Later, we shift our focus to the Robot Operating System [36] (ROS), an example of a pub-sub system, and it is the target of our experimental analyses, highlighting commonly used components and explaining how tasks are scheduled and executed within the ROS framework. This chapter lays the foundation for the concepts and techniques developed in the subsequent chapters.

2.1 Timed Automata

In this section, we give a brief introduction to the definition and semantics of Timed Automata (TA).

Definition 2.1. ***(Timed Automata).*** *A timed automaton (χ) is a tuple (L, L^0, A, C, E, I), where*

- *L is a finite set of locations,*
- *$L^0 \subseteq L$ is a finite set of initial locations,*
- *A is a finite set of actions,*
- *C is a finite set of clocks,*
- *$E \subseteq L \times A \times 2^C \times \Phi(C) \times L$ is the finite set of edges. An edge $(l, a, \lambda, \phi, l_1)$ represents an transition from location l to location l_1, on action a. ϕ is a clock constraint from the*

set $\Phi(C)$ of clock constraints that determines when the edge is enabled and the transition can occur, while $\lambda \subseteq C$ represents the set of clocks that are reset on this edge.

- *Clock constraints (ϕ) is defined by*

$$\phi := true \mid false \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid x \bowtie k \mid x - y \bowtie k$$

where x, $y \in C$ are clocks, $k \in \mathbb{N}$ and $\bowtie \in \{<, \leq, ==, \geq, >\}$. Only these types of constraints are allowed as invariants and enabling conditions. Few Examples are $x \leq 2$, $x == 10$, etc.

- *I: $L \rightarrow \Phi(C)$, is an mapping function that maps each location in L with some clock constraint (invariants) in $\Phi(C)$.*

Definition 2.2. ***(Operation Semantics).***
A clock interpretation is a function $u : C \rightarrow \mathbb{R}^+$, where each clock in C is mapped to a non-negative real number representing the current time value since it was last reset, and

$Y \subseteq C, \forall x \in C$ *and* $d \in \mathbb{R}^+$,

Time Elapse:	$(u + d)(x) = u(x) + d,$
Resetting Clock:	$u[Y := 0](p) = 0,$ *if* $p \in Y, u(p)$ *otherwise*

The semantics of the timed automata χ is defined by associating a transition system P_χ with it. A state of P_χ is a pair (l, c) where l is a location in χ and c is a clock interpretation. Let (l_0, c_0) denote the initial state, where l_0 denotes the initial location in χ and $c_o(x) = 0$ for all clocks x. The transitions in P_χ can occur due to 2 types:

- ***Delay*** *: For state (l, c) and real-valued time increment $t \geq 0$, $(l, c) \xrightarrow{t} (l, c + t)$ if for all $0 \leq t^1 \leq t$, $c + t^1$ satisfies the invariant $I(l)$.*
- ***Action*** *: For state (l, c) and edge $(l, a, \phi, \lambda, l^1)$, $(l, c) \xrightarrow{a} (l^1, c^1)$ such that $c \in \phi$, $c^1 \in I(l^1)$ and $c^1 = c[\lambda := 0]$.*

We use the following notational conventions for elements of timed automata throughout this chapter.

1. Clocks are represented in bold. For example: clock **signal**,
2. Actions are underlined. For example: action <u>go</u>,

3. Location names are denoted using *italic* styling, and the initial location is indicated with an arrow above it. For Example: Location *test*,

4. Variables are written using teletype font and initialized with zero (for integer) and false (for boolean). For example: int `shared_counter`.

2.1.1 Example of a Timed Automata

Consider the timed automaton in Figure 2.1, where (*Red*, *Green*, *Yellow*) represent locations in which time can elapse, constrained by invariants. Transitions occur when guard conditions on the clock **signal** are satisfied, potentially resetting the clock. For example, in the traffic signal system: at location *Red*, the invariant **signal** ≤ 10, limits the duration of stay at that location to a maximum of 10 units, while the guard on the edge from the *Red*, **signal** ≥ 5 ensures at least 5 units pass before transitioning to *Green* via action go, resetting the clock **signal** while at it. At *Green*, the invariant **signal** ≤ 15 and guard **signal** ≥ 10 enforce a stay of 10–15 units before transitioning to *Yellow* on action slow, resetting **signal**. Similarly, *Yellow* enforces a stay of 2–5 units with invariant **signal** ≤ 5 and guard **signal** ≥ 2, then transitions back to *Red* on action stop, resetting **signal** again. This cycle repeats indefinitely.

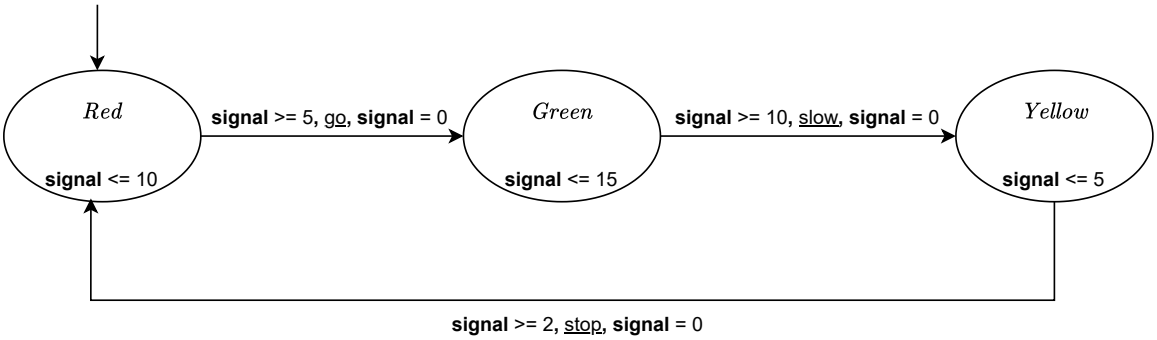


Figure 2.1: An example of timed automaton (traffic_sys).

Transitions are expressed as a tuple (*location*, clock), with edges labeled by time constraints, actions, or both. The transitions represent how long the clock **signal** can stay at each location. We present several valid transitions for the automaton system, each of which is a prefix of an infinite trace.

1. $(Red, 0) \xrightarrow{5.5} (Red, 5.5) \xrightarrow{go} (Green, 0) \xrightarrow{12} (Green, 12) \xrightarrow{slow} (Yellow, 0) \xrightarrow{3} (Yellow, 3) \xrightarrow{stop} (Red, 0)$

(a) **Starting at Red:** We begin at inital location *Red* with the clock **signal** at 0, represented by the tuple $(Red, 0)$. Our system allows a stay of 5-10 units at *Red* before transitioning. In this specific trace, we wait for 5.5 units, moving to the state $(Red, 5.5)$.

(b) **Transition to Green:** After 5.5 units, we use the $\underline{go}$ action, which transitions us from *Red* to *Green*. Upon entering *Green*, the clock resets to 0, resulting in the state $(Green, 0)$.

(c) **Staying in Green:** The system can remain in the *Green* location for 10-15 units. In this trace, we stay for 12 units, reaching $(Green, 12)$.

(d) **Transition to Yellow:** Next, the $\underline{slow}$ action is triggered, moving us from *Green* to *Yellow*. The clock resets again, taking us to $(Yellow, 0)$.

(e) **Staying in Yellow:** The *Yellow* location permits a stay of 2-5 units. Here, we remain for 3 units, arriving at $(Yellow, 3)$.

(f) **Returning to Red:** Finally, the $\underline{stop}$ action is performed, transitioning us back to *Red*. The clock resets to 0 once more, bringing us full circle to $(Red, 0)$, completing one cycle of the traffic signal.

2. $(Red, 0) \xrightarrow{7.5,\underline{go}} (Green, 0) \xrightarrow{11.4,\underline{slow}} (Yellow, 0) \xrightarrow{4.1,\underline{stop}} (Red, 0)$

3. $(Red, 0) \xrightarrow{1} (Red, 1) \xrightarrow{6,\underline{go}} (Green, 0) \xrightarrow{4.2} (Green, 4.2) \xrightarrow{10} (Green, 14.2) \xrightarrow{\underline{slow}} (Yellow, 0) \xrightarrow{3.14} (Yellow, 3.14) \xrightarrow{\underline{stop}} (Red, 0)$

2.1.2 Properties in Timed Automata

In the context of timed automata, a property refers to a statement about the system's behavior over time, often involving clock constraints. Computation Tree Logic (CTL) [44] belongs to the family of branching-time logics and is used to describe properties through the use of path quantifiers and temporal operators. The path quantifiers, A (for all computation paths) and E (for some computation path), specify the branching structure, determining whether a given property holds for all or some paths originating from a specific state. Temporal operators, which must be immediately preceded by a path quantifier, define the behavior along these paths. Two common temporal operators are F (eventually or in the future), which asserts that a property will hold in some state along the path, and G (always or globally), which states that a property holds in every state on the path. Building on this foundation, different combinations of path quantifiers and temporal operators allow the specification of more expressive system properties:

- $AG\ \varphi$: Asserts that *for all paths*, the property φ holds *globally* i.e., at every state along every possible path, φ is always true.
- $AF\ \varphi$: Indicates that *for all paths*, the property φ holds *eventually*, i.e., no matter what path is taken, the system will eventually reach a state where φ is true.
- $EG\ \varphi$: Specifies that there *exists a path* where the property φ holds *globally*, i.e., there is at least one possible path in which φ is always true.
- $EF\ \varphi$: Denotes that there *exists a path* where the property φ holds *eventually* i.e., it is possible to reach a state where φ is true.
- $A(\varphi_1 U_{\bowtie k} \varphi_2)$: Denotes that *across all paths*, φ_2 is guaranteed to become true at some time k within the constraint $\bowtie k$, and φ_1 holds continuously from now until that time k.
- $E(\varphi_1 U_{\bowtie k} \varphi_2)$: Denotes that there *exists at least one path* where φ_2 becomes true at some time k within the constraint $\bowtie k$, and φ_1 holds continuously from now until that time k.

Timed Computation Tree Logic (TCTL) extends CTL by incorporating timing constraints, allowing the specification and verification of real-time properties such as deadlines, response times, and safety conditions. The formula of TCTL for model checking can be defined as follows:

$$\begin{aligned}\varphi := p \mid true \mid x \bowtie k \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid A(\varphi_1 U_{\bowtie k} \varphi_2) \mid E(\varphi_1 U_{\bowtie k} \varphi_2) \\ \mid AG\varphi \mid EG\varphi \mid AF\varphi \mid EF\varphi\end{aligned}$$

where,

- p is an atomic proposition, which is of the form: l, where $l \in Locations$ or $expr_1 \bowtie expr_2$ where $\bowtie \in \{==, \neq, <, \leq, \geq, >\}$ and $expr_i$ denotes the set of all possible arithmetical expressions over integer variables. While standard timed automata typically do not include integer variables, we utilize them in the context of extended timed automata, which will be defined in a subsequent section.
- x is a clock, $k \in \mathbb{N}$, and $\bowtie \in \{<, \leq, ==, \geq, >\}$,

We use the following two properties to verify the timed automata shown in Figure 2.1

- $AF\,Green$: This property ensures that the system cycles through locations properly and does not get stuck in the *Red* or *Yellow* indefinitely. In other words, it guarantees that

on all possible paths, the system will eventually reach *Green*. In the given system, this property is satisfied because the system is designed to transition cyclically through the locations *Red*, *Green*, and *Yellow* in a fixed order. Since there are no deadlocks or loops that exclude *Green*, it is guaranteed that every computation path will eventually reach *Green*. Thus, *all traces* of the system satisfy the property. To illustrate this, we present *one such trace* among the many possible traces that satisfy the property. Additionally, we show only a *prefix* of that trace, a finite segment that is sufficient to demonstrate that the *Green* is eventually reached, thereby satisfying the property.

$$(Red, 0) \xrightarrow{7.5,go} (Green, 0) \xrightarrow{11.4,slow} (Yellow, 0) \xrightarrow{4.1,stop} (Red, 0) \xrightarrow{5.6,go} (Green, 0)$$

- *EF* $(Red \wedge \textbf{signal} > 10)$: This property checks whether there exists at least one path in which the system eventually reaches *Red* location and the clock **signal** has a value greater than 10. In other words, it asks: "Is it possible for the Red signal to remain active for more than 10 time units?" This property is *not satisfied* in the current system model. The reason is that the system is designed to reset the clock **signal** to 0 each time it transitions to a new location. Additionally, the *Red* location has an invariant that restricts the system from remaining in that location beyond 10 time units. As a result, no trace can remain in the *Red* with **signal** exceeding 10. Since the clock is reset upon each transition and cannot accumulate beyond the enforced time while in *Red*, no path can satisfy this property.

  To demonstrate this, we provide a finite prefix of a trace that does *not* satisfy the property:

$$(Red, 0) \xrightarrow{7.5,go} (Green, 0)$$

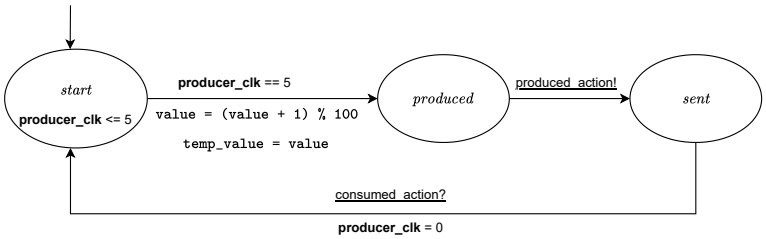
2.1.3 Product of Timed Automata

The product of timed automata is a method used to combine multiple timed automata into a single system that captures their synchronized behavior. Given two timed automata $\chi_1 = (L_1, {L^0}_1, A_1, C_1, E_1, I_1)$ and $\chi_2 = (L_2, {L^0}_2, A_2, C_2, E_2, I_2)$, assuming that their clock sets C_1, C_2 are disjoint. Then, the product automata $\chi_1 \parallel \chi_2$ is $\langle L_1 \times L_2, {L^0}_1 \times {L^0}_2, A_1 \cup A_2, C_1 \cup C_2, E, I \rangle$, where $I(l_1, l_2) = I(l_1) \wedge I(l_2)$ and the edges (E) are defined by:

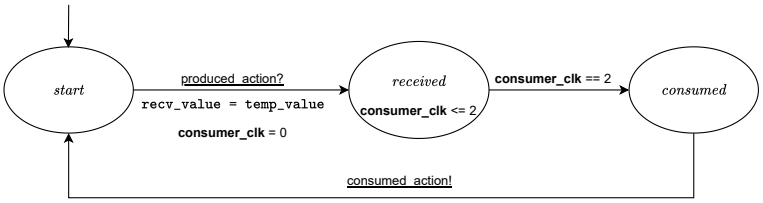
1. For $a \in A_1 \cap A_2$, for every $\langle l_1, a, \phi_1, \lambda_1, l_1^1 \rangle$ in E_1 and $\langle l_2, a, \phi_2, \lambda_2, l_2^1 \rangle$ in E_2, E has $\langle (l_1, l_2), a, \phi_1 \wedge \phi_2, \lambda_1 \cup \lambda_2, (l_1^1, l_2^1) \rangle$.

2. For $a \in A_1 \backslash A_2$, for every $\langle l, a, \phi, \lambda, l^1 \rangle$ in E_1 and every t in L_2, E has $\langle (l, t), a, \phi, \lambda, (l^1, t) \rangle$.

3. For $a \in A_2 \backslash A_1$, for every $\langle l, a, \phi, \lambda, l^1 \rangle$ in E_2 and every t in L_1, E has $\langle (t, l), a, \phi, \lambda, (t, l^1) \rangle$.



(a) Producer Timed Automata (producer_sys).



(b) Consumer Timed Automata (consumer_sys).

Figure 2.2: An example illustrating two interacting timed automata.

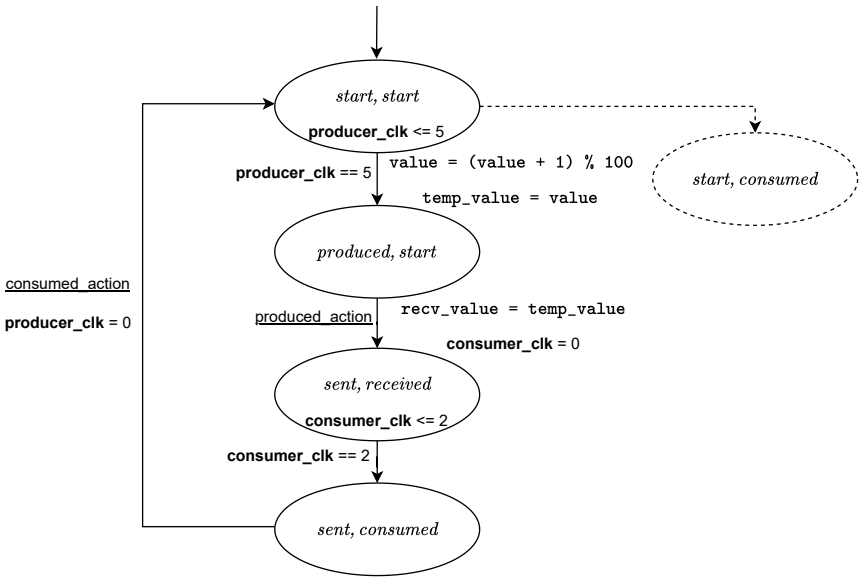


Figure 2.3: Product automaton of the example shown in Figure 2.2.

The locations of the product are the pairs of component-locations, invariants of the product location are the conjunction of the invariants of component-locations, and edges are obtained

by synchronizing with shared actions. The clocks and constraints from both automata are also considered together, ensuring correct timing behavior. The product of automata is usually obtained while performing model verification.

2.1.4 Example of Two Interacting Timed Automata

A producer takes 5 time units to generate a value and then signals the consumer. Upon receiving this signal, the consumer takes 2 time units to process the value and respond. In this model, we use variables to store and transfer data between the two automata and synchronized actions, denoted by ! (send) and ? (receive), to coordinate transitions between them. When two automata have matching actions (one with action! and the other with action?), they can only perform those transitions simultaneously, enabling precise modeling of interaction and communication. A formal definition for variables and synchronized actions will be provided in the next section when introducing UPPAAL. We assume all variables defined in the automata are initialized to zero. Among these, `temp_value` is a shared global variable accessible by both automata, while `value` and `recv_value` are local to the producer and consumer automata, respectively.

Figure 2.2 models this interaction using two timed automata named producer_sys and consumer_sys. The actions produced_action and consumed_action are used to synchronize transitions between the producer and consumer automata. In the producer automaton, the location *start* includes the invariant $\textbf{producer\_clk} \leq 5$, and the edge from *start* to *produced* has the guard $\textbf{producer\_clk} == 5$, ensuring the transition occurs exactly at 5 time units. As part of this transition, the variable `value` is incremented and taken modulo 100, and the resulting `value` is then stored in `temp_value`. The subsequent transition to *sent* uses the produced_action! to synchronize with the consumer's produced_action?, causing the consumer to move from *start* to *received*, where `recv_value` is updated. In the consumer automaton, the location *received* has the invariant $\textbf{consumer\_clk} \leq 2$, and the edge to *consumed* is guarded by $\textbf{consumer\_clk} == 2$, enforcing exactly 2 units of processing time. It then transitions back to *start*, synchronizing with the producer via the consumed_action. This cycle repeats indefinitely.

Figure 2.3 shows the product automaton derived from this interaction. While we only display the valid combined locations and transitions of the product automaton, one additional unreachable location and its transition are included with a dotted line for illustrative purposes. The system always transitions from the initial location (*start, start*) to (*produced, start*) because the producer must remain in its *start* location until its clock reaches exactly 5 time units, as enforced by the invariant $\textbf{producer\_clk} \leq 5$ and the guard $\textbf{producer\_clk} == \textbf{5}$. At that point, it independently transitions to *produced*, incrementing the `value` and taking modulo 100, fol-

lowed by updating `temp_value`. Meanwhile, the consumer remains in its *start* location, which has no timing constraint and no enabled transitions until synchronization occurs. Therefore, (*produced, start*) is the only valid next location from (*start, start*). Other possible transitions, such as to (*start, received*) or (*start, consumed*), are invalid at this stage, as they would require the consumer to move independently, which it cannot do without first synchronizing with the producer. Similarly, transitions involving synchronized actions like produced_action or consumed_action cannot occur until both automata are in the appropriate locations. Additionally, in the figure, the synchronized actions produced_action? and produced_action! are combined and labeled simply as produced_action for clarity.
We use the following two properties to verify the two timed automata shown in Figure 2.2

- *AG* (producer_sys.*sent* $\implies$ (consumer_sys.`recv_value` == producer_sys.`value`)): This property verifies that whenever the system reaches the *sent* location in the producer automaton, the value received in the consumer automaton `recv_value` is always equal to the value produced by the producer `value`. In other words, once the signal has been successfully sent to the consumer, the data integrity between the producer and consumer must be preserved. This property is satisfied. All traces of the system satisfy this property. To illustrate this, we present one example trace from the many possible traces that fulfill the property. Additionally, we show only a prefix of the trace, a finite segment that is sufficient to demonstrate that the property holds below:

Producer Location	*start*	*produced*	*sent*	*sent*	*start*
producer_clk	0	5	7.7	9.7	0
Consumer Location	*start*	*start*	*received*	*consumed*	*start*
consumer_clk	0	5	0	2	4.1

- *EF* (consumer_sys.*consumed* and producer_sys.*produced*): This property checks whether there exists a case where a value is consumed before it is produced. The property is not satisfied. However, this is expected, as the synchronization of actions between the producer and consumer ensures that such a scenario cannot occur. Specifically, the system guarantees that we will never reach the *consumed* location in the consumer automaton while still being in the *produced* location of the producer automaton. To demonstrate this, we provide a finite prefix of a trace that does *not* satisfy the property:

Producer Location	*start*	*produced*	*sent*	*sent*	*start*
producer_clk	0	5	13.4	15.4	0
Consumer Location	*start*	*start*	*received*	*consumed*	*start*
consumer_clk	0	5	0	2	9.9

2.2 UPPAAL

At its core, UPPAAL[26] is a specialized tool for modeling, simulating, and verifying systems using networks of timed automata. A network of timed automata is a set of individual automata that execute concurrently and interact via synchronization channels and shared variables, enabling the modeling of complex system behavior. It extends classical timed automata by incorporating features such as data variables, complex data types, synchronization mechanisms via channels, and urgent/committed locations, enabling a more expressive representation of real-time behavior. However, it restricts the use of TCTL for verification by disallowing nested TCTL formulas, with timing aspects limited to clock constraints in conditions. Additionally, the temporal operators G and F are denotes as $[]$ and $<>$, respectively. The verification engine can be used to formally verify the following properties:

- **Reachability:** Whether a specific location can be reached.
  - **Example:** Checking if a location named *unsafe* is reachable.
  - **TCTL:** $E <>$ *unsafe*.
- **Safety:** Ensures something bad never happens.
  - **Example:** Temperature should never exceed a certain `threshold`.
  - **TCTL:** $A[]$ (**temperature** $\leq$ `threshold`)
- **Liveness:** Ensures that something good eventually happens.
  - **Example:** A resource is always eventually granted.
  - **TCTL:** $A[]$ (*resource* $\implies$ (A$<>$ *grant*)).
  - This is a special case, as UPPAAL only allows this particular property to be expressed with nesting [32].
- **Deadlock freedom:** Confirms no system component halts indefinitely.
  - **TCTL:** $A[]$ (not deadlock)

– "deadlock" is a special keyword in UPPAAL which can be used accordingly.

These capabilities make UPPAAL particularly useful for verifying real-time embedded systems, communication protocols, and safety-critical applications. When a property is violated, UPPAAL generates a single counterexample trace, aiding in debugging by pinpointing errors.

2.2.1 Extended Timed Automata

In this section, we formally introduce the definitions and semantics of Extended Timed Automata (ETA) by referring to the work presented in [49]. To support features such as discrete variables, function calls, non-blocking channel-based synchronization via broadcast, and location constraints (urgent and committed), we extend classical timed automata with additional constructs. These enhancements enable the modeling of data-dependent behavior, precise timing control, and coordinated interactions among automata. In general, timed automata are composed into a network of timed automata, typically consisting of a set of n processes, each representing a timed automaton, as done in tools like UPPAAL. Each automaton operates over a shared set of clocks and actions. Synchronization between processes is achieved through input and output actions, enabling tightly coupled communication. To model synchronization, we assume the action alphabet A includes:

1. Input actions (receive) denoted as action?,
2. Output actions (send) denoted as action!,
3. Internal (local) actions denoted as action.

The resulting Extended Timed Automata formalism integrates variable updates, function calls, clock resets, transition guards, and enriched location types into a unified transition model, offering a flexible yet precise framework for modeling and verifying real-time systems.

Definition 2.3. ***(Extended Timed Automata).*** *Let* $\mathcal{T} = \langle L, L^0, \mathcal{F}, \mathcal{V}, A, C, E, I \rangle$, *where*

- L *is a finite set of locations, ranged over by* l,
  - *Locations can be marked urgent (#U) or committed (#C) or not marked at all.*
  - *Urgent locations: Time is not allowed to progress, but transitions can still be delayed if guards are not satisfied.*
  - *Committed locations: Time is not allowed to progress, and the next transition must involve only committed locations, and no interleaving with other automata is allowed.*

- *L^0 is the initial location, allowing only one location as the initial location,*
- *$\mathcal{F}$ is a set of function declarations, where f is an non-recursive function used for updating the discrete variables $\mathcal{V}$,*
  - *$funcCall := f(expr_1, expr_2, \cdots, expr_n)$, where $f \in \mathcal{F}$*
- *$\mathcal{V}$ is a finite set of discrete variables, ranged over by v,*
  - *Variable valuation ω is mapping from set of variables $\mathcal{V}$ to set of integers $\mathbb{Z}$.*
  - *$expr := m \mid v \mid expr \bowtie expr \mid -expr \mid funcCall$, where $\bowtie \in \{\times, +, -, \div\}$, $m \in \mathbb{Z}$, $v \in \mathcal{V}$. The division operator ($\div$) in the expression grammar represents UPPAAL's C-style semantics of integer division. Since the domain of variable valuations (ω): $\mathcal{V} \to \mathbb{Z}$ is restricted to integers, the result of any arithmetic expression on division must also be an integer. This ensures the semantics remain consistent within the integer domain and no non-integer results arise.*
  - *Act($\mathcal{V}, \mathcal{F}$) denotes set of all assignments over $\mathcal{V}$ and $\mathcal{F}$.*
    - *single_act := funCall $\mid v = expr \mid$ skip, where $v \in \mathcal{V}$*
    - *act_seq := $\epsilon \mid$ single_act act_seq*
  - *$\Phi(\mathcal{V}, \mathcal{F})$ denotes set of all boolean expression over $\mathcal{V}$ and $\mathcal{F}$.*
- *A is a finite set of actions, ranged over by a,*
- *C is a finite set of clocks, ranged over by u,*
  - *Clock constraints (Ψ) is defined by*

$$\psi := true \mid false \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \neg\psi \mid x \bowtie k \mid x - y \bowtie k$$

    where x, $y \in C$ are clocks, $k \in \mathbb{N}$ and $\bowtie \in \{<, \leq, ==, \geq, >\}$. Only these types of constraints are allowed on edge-guards and location-invariants. Few Examples are $x \leq 2$, $x == 10$, etc.
  - *Asg(C) denotes set of all assignments over C*
    - *single_asg := $u = expr \mid$ skip, $u \in C$*
    - *asg_seq := $\epsilon \mid$ single_asg asg_seq*

- $E \subseteq L \times A \times \eta(\Phi(\mathcal{V}, \mathcal{F}), \Psi(C)) \times Asg(C) \times Act(\mathcal{V}, \mathcal{F}) \times L$ *is finite set of edges where* $\eta(\Phi(\mathcal{V}, \mathcal{F}), \Psi(C))$ *denotes the set of all conjunctions over* $\Phi(\mathcal{V}, \mathcal{F})$ *and* $\Psi(C)$ *and* $Update(\mathcal{V}, C, \mathcal{F})$ *is* $Act(\mathcal{V}, \mathcal{F}) \times Asg(C)$ *and defined by the rule:*
  - *update* $:= \epsilon \mid update\ update \mid act\_seq \mid asg\_seq$, *where single_act and single_asg are defined in* $Act(\mathcal{V}, \mathcal{F})$ *and* $Asg(C)$.
  - *An edge* $(l_i, a, \psi, \lambda, v, l_j)$ *represents an transition from location* l_i *to location* l_j, *on action* a. ψ *is the clock constraints from* $\eta(\Phi(\mathcal{V}, F), \Psi(C))$ *that determine when the edge is enabled and a transition can occur.* λ *represents all the clocks that needs to be reset (*$Asg(C)$*) and* v *represents function calls and variable updates (*$Act(\mathcal{V}, F)$*).*
- I*:* $L \rightarrow \eta(\Phi(\mathcal{V}, \mathcal{F}), \Psi(C))$, *is an mapping function that maps each location in* L *with some clock constraint (invariants) in* $\Psi(C)$.

Definition 2.4. ***(Operation Semantics).*** *We give a brief overview of semantics in extended timed automata, for in-depth semantics, refer to Section 3.2 of Slomp, G. H.'s work [49]. Let* $\mathcal{T}_i = \langle L_i, {L^0}_i, \mathcal{F}_i, \mathcal{V}_i, A_i, C_i, E_i, I_i \rangle$ *where* $1 \leq i \leq n$ *be set of* n *timed automata. A network of* n *timed automata* $\mathcal{T}_i$ *written as* $(\mathcal{T}_1 \parallel \mathcal{T}_2 \parallel \cdots \parallel \mathcal{T}_n)$ *is defined in terms of a transition system* $(S, s_0, \rightarrow)$ *where* $\bar{l} = (l_1, l_2, \cdots, l_n)$ *is an location vector and* $S = (L_1, L_2, \cdots, L_n) \times C \times \mathcal{V}$ *is the set of states. Updates to the location vector are written as* $\bar{l}[\hat{l}_i / l_i]$ *to denote that extended timed automata* $\mathcal{T}i$ *moves from location* l_i *to* $\hat{l}_i$. *Let* $\langle \bar{l}, \sigma, w \rangle$ *be element in set of states* S. *The transition relation* $\rightarrow \subseteq S \times S$ *is defined using the following rules:*

- ***Delay:***

$$\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{d} \langle \bar{l}, \sigma + d, \omega \rangle \text{ if:}$$

  - $\forall d'$ *where* $0 \leq d' \leq d : (\sigma + d', \omega) \models I(\bar{l})$
  - *And* $\forall d', l \in \bar{l} : d'$ *does not result in an edge* e *being enabled for any* l, *which is either* urgent *or* committed.

- ***Action:***

$$\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{a} \langle \bar{l}[l'_i / l_i], \sigma', \omega' \rangle \text{ if:}$$

  - *There exists an edge* $l_i \xrightarrow{a, \psi, \lambda, v} l'_i$ *where*
  - $(\sigma, \omega) \models \psi$
  - $a = \epsilon$
  - $(\sigma', \omega') = [update](\sigma, \omega)$

- $(\sigma', \omega') \models I(\bar{l}[l'_i/l_i])$
- *And l_i is committed or there is no edge e that is enabled for any l_j such that l_j is committed*

- ***Sync:***

$$\langle \bar{l}, \sigma, \omega \rangle \xrightarrow{\tau} \langle \bar{l}[l'_i/l_i, l'_j/l_j], \sigma', \omega' \rangle \text{ if:}$$

  - *There exist edges $l_i \xrightarrow{a!,\psi_i,\lambda_i,v_i} l'_i$ and $l_j \xrightarrow{a?,\psi_j,\lambda_j,v_j} l'_j$ and a state (σ'', ω'') such that:*
    - $(\sigma, \omega) \models \psi_i \wedge \psi_j$
    - *(output)* $(\sigma'', \omega'') = [update](\sigma, \omega)$ *(followed by)*
    - *(input)* $(\sigma', \omega') = [update](\sigma'', \omega'')$
    - $(\sigma', \omega') \models I(\bar{l}[l'_i/l_i, l'_j/l_j])$
  - *And l_i and/or l_j are committed or there is no edge e that is enabled for any $l_k \in \bar{l}$ such that l_k is committed*

1. $Act(\mathcal{V}, \mathcal{F})$ in Definition 2.3 denotes the set of all assignments over discrete variables ($\mathcal{V}$) and Functions ($\mathcal{F}$). By assignment, we refer to any local computational step that updates the internal state of the automaton, for instance, assigning values to variables (z = x + 1) or invoking a function (deq_val = dequeue()), as opposed to synchronization actions over channels denoted by A. The latter occurs between parallel automata, using the symbols "!" and "?" to achieve synchronization for sending and receiving, respectively.

2. $Update(\mathcal{V}, C, \mathcal{F})$ in Definition 2.3 denotes the set of all possible update actions that can occur over variables, clocks, and functions. Hence, it can be viewed as the Cartesian product $Act(\mathcal{V}, \mathcal{F}) \times Asg(C)$, where $Act(\mathcal{V}, \mathcal{F})$ is the set of actions over discrete variables and functions (as defined in previous point), and $Asg(C)$ is the set of assignments over clocks C. In contrast, *update* represents the concrete grammar that defines the syntactic structure of elements belonging to $Update(\mathcal{V}, C, \mathcal{F})$. From the grammar, it can be observed that *update* consists of *act_seq* and *asg_seq*, which are the concrete grammar rules for $Act(\mathcal{V}, \mathcal{F})$ and $Asg(C)$, respectively.

The Producer Timed Automaton in Figure 2.2 is defined by the locations $L = \{start, produced, sent\}$, with *start* as the initial location L^0. It uses the discrete variables $\mathcal{V} = \{\texttt{value}\}$, the actions $A = \{\underline{produced\_action}, \underline{consumed\_action}\}$, and a single clock $C = \{\textbf{producer\_clk}\}$. Similarly, the Consumer Timed Automaton in the same Figure is defined by $L = \{start, received,$

consumed} with *start* as its initial location. It uses the variable set $\mathcal{V} = \{\texttt{recv\_value}\}$, synchronizes with the Producer via the same actions $A = \{\underline{produced\_action}, \underline{consumed\_action}\}$, and measures time using the clock $C = \{\textbf{consumer\_clk}\}$. `temp_value` is a global variable accessible to both timed automata. The invariants on the locations *start* and *received*, together with the clock guards on edges originating from these locations, correspond to the delay semantics—time may elapse as long as these conditions remain satisfied. In contrast, the actions *produced_action* and *consumed_action* in both automata govern their synchronized behavior, enforcing the synchronization semantics between the two automata.

Transitions are expressed as a tuple ([*Producer location*,*Consumer location*], [**Producer clock**,**Consumer clock**], [`Global Variable`,`Producer Variable`,`Consumer Variable`]), with edges labeled by time constraints, actions, or both. The variables and clocks are initialized to 0. The transitions represent how long the clocks **producer_clk** and **consumer_clk** can stay at each location and the variable values at that time. We present a valid example transition for the automaton system in Figure 2.2, which is a prefix of an infinite trace.

Example Transition:

([*start*,*start*],[0,0],[0,0,0]) $\xrightarrow{5}$ ([*produced*,*start*],[5,5],[1,1,0]) $\xrightarrow{1.1}$ ([*produced*,*start*],[6.1,6.1],[1,1,0]) $\xrightarrow[\underline{produced\_action?}]{\underline{produced\_action!}}$ ([*sent*,*received*],[6.1,0],[1,1,1]) $\xrightarrow{2}$ ([*sent*,*consumed*],[8.1,2],[1,1,1]) $\xrightarrow{1.8}$ ([*sent*, *consumed*],[9.9,1.8],[1,1,1]) $\xrightarrow[\underline{consumed\_action?}]{\underline{consumed\_action!}}$ ([*start*,*start*],[0,1.8],[1,1,1])

1. **Starting at** [*start*, *start*]: The execution begins at the initial locations *start* and *start* of the Producer and Consumer TA. The invariant **producer_clk** ≤ 5 on the Producer's *start* location, together with the clock guard on its outgoing edge, permits time to elapse until exactly 5 time units have passed. At that point, the Producer transitions to *produced* while the Consumer remains at *start*. During this transition, the Producer updates its local variable `value` and writes the generated data to the global variable `temp_value`.

2. **Staying at** [*producted*, *start*]: Once the Producer reaches *produced* at time 5 while the Consumer remains at *start*, both clocks hold the value 5. At this point, neither automaton is forced to transition immediately. An delay of 1.1 time units is permitted by the invariants and guard conditions. This leads to the state $([produced, start], [6.1, 6.1])$ with no variable changes, since no transition has fired.

3. **Transition to** [*sent*, *received*]: When the Producer is ready to send the produced `value`, it performs the output action *produced_action*! from location *produced*. The Consumer simultaneously performs the matching input action *produced_action*? from *start*. By the synchronization rule of ETA operational semantics, both automata must take

these transitions simultaneously. The Consumer resets its clock **consumer_clk** to begin measuring processing time, and it updates its local variable `recv_value` using the global variable `temp_value`. The Producer moves to *sent*, the Consumer to *received*, while the `temp_value` and `value` variables remain unchanged.

4. **Transition to** [*sent*, *consumed*]: After the synchronization between the Producer and Consumer TA, the Producer is at *sent* while the Consumer is now at *received*. The invariant **consumer_clk** ≤ 2 on the Consumer's *received*, together with the clock guard on its outgoing edge, permits time to elapse until exactly 2 time units have passed. During this delay, **producer_clk** continues from 6.1 to 8.1, while **consumer_clk** increases from 0 to 2. No variables change during this waiting period.

5. **Staying at** [*sent*, *consumed*]: A delay of 1.8 time units occurs, bringing the system to the state $([sent, consumed], [9.9, 1.8])$. Variables remain unchanged, reflecting that the Consumer has processed but not yet acknowledged consumption to the Producer.

6. **Transition to** [*start*, *start*]: Finally, the Consumer is ready to signal that it has finished processing. The Consumer performs *consumed_action*! from *consumed*, and the Producer performs *consumed_action*? from *sent*. By the ETA sync rule, these transitions occur simultaneously. The Producer resets **producer_clk** and returns to *start* to begin producing the next value, while the Consumer also moves to *start*, retaining its current **consumer_clk** value since its clock is not reset on this transition. All variable values remain consistent with the previous state, finalizing the cycle.

2.2.2 Verification Results on Properties using UPPAAL

Figure 2.4 shows the UPPAAL representation of the timed automata described in Example 2.1. The automaton is named *traffic_sys* to allow its components, such as locations, clocks, and variables, to be accessed during verification. Figure 2.6 presents the verification results of the corresponding queries.

Figure 2.5 is the network of timed automata representation of Example 2.2 in UPPAAL. The two automata are named *producer_sys* and *consumer_sys* to allow their components, such as locations, clocks, and variables, to be accessed during verification. Figure 2.7 presents the verification results of the queries.

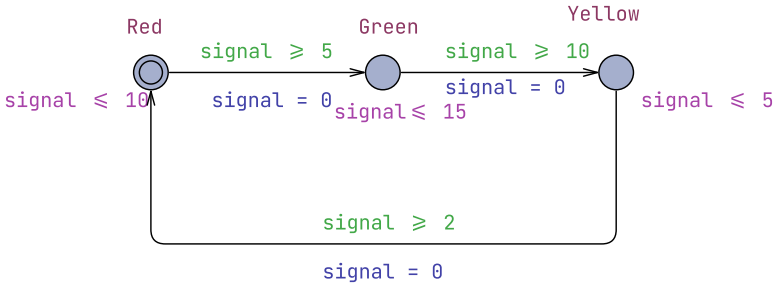
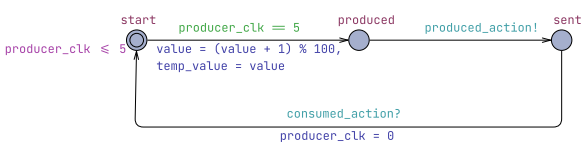
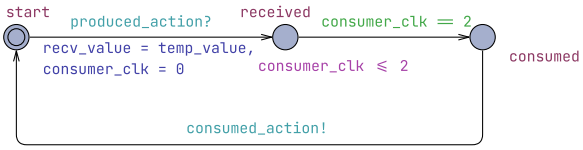


Figure 2.4: UPPAAL Timed Automata for Example shown in Figure 2.1.



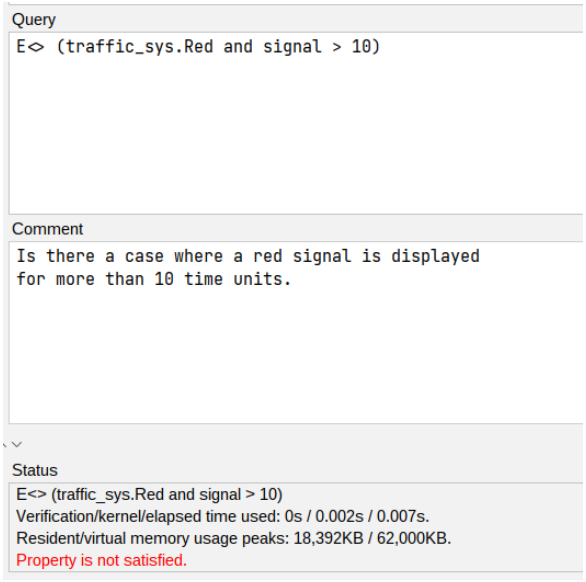
(a) Producer TA (produced_sys).



(b) Consumer TA (consumer_sys).

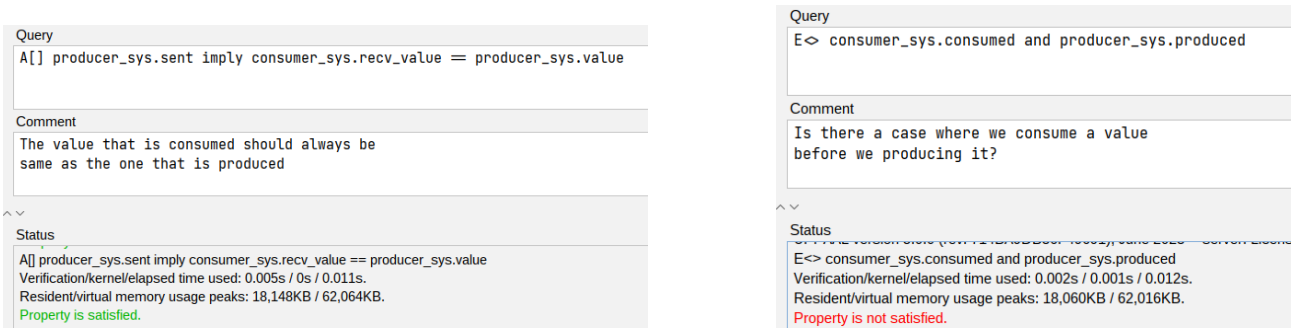
Figure 2.5: UPPAAL Network of Timed Automata for Example shown in Figure 2.2.

(a) Query satisfied



(b) Query not satisfied

Figure 2.6: Model verification result of UPPAAL Example shown in Figure 2.4.



(a) Query satisfied

(b) Query not satisfied

Figure 2.7: Model verification result of UPPAAL Example shown in Figure 2.5.

2.3 Robot Operating System (ROS)

ROS [36] is an open-source framework widely adopted by industry and academia for the rapid prototyping and development of distributed robotic systems. Although commonly referred to as an operating system, ROS is in fact composed of packages (libraries) and tools that run on top of an existing OS. At the heart of ROS lies a publish-subscribe (pub-sub) communication model (Section 1.1), which is the primary mechanism for data exchange between distributed systems. This model, which we aim to formally model and analyze, enables asynchronous and event-driven communication, promoting modularity and decoupling within complex robotic systems. While the pub-sub paradigm is central to ROS communication, the framework also supports additional interaction patterns, such as request-response (services) [40] and goal-oriented (action) [38] communication models. These alternatives enable synchronous exchanges and support long-running operations that are often necessary in robotic applications. To further enhance communication, ROS includes Quality of Service (QoS) policies [39], which allow developers to fine-tune aspects such as reliability, latency, and resource usage. QoS profiles can be independently configured for publishers, subscribers, services, and actions, offering flexibility across various communication entities. However, mismatches in QoS configurations may lead to communication failure, emphasizing the need for careful design. Despite its flexibility and widespread adoption, ROS lacks built-in timing guarantees, as it is not designed as a real-time system. This limitation poses challenges in time-critical robotic domains such as autonomous vehicles, industrial automation, and robotic arms, where timing correctness and predictability are essential. This motivates the need for formal modeling and verification of ROS-based systems, particularly their pub-sub communication behavior, to ensure correctness under timing constraints.

2.3.1 Communication Mechanisms in Packages

A package in ROS is a fundamental software engineering construct that encapsulates various files, including source code, configuration files, dependencies, and other resources grouped together to provide specific functionality. This modular packaging approach enables independent development, allowing different teams to work on distinct applications of a system in isolation. In ROS, a package is like a container that holds everything needed for a specific part of a robot's functionality. Inside this package, there can be multiple ***components***, each one responsible for a smaller, specific task. For example, one component handles reading sensor data and processes that data, while another publishes commands to the robot's wheels. This modular design makes it easier to build, understand, and reuse parts of the system. Despite being developed separately, they can still seamlessly communicate with each other when deployed, thanks to the common pub-sub communication model, where components within different packages exchange data using agreed-upon topic[1] names. In this work, we focus exclusively on the pub-sub communication model. Within this model, any node[2] that sends messages on a topic is referred to as a publisher, while a node that receives and processes these messages through an event-handler is called a subscriber. A single node can act as both a publisher and a subscriber, depending on the application logic. In practice, publishers and subscribers are instantiated as objects using the node, which provides the necessary APIs for communication. Messages are published by invoking the publish method on the publisher object. This invocation can occur either at a user-defined frequency, as demonstrated by the value within the sleep function in the Publisher1 and Publisher2 functions in the pseudo-code shown in Figure 1.2 in Chapter 1, or within an event-handler, such as in response to a message received by a subscriber (as illustrated in ASubscriber from the same pseudo-code example). Messages published to a topic are first enqueued in a queue before being dequeued and processed by their corresponding event-handlers. The number of queues in the system is uniquely determined by the tuple (subscriber, topic, MessageType). Event-handlers typically process messages from the queue and may update internal state variables, which can subsequently influence the behavior of other event-handlers. If multiple messages arrive simultaneously, the execution order of their respective event-handlers is determined by the priority of the subscriber that registered its subscription first. This introduces additional complexity, as subscriptions can be conditionally created within control structures such as `if`-statements, making the registration order dependent on runtime behavior. Furthermore, publishers and subscribers can be dynamically added or removed even after

[1] A topic is a named communication channel used for message exchange in ROS.

[2] A node in ROS is a logical unit that performs a specific task and communicates with other nodes. In code, it is typically implemented as a class.

the system has been initialized, requiring the execution model to handle dynamic changes effectively. To manage the scheduling and invocation of these event-handlers consistently and efficiently, ROS introduces the concept of *executors*.

2.3.1.1 Executors

In ROS 2, *executors* [34] are responsible for managing the scheduling and execution of functions associated with subscriptions, timers, services, action servers, and other event-driven components. Executors make use of one or more threads from the underlying operating system to invoke these functions. The order in which functions are dispatched depends not only on their arrival times but also on the order in which their corresponding components were created (e.g., via `create_subscription`, `create_timer`, etc.). ROS provides different executor types to support a range of concurrency and performance needs:

1. **Single-Threaded Executor:** This executor runs all functions (of timers, subscribers, etc) sequentially using a single thread. It is the default in many cases and is simple to reason about. However, it can become a bottleneck if one function blocks for too long, potentially delaying other important tasks such as message processing or publishing.

2. **Multi-Threaded Executor:** This executor allows functions to be handled concurrently using multiple threads. It is useful when multiple subscriptions, timers, or services may block or perform long-running computations. Developers can configure the number of threads and use *groups* to control whether specific functions can run in parallel or must remain mutually exclusive.

3. **Static Single-Threaded Executor:** Similar to the regular Single-Threaded Executor, but with one key difference, it discovers all functions only once during initialization. It does not poll for changes at runtime. As a result, it does not respond to dynamically added or removed components (e.g., new subscriptions/publishers created). This makes it unsuitable for highly dynamic systems.

In summary, the executor model plays a central role in determining how a system responds to incoming messages and other events. Selecting an appropriate executor type and configuring it correctly is therefore crucial for achieving timely and predictable behavior, especially in systems with multiple interacting components. In this work, we focus exclusively on the *Static Single-Threaded Executor* for system modeling. It is important to clarify that static analysis in our framework is used solely to extract structural information from the ROS package, such as subscriptions, publishers, callback mappings, and loop bounds, by analyzing the LLVM-IR.

This information is then used to construct the corresponding extended timed-automata model, which will vary depending on the executor model chosen. After this model is constructed, all timing verification is performed using UPPAAL, which is a model-checking tool and not a static-analysis tool. Thus, static analysis serves only as a preprocessing step for model construction and is independent of the choice of executor model. Our decision to use the *Static Single-Threaded Executor* is based on empirical observations from real-world ROS packages relevant to our problem statement: callbacks typically execute sequentially, and systems rarely introduce new publishers or subscribers after initialization. This executor model, therefore, reflects common practice while enabling a sound and tractable timed-automata representation. Supporting more complex executor models, such as multi-threaded execution or dynamically evolving communication structures introduces additional challenges, including reasoning about concurrent callback execution, and dynamically changing interaction patterns. Addressing these challenges requires extending the modeling layer, particularly the interaction between parallel automata and their synchronization semantics. We outline how our framework can be extended to support richer executor models through limited user annotations in Chapter 6. Furthermore, in Section 3.3.1, we analyze the ordering semantics of event-handler execution

2.3.2 Inter and Intra Package Communication

To illustrate in detail how packages communicate using topics, we use Figure 2.8 as a running example. This is a fabricated example created for illustrative purposes, but it closely resembles how packages typically interact in real-world ROS systems. The Camera Package plays a crucial role in supplying visual data to the system. The RawCamera component of the Camera Packages includes multiple publishers: Front Camera Publisher1, Front Camera Publisher2, and Back Camera Publisher, which use the underlying hardware to capture image data and publish it to the /f_camera, /f_camera, and /cam_data topics, respectively. This raw visual data is further processed by the ProcessedCamera component within the same package. For instance, the Front Camera Data component subscribes to the /f_camera topic, processes the incoming stream, and then republishes the refined data to the /cam_data topic. Both front and back camera data streams on /cam_data are consumed by the Camera Data Listener, which aggregates and processes them to produce a unified output published on the /cam_info topic. Although all of this communication occurs internally within the Camera Package, the data published on /cam_info is consumed externally. Specifically, the Detector Component within the Odometry Package, which contains the Obstacle Detector, subscribes to the /cam_info topic to access comprehensive camera data of the system. This information is critical for performing collision detection and navigation-related decision-making.

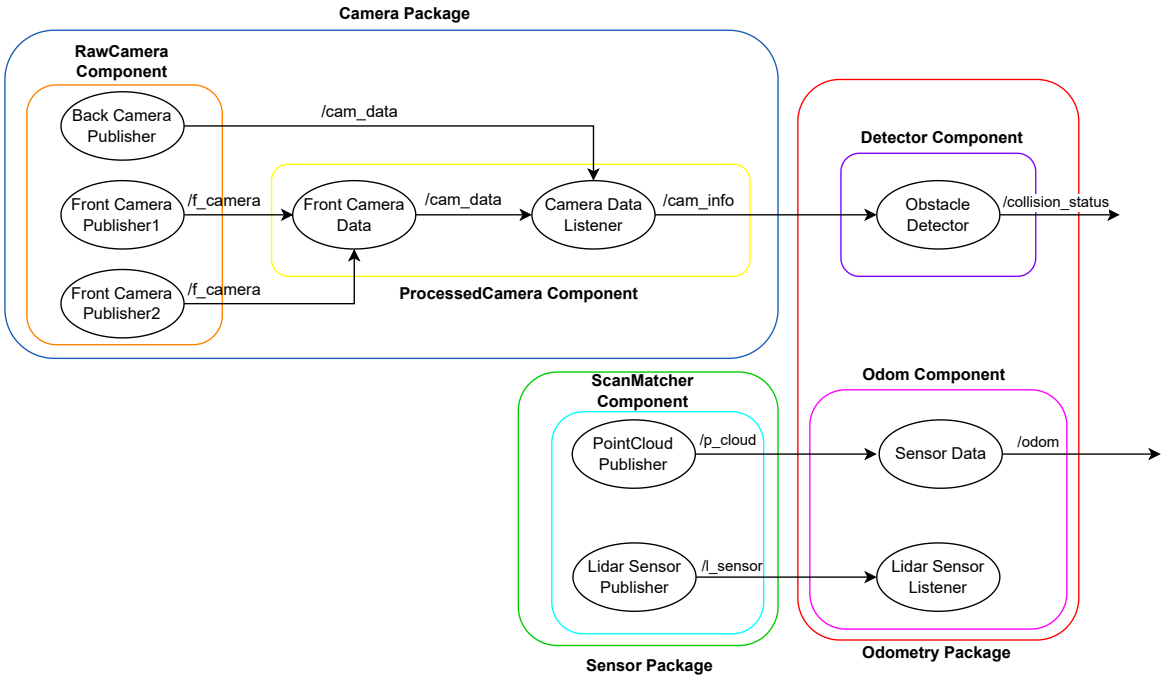


Figure 2.8: Communication within and across different packages in ROS application.

Similarly, the ScanMatcher Component within the Sensor Package is responsible for collecting spatial and environmental data from various hardware sources. It contains publishers such as the Lidar Sensor Publisher and the PointCloud Publisher, which publish data to the /l_sensor and /p_cloud topics, respectively. These topics stream raw sensor readings, which are essential for spatial awareness and environmental mapping. This data is then consumed by the Odom Component within the Odometry Package, specifically by the Sensor Data, which subscribes to the /p_cloud topic. Additionally, the Lidar Sensor Listener within the same component listens to the /l_sensor topic using its event handler. Upon receiving new LIDAR data, it updates certain internal variables that may influence the execution of the Sensor Data event-handler. These internal variables, combined with the incoming point cloud data from /p_cloud, are used by Sensor Data to compute the system's motion and positional state. The result of this processing is published to the /odom topic, which represents the robot's real-time odometry data. The /odom topic represents the robot's real-time odometry data, which can be used by other packages for localization, path planning, or navigation.

2.3.3 Popular ROS packages

Several open-source packages in the ROS community are widely used for developing robotic systems, a few of them are described below:

1. Autoware[4] uses ROS for deploying their open-source autonomous driving stack across a wide range of vehicles. It integrates multiple packages for tasks like object detection, path planning, and vehicle control.

2. Nav2[31] is used to provide capabilities such as path planning, collision avoidance, and recovery behaviors.

3. MoveIt[41] is used for motion planning and manipulation of robotic arms.

4. lidar-slam[45], an SLAM (Simultaneous Localization and Mapping) algorithm package using lidar-sensors, is used for autonomous navigation in unknown environments.

These packages incorporate sophisticated algorithms crucial for autonomous systems. Thus, it is crucial to rigorously test these packages to ensure that they meet the stringent timing constraints required for their applications.

Chapter 3

Approach

In this chapter, we formally define the problem statement and present the foundational definitions that serve as inputs to the core algorithms of our proposed methodology. This methodology emphasizes the systematic construction of a Timed Automata model, followed by formal verification, feedback generation based on analysis results, corrective modifications to the model, and iterative re-verification to ensure compliance with specified timing constraints. Figure 3.1 offers a comprehensive overview of the entire approach. The **Algorithm for resolving loop conditions (step-4)** in the red-highlighted region and the **Model Construction** in the green-highlighted region of Figure 3.1 represent the specific focus of this chapter.

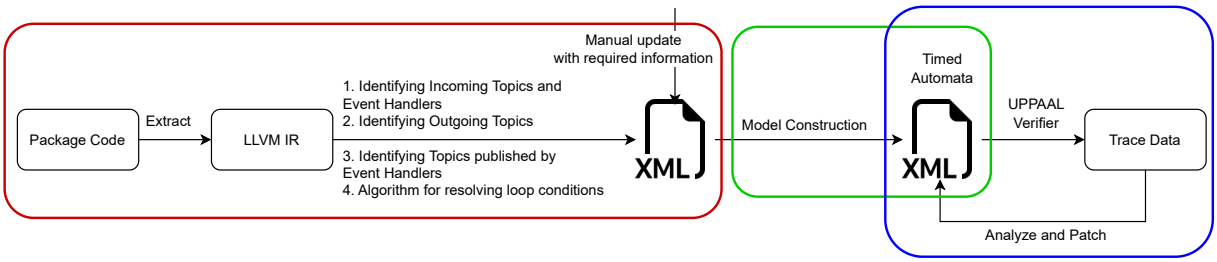


Figure 3.1: Overview of our framework's approach.

The model construction step involves generating a Timed Automata representation of the system, encoded in XML format. This representation captures key elements such as locations, transitions, clocks, variables, and synchronization mechanisms between components, making it well-suited for automated analysis and verification.

We provide a detailed walkthrough of the algorithms used in this phase, explaining the underlying logic. To illustrate the outcome of the model construction process, we use a representative ROS package example, as shown in Figure 3.2. The remaining components of the approach are discussed in subsequent chapters.

3.1 Problem Statement

Definition 3.1. *Given source code of the package as shown in Figure 3.2, which contains*

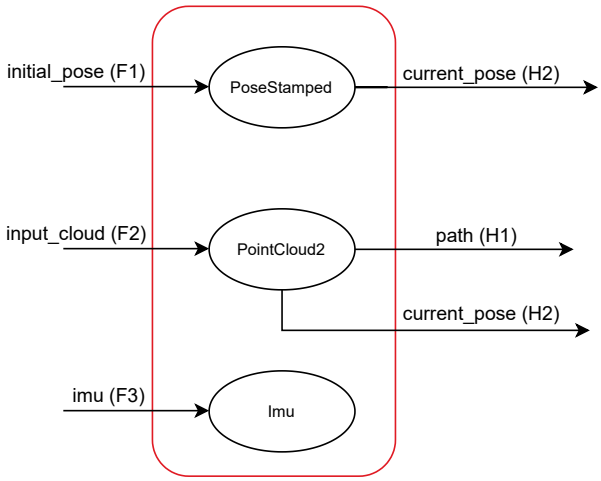


Figure 3.2: Modified Lidarslam ROS Package [45]

- *Set of Incoming Topics, $F = \{F_1, F_2, \cdots, F_n\}$, where each F_i represents a topic on which the corresponding subscriber listens for incoming messages. We discuss how to compute this set in Section 4.1,*
  - *Timing Assumptions on Incoming Topics, $T : F \to \mathbb{R}^+$, where $T = \{T_1, T_2, \cdots, T_n\}$ and $T(F_i) = T_i$ denotes the maximum time gap between two consecutive incoming messages on the incoming topic F_i.*
  - *Set of Outgoing Topics, $H = \{H_1, H_2, \cdots, H_m\}$, where each H_j denotes the topic on which the event-handler publishes messages. We discuss how to compute this set in Section 4.2,*
    - *Timing Constraint Evaluation on Outgoing Topics, $\tau : H \to \mathbb{R}^+ = \{\tau_1, \tau_2, \cdots, \tau_m\}$, where $\tau(H_i) = \tau_i$ denotes the maximum allowable time within which at least one message is expected to be published on the outgoing topic H_i,*

– *Event Handler of Incoming Topics, $G : F \rightarrow \mathcal{G}$, where $\mathcal{G}$ represents the set of all possible event-handlers represented by control flow graphs. We discuss how to identify this set in the Section 4.1. $G = \{G_1, G_2, \cdots, G_n\}$, where G_i is control flow graph of the incoming topic F_i represented using a directed-graph (B_i, E_i), where*

  * *B_i represents a set of basic blocks,*
  * *$E_i \subseteq B_i \times B_i$ represents directed edges between the basic blocks,*
  * *Block Timing Map, $\lambda_T : B_i \rightarrow \mathbb{R}^+$, $\lambda_T(b) = t, \forall b \in B_i$ where t is the worst-case execution time (WCET) of the basic-block b in some time units that needs to be computed using an external WCET tool. We discuss the WCET values that we use in our work in Section 4.5,*
  * *Block Publish Map, $\lambda_H : B_i \rightarrow 2^H$, $\lambda_H(b) = H_b, \forall b \in B_i$ where H_b represents the set of outgoing topics associated with block b; ϕ if no outgoing topics exists. We discuss how to compute this set in Section 4.3,*

Given the source code of a package, along with user-defined configuration and timing specifications data, we construct a Timed Automata model to verify whether the system adheres to its timing guarantees. The model is analyzed using the UPPAAL verifier to identify any timing violations. If violations are detected, we analyze their root causes and provide actionable feedback. The model is then iteratively patched and re-verified. This cycle of verification, feedback, and patching continues until all timing constraints are satisfied, or it is determined that the constraints are too strict and need to be relaxed.

Example ROS Package

```
1  initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>("initial_pose",
       rclcpp::QoS(10), initial_pose_event_handler);

3  input_cloud_sub_ = create_subscription<<sensor_msgs::msg::PointCloud2>("input_cloud", rclcpp::
       QoS(10), input_cloud_event_handler);

5  imu_sub_ = create_subscription<sensor_msgs::msg::Imu>("imu", rclcpp::QoS(10),
       imu_event_handler);

7  pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose", rclcpp::QoS(10))
       ;

9  path_pub_ = create_publisher<nav_msgs::msg::Path>("path", rclcpp::QoS(10));

11 geometry_msgs::msg::PoseStamped current_pose_stamped_;
12 nav_msgs::msg::Path path_msg;

14 auto initial_pose_event_handler =
15 [this](const typename geometry_msgs::msg::PoseStamped::SharedPtr msg) -> void
```

```
16 {
17     // Block1 (B1)
18     ...
19     ...
20     current_pose_stamped_ = *msg;
21     pose_pub_->publish(current_pose_stamped_);
22 };

24 auto input_cloud_event_handler =
25 [this](const typename sensor_msgs::msg::PointCloud2::SharedPtr msg) -> void
26 {
27     // Block1 (B1)
28     ...
29     if(long_path) {
30     // Block2 (B2)
31         ...
32         PointCloud2::converter(msg, path_msg);
33         path_pub_->publish(path_msg);
34     } else {
35     // Block3 (B3)
36         ...
37         pose_pub_->publish(current_pose_stamped_);
38     }
39     // Block4 (B4)
40     ...
41 };

43 auto imu_event_handler =
44 [this](const typename sensor_msgs::msg::Imu::SharedPtr msg) -> void
45 {
46     // Block1 (B1)
47     ...
48 };
```

Listing 3.1: C++ Code of Modified Lidarslam ROS Package [45]

Figure 3.2 presents a pictorial representation of a Modified Lidarslam ROS package example, corresponding to the illustrative code shown in Listing 3.1. To make the Model construction process easier to understand, we removed a few publishers and simplified the input_cloud_event_handler code. We formally define this constructed example as follows.

- Set of Incoming Topics, $F = \{\text{initial\_pose, input\_cloud, imu}\}$
  - Timing Assumptions on Incoming Topics, $T = \{2, 2, 5\}$ denotes the maximum time units gap between enqueueing two consecutive incoming messages of topics initial_pose, input_cloud, imu respectively. The time units are assumed to be in abstract time units. These values are provided by the user,
  - Set of Outgoing Topics, $H = \{\text{path, current\_pose}\}$

* Timing Constraint Evaluation on Outgoing Topics, $\tau = \{4, 8\}$ denotes the maximum allowable time within which a message is expected to be published to the path and current_pose outgoing topics, respectively. The time units are assumed to be in abstract time units. In this case, the message to current_pose can be published by two different event-handlers, therefore, the requirement is satisfied as long as at least one of them publishes within 8 time units. These values are provided by the user,

– Event Handlers of Incoming Topics, $G = \{G_1, G_2, G_3\}$ = {initial_pose_event_handler, input_cloud_event_handler, imu_event_handler} where,

* $G_1 = \{(\{B_1\}, \phi)\}$, Block Timing Map, $\lambda_T = \{4\}$ denotes the worst-case execution time of the block B_1. In this case, we assume it's provided by an external WCET tool. Block Publish Map, $\lambda_H = \{\{\text{current\_pose}\}\}$ denotes the set of outgoing topic names to which a given basic block within the event-handler publishes messages during its execution. Since a block may publish to multiple outgoing topics, each block is associated with its own set of outgoing topics. In this case, the handler pose_pub_ is responsible for publishing messages within the block and is specifically created to publish on the current_pose outgoing topic, as defined in Line 21 of Listing 3.1. Figure 3.3 shows the CFG of the initial_pose_event_handler. As there are no control-flow statements in this handler, it consists of a single basic block.

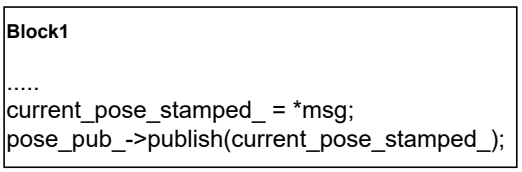


Figure 3.3: CFG representation of Lines 14–22 from the initial_pose_event_handler function in Listing 3.1.

* $G_2 = \{(\{B_1, B_2, B_3, B_4\}, \{(B_1, B_2), (B_1, B_3), (B_2, B_4), (B_3, B_4)\})\}$,
Block Timing Map, $\lambda_T = \{1, 10, 3, 1\}$,
Block Publish Map, λ_H = {ϕ,{path}, {current_pose},ϕ}, Figure 3.4 shows the CFG of the input_cloud_event_handler function. As there are control-flow statements (if-else branch), we have more than one basic block in the CFG.

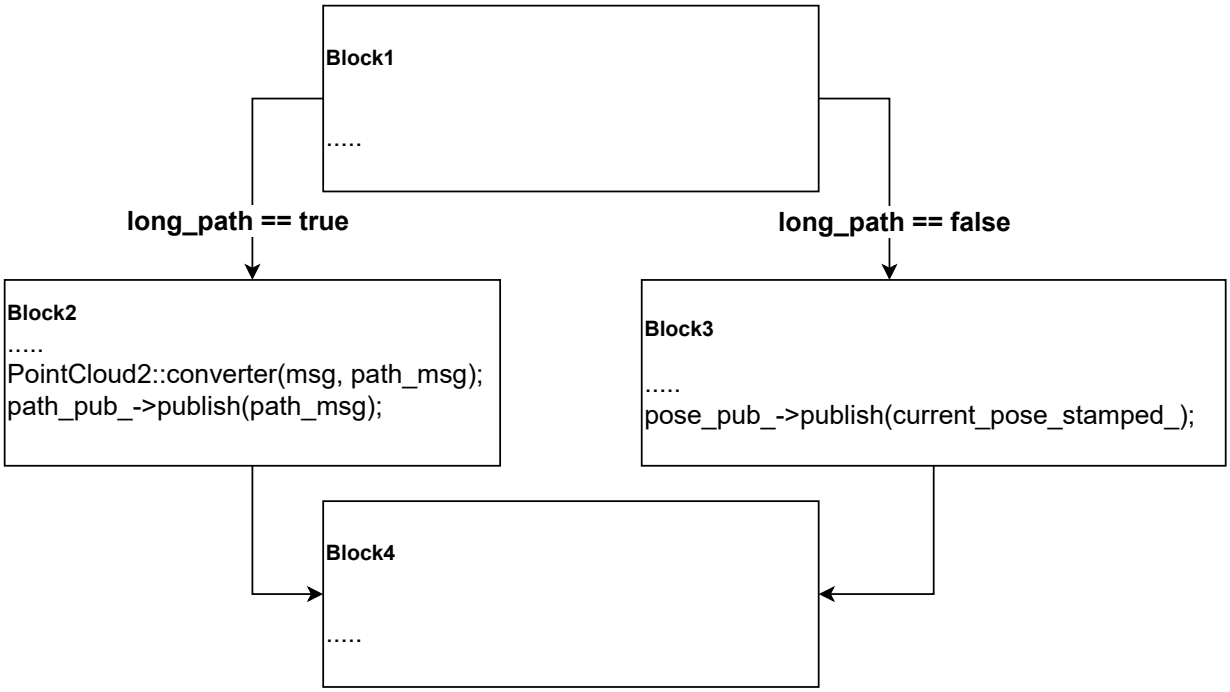


Figure 3.4: CFG representation of Lines 24-41 from the input_cloud_event_handler function in Listing 3.1.

* $G_3 = \{(\{B_1\}, \phi)\}$, Block Timing Map, $\lambda_T = \{1\}$, Block Publish Map, $\lambda_H = \{\phi\}$, Figure 3.5 shows the CFG of the imu_event_handler function. As there are no control-flow statements, we only have a single basic block in the CFG.

Figure 3.5: CFG representation of Lines 43-48 from the imu_event_handler function in Listing 3.1.

We use the above package data as a running example to demonstrate the results of the model construction algorithms described in the section 3.3.

3.2 Algorithm to resolve loop conditions

The purpose of this section is to describe an algorithm that identifies and resolves loop conditions within event handlers to assign finite bounds, ensuring that the loops in the modeled

system are bounded and thereby enabling tractable and accurate model construction for verification. If the loop conditions are not resolved, their bounds remain undefined, which may result in potentially infinite loops and render the generated model unsuitable for verification. This makes the proposed algorithm a crucial component of the overall framework. The algorithm maps each loop's conditional block to a concrete condition value, which is derived from a configuration parameter, a message field, or an integer constant. We begin with the conditional block identified as part of a loop using the dominator tree and back edge analysis. From this block, we isolate the condition value that controls the loop. Using use-def chains, we perform a backward traversal starting from this condition to trace its reaching definitions. This analysis helps determine whether the loop condition originates from a message field or from a user-defined configuration parameter. Specifically, the algorithm targets condition expressions of the form "$index \bowtie value$", where *index* refers to the loop iterator variable, $\bowtie \in \{\leq, <, >, \neq, ==, \geq\}$ is a conditional operator, and *value* is an integer constant or a field from the message or a configuration parameter. If a condition expression is more complex, such as "$index \bowtie (value1 \times 3) \wedge value2$", the algorithm deems it unresolvable as the match pattern fails and assigns ϕ as the resolved condition for that block.

Our goal is to assign an integer bound to loops to ensure they remain finite. This is important because we import the Control Flow Graph (CFG) of the event-handler into the model construction step, which will be discussed in the following section. To ensure the loops are always bounded, we attempt to resolve their bounds either to a constant integer or to a message field, or to a configuration parameter. Once a loop bound is resolved to a specific message field or configuration parameter, we can retrieve its corresponding integer value from an external file and substitute it during the model construction phase. This approach helps maintain tractability during verification by ensuring all loops are finitely bounded.

We begin by presenting a set of resolvable and unresolvable conditions using C++ code snippets from real-world ROS packages, along with their corresponding Resolved Conditions, which represent the outputs of our algorithm. Resolved Conditions are highlighted in green, while Unresolvable Conditions are highlighted in red, with accompanying explanations for why they cannot be resolved. This is followed by an illustrative example demonstrating the analysis flow using a simplified three-address SSA code format. We use this format to make the explanation more accessible and to help the reader understand the reasoning process and challenges involved in the analysis. However, in practice, our analysis is performed directly on LLVM-IR, which offers a similar level of complexity and precision. Finally, we present the algorithm.

A few examples of conditions that we are trying to resolve are as follows:

```
void ScanMatcherComponent::updateMap(...)
{
  .....
  for (int i = 0; i < num_targeted_cloud_ - 1; i++) {
        .....
  }
  .....
}
```

Listing 3.2: Resolvable Condition from Lidarslam Package [45]

```
index < num_targeted_cloud_ - 1
```

Listing 3.3: Resolved Condition of Listing 3.2.

- Listing 3.2 shows a resolvable loop condition (highlighted in green) that is bounded by the configuration parameter `num_targeted_cloud_` from the Lidarslam [45] ROS package at the source-code level. The original condition in the loop, `i < num_targeted_cloud_ - 1`, is resolved as shown in Listing 3.3. Since this is a configuration parameter, we leave it unchanged. While it may appear trivial when illustrated through source code, the actual analysis is performed at a low-level intermediate representation, making the resolution process significantly complex and non-trivial. During the model construction step, the value of this configuration parameter is read from the external file and substituted accordingly.

```
void scan2occ_grid::Plugin::on_laser_scan(const sensor_msgs::msg::LaserScan::SharedPtr
    SensorMsg)
{
  .....
  for (int i = 0; i < SensorMsg->ranges.size(); i++) {
        ....
  }
  .....
}
```

Listing 3.4: Resolvable Condition from Aerostack2 Package [14]

```
index < SensorMsg:9
```

Listing 3.5: Resolved Condition of Listing 3.4.

- Listing 3.4 shows a resolvable loop condition (highlighted in green) that is bounded by the `range` field from an incoming message in the Aerostack2 [14] ROS package. The original condition, `i < SensorMsg->ranges.size()`, is resolved to the simplified form `index < SensorMsg:9`, as shown in Listing 3.5. In our analysis, we focus solely on field access

within the message and disregard any method calls applied to those fields. If the message access involves nested structures or subfields, we resolve the loop condition using all the fields that are part of the access chain, as it typically represents the value that bounds the loop. In this case, `SensorMsg->ranges` is resolved to a simplified representation, where `ranges` corresponds to the 9th field in the message type definition. As a result, `SensorMsg:9` denotes that the loop is bounded by the value of the 9th field in the incoming message. During model construction, this value is retrieved from the message bounds file and substituted accordingly.

```
void voxelCallback(const nav2_msgs::msg::VoxelGrid::ConstSharedPtr grid)
{
  .....
  const uint32_t x_size = grid->size_x;
  const uint32_t y_size = grid->size_y;
  const uint32_t z_size = grid->size_z;
  ....
  for (uint32_t y_grid = 0; y_grid < y_size; ++y_grid) {
    for (uint32_t x_grid = 0; x_grid < x_size; ++x_grid) {
      for (uint32_t z_grid = 0; z_grid < z_size; ++z_grid) {
                .....
      }
    }
  }
}
```

Listing 3.6: Resolvable Condition from Navigation2 Package [53]

```
index  < grid:6
index1 < grid:5
index2 < grid:7
```

Listing 3.7: Resolved Condition of Listing 3.6.

- Listing 3.6 presents three resolvable loop conditions (highlighted in green) in a triple-nested loop, where the bounds are determined by the fields `size_y`, `size_x`, and `size_z` from an incoming message. This code is taken from the Navigation2 [53] ROS package. The resolved conditions are shown in Listing 3.7. The original conditions, `y_grid < y_size`, `x_grid < x_size`, and `z_grid < z_size`, are mapped to `index` $<$ `grid:6`, `index1 < grid:5`, and `index2 < grid:7` respectively, indicating that the loop bounds correspond to the values of 6th, 5th, and 7th fields of the incoming message. These values are read from the message bounds file during model construction and substituted accordingly.

```
std::vector<std::vector<int>> scan2occ_grid::Plugin::bresenham_line(
```

```
  int x1, int y1, int x2, int y2)
{
        // Calculation based on x1, y1, x2, y2
        .....
}

void scan2occ_grid::Plugin::on_laser_scan(const sensor_msgs::msg::LaserScan::SharedPtr
    LaserMsg)
{
  .....
  for (int i = 0; i < LaserMsg->ranges.size(); i++) {
        .....
    std::vector<std::vector<int>> middle_cells = bresenham_line(
      drone_cell[0], drone_cell[1], cell[0], cell[1]);
    for (const std::vector<int> &p :  middle_cells) {
                .....
    }
  }
}
```

Listing 3.8: Unresolvable Condition from Aerostack2 Package [14]

- Listing 3.8 shows a nested loop where one loop contains a resolvable condition (highlighted in green), bounded by the `range` field from an incoming message `LaserMsg`, and the other contains an unresolvable condition (highlighted in red), which is computed through complex operations based on the message data. In this case, the red-highlighted condition cannot be resolved statically. Manual intervention is required to provide a bounded integer value for this loop to keep the verification process tractable. The details of the manual intervention process will be discussed following the explanation of the algorithm. This code is taken from the Aerostack2 [14] ROS package. The resolved condition for the green-highlighted loop is shown in the Listing 3.5, while the unresolvable condition is represented by a placeholder value, to be substituted manually.

```
void pointCloud2Helper(uint32_t num_channels, ....)
{
  .....
  for (uint32_t i = 0; i < num_channels; ++i) {
        .....
  }
  .....
}

void voxelCallback(const nav2_msgs::msg::VoxelGrid::ConstSharedPtr grid)
{

  const uint32_t * data = &grid->data.front();
  const uint32_t x_size = grid->size_x;
```

```
  const uint32_t y_size = grid->size_y;
  const uint32_t z_size = grid->size_z;
  uint32_t num_marked = 0;
  uint32_t num_unknown = 0;
  for (uint32_t y_grid = 0; y_grid < y_size; ++y_grid) {
    for (uint32_t x_grid = 0; x_grid < x_size; ++x_grid) {
      for (uint32_t z_grid = 0; z_grid < z_size; ++z_grid) {
        nav2_voxel_grid::VoxelStatus status = nav2_voxel_grid::VoxelGrid::getVoxel(x_grid
            , y_grid, z_grid, x_size, y_size, z_size, data);
        if (status == nav2_voxel_grid::UNKNOWN) {
            .....
            ++num_unknown;
        } else if (status == nav2_voxel_grid::MARKED) {
            .....
            ++num_marked;
        }
      }
    }
  }
  .....
  pointCloud2Helper(cloud, num_marked, pcl_header, g_marked);
  pointCloud2Helper(cloud, num_unknown, pcl_header, g_unknown);
  .....
}
```

Listing 3.9: Unesolvable Condition from Navigation2 Package [53]

- Listing 3.9 presents a triple-nested loop with three resolvable conditions (highlighted in green), determined by the fields `size_y`, `size_x`, and `size_z` from an incoming message. Additionally, it includes an unresolvable condition (highlighted in red), whose bound is derived from computations based on the message data and cannot be statically determined. As in the previous case, a bounded integer value must be manually annotated for the red-highlighted loop to ensure tractability during verification. This code is taken from the Navigation2 [53] ROS package. The resolved conditions for the green-highlighted loops are shown in the Listing 3.6, while the unresolvable condition will be updated manually.

The resolved conditions shown above as part of various ROS package Listings are stored in the map λ_C at block-level per CFG G_i of the incoming topic, it's defined as follows:

- Block Condition Map, $\lambda_C : CB \rightarrow$ Resolved Condition, maps each loop conditional block CB in B_i within the CFG G_i to its corresponding Resolved Condition using Algorithm 1, or to a placeholder value if it cannot be resolved, which needs to be manually updated. The Resolved Condition as seen from Listings 3.3, 3.5, and 3.7 is of the form "$index \bowtie value$", where $index$ is an iterator variable, $\bowtie \in \{\leq, <, >, \neq, ==, \geq\}$, and $value$ is an constant or either related to a field in one of the messages or to one of the configuration parameters.

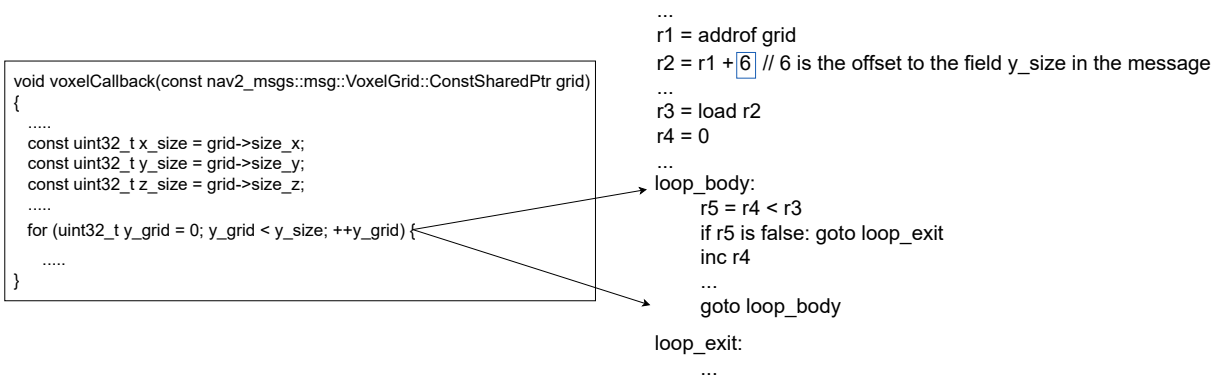


Figure 3.6: Tracing the origin of the loop condition of the outermost loop in Listing 3.6.

Figure 3.6 illustrates the dependency tracing involved in resolving the loop condition of the outer-most loop from Listing 3.6 of the Navigation2 [53] ROS package. For reference, the source code is shown on the left, while the three-address SSA code format on the right highlights only those instructions relevant to our analysis.
The loop condition block, identified via dominator tree analysis, contains the condition "y_grid $<$ y_size", which appears within the loop_body. In the corresponding SSA form, this condition is encoded as a comparison instruction: "r5 = r4 $<$ r3". Here, r4 holds the loop index "y_grid", and r3 holds the loop bound "y_size". To resolve the origin of this condition, our analysis begins from r3 and traces its definitions using use-def chains to determine how this value was constructed. We find that "r3 = load r2", indicating that the value was loaded from a memory location pointed to by r2. Tracing further back, "r2 = r1 + 6" shows that the value is located at an offset of 6 from the base address stored in r1. Finally, "r1 = addrof grid" reveals that the base address corresponds to the incoming message grid. By following this chain of definitions, we determine that the loop is bounded by the value stored in the sixth field of the message. The offset 6 is critical for identifying which specific message field is involved. This resolution confirms that the loop condition is data-dependent on the message content and enables precise modeling of such dependencies during timing analysis. Accordingly, the Block Condition Map λ_C corresponding to `voxelCallback` stores the mapping {loop_block: `index < grid:6`}, indicating that the loop bound for this block is determined by the 6^{th} field of the message. While we use three-address SSA code for illustration to improve readability, our actual implementation performs this analysis directly over LLVM IR, which offers the same complexity and resolution capabilities.

Although use-def chain analysis itself is a precise and deterministic technique for tracking

Algorithm 1 Algorithm to resolve loop conditions

Input: **UseDef**$(r) \rightarrow i$ - A map from a register r to the instruction i that defines it,
$\mathbf{i_{cond}}$ - The comparison instruction (e.g., $r_{cmp} = r_{idx} \bowtie r_{bound}$, where $\bowtie \in \{\leq, <, >, \neq, ==, \geq\}$),
$\mathbf{B_{loop}}$ - The block representing the loop,
MB — Message List of Incoming Topics,
CD — List of configuration parameters
Output: A map $\boldsymbol{\lambda_C}$ from block to their resolved conditions

$r_{bound} \leftarrow$ getBoundRegister(i_{cond})
$i_{def} \leftarrow$UseDef(r_{bound})
$worklist \leftarrow [(i_{def}, r_{bound})]$
$visited \leftarrow$ new set()
while $worklist$ is not empty **do**
  $(i_{curr}, r_{curr}) \leftarrow worklist$.pop()
  if isResolvableToMessage(r_{curr}, B_{loop}, MB, λ_C) or isResolvableToConfig(r_{curr}, B_{loop}, CD, λ_C) **then**
    break
  end if
  if $r_{curr} \notin visited$ **then**
    $visited \leftarrow visited \cup \{r_{curr}\}$
    $r_{operands} \leftarrow$ getOperandRegisters(i_{curr}) {Get all registers part of instruction}
    for all $r \in r_{operands}$ **do**
      $worklist$.append((UseDef(r), r))
    end for
  end if
end while

function
isResolvableToMessage(r_{curr}, B_{loop}, MB, λ_C):

for all $r_{msg} \in$ MB **do**
  if $r_{curr} == r_{msg}$ **then**
    offset = getOffsetValue(r_{msg})
    {offset to access message field from
    base address}
    $\lambda_C[B_{loop}] \leftarrow r_{msg}$:offset
    return true
  end if
end for
return false
end function

function
isResolvableToConfig(r_{curr}, B_{loop}, CD, λ_C):

for all $r_{config} \in$ CD **do**
  if $r_{curr} == r_{config}$ **then**
    $\lambda_C[B_{loop}] \leftarrow r_{config}$
    return true
  end if
end for
return false
end function

data dependencies, it is important to note that this approach is heuristic-based and not guaranteed to succeed in all cases. While it is effective for common patterns such as direct field accesses, simple copy propagation, or even nested field accesses, it may fail when dealing with more complex constructs like pointer aliasing, function indirection, etc. The algorithm performs a backward traversal over the variable's previous definitions until it either reaches the message or configuration structure or determines that the condition cannot be resolved. We adopt a consistent representation format of "$index \bowtie value$" for resolved loop conditions, where the value must be statically traceable and syntactically simple. In cases where the algorithm fails to resolve the loop bound, we still annotate the corresponding conditional block using the partial format "$index \bowtie$", leaving the value unspecified. Additionally, we provide the source code line number of the unresolved condition, offering the user a helpful hint to manually supply the required value based on their understanding of the application. These unresolved cases require manual intervention, where the user is expected to provide the appropriate value based on application-specific domain knowledge.

3.3 Model Construction

In this section, we describe the process of constructing the Timed Automata model used for verification, and demonstrate the resulting model using the package code shown in Figure 3.2. Since standard Timed Automata do not support variable updates, we use the Extended Timed Automata formalism introduced in Section 2.2.1.

3.3.1 Limiting Messages received on Incoming Topics

```
std::time_t current_time = std::time(nullptr);
std::tm *local_time = std::localtime(&current_time);
int current_hour = local_time->tm_hour;

if (current_hour >= 18) {
    Sensor_Subscriber_Node = this->create_subscription<std_msgs::msg::String>(
        "/sensor_feed", 10,
        std::bind(&ROSPubSubApplication::Sensor_Message_Callback, this, _1));
    Camera_Subscriber_Node = this->create_subscription<std_msgs::msg::String>(
        "/camera_feed", 10,
        std::bind(&ROSPubSubApplication::Camera_Message_Callback, this, _1));
} else {
    Camera_Subscriber_Node = this->create_subscription<std_msgs::msg::String>(
        "/camera_feed", 10,
        std::bind(&ROSPubSubApplication::Camera_Message_Callback, this, _1));
    Sensor_Subscriber_Node = this->create_subscription<std_msgs::msg::String>(
        "/sensor_feed", 10,
        std::bind(&ROSPubSubApplication::Sensor_Message_Callback, this, _1));
}
```

Listing 3.10: Registration order of Incoming Topic Event-Handlers in ROS.

In Listing 3.10, which is made up for illustrative purposes, we see based on the "current_hour" value, which is determined at runtime, the order in which incoming topics are created changes. If we take the if-branch /sensor_feed topic is given priority over the /camera_feed topic and vice versa. This, in turn, affects the sequence in which their event-handlers are dequeued and dispatched, as ROS executors follow a priority-based scheduling mechanism using the registration-order of the "create_subscription" method as seen in Chapter 2.3.1.1. Due to this runtime-dependent behavior, different executions may result in different execution orders. However, since our analysis does not account for runtime variations, we assume that all possible orderings of subscriber creation can occur. Consequently, we do not explicitly model the internal scheduling order of event-handlers. Instead, we allow the UPPAAL verifier to non-deterministically select the next event-handler for execution based on the current count of received messages of the subscribers. This method, although it offers an over-approximation,

leads to a substantial increase in the state-space explosion caused by the combinatorial expansion of execution options.

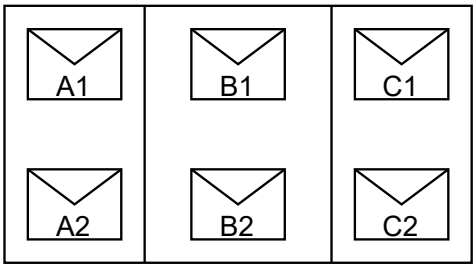


Figure 3.7: Snapshot of `event_counter`: Shared Counter Array Tracking Message Counts per Incoming Topic F_i.

Consider Figure 3.7, which shows the current status of the globally shared counter array `event_counter`. Its size is 3, corresponding to the 3 different incoming topics, $F = \{\text{A, B, C}\}$. Each topic has generated 2 messages so far {A1, A2}, {B1, B2}, and {C1, C2}, respectively. When UPPAAL attempts to non-deterministically choose the next message to dequeue and dispatch the event-handler associated with the incoming topic, and if we consider there is ordering among the messages within the same topic i.e., A1 is always dispatched before A2 (which happens in actual ROS executors because it uses an queue) and similarly for the other topics, then the total number of different valid orderings is given by:

$$\frac{6!}{2 \times 2 \times 2} = \frac{720}{8} = 90$$

And it further reduces to 1 if we impose the priority of message handling based on the registration order of the event-handlers. However, in our case, we do not explicitly model any message queue entity. All messages, both within and across topics, are treated as identical in that they are simply values used to invoke their respective event-handlers. Therefore, the total number of possible interleavings for these 6 messages is the number of unrestricted permutations of 6 distinct messages, which is:

$$6! = 720$$

This number keeps growing at a rate faster than exponential, with an increase in the number of messages generated. To address this issue, we implement a limit on the number of messages, which helps to restrict the potential execution scenarios. This has also been empirically

observed, the time taken to obtain the verification results from UPPAAL increases drastically when we increase the message limit from 10 to 20. This constraint is essential for maintaining the computational feasibility of the verification process within a manageable time period. Thus, we represent the limit on the incoming topic messages using Δ and define it as follows:

- Message limit on Incoming Topic, $\Delta : F \rightarrow \mathbb{R}^+ = \{\Delta_1, \Delta_2, \cdots, \Delta_n\}$, where Δ_i represents the maximum number of messages the incoming topic F_i is allowed to receive.

Over-approximating the scheduling of event-handlers using non-determinism instead of priority-based ordering has the advantage of ensuring soundness, as it captures all possible interleavings regardless of specific execution behavior. This makes the analysis more general and avoids relying on platform-specific scheduling details, such as subscriber registration order in ROS. However, this comes at the cost of significantly increasing the state-space, which can lead to false positives, reporting timing violations that may not actually occur due to practical execution constraints like FIFO or priority ordering.
Although model checkers like UPPAAL do not inherently require bounded inputs, introducing bounds it is often necessary to keep the state space manageable. Without such limits, even a single unbounded integer variable can lead to a vast number of states, severely impacting verification time. Moreover, as we deal with fine-grained modeling of event-handlers, each resulting in automata with a large number of locations, combining such detailed models with over-approximation and unbounded message limit would significantly exacerbate the state-space explosion problem, making verification computationally infeasible.
To mitigate the state-space explosion caused by over-approximation, we impose a limit on the number of messages each incoming topic can receive. While this improves tractability and ensures the verification process completes in a reasonable timeframe, it also carries the risk of missing real timing violations that could occur in long-running executions or under more message arrivals. Thus, while this approach is conservative and scalable, it may compromise precision or completeness depending on the chosen message limit. However, since the message-bound values can be adjusted by the user before verification, this limitation is not a major concern. The model will still report potential timing violations if they exist, though higher message bounds may result in longer verification times due to increased state space.

While showing the result of the algorithm on the package code shown in Figure 3.2, we assume $\Delta = \{3, 3, 3\}$. Before presenting the automata construction and their corresponding output, we define several notational conventions used throughout the results in this section:

1. Clocks are represented in bold. For example: clock **signal**

2. Actions are underlined. For example: action go!

3. Location names are represented using numbers (except for one descriptive location)

4. Variables are written using teletype font and initialized with zero (for integer) and false (for boolean). For example: int `shared_counter`

5. Special locations: Urgent locations are marked with a U inside the location, initial locations are denoted by concentric circles, and committed locations are marked with a C inside the location.

6. Shorthand notations, eh = event-handler

3.3.2 Incoming Message Generator

A Timed Automaton is constructed for each incoming topic F_i to model the periodic arrival of messages under a message limit constraint Δ_i. Each automaton simulates message arrivals at fixed time intervals, capturing the timing behavior of that topic within the system. These automata incorporate a timer, a looping transition structure to model periodic behavior, and a global shared counter that tracks the total number of messages generated per topic. This counter is later utilized by other automata components in the system to initiate actions based on message arrival. For example, the counter is decremented when the Incoming Message Dispatcher Automata consumes the message for event-handler execution corresponding to the message's topic. Additionally, each automaton maintains a local variable to record the number of messages generated so far for its topic, which is never decremented. This makes sure we stop the message generation when we have reached the message limit on the variable. A globally shared boolean array is introduced, which is updated based on the local variable, indicating whether the message limit for each topic has been reached. While the boolean array allows other automata to monitor message-generation status, the shared counter facilitates actions triggered by message arrivals. As previously discussed, the message limit Δ ensures computational tractability by capping the number of potential executions. Each transition in the automaton is defined by a 6-tuple: (Source Location, Action, Guard Condition, Clock Resets, Variable Updates, Target Location), aligning with the formal semantics in Definition 2.3.

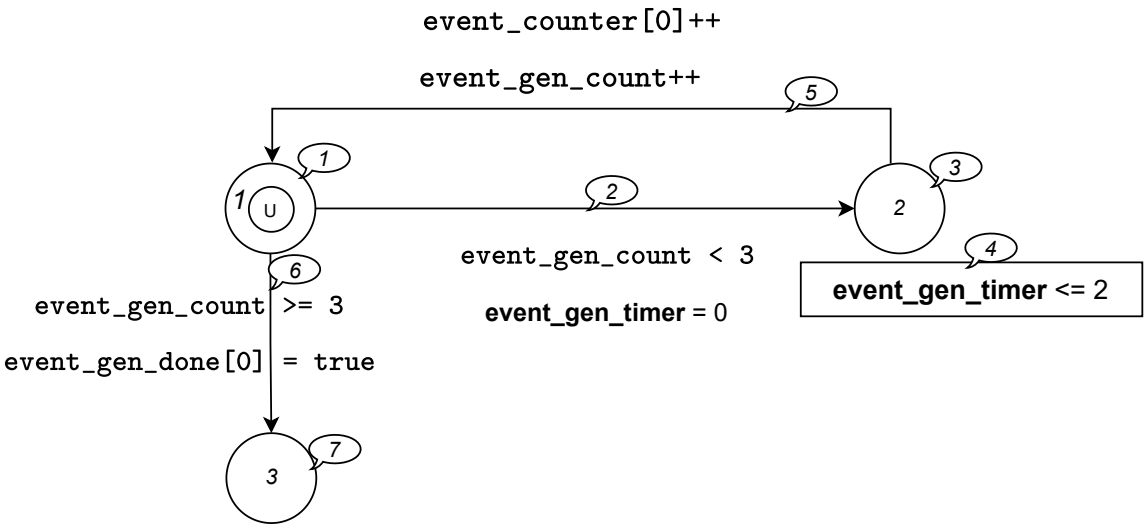
Result

Figures 3.8a, 3.8b, and 3.8c present the Timed Automata constructed for the example ROS package illustrated in Figure 3.2, with each figure corresponding to one of the three incoming topics. A global counter array, `event_counter`, initialized to zero and sized according to the

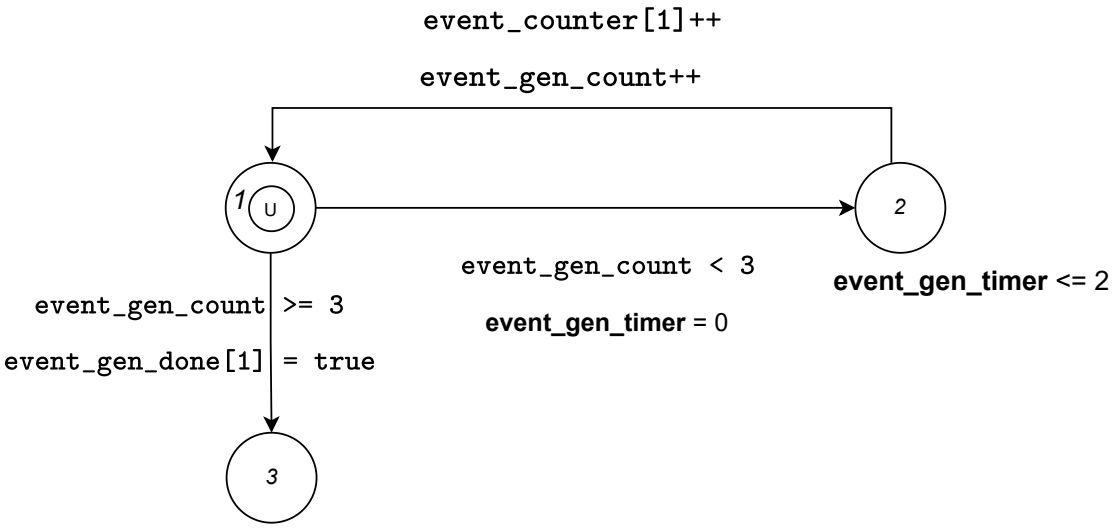
number of incoming topics, serves as a shared variable across other automata. It allows other automata components in the system to observe, react to, and modify the number of messages generated for each topic. Its interactions will be explored in subsequent sections. Similarly, a global boolean array, `event_gen_done`, also sized by the number of incoming topics and initialized to false, is shared across automata to indicate whether message generation has completed for a given topic. Each generator automaton also maintains a local clock, **event_gen_timer**, and a local counter variable, `event_gen_count`, which tracks the number of messages generated so far. This local counter is monotonically increasing, never decremented, and is used solely for comparison against the message limit, Δ_i.

To enforce periodic message generation, an invariant (Callout 4) is placed on Location 2. The invariant value is derived from the timing assumption of the respective topic, ensuring that the automaton cannot remain in this location for more than the specified duration. Specifically, the invariant value is set to 2 time units for the initial_pose and input_cloud topics, and 5 time units for the imu topic. These values are derived from the timing assumption values T specified in the ROS example shown in Figure 3.2. The timing assumption for each topic represents the maximum allowed time gap between two consecutive messages. Therefore, these values are directly used in the invariant to ensure that message generation adheres to the expected periodic behavior.

Message generation is further controlled by guard conditions (Callouts 2,6), which ensure that a message is generated only if the local counter `event_gen_count` is still less than the topics' message limit, Δ_i=3 in all three examples. Once the timing constraint is satisfied at Location 2, the transition is triggered. At this point, both the global counter `event_counter` (at the topic-specific index) and the local counter `event_gen_count` are incremented (Callout 5), and the clock **event_gen_timer** is reset (Callout 2) to maintain the periodic structure of the automaton. When the local counter `event_gen_count` reaches the message limit, the guard condition in Callout 2 evaluates to false, and its complementary condition in Callout 6 evaluates to true. This causes the automaton to transition to Location 3, where message generation permanently halts for that topic. Additionally, the corresponding boolean flag in the `event_gen_done` array is set to true, allowing other automata in the system to detect that message generation for that topic has completed.

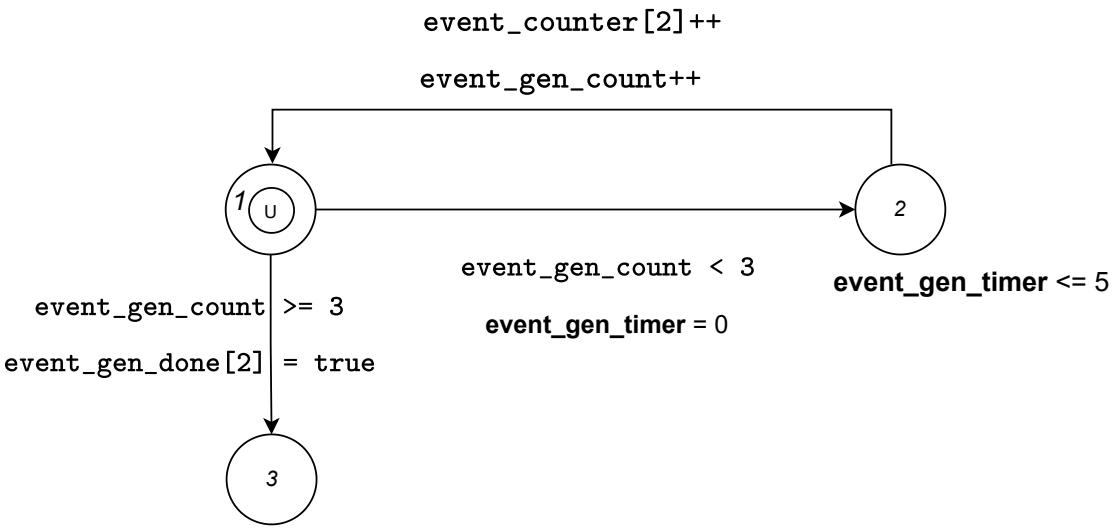


(a) Incoming Message Generator for Incoming Topic initial_pose.



(b) Incoming Message Generator for Incoming Topic input_cloud.

Figure 3.8: Incoming Message Generators of ROS Package 3.2.



(c) Incoming Message Generator for Incoming Topic imu.

Figure 3.8: (Continued) Incoming Message Generators of ROS Package 3.2.

3.3.3 Incoming Message Dispatcher

A Timed Automaton is constructed to model the behavior of dispatching incoming messages and coordinating the execution of event-handlers associated with incoming topics. This automaton continuously monitors the globally shared counter array, `event_counter`, which is updated by the Incoming Message Generator Automata as described in the previous section. If a pending message is detected for any incoming topic, the automaton initiates the corresponding event-handler by sending a signal via the handle_eh[index]! action where index $\in$ 0, 1, 2 maps to a specific incoming topic in this example, as shown in Figure 3.2, thereby triggering the execution of the corresponding event-handler automaton. After dispatching, the automaton waits for a response from the respective event-handler automaton via the handled_eh? action. Once this acknowledgment is received, indicating the completion of event-handler execution, the dispatcher resumes checking for other pending messages. This message dispatch cycle is repeated in a loop, with checks occurring at fixed intervals of one time unit. The termination condition for this loop is evaluated using the shared boolean array `event_gen_done`. If there are no pending messages to dispatch and all entries in `event_gen_done` are true, meaning that message generation for all topics has completed, the automaton transitions to Location 6. This location signifies the end of the dispatching process. Upon entering this location, the automaton emits a stop_automata! action, which is received by the Outgoing Message Checker Automata to finalize their monitoring process. Each transition in this automaton is formally represented

using a 6-tuple: (Source Location, Action, Guard Condition, Clock Resets, Variable Updates, Target Location), following the formal semantics described in Definition 2.3.

Result

Figure 3.9 illustrates the Timed Automaton constructed for dispatching incoming messages. This automaton is responsible for invoking the appropriate event-handler automaton for each incoming topic, one at a time, based on the availability of unprocessed messages. Message availability is determined by inspecting the global counter array `event_counter` (Callout 2), which is populated by the Incoming Message Generator Automata. The result of this check is stored in the local variable `eh_status`.

If no pending messages remain to be dispatched, the guard condition at Callout7 evaluates to true, and the automaton transitions to Location 5. From there, it either proceeds to Location 6, if all entries in the boolean array `event_gen_done` are true (Callout14), signifying that message generation and dispatching have completed for all topics. It also emits a signal via the stop_automata! action to inform the Outgoing Message Checker Automata to transition accordingly. Alternatively, if any entry in the `event_gen_done` array is still false, the automaton loops back to the initial Location 1 (Callouts 13, 12) and continues checking for pending messages.

When multiple messages are pending, the automaton non-deterministically selects an event-handler of one topic for dispatch. This modeling choice aligns with our use of the Static Single-Threaded Executor, as discussed in Section 2.3.1.1. Unlike ROS's actual executor model, which enforces a deterministic event-handler invocation order based on registration, the non-deterministic approach in our model allows UPPAAL to explore all possible interleavings of event-handler executions. This is essential to account for various possible registration orders, but it introduces challenges in traceability during verification. To manage this complexity, we enforce a message limit constraint, as discussed in Section 3.3.1.

In this case, three edges are present from Location 2 to Location 3 (Callouts 4, 5, 6), corresponding to the three incoming topics based on the example in the Figure 3.2. Before taking any of these edges, a guard condition ensures that the selected topic has pending messages. Once an edge is selected, the dispatcher sends a signal via the handle_eh[index]! action, where index $\in$ 0, 1, 2 maps to a specific incoming topic, thereby triggering the execution of the corresponding event-handler automaton.

After dispatching, the message count for the selected topic is decremented in the `event_counter`, and the automaton transitions to Location 3. At this location, it waits for an acknowledgment

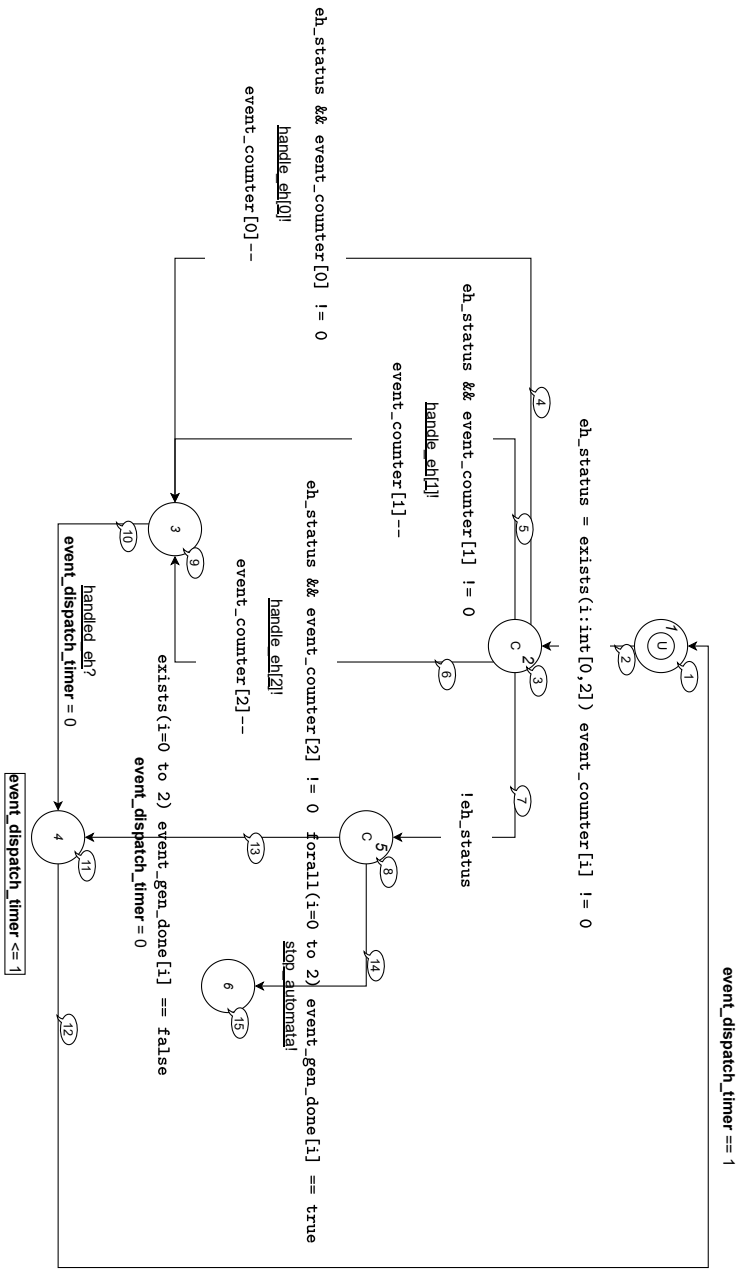


Figure 3.9: Incoming Message Dispatcher.

via the handled_eh? action (Callout 10) from the respective Incoming Topic Event-Handler Automaton, signaling the completion of event-handler execution. Upon receiving the acknowledgment, it resets the clock **event_dispatch_timer** to 0 and remains in Location 4 for exactly one time unit as governed by its invariant. The invariant makes sure we stay at that location 4 for 1 time unit before transitioning back to the initial location 1. This behavior enables the dispatcher to periodically check and react to incoming messages in an event-driven manner.

3.3.4 Incoming Topic Event-Handler

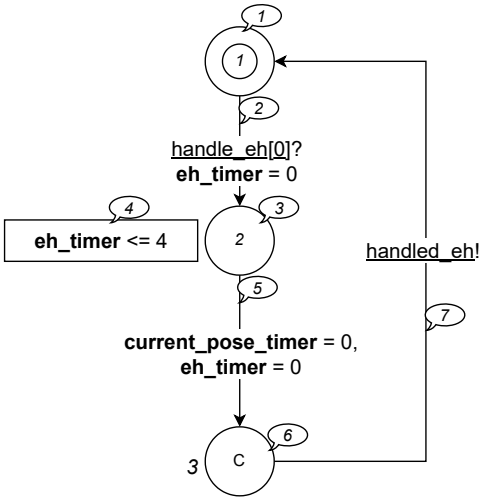
A Timed Automaton is constructed for each incoming topic F_i, guided by its corresponding Control Flow Graph (CFG) G_i. The execution of this automaton begins upon receiving a signal on the handle_eh[index]? action from the Incoming Message Dispatcher Automaton, indicating that a message is pending for the topic and that the event-handler associated with the indexed topic should be invoked. The automaton then traverses the execution path defined by the CFG, modeling key control structures such as computations, conditionals, and loops. Internal computations are abstracted away, but their timing is captured using Worst-Case Execution Time (WCET) annotations (more about this in Section 4.5), represented through clock time progression within the automaton. Simple, statically resolvable conditions, especially those based on configuration parameters, are modeled explicitly. In contrast, complex or dynamic conditions are conservatively represented non-deterministically. Loop bounds are resolved using the Algorithm (Section 3.2) when possible otherwise, user intervention is required to manually specify the bounds. Additionally, any publish operations encountered during the CFG traversal are incorporated into the automaton and modeled using clock resets to accurately capture their timing behavior. Correspondingly, the Outgoing Message Checker Automata described in the next section monitor whether these clocks are reset within their specified timing constraints, i.e., whether the messages are being published in accordance with their timing requirements. Once the event-handler execution concludes, the automaton sends an acknowledgment back to the dispatcher via the handled_eh! action, signaling completion. Each transition in the constructed automaton is defined using a 6-tuple: (Source Location, Action, Guard Condition, Clock Resets, Variable Updates, Target Location), which captures the formal semantics of transitions as described in Definition 2.3.

Result

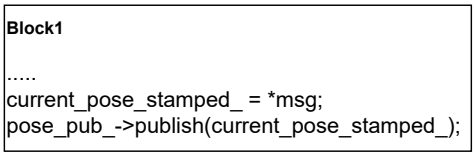
Figures 3.10a, 3.10c, and 3.10e present the Timed Automata constructed for the example ROS package illustrated in Figure 3.2, with each automaton corresponding to the event-handler of one of the three incoming topics defined in the system. For instance, Figure 3.10a shows the

Timed Automaton generated for the incoming topic F_1 = initial_pose, constructed based on its Control Flow Graph (CFG) G_1 shown in Figure 3.10b. The execution of this automaton begins when it receives a synchronization action handle_eh[0]? (Callout 2) from the Incoming Message Dispatcher Automaton, indicating that a message from the initial_pose topic (index = 0) is ready to be processed and the corresponding event-handler should begin execution.
Upon receiving the synchronization action, the automaton transitions to Location 2, resetting the clock **eh_timer** (Callout 2). This reset marks the beginning of the event-handler execution and enables precise tracking of the timing behavior for each basic block. The clock **eh_timer** is reset upon entering each new location to measure the execution time of that specific basic block independently and to enforce its corresponding Worst-Case Execution Time (WCET) constraint. As the automaton progresses through the structure defined by the CFG, each location is annotated with an invariant representing the WCET of the associated basic block. For instance, Location 2 is annotated with the invariant **eh_timer** ≤ 4, indicating that the basic block corresponding to this location has a WCET of 4 time units, as obtained from the λ_T of the CFG of the Example 3.2. This also implies that the automaton may spend at most 4 time units at this location before transitioning and resetting the clock for the next segment (Callout 4).
Instead of explicitly modeling all computational instructions, the automaton abstracts these low-level details and uses WCET-based annotations to accurately capture the timing behavior of the event-handler logic. One critical operation that is explicitly modeled, however, is the publication of messages. In the example shown, the publish statement "pose_pub_->publish(..)" located within a basic block of the event-handler, as illustrated in Figure 3.10b, is represented by a clock reset: **current_pose_timer** = 0, in the automata as shown in Callout 5. Since the handler object pose_pub_ is associated with the outgoing topic current_pose, the corresponding clock (**current_pose_timer**) is reset to indicate that a message has been published on this topic. This clock reset is essential for the Outgoing Message Checker Automaton, which relies on it to verify that messages are being published within the required timing constraints. As the event-handler continues execution, additional locations may be visited, each representing further blocks in the CFG, and each with its own timing constraints and potential clock resets. Once the execution path completes, the automaton reaches Location 3 and emits a handled_eh! action (Callout 7), signaling to the Incoming Message Dispatcher Automaton that the message has been fully processed and the event-handler is ready to handle the next message.
Similarly, conditional statements are modeled using guard conditions, as illustrated in Figure 3.10e, where each outgoing edge from a conditional block is guarded by its respective condition. This allows the automaton to follow a precise execution path based on the evalu-

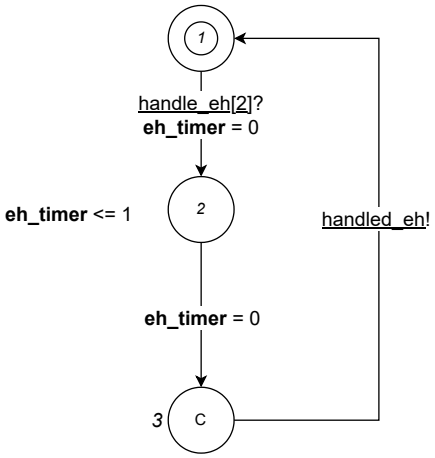
ated condition value. The resulting branching structure ensures that the automaton accurately mirrors the control-flow logic defined in the source code.



(a) Incoming Topic Event-Handler Automata for Incoming Topic initial_pose



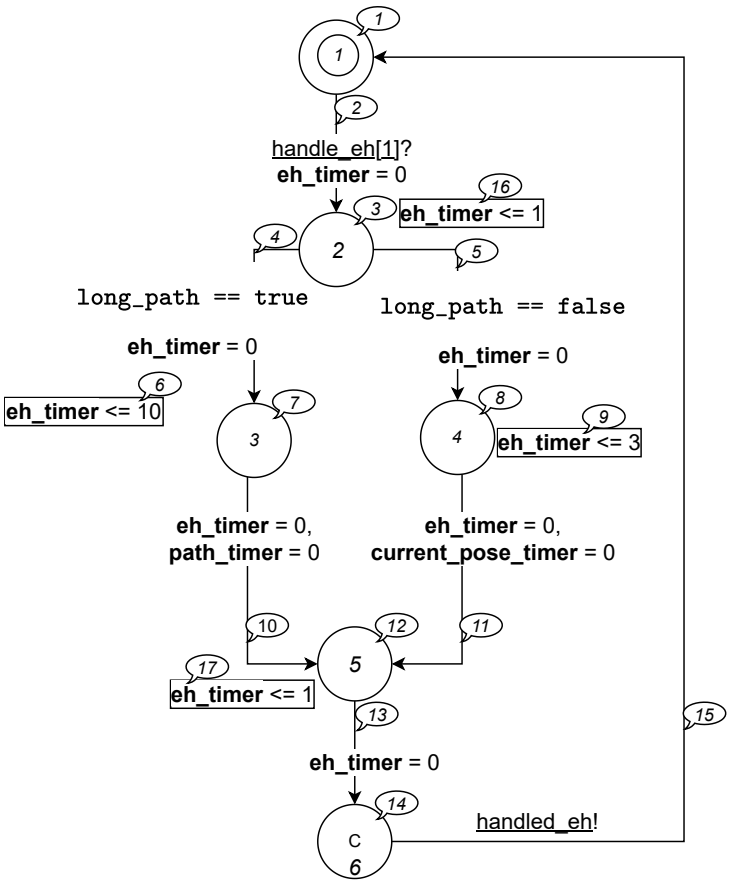
(b) CFG representation of Lines 14–22 from the initial_pose_event_handler function in Listing 3.1



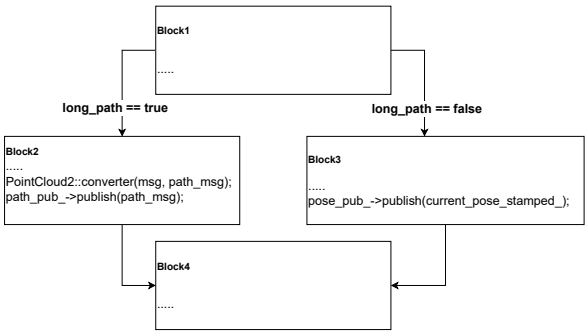
(c) Incoming Topic Event-Handler Automata for Incoming Topic imu

(d) CFG representation of Lines 24-41 from the imu_event_handler function in Listing 3.1

Figure 3.10: Incoming Topic Event-Handler Automata of Example ROS Package 3.2



(e) Incoming Topic Event-Handler Automata for Incoming Topic input_cloud



(f) CFG representation of Lines 43-48 from the input_cloud_event_handler function in Listing 3.1

Figure 3.10: (continued) Incoming Topic Event-Handler Automata of Example ROS Package 3.2

3.3.5 Outgoing Message Checker

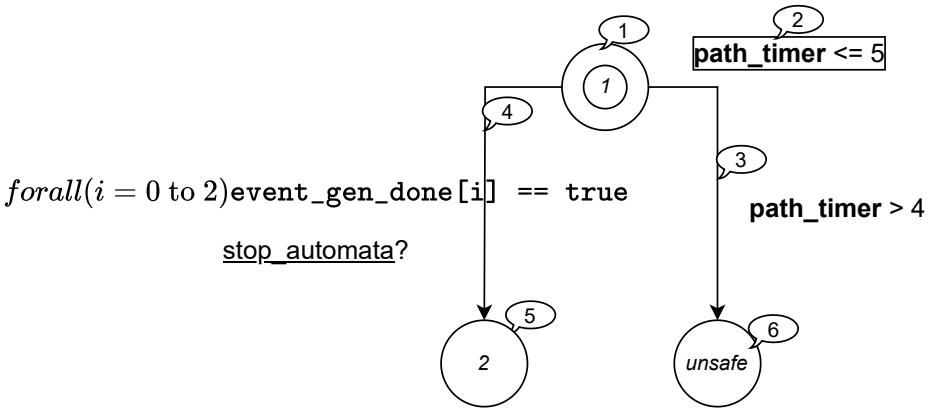
A Timed Automaton is constructed for each outgoing topic H_k. Each automaton is responsible for verifying whether messages published on the corresponding output topic comply with the specified timing constraint τ_k. Message publications are modeled as clock resets within the Incoming Topic Event-Handler automata. Therefore, if a message publication is delayed, i.e., the clock associated with the outgoing topic is not reset within the specified timing constraint, the automaton transitions to an *unsafe*, indicating a potential violation of the timing requirement. However, due to the message limit imposed on incoming topics, there may come a point when no more messages are available to be dispatched. In such cases, even if no timing violation has occurred, the automaton would eventually transition to the *unsafe* location simply because no further messages are being published, resulting in false positives. To address this, we introduce a new Location 2. The automaton transitions to this state only when two conditions are met: (1) no publication for the topic has been delayed yet, and (2) no further messages remain to be dispatched. This transition is triggered by receiving the stop_automata action, which is sent by the Incoming Message Dispatcher Automaton. Thus, this automaton effectively evaluates whether each outgoing topic H_k adheres to its timing constraint τ_k, without falsely flagging violations due to message exhaustion. Each transition in the automaton is formally defined using a 6-tuple: (Source Location, Action, Guard Condition, Clock Resets, Variable Updates, Target Location), in accordance with the semantics described in Definition 2.3.

Result

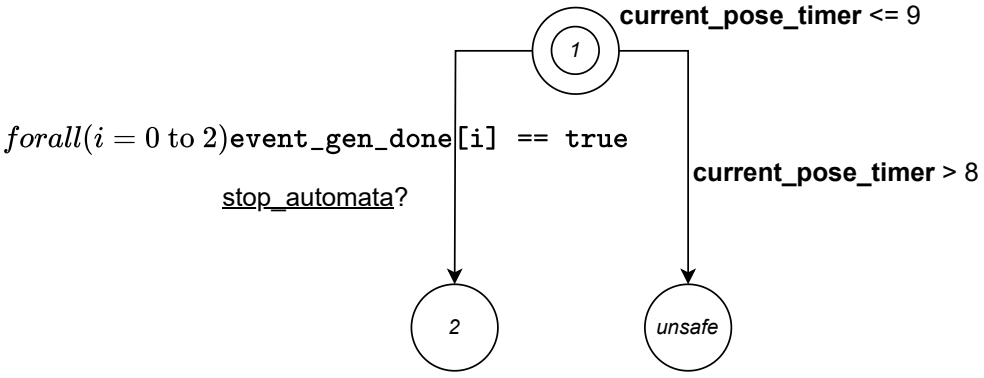
Figures 3.11a and 3.11b present the Timed Automata constructed for the example ROS package shown in Figure 3.2, with each figure corresponding to one of the two outgoing topics. For instance, Figure 3.11a illustrates the Timed Automaton for the outgoing topic $H_1 = \text{path}$, which is associated with a timing constraint $\tau_1 = 4$ and it's clock **path_timer**, while Figure 3.11b illustrates the Timed Automaton for the outgoing topic $H_1 = \text{current\_pose}$, which is associated with a timing constraint $\tau_1 = 8$ and it's clock **current_pose_timer**.

The timing constraint values τ are obtained from the ROS example shown in the Figure 3.2 and are enforced at two key places. First, it appears as a guard condition (Callout 3) on a transition that checks whether the timing bound has been violated, that is, whether a message was not published within the allowable time window. Second, it is incorporated into the invariant on Location 1 (Callout 2). The invariant ensures that the automaton cannot remain at that location for more than the specified duration. Without this invariant, even if the guard condition becomes true, the automaton could indefinitely remain in the same location, delaying

the detection of timing violations. To ensure prompt detection, the invariant is set to $\tau_1 + 1$ (5 for the path topic), forcing the automaton to exit Location 1 and transition to the designated *unsafe* once the bound is exceeded. Conversely, if all messages are successfully processed within the allowed time, the automaton transitions to Location 2. This transition is triggered through synchronization with the stop_automata? action (Callout 4), which originates from the Incoming Message Dispatcher Automaton. It signals that no further messages are expected. In addition, the transition is guarded by a condition on the shared boolean array `event_gen_done`, which ensures that message generation has completed for all incoming topics. Together, these conditions allow the system to safely conclude message monitoring.



(a) Outgoing Message Checker for Outgoing Topic path



(b) Outgoing Message Checker for Outgoing Topic current_pose

Figure 3.11: Outgoing Message Checkers of Example ROS Package 3.2

We established an *unsafe* location within the Timed Automata that is visited in the event of a potential timing violation. Consequently, we can utilize the query "E<> unsafe" to determine whether any potential timing violations exist.

Chapter 4

Implementation

In the previous chapter, we presented the algorithms used to construct the Timed Automata model, which is central to our approach. This chapter shifts focus to the initial phase of the approach, the first 3 steps in the red highlighted region in the Figure 3.1, which involves extracting the required input data using heuristics from the package source code. This extracted data serves as the input for the model construction phase discussed earlier. Specifically, we detail the process of retrieving the following information:

- Incoming Topics F and their corresponding Event Handlers G,
- Outgoing Topics H,
- Outgoing Topics that are published through the Event-Handlers.

We then turn our attention to the blue-highlighted region of Figure 3.1, which represents the post-model construction phase, encompassing trace analysis, feedback, and patch generation. This phase begins once the verification results from the Timed Automata model become available.

Additionally, we justify the use of placeholder values for the Worst-Case Execution Time (WCET) of basic blocks in our model. Precise WCET estimation remains a longstanding challenge, particularly in systems with complex control flow and huge package dependencies. While using placeholder values may reduce the precision of the results, our goal is to develop a flexible and extensible analysis framework. To that end, the framework allows users to supply custom WCET values, enabling more accurate and tailored analysis when such information is available.

4.1 Identifying Incoming Topics and Event Handlers

In this section, we present the technique used to identify and extract the set of incoming topics F and their corresponding event-handlers G from the package source code, which serves as input to the model construction phase. In our current approach, we assume that incoming topic creation is not enclosed within loops or conditional branches. If such constructs are encountered, they are ignored, as resolving their runtime behavior would require dynamic analysis to accurately determine execution paths and loop iterations, an aspect beyond the scope of this work. Some timing violations may go undetected if critical message sources are instantiated within these ignored constructs. We first present the output, accompanied by an explanation through a concrete example. This is followed by an illustrative example demonstrating the analysis flow using a simplified three-address SSA code format to illustrate how such constructs are identified.

Result

Input Package Code

```
1 int main() {
2   .....
3   initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>(
4     "initial_pose", rclcpp::QoS(10), initial_pose_callback);
5   imu_sub_ = create_subscription<sensor_msgs::msg::Imu>(
6     "imu", rclcpp::SensorDataQoS(), imu_callback);
7   .....
8 }
```

Listing 4.1: Lidarslam Package Code in ROS [45]

Output

- F = {initial_pose, imu}
- G = {initial_pose_callback, imu_callback}

We start our analysis from the main function and its transitive calls (inlined into main) to identify incoming topics and their corresponding event-handlers. This is done using a heuristic based on the presence of the keyword "create_subscription" (Lines 3,5 of Listing 4.1). For each identified subscription, we extract its topic name from the first parameter and the event-handler from the third parameter, storing them in respective maps F and G. For instance, in Listing 4.1, we identify the topic name "initial_pose" and the event-handler "initial_pose_callback" from the arguments of the first create_subscription invocation and, "imu" and "imu_callback"

from the subsequent call as shown in the output. While such mappings are relatively easy to interpret at the source-code level, our analysis is performed entirely at the LLVM Intermediate Representation (IR) level, where extracting this information, especially from the control-flow graph (CFG) of the event-handler, is non-trivial.

```
1  .string_value = "initial_pose"

3  initial_pose_callback:
4          ...

6  main:
7          ...
8          callback_ptr = addrof initial_pose_callback
9          string_data_ptr = addrof @string_value
10         ...
11         topic_name_param = call std::string::constructor(string_data_ptr)
12         ...
13         call create_subscription(topic_name_param, ..., callback_ptr)
14         ....
```

Listing 4.2: Simplified Intermediate Representation of Listing 4.1

To illustrate how this works, we provide a simplified IR flow in Listing 4.2. As described, we begin from the "create_subscription" call and examine the first parameter, which contains a pointer to the topic name. We perform a backward traversal over its previous definitions to trace back to the corresponding string constructor. From the constructor, we continue tracing to retrieve the actual pointer to the string data and, ultimately, the string literal itself. Similar traversal is done for identifying the event-handler as well, which is the third parameter in the "create_subscription" call. While this simplified view helps with understanding, actual LLVM IR analysis involves navigating a more intricate web of variables due to the SSA (Static Single Assignment) nature of the IR. Each definition introduces a new variable, requiring more backward traversal logic. We acknowledge that, due to the heuristic nature of our approach, certain edge cases, such as topic names constructed as part of user input or heavily aliased references, may remain unresolved. To mitigate this, we use placeholder topic names in cases where resolution is not possible. While topic names help narrow down which part of the system is being analyzed, they do not directly influence the generation or detection of timing violations. As such, using placeholders allows the analysis to proceed without impacting the integrity of the timing model. Nonetheless, we successfully extract the necessary topic and event-handler mappings. This process involves systematically walking through basic blocks and inspecting instruction operands.

4.2 Identifying Outgoing Topics

In this section, we present a technique used to identify Outgoing Topics H that are published within the event-handlers of Incoming topic G from the package source code, which serves as input to the model construction phase. In our current approach, we assume that outgoing topic creation is not enclosed within loops or conditional branches. If such constructs are encountered, they are ignored, as resolving their runtime behavior would require dynamic analysis to accurately determine execution paths and loop iterations, an aspect beyond the scope of this work. Some timing violations may go undetected if critical message sources are instantiated within these ignored constructs. We first present the output, accompanied by an explanation through a concrete example.

Result

Input Package Code

```
1  int main() {
2    .....
3    pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose",
         rclcpp::QoS(10));
4    map_pub_ = create_publisher<sensor_msgs::msg::PointCloud2>("map", rclcpp::QoS(10));
5    map_array_pub_ = create_publisher<lidarslam_msgs::msg::MapArray>(
6      "map_array", rclcpp::QoS(rclcpp::KeepLast(1)).reliable());
7    path_pub_ = create_publisher<nav_msgs::msg::Path>("path", rclcpp::QoS(10));
8    .....
9  }
```

Listing 4.3: Lidarslam Package Code in ROS [45]

Output

- H = {current_pose, map, map_array, path}

We start our analysis from the main function and its transitive calls (inlined into main) to identify outgoing topic data. This identification relies on a heuristic based on the presence of the keyword "create_publisher" (as seen in Lines 3, 4, 5, 7 of Listing 4.3). For each detected invocation, we extract the topic name from the first parameter and store it in the appropriate mapping structure H. For example, in Listing 4.3, we extract the topic name current_pose from the first parameter of the initial "create_publisher" call and map, map_array, path from its subsequent calls as shown in the output. Although this is straightforward to observe in the source code, our analysis is conducted entirely at the LLVM IR level, where such extraction is

significantly more complex. To handle this, we apply the analysis flow described in the previous section, which enables us to systematically populate the H map. The only difference is the replacement of the keyword "create_subscription" with "create_publisher" to match outgoing topic patterns. We acknowledge that, due to the heuristic nature of our approach, certain edge cases, such as topic names constructed as part of user input or heavily aliased references, may remain unresolved. To mitigate this, we use placeholder topic names in cases where resolution is not possible. While topic names help narrow down which part of the system is being analyzed, they do not directly influence the generation or detection of timing violations. As such, using placeholders allows the analysis to proceed without impacting the integrity of the timing model.

4.3 Identifying Topics published through Event Handlers

In this section, we present a technique used to identify and extract the outgoing topics being published within the event-handlers of incoming topics from the package source code, which serves as input to the model construction phase. In our current approach, we assume that incoming topics and outgoing topics creation are not enclosed within loops or conditional branches. If such constructs are encountered, they are ignored, as resolving their runtime behavior would require dynamic analysis to accurately determine execution paths and loop iterations, an aspect beyond the scope of this work. Some timing violations may go undetected if critical message sources are instantiated within these ignored constructs. We first present the output, accompanied by an explanation through a concrete example.

Result

Input Package Code

```
2    ...
3    pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose", rclcpp::QoS
         (10));
4    map_pub_ = create_publisher<sensor_msgs::msg::PointCloud2>("map", rclcpp::QoS(10));
5    path_pub_ = create_publisher<nav_msgs::msg::Path>("path", rclcpp::QoS(10));
6     ...

8    auto imu_callback =
9      [this](const typename sensor_msgs::msg::Imu::SharedPtr msg) -> void
10     {
11       // Block1
12       if (initial_pose_received_) {
13         path_pub_->publish(msg);}
14       // Block2
```

```
15        pcl::PointCloud<pcl::PointXYZI>::Ptr map_ptr(new pcl::PointCloud<pcl::PointXYZI>);
16        sensor_msgs::msg::PointCloud2::SharedPtr map_msg_ptr(new sensor_msgs::msg::PointCloud2)
              ;
17        map_ptr = &msg;
18        pcl::toROSMsg(*map_ptr, *map_msg_ptr);
19        map_msg_ptr->header.frame_id = map_frame_id;
20        map_pub_->publish(*map_msg_ptr);
21      };

23    auto initial_pose_callback =
24      [this](const typename geometry_msgs::msg::PoseStamped::SharedPtr msg) -> void
25      {
26        // Block1
27        if (msg->header.frame_id != global_frame_id_) {
28          RCLCPP_WARN(get_logger(), "This initial_pose is not in the global frame");
29          return;
30        }
31        // Block2
32        RCLCPP_INFO(get_logger(), "initial_pose is received");

34        current_pose_stamped_ = *msg;
35        previous_position_.x() = current_pose_stamped_.pose.position.x;
36        previous_position_.y() = current_pose_stamped_.pose.position.y;
37        previous_position_.z() = current_pose_stamped_.pose.position.z;
38        initial_pose_received_ = true;

40        pose_pub_->publish(current_pose_stamped_);
41      };
```

Listing 4.4: Lidarslam Package Code in ROS [45]

Output

- λ_H for G_1 (initial_pose_callback) of F_1 (initial_pose) = {Block1: ϕ , Block2: {current_pose}}, λ_H for G_2 (imu_callback) of F_2 (imu) = {Block1: {path}, Block2: {map}}

We analyze the event-handlers of each incoming topic G to identify outgoing topics that are used in publish statements using a heuristic based on the presence of the keyword "publish(", as shown in Lines 13, 20, 40 of Listing 4.4. It then constructs a mapping from each basic block to the set of outgoing topics published within that block. For instance, in Listing 4.4, the statement "pose_pub_->publish" call appears within Block2 of the "initial_pose_callback" function. Since "pose_pub_" is the handler used to create the publisher for the output topic "current_pose" (Line 3), this block is mapped to the topic "current_pose" accordingly. Similarly, Block1 and Block2 of imu_callback event-handler are mapped to output topics path and map as respectively.

Although the mapping is relatively intuitive when observing the source code, our analysis is

conducted entirely at the LLVM IR level, where determining the association between outgoing topic publish statements and their enclosing control flow context is non-trivial. To address this, we leverage the analysis flow described in the section 4.1 and adapt it to meet our current needs to populate the λ_H map. Specifically, we trace the name of the publish handler used in the publish call and match it against the handler registered during the publisher's creation. Once the handler is resolved, we apply the backward flow analysis algorithm to identify the corresponding topic name associated with it. In practice, we observed that these handler objects are typically defined as global variables, which makes the heuristic both effective and straightforward. However, in rare scenarios where the handler is passed as a function parameter (something that rarely occurs in such systems), the algorithm may not immediately resolve the topic. In such cases, we can extend our analysis to include an additional step: identifying the function parameter, then examining the call site of the function to determine which specific topic was passed in. This extension makes the heuristic robust to slightly more dynamic patterns of topic publication while maintaining the scalability of our approach.

4.4 End-to-End workflow

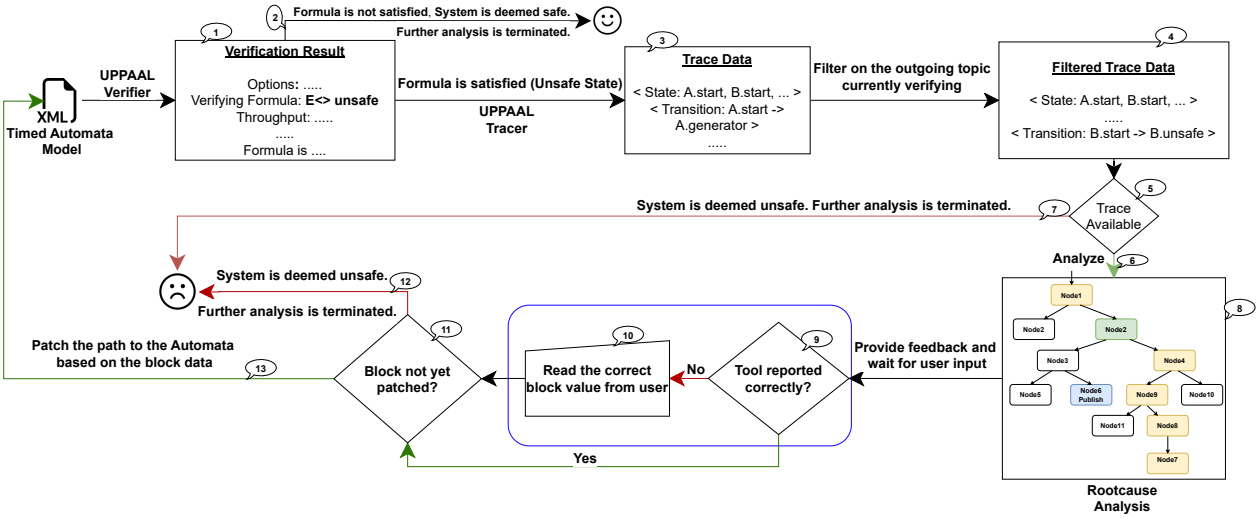


Figure 4.1: End-to-end workflow of post model construction phase.

In this section, we detail the workflow that follows the model construction phase. Figure 4.1 provides a high-level overview of the complete end-to-end evaluation process and serves as a detailed expansion of the blue-highlighted region in Figure 3.1, which represents the post-model construction phase of our framework. The figure outlines the sequence of steps from model

verification to trace analysis, and finally to feedback and patch generation, offering a comprehensive view of how the system detects and addresses potential timing violations. Among these steps, only the portion highlighted in dark blue requires manual intervention from the user. We elaborate more on this in one of the subsequent paragraphs. Once the model is constructed, we use the UPPAAL verification tool to formally check the system's timing properties. If the verification confirms the absence of timing violations (Callout 2), further analysis is terminated and the benchmark is considered to satisfy the specified timing constraints.
However, if a violation is detected, a more detailed diagnostic process is triggered. Specifically, we use UPPAAL's Trace Utility to extract the execution trace corresponding to the violation from the verification results. A trace represents the sequence of transitions across various automata that led to the violation state. It includes transitions between the various automata constructed during the model construction phase, namely, the incoming message generator, outgoing message checker, incoming message dispatcher, and incoming topic event-handler automata, as illustrated in Figure 4.2.

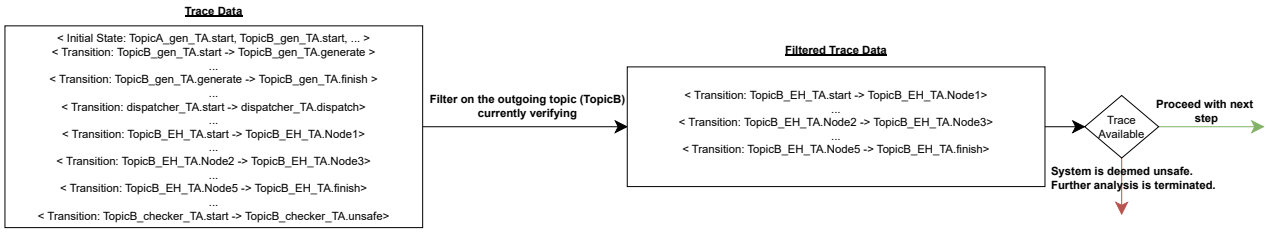


Figure 4.2: Filtering Trace data for rootcause analysis for outgoing topic TopicB.

These are represented respectively as: TopicB_gen_TA (Incoming Message Generator for TopicB), TopicB_checker_TA (Outgoing Message Checker for TopicB), dispatcher_TA (Incoming Message Dispatcher), and TopicB_EH_TA[1] (Incoming Message Event-Handler for TopicB). The labels following each automaton name (e.g., start, generate) refer to the specific locations within each automaton. However, transitions from automata other than the incoming topic event-handler are generally not useful for diagnosing timing violations. Even within this event-handler automaton, we are primarily interested in transitions related to the specific event-handler that contains the publish statement for the relevant outgoing topic H under analysis. To streamline the analysis, the trace is filtered to retain only the final transitions associated with the incoming topic event-handler responsible for publishing the message on the outgoing topic in question. In the example where we are analyzing the outgoing topic TopicB, only transitions from TopicB_EH_TA (the event-handler automaton for TopicB) are retained, as highlighted in

[1]We abbreviate Timed Automata as TA and Event-Handler as EH

Figure 4.2. Transitions from unrelated automata are excluded, as they do not offer meaningful insight into the root cause of the violation. After this filtering step, three possible scenarios may arise:

1. No relevant transitions remain: This occurs when the timing constraint is too strict, preventing the event-handler from executing at all, often because other event-handlers consumed the available time budget. In such cases, the violation is due to the system's inability to even begin executing the relevant handler (Callout 7), and further analysis is terminated as there is no trace to analyze.

2. Multiple relevant event-handler traces exist: Since an outgoing topic may be published by more than one incoming topic event-handler, the trace may include transitions from several such handlers. In this situation, we analyze the trace corresponding to the last executed event-handler, as it most likely contributed to the observed violation (Callout 6).

3. A single relevant event-handler trace is available: In this straightforward case, we proceed with analyzing the available event-handler trace to diagnose the timing violation (Callout 6).

The filtered trace data from the above step is then mapped onto the dominator tree of the corresponding Incoming Topic Event-handler, as illustrated in yellow in Figure 4.3. We use the dominator tree as a heuristic because it effectively captures control dependencies within a program. Specifically, it identifies which basic blocks must be executed before others, thereby encoding the essential flow structure of the CFG. A basic block B is said to dominate another block C if every possible path from the entry point to block C passes through block B. This dominance relationship allows us to infer which parts of the code must execute before a particular point can be reached. By analyzing these relationships, we can efficiently identify common ancestors and narrow down the potential root causes of control-flow deviations. In the context of our timing analysis, a typical violation occurs when the system fails to reach a publish statement (represented as a clock reset in the automaton, and shown in blue in Figure 4.3) within the specified time bounds. To diagnose the root cause of this failure, we compute the Lowest Common Ancestor (LCA) between the basic blocks that are part of the trace (yellow) and the block containing the relevant publish statement (blue). This LCA, provided by our analysis tool and highlighted in green in the figure, is flagged as the root cause of the timing violation. The UPPAAL tracing utility uses an internal A* algorithm to generate the execution trace corresponding to the timing violation. Notably, for a given violation, it consistently returns the same single trace. Due to this limitation in the number of available traces, we rely on our

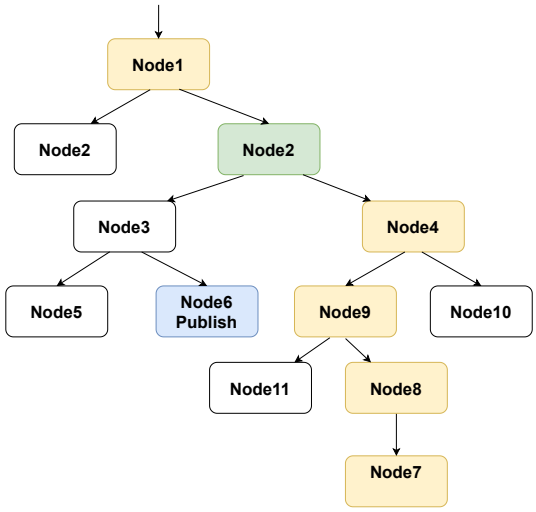


Figure 4.3: Rootcause analysis by mapping filtered trace (yellow) onto the dominator tree of incoming topic event-handler. Root cause (using LCA) highlighted in green.

heuristic to extract meaningful diagnostic information from just one trace. While this approach may not always pinpoint the root cause with perfect accuracy, it serves as a reasonable and effective heuristic in practice. This is because dominator trees reflect control dependencies that directly influence whether certain execution paths, including those reaching critical publish statements, can be taken. By analyzing dominance patterns, we gain valuable insights into how deviations in control flow may prevent the system from meeting its timing guarantees.
Next, we analyze the terminating condition (i.e., branch predicate) within the identified root-cause block to infer why the verifier followed an alternate control path instead of the one leading to the block containing the publish statement. If a block lacks a meaningful branch predicate, it may not be possible to determine a specific reason for the observed deviation. However, when a predicate is present, we generate clear feedback. To map the root-cause block to the corresponding source-level code snippet, we compile the program in debug mode. This allows us to access the debug information necessary for localizing the line number of the error within the source code. The predicate within the conditional statement of the root-cause block is used to derive the reason for failure. To articulate this to the user, we rely on a set of predefined templates. These templates are designed based on empirical observations of common issues encountered in code. Based on the evaluated predicate in the root-cause block, we select a suitable template and customize it with the specific predicate details to generate meaningful, context-aware feedback. Examples of such feedback based on templates will be shown in the results section. The final output presented to the user includes the identified root-cause block,

the corresponding source-code line number, and context-aware feedback derived from the selected template. This approach helps the user understand that the tool is focusing on which part of the code.

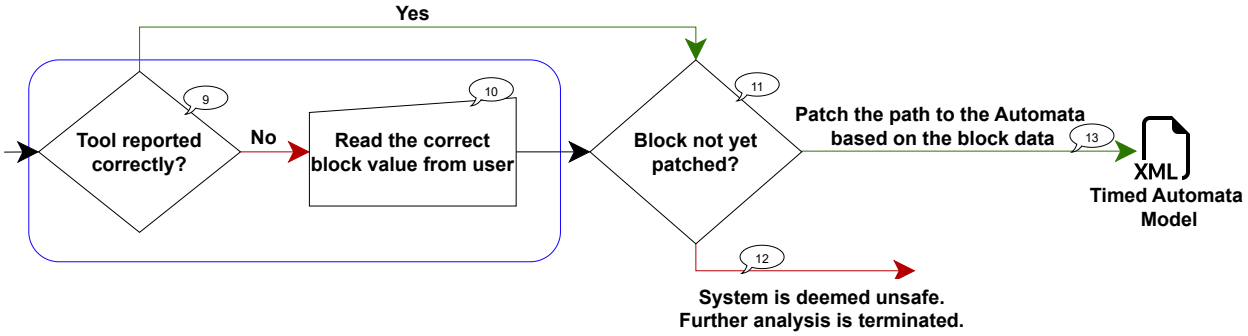


Figure 4.4: Manual Intervention by the user for patching the automata

At this stage, the user may either accept the block identified by our analysis or manually specify an alternative they believe more effectively addresses the potential timing violation based on their analysis. This is the only point in the workflow that requires manual intervention. Once the user confirms the choice of a block, either by approving the one suggested by our analysis or selecting an alternative based on their own reasoning, we first check whether the block has already been patched (Callout 11). This means determining whether the execution path originating from this block has previously been disabled in the TA model. If the block has already been patched, it indicates that no viable alternative paths remain to safely reach the publish statement. This suggests that the timing constraints are too strict, as even the shortest feasible path to the publish statement has already been eliminated. In such cases, the system is declared unsafe, and the analysis is terminated. If the block has not been patched before, we use the block information and trace data to identify the specific execution path that should be disabled. Instead of modifying the source code, we directly update the TA model to exclude this path from further consideration. Patching at the automata level offers significant advantages: it is much faster than altering the source code and re-running the entire model construction process for every change. This makes the iterative verification process more efficient and practical. The patching process is applied iteratively. After each update, the verifier is re-executed to determine whether the patched model still results in a timing violation. This cycle continues until the model reaches a safe state, indicating that no further timing violations are reported (Callout 2). In cases where the verifier correctly follows the logical path leading to the publish statement but still fails due to overly strict timing constraints, we temporarily relax these constraints and re-run the verification as a sanity check. Although such relaxation

may not be feasible in real-world systems where timing requirements are fixed, it helps confirm that the underlying automata structure is functionally correct and that the failure is solely due to tight timing constraints.

4.5 WCET Analysis

In our work, we require Worst-Case Execution Time (WCET) estimates at the basic-block level for a select set of event-handlers within C++ programs. Based on our research, LLVMTA emerged as a promising academic tool for performing static WCET analysis using LLVM Intermediate Representation (IR). However, despite its potential, LLVMTA is not practically feasible for analyzing the ROS-based C++ programs under consideration. The core issue stems from a fundamental mismatch between the assumptions and capabilities of LLVMTA and the complexities inherent in modern ROS applications. LLVMTA is designed to analyze relatively small, statically analyzable programs with well-defined control flows, fixed loop bounds, and predictable execution paths, typically targeting academic benchmarks or simple embedded systems. In contrast, the ROS program in question is built atop a complex, highly abstracted framework involving asynchronous callback mechanisms, function templates (Listing 4.5), dynamic memory management, multithreading, and heavy use of third-party libraries such as TF2, PCL, and Eigen. It also relies significantly on runtime behavior, such as sensor input processing and real-time transform lookups. These factors collectively generate large, intricate LLVM IR, characterized by deeply nested inlined templates, data-dependent branches, and highly dynamic control paths—patterns that static analyzers like LLVMTA struggle to scale to effectively. Further compounding the challenge, LLVMTA requires a custom-patched version of LLVM/Clang to extract IR in a compatible form, which is incompatible with the standard ROS 2 build system (colcon and CMake).

```
initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>("initial_pose",
    rclcpp::QoS(10), initial_pose_callback);

map_pub_ = create_publisher<sensor_msgs::msg::PointCloud2>("map", rclcpp::QoS(10));
```

Listing 4.5: ROS Function Templates to create a publisher and a subscriber based on the message type

As an effort on our part, we attempted to run LLVMTA on the LLVM IR extracted from our ROS-based application, following the procedures described in the previous section. Unfortunately, LLVMTA failed to process the supplied IR correctly, even though the IR was properly generated and structurally valid, indicating that the failure was not due to compatibility issues

but rather due to the tool's inherent limitations.
As a result, we were unable to derive meaningful WCET estimates for the basic blocks within each event-handlers. To enable continued analysis and system modeling, we assigned a placeholder WCET value of one time unit to each basic block in these event-handlers. Since our goal is to build a flexible analysis framework, we support user-supplied WCET values: all block-level timing metadata, including basic block names and WCET values, are exported in an intermediate XML file that feeds into the model construction phase. This XML file is easily editable, allowing users to update it with real WCET values if they are able to obtain such estimates through more suitable WCET analysis tools in the future.

4.6 Model based Approach

UPPAAL offers explicit and mathematically grounded timing verification capabilities that static analysis and simulation fundamentally cannot match. While our current framework does not incorporate true WCET information, its precision would further improve with accurate WCET values from an external oracle or commercial tools such as AbsInt. Unlike static analysis, which suffers from path explosion across branching event-handlers and lacks a semantics for real-valued clocks and nondeterministic timing, UPPAAL uses model-checking techniques such as state-space reduction, slicing, and DBMs to explore all possible temporal behaviors efficiently. Simulation-based approaches are similarly limited because they only observe the execution traces they happen to run. In contrast, UPPAAL's timed-automata framework symbolically explores complete behaviors, provides counterexample traces for timing violations, and has been shown in prior work, including automata-based bug discovery in protocol implementations [15], safety assessments of industrial robots [7, 25], verification of access control to critical zones in drones [27], energy wastages in embedded systems [47] and timing-sensitive studies detecting subtle errors exposed only under slight system variations [30].

Chapter 5

Evaluation

In this chapter, we evaluate the effectiveness of our framework through a structured analysis organized into several parts. We begin by outlining the core objectives of the evaluation, outlining the key aspects we intend to analyze and validate. Next, we describe the experimental setup and selected benchmarks used to assess our approach across diverse ROS packages. To illustrate this stage in detail, we present a representative case study using a ROS package from the benchmark set. The case studies demonstrate the framework's capability to identify and reason about potential timing issues and highlight its end-to-end effectiveness from model construction through verification to actionable feedback.

5.1 Objectives

This section defines the primary goals of our evaluation, which focus on rigorously assessing the practical effectiveness, correctness, and scalability of the proposed model construction and verification pipeline when applied to real-world robotic software systems.

The evaluation is designed with the following core objectives:

- **Diverse Case Study Evaluation:** Apply the framework to a broad set of open-source ROS packages spanning different domains (e.g., localization, mapping, navigation), treating each component in a package as an individual case study. This demonstrates the generalizability of the approach across varied software architectures and operational constraints.

- **Detection and Analysis of Potential Timing Violations:** Identify potential timing-related issues across the evaluated packages and provide a reasoning mechanism that explains their root causes. This objective is critical for showing the framework's diagnostic capabilities and its utility in uncovering latent bugs or performance bottlenecks.

By formally establishing these objectives, we provide a clear foundation for evaluating the framework's robustness, extensibility, and effectiveness in analyzing and verifying timing behavior across a wide spectrum of ROS packages.

5.2 Experimental Setup and Benchmarks

This section outlines the experimental setup, detailing the hardware and software environment, as well as the chosen benchmarks. To conduct a meaningful evaluation, we selected a diverse set of open-source and widely adopted ROS packages from GitHub as benchmarks. The selected packages span multiple domains, including localization, navigation, and mapping, enabling us to assess the versatility and robustness of our framework under varying operational conditions. To guide our selection, we searched for open-source ROS packages on GitHub using tags such as ROS2, ROS, robots, etc, while also ensuring that the packages had at least 150 stars (with the exception of axebot), indicating their popularity and adoption within the ROS community. Additionally, the selected packages should contain at least one publish statement within any of the event-handlers, which is essential for our analysis.

Table 5.1 presents the different ROS packages and their corresponding components on which we evaluate our framework. For all benchmarks, we verify the same property, "E<> *unsafe*", which checks whether there exists a state in which the system reaches the *unsafe* location within the constructed Timed Automata model. Reaching this location indicates that the system failed to receive a message within the specified timing constraint for the outgoing topic of the component under verification. Our experiments were conducted on a server machine equipped with an Intel i9-7920X 64-bit processor (24 cores, 2.90 GHz) and 64 GB of DDR4 RAM. However, the number of cores had minimal impact on performance, as running the models on a 4-core machine yielded similar execution times. To maintain language agnosticism during code analysis, each package was compiled down to LLVM Intermediate Representation (LLVM-IR) using LLVM 14.0.0. This IR then served as the input to our

model construction phase. For verification, we used UPPAAL 5.0.0 to analyze the generated Timed Automata models. The UPPAAL Tracer, included in the toolkit, was utilized to extract trace data from the verification results. Finally, to process and filter the traces generated from the verification phase, we utilize Python 3.8. This filtered trace is then used to provide actionable feedback and patching of the automata.

Table 5.1: Benchmarks used for evaluating the framework.

Package Name	Component Name	Incoming Topics Count	Outgoing Topics Count	Outgoing Topic Name	Model Size (KB)	Violated?	Iterations	Time Taken[a] (secs)
Lidarslam [45] .	ScanMatcher	3	4	current_pose	62	Yes	4	34 secs
				path	55	Yes	4	37 secs
				map_array	55	Yes	5	47 secs
				map	55	Yes	5	44 secs
Aerostack2 [14]	DetectArucoMarkers Behavior	2	1	aruco_pose	206	Yes	5	52 secs
	Scan2occ grid	1	2	map	58	Yes	2	16 secs
				map_filtered	58	Yes	2	16 secs
Axebot[33]	GoToGoal	2	1	omnidirectional_controller /cmd_vel_unstamped	20	No	1	3 secs
Kiss-ICP[57]	OdometryServer	1	1	kiss/odometry	55	Yes	3	40 secs
MRPT-navigation[24]	TPS_Astar_Planner_ Node	1	2	Pub_Topic1	54	Yes	4	55 secs
				Pub_Topic2	54	Yes	4	55 secs
Navigation2[53]	Costmap_2d_markers	1	1	visualization_marker	52	Yes	4	38 secs
	Costmap_2d_cloud	1	2	voxel_marked_cloud	60	Yes	4	38 secs
				voxel_unknown_cloud	60	Yes	4	3 secs
Tello [46]	TelloJoyNode	1	1	cmd_vel	17	No	2	11 secs

[a]This is the total time taken just for verification of the timing property on the model using UPPAAL. It doesn't account for generating the trace file, followed by some manual analysis, which might take from seconds to minutes.

5.3 Case-studies

In this work, we focus on components that contain at least one publish statement within any of their event-handlers, as these are relevant to our analysis. As discussed in Chapter 2, a single package may include multiple components, each responsible for a distinct set of tasks. Each component is treated as an individual case study in our evaluation. Table 5.1 lists the various benchmarks that meet this criterion. We provide a detailed explanation to illustrate the complete operation of the framework, using the ScanMatcher component from the Lidarslam package and its associated outgoing topic, current_pose. For the remaining outgoing topics and other benchmarks, we focus on summarizing the results and highlighting any identified potential timing violations, without repeating the full evaluation workflow.

5.3.1 Lidarslam Package

5.3.1.1 ScanMatcher component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "ScanMatcher" component of the Lidarslam [45] package, as depicted in Figure 5.1. It's formally defined as follows:

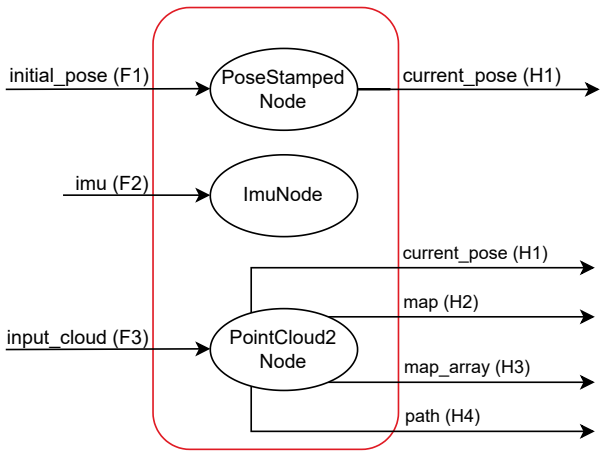


Figure 5.1: ScanMatcher Component of Lidarslam ROS Package

- Set of Incoming Topics, F = {initial_pose, imu, input_cloud}

- Timing assumptions on Incoming Topics, $T = \{2, 3, 5\}$, where time units are assumed to be in abstract time units. If different units, such as seconds or milliseconds, are provided, they can be rescaled and adjusted accordingly. These values, which reflect the incoming message generation timing, are provided by the user and, in our case, are arbitrary and fixed for the purpose of evaluating our timed automata model.
- Set of Outgoing Topics, $H = \{$current_pose, map, map_array, path$\}$
  - Timing Constraint Evaluation on Outgoing Topics, $\tau = \{50, 200, 150, 100\}$, where time units are assumed to be in abstract time units. If different units, such as seconds or milliseconds, are provided, they can be rescaled and adjusted accordingly. These values, which reflect the outgoing message checker timing, are provided by the user and, in our case, are arbitrary and fixed for the purpose of evaluating our timed automata model.
- Event-Handlers of Incoming Topics, $G = \{G_1, G_2, G_3\} = \{$initial_pose_callback, imu_callback, cloud_callback$\}$ where,
  - Block Timing Map, λ_T is a duration of 1 time unit to every block across all event-handlers,
  - Block Publish Map, λ_H of $G_1 = \{$current_pose$\}$, is the outgoing topic associated with some of the blocks in G_1,
  - Block Publish Map, λ_H of $G_2 = \{\phi\}$, no outgoing topics are associated with the blocks of G_2,
  - Block Publish Map, λ_H of $G_3 = \{$current_pose, map, map_array, path$\}$, are the outgoing topics associated with some of the blocks in G_3
- Message limit on Incoming Topics, $\Delta = \{3, 3, 3\}$, used for the purpose of evaluating our timed automata model.

In addition to the above data, user-defined configuration parameters that appear in conditional statements and loops are also included as inputs, as shown in Table 5.2. While the component may contain numerous configuration parameters, we include only those that are relevant and actively used in the model construction process.

Field Name	Field Value
initial_pose_received_	false
set_initial_pose_	true
use_imu_	false
use_min_max_filter_	false
initial_cloud_received_	false
mapping_flag_	true
is_map_updated_	false
publish_tf_	true
use_odom_	false
debug_flag_	false
num_targeted_cloud_	2

Table 5.2: Additional inputs provided for the verification of potential timing violations in the ScanMatcher component of the Lidarslam package

All the boolean fields shown in Table 5.2 represent actual configuration values defined within the package. We construct the Timed Automata model as described in the previous chapters. This is followed by the verification of timing constraints (τ) on outgoing topics (H) using the query E<> *unsafe*. This query checks whether it is possible to reach an unsafe state in the outgoing message checker automaton, which would indicate a failure to publish a message within the user-defined deadline τ signaling a timing violation.
It has to be noted that in our approach, we analyze the system using one set of user-provided timing values and configuration at a time. However, this does not guarantee the absence of timing violations for other input values or configurations. A different set may drive execution through alternate code paths, potentially triggering new timing violations. For this reason, we allow the user to specify these values, which enables them to verify whether, under that specific combination of timing parameters and configuration, the system violates any timing guarantees.

Verification of timing constraint of the outgoing topic "current_pose" :- After generating the Timed Automata model, we proceed to the post-construction

phase (Section 4.4), where we verify whether the system can reach an unsafe state. The analysis is performed per outgoing topic, in this case, we examine whether the "current_pose" outgoing topic can lead to an unsafe state under the user-provided configuration.

- **Iteration-1 :-**
  "Tool Patched :- Message from input_cloud-topic arrived before that of initial_pose-topic, dependency due to initial_pose_received_."
  The above statement is the feedback provided by our tool when we try to verify the automata for the outgoing topic "current_pose" with a timing constraint value $\tau = 50$ time units. We will explain in detail what the feedback means using the simplified Listing 5.1 of the Lidarslam ROS Package.

```
1  initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>("
       initial_pose", rclcpp::QoS(10), initial_pose_callback);

3  input_cloud_sub_ = create_subscription<sensor_msgs::msg::PointCloud2>("
       input_cloud", rclcpp::SensorDataQoS(), cloud_callback);

5  pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose",
       rclcpp::QoS(10));

7  auto initial_pose_callback = [this](const typename geometry_msgs::msg::
       PoseStamped::SharedPtr msg) -> void
8  {
9          ...
10         initial_pose_received_ = true;
11         pose_pub_->publish(...);
12 };

14 auto cloud_callback = [this](const typename sensor_msgs::msg::PointCloud2::
       SharedPtr msg) -> void
15 {
16         if (!initial_pose_received_)
17         {
18                 ...
19                 return;
20         }
21         ...
22         ...
23         pose_pub_->publish(...);
24 }
```

Listing 5.1: C++ Code of Modified Lidarslam ROS Package [45]

The term "Tool Patched" refers to the case where the root-cause block identified by our analysis is correct and used by the user to patch the automaton

in the iteration. Feedback is generated from the root-cause block using predefined templates, in this case: *"Message from XXX-topic arrived before that of YYY-topic, dependency due to ZZZ"*, where XXX and YYY are incoming topics and ZZZ is a configuration parameter.

In the Listing 5.1, initial_pose_callback (Line 7) is the event-handler for the incoming message topic initial_pose, which is linked via a create_subscription call at Line 1. Similarly, cloud_callback (Line 14) is the event-handler for the incoming topic input_cloud, connected through a create_subscription at Line 3. The handler pose_pub_ (Line 5) is responsible for publishing messages to the outgoing topic current_pose, with publish statements appearing at Line 11,23. The configuration in Table 5.2 has initial_pose_received_ set to false initially, which is serving as a guard in cloud_callback. When cloud_callback gets dispatched before initial_pose_callback, cloud_callback returns early (Line 19), skipping the publish statement and preventing message publication on current_pose topic. This violates expected timing constraints, as no message gets published on the current_pose topic.
Ideally, initial_pose_callback should run first, setting initial_pose_received_ to true (Line 10). Since the issue stems from message ordering, the above template is instantiated with the relevant topics and parameters. To fix it, the automaton is patched to disallow the early-return branch, ensuring the publish at Line 23 is reachable. The time taken to obtain the verification result for this iteration, i.e., verifying the formula on the non-patched timed automaton using the UPPAAL model checker (Callout 1 in Figure 4.1) is 9.2 seconds. We then proceed with further verification on the patched automaton.

- **Iteration-2 :-**
  "Tool Patched :- Error due to issue in the message field frame_id of initial_pose-topic."
  The above statement is the feedback provided by our tool when we try to verify the patched automata from iteration-1 for the outgoing topic "current_pose" with a timing constraint value $\tau = 50$ time units.

```
1  initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>("
       initial_pose", rclcpp::QoS(10), initial_pose_callback);

3  pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose",
       rclcpp::QoS(10));
```

```
5  auto initial_pose_callback =[this](const typename geometry_msgs::msg::
       PoseStamped::SharedPtr msg) -> void
6  {
7      if (msg->header.frame_id != global_frame_id_) {
8          ...
9          return;
10     }
11     ...
12     pose_pub_->publish(...);
13 };
```

Listing 5.2: C++ Code of Modified Lidarslam ROS Package [45]

As discussed in the workflow, feedback to the user is generated using predefined templates. In this case, the template selected is: *"Error due to issue in the message field XXX of YYY-topic"*, where XXX is the message field and YYY is the corresponding incoming topic.

In Listing 5.2, initial_pose_callback (Line 5) handles the incoming initial_pose topic, and pose_pub_ handler (Line 3) publishes to current_pose (Line 12). After patching the automaton from Iteration-1, a new issue arose due to an unmodeled condition depending on the runtime messages frame_id value. The model checker could freely take either branch, it chose the branch where the messages frame_id value did not match global_frame_id (Line 7), causing an immediate return and skipping the publish statement (Line 12), resulting in a timing violation.

To prevent this, we further patch the automaton by modifying it to disallow the true branch of the condition at Line 7, essentially assuming that only valid message content will be received. The time taken to obtain the verification result for this iteration, i.e., verifying the formula on the patched timed automaton from iteration-1 using the UPPAAL model checker (Callout 1 in Figure 4.1), is 8.6 seconds. We then proceed with verification on the patched automaton.

- **Iteration-3 :-**

  "Manually Patched :- Error couldn't resolve the condition. Verify Manually."
  The above statement is the feedback provided by our tool when we try to verify the patched automata from iteration-2 for the outgoing topic "current_pose" with a timing constraint value $\tau = 50$ time units.

```
1  initial_pose_sub_ = create_subscription<geometry_msgs::msg::PoseStamped>("
       initial_pose", rclcpp::QoS(10), initial_pose_callback);
```

```
3  pose_pub_ = create_publisher<geometry_msgs::msg::PoseStamped>("current_pose",
       rclcpp::QoS(10));

5  auto initial_pose_callback =[this](const typename geometry_msgs::msg::
       PoseStamped::SharedPtr msg) -> void
6  {
7      if (msg->header.frame_id != global_frame_id_) {
8          ...
9          return;
10     }
11     RCLCPP_INFO(get_logger(), "initial_pose is received");
12     ...
13     pose_pub_->publish(...);
14 };
```

Listing 5.3: C++ Code of Modified Lidarslam ROS Package [45]

The term "Manually Patched" refers to cases where the root-cause block identified by our analysis is incorrect, requiring the user to select a different block to patch the automaton (Callout 9-10, Figure 4.1). The feedback template used is: *"Error couldn't resolve the condition. Verify Manually."* This occurs when the tool cannot interpret a complex predicate or identifies a non-conditional block. In both scenarios, we present the above generic message (along with the block name for manual analysis), indicating our tool's inability to reason about the issue. At this point, the user must manually inspect the reported root-cause block, determine if it is truly responsible for the issue, and if so, provide it as input for the tool to patch and proceed with further verification. Manual inspection revealed Line 11 in Listing 5.3 as the source of the issue. Logging statement forced execution along an exception-handling path, preventing the publish statement (Line 13) from being reached. The presence of a try-catch block in a function within a file causes all functions in that file to adopt similar exception-handling constructs (at the IR level), which occurred in this case. To address this, we manually provide the correct block information as input to the tool (Callout 10 in Figure 4.1). This enables the tool to patch the automaton associated with the logging statement, which is the cause of the timing violation. As a result, the model checker is guided away from the exception path, increasing the likelihood of reaching the publish statement at Line 13 within the timing constraints. The time taken to obtain the verification result for this iteration, i.e., verifying the formula on the patched timed automaton from iteration-3 using the UPPAAL model checker (Callout 1 in Figure 4.1), is 8.8 seconds. We then proceed with verification on the patched

automaton.

- **Iteration-4 :-**
  "Already Patched. System is Unsafe."
  The above statement is the feedback provided by our tool when we try to verify the patched automata from iteration-3 for the outgoing topic "current_pose" with a timing constraint value $\tau = 50$ time units.
  This situation arises when the tool reports a block that has already been patched in a previous iteration (Callout 11 in Figure 4.1). Despite patching multiple blocks in the automaton, the system continues to violate the timing constraints for the outgoing topic. This suggests that the publish statement cannot be reached within the user-defined timing constraint. Even when the automaton is patched to follow the shortest possible path to the publish statement, it still fails to reach it within the allowed time. This strongly indicates that the constraint itself may be overly strict.

Verification of timing constraint of the outgoing topic "path" :- For this outgoing topic, our tool required four iterations but was still unable to fully resolve the timing violations for the user-defined constraint $\tau = 100$ time units. As with the previous outgoing topic, the first issue arose from the order of message arrivals: a message from the input_cloud topic arrived before one from the initial_pose topic, due to dependency on initial_pose_received_. The second issue resulted from the configuration parameter mapping_flag_, which triggered additional time-consuming operations, leading to a timing violation. The third issue was caused by the configuration parameter publish_tf_ for a similar reason, and the fourth issue involved a repeated patch to a block already patched in a prior iteration. These findings indicate that, even when guiding the automaton along the shortest execution path, the publish statement within the event-handler for this outgoing topic cannot be reached within the specified timing constraint.

Verification of timing constraint of the outgoing topic "map_array" :- For this outgoing topic, our tool required five iterations but was still unable to resolve the timing violations for the user-defined constraint $\tau = 150$ time units. As with the previous outgoing topic, the first issue arose from the order of message arrivals, specifically a message from the input_cloud topic arriving before one from the initial_pose topic, creating a dependency on initial_pose_received_. The sec-

ond issue resulted from the configuration parameter mapping_flag_, which triggered additional time-consuming operations. The third issue was caused by the configuration parameter publish_tf_ for a similar reason. The fourth issue stemmed from an unmodeled if condition whose predicate depends on a run-time variable, leading to further time-consuming execution paths. The fifth issue involved a patch to a block already patched in a prior iteration. These findings indicate that, even when guiding the automaton along the shortest execution path, the publish statement within the event-handler for this outgoing topic cannot be reached within the specified timing constraint.

Verification of timing constraint of the outgoing topic "map" :- For this outgoing topic, our tool required five iterations but was still unable to resolve the timing violations for the user-defined constraint $\tau = 200$ time units. As with the previous outgoing topic, the first issue arose from the order of message arrivals: a message from the input_cloud topic arrived before one from the initial_pose topic, creating a dependency on initial_pose_received_. The second issue resulted from the configuration parameter mapping_flag_, which triggered additional time-consuming operations. The third issue was caused by the configuration parameter publish_tf_, for a similar reason. The fourth issue stemmed from an unmodeled if-condition whose predicate depends on a run-time variable, leading to further execution along time-consuming paths. The final issue involved a patch to a block already patched in a prior iteration. These findings indicate that, even when guiding the automaton along the shortest execution path, the publish statement within the event-handler for this outgoing topic cannot be reached within the specified timing constraint.

Verification of timing constraint of the outgoing topic "map" (with block timing values of event-handlers set to 2 time-units) :- In this version, we update the WCET of blocks within the event-handler to 2 time-units and continue the analysis. For this outgoing topic, our tool required three iterations but was still unable to resolve the timing violations for the user-defined constraint $\tau = 200$ time units. As with the previous outgoing topic, the first issue arose from the order of message arrivals: a message from the input_cloud topic arrived before one from the initial_pose topic, creating a dependency on initial_pose_received_. The second issue resulted from the configuration parameter mapping_flag_, which triggered additional time-consuming operations. The final issue involved a patch to a block already patched in a prior iteration. These findings indicate that, even when guiding the

automaton along the shortest execution path, the publish statement within the event-handler for this outgoing topic cannot be reached within the specified timing constraint.

We observe similar issues across multiple outgoing topics because they originate from the same event-handler. During the inlining process, the respective publish statements that are part of different functions are inlined within the event-handler. As a result, the conditional blocks related to mapping_flag_, publish_tf_, and the unmodeled if-condition appear sequentially (with only a few intervening statements), leading to a series of related timing violations during verification. While presenting results for other benchmarks, we do not explicitly indicate the final iteration if it involves patching a block that has already been patched in previous iterations, unless it is necessary to highlight a different issue.

5.3.2 Aerostack2 Package

5.3.2.1 DetectArucoMarkersBehavior Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "DetectArucoMarkersBehavior" component of the Aerostack2 [14] package, as shown in Figure 5.2. We use timing assumptions of $T = 2, 3$ time-units for the incoming topics camera/image_topic and camera/camera_info, respectively, along with a Block Timing Map $\lambda_T = 1$ time-unit for each block within the corresponding event-handlers. Both topics have a message constraint $\Delta = 3$ messages, and the timing constraint for the outgoing topic aruco_pose is set to $\tau = 200$ time-units.

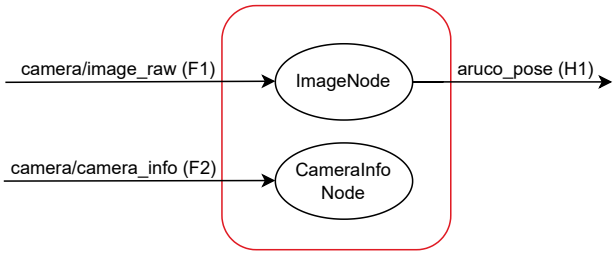


Figure 5.2: DetectArucoMarkersBehavior Component of Aerostack2 ROS Package

Verification of timing constraint of the outgoing topic "aruco_pose" :- For this outgoing topic, our tool required five iterations but was still unable to fully

resolve the timing violations for the user-defined constraint $\tau = 200$ time units. The first issue arose from a statement (function call) that caused execution to follow an exception-handling path, preventing the publish statement from being reached. The second issue was due to the configuration parameter camera_params_available_, which resulted in an early return and skipped the publish statement, causing a timing violation. The third issue stemmed from an unmodeled if-condition whose predicate depends on a run-time variable, allowing the model checker to explore longer execution paths. Finally, two additional issues were caused by logging statements, which similarly forced execution along an exception-handling path, preventing the publish statement from being reached.

5.3.2.2 Scan2occ_grid Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "Scan2occ_grid" component of the Aerostack2 [14] package, as depicted in Figure 5.3. We use a timing assumption of $T = 2$ time-units for the incoming topic sensor_measurements/lidar/scan, along with a Block Timing Map $\lambda_T = 1$ time-unit for each block within its event-handler. The message constraint Δ is set to 3 messages. The timing constraints for the outgoing topics map and map_filtered are set to $\tau = 100$ time units.

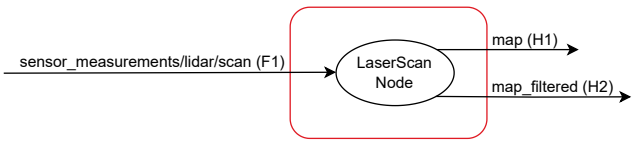


Figure 5.3: Scan2occ_grid Component of Aerostack2 ROS Package

Verification of timing constraint of the outgoing topic "map" and "map_filtered" :- We use a single section, as the publish statements for both outgoing topics appear consecutively within the event-handler. Although there is only one event-handler that executes each time, it required two iterations and still failed to resolve the timing violations for the user-defined constraint $\tau = 100$ time units for both topics. The first issue is caused by a statement (function call) that caused the execution to follow an exception-handling path and prevented the publish statement from being reached. The second issue involved a loop iterating over a message field value. Its bound was set to a value that exceeded the available timing

budget, resulting in execution traces that violated the constraint and highlighted a potential timing violation.

Verification of timing constraint of the outgoing topic "map" and "map_filtered" (with Timing Constraint of 500 time-units) :- In this version, we update the timing constraint for both outgoing topics to 500 time units and continue the analysis. We use a single section, as the publish statements for both outgoing topics appear consecutively within the event-handler. Although there is only one event-handler that executes each time, it requires one iteration to resolve the timing violations for the user-defined constraint $\tau = 500$ time units for both topics. The issue is caused by a statement (function call) that caused the execution to follow an exception-handling path and prevented the publish statement from being reached.

5.3.3 Axebot Package

5.3.3.1 GoToGoal Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "GoToGoal" component of the Axebot [33] package, shown in Figure 5.4. For this case study, we assume timing assumptions of $T = 2, 3$ time-units for the incoming topics /gazebo_ground_truth/odom and ∼/goal, respectively, a Block Timing Map $\lambda_T = 1$ time-unit for each block within both the event-handlers and with message constraint values $\Delta = 3$ messages for both topics. The timing constraint for the outgoing topic /omnidirectional_controller/cmd_vel_unstamped is set to $\tau = 100$.

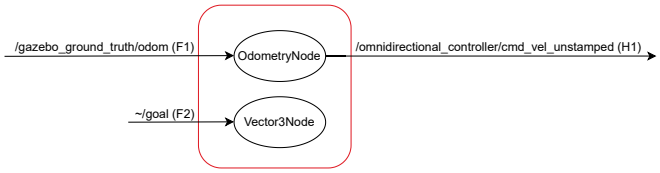


Figure 5.4: GoToGoal Component of Axebot ROS Package

Verification of timing constraint of the outgoing topic "omnidirectional_controller/cmd_vel_unstamped" :- We do not observe any timing violations in this case. The provided timing constraint for the outgoing topic

is sufficiently large for the assumed execution times and message intervals, ensuring that all event-handlers complete their operations without violations. This occurs because the event-handler are very simple, and the message is published even in the case of early returns with appropriate message content.

5.3.4 Kiss-ICP Package

5.3.4.1 OdometryServer Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "OdometryServer" component of the kiss-icp [57] package, as depicted in Figure 5.5. We use timing assumptions T of 2 for the incoming topic pointcloud_topic, along with a Block Timing Map $\lambda_T = 1$ time-unit for each block within its event-handler. Its message constraint value Δ is set to 3 messages. The timing constraint τ for the outgoing topic kiss/odometry is set to 100 time-units.

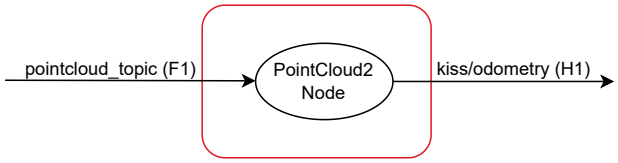


Figure 5.5: OdometryServer Component of kiss-icp ROS Package

Verification of timing constraint of the outgoing topic "kiss/odometry":- . Although there is only a single event-handler that executes each time, it required four iterations and still failed to resolve the timing violations for the given parameters and configuration. The first 3 iterations were caused by statements (function calls) that led the execution to follow an exception-handling path, thereby preventing the publish statement from being reached. The final iteration was caused by a configuration field used in an if-condition, which similarly delayed the model checker from reaching the publish statement within the event-handler.

5.3.5 Mrpt-navigation Package

5.3.5.1 TPS_Astar_Planner_Node Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "TPS_Astar_Planner_Node" component of the mrpt-navigation [24]

package, as depicted in Figure 5.6. We use a timing assumption of $T = 2$ for the incoming topic Sub_Topic1, along with a Block Timing Map $\lambda_T = 1$ time-unit for each block within its event-handler. The message constraint Δ is set to 3 messages. The timing constraints for the outgoing topics Pub_Topic1 and Pub_Topic2 are set to $\tau = 100$. As discussed in Sections 4.1 and 4.2, we were unable to resolve the topic names because their values are built dynamically. Therefore, we assigned them unique placeholder names, which do not affect the results.

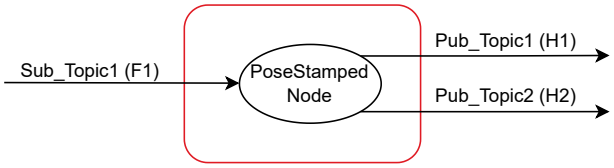


Figure 5.6: TPS_Astar_Planner_Node Component of mrpt-navigation ROS Package

Verification of timing constraint of the outgoing topic "Pub_Topic1" and "Pub_Topic2" :- Although there is only a single event-handler that executes each time, it took four iterations to address the timing violations, and the tool was unable to fully resolve them for the provided timing parameters and configuration. The first 2 issues are caused by statements (function calls) that caused the execution to follow an exception-handling path and prevented the publish statement from being reached. The third issue is from a variable whose value is determined at runtime and is not represented in our automata model, necessitating manual analysis. After supplying the correct block data and patching the automaton, this issue was addressed. The fourth issue was caused by a loop whose bound value increased the execution time of the blocks, resulting in a timing violation.

5.3.6 Navigation2 Package

5.3.6.1 Costmap_2d_cloud Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "Costmap_2d_cloud" component of the Navigation2 [53] package, as depicted in Figure 5.7. We use a timing assumption of $T = 2$ time-units for the incoming topic voxel_grid, a Block Timing Map $\lambda_T = 1$ time-unit for each block within its event-handler, and a message constraint $\Delta = 3$ messages. The timing

constraints for the outgoing topics voxel_marked_cloud and voxel_unknown_cloud are set to $\tau = 100$.

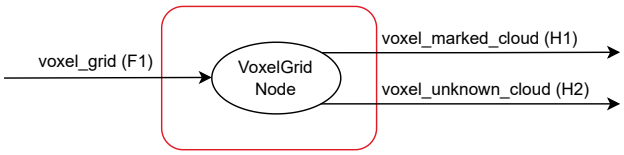


Figure 5.7: Costmap_2d_cloud Component of Navigation2 ROS Package

Verification of timing constraint of the outgoing topic "voxel_marked_cloud" and "voxel_unknown_cloud" :-

We consider a single section for both outgoing topics, as their publish statements appear consecutively within the same event-handler, producing identical output. For these two topics, our tool required four iterations and was still unable to resolve the timing violations for the user-defined constraint $\tau = 100$ time units. The first issue arose from an early return because of a condition dependent on a field of the incoming message. Since our automata model does not explicitly represent message content, the model checker non-deterministically followed the branch corresponding to invalid data, preventing the publish statement from being reached. While a more detailed modeling approach that accounts for message content could avoid such violations, we assume in this work that only valid message content is received. Thus, the automaton is patched to block the invalid branch. The second issue was introduced by a logging statement, which caused the execution to follow an exception-handling path and prevented the publish statement from being reached. The third issue involves a timer-related statement, which similarly chooses the exception path. The fourth issue involved a loop iterating over a message field value. Its bound was set to a value that exceeded the available timing budget, resulting in execution traces that violated the constraint and highlighted a potential timing violation.

5.3.6.2 Costmap_2d_markers Component

In this section, we illustrate potential timing issues that may arise when applying our framework to the "Costmap_2d_markers" component of the Navigation2 [53] package, as depicted in Figure 5.8. We use a timing assumption of $T = 2$ time-units for the incoming topic voxel_grid, a Block Timing Map $\lambda_T = 2$ time-units for

each block within its event-handler, and a message constraint $\Delta = 3$ messages. The timing constraint for the outgoing topic visualization_marker is set to $\tau = 100$.

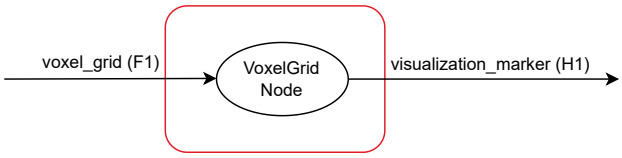


Figure 5.8: Costmap_2d_markers Component of Navigation2 ROS Package

Verification of timing constraint of the outgoing topic "visualization_marker":-

While there is only 1 event-handler that will execute every time, it took four iterations and still failed to resolve timing violations for the provided timing parameters and configuration. We observe issues similar to those in the previous component, as the underlying event-handler code is pretty similar. The first issue arose from an early return because of a condition dependent on a field of the incoming message. Since our automata model does not explicitly represent message content, the model checker non-deterministically followed the branch corresponding to invalid data, preventing the publish statement from being reached. While a more detailed modeling approach that accounts for message content could avoid such violations, we assume in this work that only valid message content is received. Thus, the automaton is patched to block the invalid branch. The second issue was introduced by a logging statement, which caused the execution to follow an exception-handling path and prevented the publish statement from being reached. The third issue involves a timer-related statement, which similarly chooses the exception path. The fourth issue involved a loop iterating over a message field value. Its bound was set to a value that exceeded the available timing budget, resulting in execution traces that violated the constraint and highlighted a potential timing violation.

5.3.7 Tello Package

5.3.7.1 TelloJoyNode Component

In this section, we demonstrate potential timing issues that may arise when applying our framework to the "TelloJoyNode" component of the Tello [46] package, shown in Figure 5.9. For this case study, we assume a timing interval $T = 2$ time-units for the incoming topic joy, a Block Timing Map $\lambda_T = 1$ time-unit for each block

within its event-handler, a message constraint $\Delta = 3$ messages for joy, and a timing constraint $\tau = 100$ time-units for the outgoing topic cmd_vel.

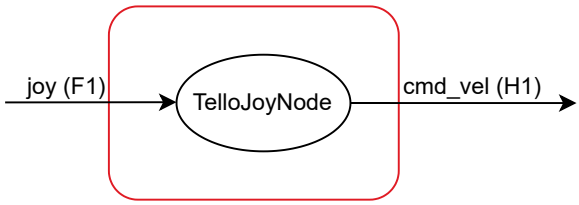


Figure 5.9: TelloJoyNode Component of Tello ROS Package

Verification of timing constraint of the outgoing topic "cmd_vel" :-

Although there is only a single event-handler that executes each time, resolving the timing violations required two iterations. In both cases, the violations were caused by conditions on message fields within an if statement, which prevented the model checker from reaching the publish statement. Since our automata model does not explicitly represent message content, such violations arise. A more accurate modeling approach that incorporates message content would eliminate these spurious violations for the given configuration.

Verification of timing constraint of the outgoing topic "cmd_vel" (with block timing values of event-handlers set to 2 time-units) :-

In this version, we update the WCET of blocks within the event-handler to 2 time-units and continue the analysis. We get the same result as above. It required two iterations. In both cases, the violations were caused by conditions on message fields within an if statement, which prevented the model checker from reaching the publish statement.

Chapter 6

Extensions

6.1 Single-Threaded Executor

In this section, we describe how our work can be extended to support the Single-Threaded Executor. In this executor variant, publishers and subscribers can be dynamically added or removed after the initialization phase. To model this behavior, users must specify two additional fields: the time at which they are added and the time at which they are removed. With this information, the Generator Automata (Section 3.3.2) can be adapted to incorporate the dynamic behavior. Additions can be represented by delaying the activation of message generation using a sleep timer for the specified duration, while removals can be modeled by transitioning to a finish location, after which the automaton stops enqueuing messages for that topic.

6.2 Multi-Threaded Executor

In this section, we describe how our framework can be extended to support a Multi-Threaded Executor. Building upon the modifications introduced in the previous section, this executor requires the user to explicitly specify both the number of threads available for concurrent event-handler execution and the mapping of each event-handler to a specific thread. The number of threads determines how many Dispatcher Automata (Section 3.3.3) are instantiated, with each dispatcher maintaining its own queue for event-handlers awaiting execution. To ensure correct dispatching, the grouping information provided by the user is leveraged by the Incoming Message Generator to enqueue each event-handler into the queue corresponding to its assigned dispatcher.

Chapter 7

Related Work

Dust et al.[11] address the need for an automated method to verify the correctness of ROS applications, with particular emphasis on two critical aspects: event-handler execution latencies[1] and event-handler queue overflows. Their work demonstrates that different versions of the single-threaded ROS executor exhibit subtle semantic variations, which can lead to unexpected behavior and hard-to-detect bugs. To address these challenges, they propose a formal verification approach using the UPPAAL model checker. At the core of their approach is a set of reusable Timed Automata (TA) templates that model the behavior of various ROS components along with multiple versions of the single-threaded executor. By composing these templates, they construct a holistic system-level model that enables formal verification to ensure compliance with specified properties. The model is validated by comparing its generated execution traces against logs obtained from a real ROS system. Like our work, Dust et al. leverage Timed Automata and the UPPAAL toolchain to enable formal verification of the correctness of ROS applications through exhaustive state-space exploration. They focus on properties such as event-handler execution latency, event-handler queue overflows, deadlocks, and scheduling issues, whereas our work emphasizes timing violations arising from the internal execution of event-handlers. The primary distinction lies in the modeling focus and level of granularity. Dust et al. concentrate on the ROS internals, constructing a high-fidelity model of the executor to detect system-level errors such as queue overflows and scheduling-induced latency spikes. Their modeling of application event-handlers is intentionally coarse-grained: each event-handler is abstracted as a single worst-case execution time (WCET) number. In contrast, our work abstracts away the executor's internal scheduling mechanics and develops a fine-grained model of the application's event-handlers. Specifically, we capture internal control flow, including loops and

[1]Measured as the time from the arrival of an event in the executor's queue until the completion of its corresponding event-handler execution.

configuration-dependent conditional branches, enabling us to pinpoint application-level logic that can give rise to timing violations. In this way, our approach complements that of Dust et al. by shifting the focus from ROS behavior to the detailed execution structure of application code.

Wilson et al. [59] address the challenge of verifying the safety of latency-aware cyber-physical systems, arguing that traditional analysis methods are often overly pessimistic. These traditional approaches typically decouple the analysis of continuous physical dynamics from discrete computational constraints and rely on worst-case scenarios, which result in overly strict timing bounds that are difficult to satisfy in practice or that restrict the system to very simple computation modules. To overcome this limitation, they propose SOTERIA, a formal, digital-twin-enabled framework that verifies system safety within a specific operating environment rather than under general assumptions. Their methodology follows a two-stage process. In the first phase, a formal timing model of the ROS executor and the application workload is constructed to derive event-handler latencies. Execution times of the event-handlers are measured empirically by running the application, and these measurements are provided as inputs to the timing model, from which the corresponding latencies are obtained. In the second phase, the derived latency values are integrated into a separate model that simulates the physical system within the target environment, enabling verification of whether the system remains safe under its worst-case computational performance. Both Wilson et al. and our work leverage formal modeling to analyze the impact of timing on system safety. Wilson et al. emphasize holistic CPS safety assurance by evaluating whether system latencies are physically acceptable for a particular environment. They link latency to physical safety and simulate the system within a digital twin, explicitly assessing whether the obtained latency values are suitable for the target environment. Their ROS application model is coarse-grained, using a single WCET value per event-handler, and their ROS executor model is less detailed compared to the fine-grained executor model of Dust et al., though it is still more detailed than the executor model that we use in this work. In contrast, our framework does not run or simulate the application. Instead, it models the application event-handlers with fine-grained detail, providing actionable feedback on the root causes of timing violations.

Backeman and Seceleanu [7] extend the formal analysis of ROS applications by incorporating non-determinism using Stochastic Timed Automata (STA). They observe that traditional deterministic WCET analysis is often overly pessimistic, as tasks rarely reach their absolute worst-case execution time, and some systems exhibit inherently probabilistic behaviors—for example, a sensor that only sends data when its value changes. Deterministic models cannot realistically capture these uncertainties. To address this, they model task execution times as

variables bounded between a best-case execution time (BCET = WCET/2) and a worst-case execution time (WCET), and introduce probabilistic data generators that trigger events with a certain probability, providing more realistic performance insights and failure probabilities. Rather than producing a binary pass/fail outcome, their approach calculates the probability of a timing violation for a given system configuration, offering statistical guarantees of performance under uncertainty. Similar to previous works, their model uses a single WCET value per event-handler and adopts a coarse-grained design for the ROS executor. While both their work and ours embrace non-determinism as a core aspect of system behavior, the application and granularity differ: Backeman and Seceleanu focus on system-level non-determinism, such as variable task execution times and probabilistic event arrivals, to evaluate the likelihood of timing violations under uncertain conditions. In contrast, our framework applies non-determinism at the level of internal event-handler logic, modeling conditional branching and behavior to identify timing violations and provide actionable feedback rather than estimating probabilistic outcomes. This distinction enables our approach to pinpoint the root causes of timing violations, complementing the probabilistic perspective offered by STA based analysis.

While formal methods focus on exhaustive verification, an orthogonal line of work empirically studies the physical impact of timing violations. For instance, Li et al. [28] investigate the real-world physical impact of performance interference in cyber-physical systems (CPS). They note that simply maximizing delays through a Denial-of-Service (DoS) attack is often ineffective, as many systems incorporate fail-safe mechanisms that detect such covert attacks and shut down safely. The real threat lies in stealthy, fine-grained delays that degrade control performance without triggering these fail-safes. To address this, they present TimeTrap, an automated framework for analyzing the end-to-end impact of performance interference on CPS platforms. TimeTrap identifies harmful task execution patterns caused by software implementation flaws that may lead to timing issues and control degradation, and it automatically synthesizes adversarial aggressor workloads to expose these patterns. Rather than formally verifying all possible timing behaviors, TimeTrap takes an adversarial, empirical approach to discover and exploit timing violations. It first uses software fault injection to introduce small delays at various points in the target ROS application, identifying specific timing patterns that cause adverse physical outcomes, such as a robot arm collision. Once a vulnerable temporal displacement is identified, TimeTrap profiles the application's sensitivity to resource contention, such as cache or I/O interference, and constructs a targeted adversarial workload designed to reproduce the exact delay pattern, thereby confirming that the vulnerability can be triggered in a real-world scenario. While both our work and Li et al. focus on timing violations in ROS applications that can lead to system failures, the approaches are fundamentally different. TimeTrap is an

empirical tool that observes the physical effects of induced delays to find exploitable vulnerabilities in a running system. In contrast, our framework employs formal methods to statically verify the internal logic of an application's source code. Our goal is to exhaustively analyze all possible execution paths within the model to identify timing violations, rather than actively triggering them on a physical system.

Chapter 8

Conclusion

This thesis presents a framework for analyzing and ensuring timing correctness in event-driven applications, with a particular focus on the publish–subscribe communication model commonly used in modern robotics. The primary challenge addressed is guaranteeing that such systems, under given configurations, comply with their real-time timing constraints. Our approach constructs a network of Timed Automata to model the behavior of publish–subscribe systems. This involves analyzing the source code to extract key components, including topics, event-handlers, and internal control-flow logic, while also incorporating user-defined timing and configuration parameters. The resulting model is verified using a model checker, which explores all possible execution paths to identify potential timing violations. Evaluation across a diverse set of real-world ROS packages demonstrates the practical utility of the framework. We observed various timing issues, including event order dependencies, errors related to message fields, misconfigured parameters, timing budget exhaustion due to loops, and complex control flow within event-handlers. While some of these issues may be spurious and could be mitigated through improved modeling, they highlight the practical challenges of ensuring timing correctness in event-driven systems. Also, we vary the execution time of blocks of event-handlers and timing constraint values of outgoing topics to see how it affects the overall results for a few benchmarks. A key contribution of this work lies not only in detecting such violations but also in providing actionable feedback to developers, such as adjusting configuration values, reordering event-handler priorities, or relaxing overly strict timing constraints, to help ensure a safe and reliable system.

Chapter 9

Responses to reported issues

To validate the impact of our tool, we contacted five individuals (four package developers and one user) and received two responses. The questions asked and their corresponding replies are summarized below. Response1 is from the developer of the lidarslam package, while Response2 is from the developer of the Aerostack2 package.

Question1

> One common challenge in ROS package development is ensuring that published messages reach subscribers within the required timing constraints otherwise it might lead to crashes in the robotic system. Typically, publishers are driven by timer callbacks, but I noticed that in your implementation, messages are also published from within subscriber callbacks. Could you share how you ensure that subscribers on the receiving end receive these published messages within the expected time period in such a setup? Or is it the case that timing constraints are not a primary consideration when developing this package?

Response1

To be honest, I wrote this package when I was still a beginner with SLAM, so there are some problems with it... At the time of development, I did not really consider timing constraints.The main purpose of this package was offline map creation through post-processing, and I didn't really expect it to be embedded in a real-time robotic system. The reason for publishing inside sensor callbacks was simply because I followed some existing open-source implementations I was referencing back then, not because of a strong design decision.

Response2

Regarding question 1, we generally avoid publishing redundant information unless it is necessary to ensure continuity (such as with the TF tree, control signals, or estimation data). For ArUco detections, we only want to publish valid detections—so if there are none, we do not publish anything. As for the map, we wait until it is properly created before publishing, which means all cells must be traversed first.

Question2

> In specific part of the code if certain parameters are invalid, the function performs an early return without publishing. Do you consider skipping publication in such cases as a valid issue for missing publication?

Response1

I honestly don't remember why I implemented that early return... It might not even be necessary. I wrote it about five years ago, so I'm not entirely sure.

Response2

No comment

Question3

> In specific part of the code involving loop, the condition depends on a value computed from transformations on the input message. Do you have an rough estimate or upper bound on how many iterations this loop can take? Since the publish statement occurs after this loop, how do you ensure that the message is still published within the required timing constraints? or do you just accept that the message publication can be delayed?

Response1

I don't have a rough estimate or upper bound for that loop, but this kind of processing is quite common, so I assumed it wouldn't be a big problem.

Response2

No comment

Question4

> Are you familiar with the QoS deadline feature in ROS, which can be used to detect missed or delayed message publications? If so, I was curious why it wasn't applied in this package. Is there a particular design decision behind not using it?

Response1

At the time I developed this package, there weren't many well-prepared resources or documentation for ROS2 QoS features, so I simply didn't take them into account.

Response2

We are not very familiar with the QoS deadline feature, so we haven't explored how it could be used to improve system reliability. It would be great to consider this as a potential improvement in the future.

Bibliography

[1] Ably. 10 Event-Driven Architecture Examples: Real-World Use Cases. *https://ably.com/topic/event-driven-architecture*. 1

[2] Adya, Atul and Howell, Jon and Theimer, Marvin and Bolosky, William J and Douceur, John R. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002. 1

[3] Antonić, Aleksandar and Marjanović, Martina and Skočir, Pavle and Žarko, Ivana Podnar. Comparison of the CUPUS middleware and MQTT protocol for smart city services. In *2015 13th International Conference on Telecommunications (ConTEL)*, pages 1–8, 2015. doi: 10.1109/ConTEL.2015.7231225. 12

[4] Autoware Foundation. Autoware. *https://autoware.org/*. 12, 38

[5] AWS. Pub-Sub Systems in Amazon Web Services. *https://aws.amazon.com/what-is/pub-sub-messaging/*. 13

[6] Azure. Pub-Sub Systems in Microsoft Cloud. *https://learn.microsoft.com/en-us/azure/azure-web-pubsub/*. 13

[7] Backeman, Peter and Seceleanu, Cristina. Verifying ROS-Based Applications Using Timed and Stochastic Timed Automata. In *Rebeca for Actor Analysis in Action: Essays Dedicated to Marjan Sirjani on the Occasion of Her 60th Birthday*, pages 295–325. Springer, 2025. 80, 104

[8] Barry, Richard and others. FreeRTOS. *Internet, Oct*, 4:18, 2008. 1

[9] Dabek, Frank and Zeldovich, Nickolai and Kaashoek, Frans and Mazieres, David and Morris, Robert. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189, 2002. 1

[10] Desai, Ankush and Gupta, Vivek and Jackson, Ethan and Qadeer, Shaz and Rajamani, Sriram and Zufferey, Damien. P: safe asynchronous Event-driven programming. *ACM SIGPLAN Notices*, 48(6):321–332, 2013. 1

[11] Dust, Lukas and Gu, Rong and Seceleanu, Cristina and Ekström, Mikael and Mubeen, Saad. Pattern-based verification of ROS 2 applications using UPPAAL. *International Journal on Software Tools for Technology Transfer*, pages 1–20, 2025. 103

[12] Eckhardt, Andreas and Müller, Sebastian and Leurs, Ludwig. An evaluation of the applicability of OPC UA Publish Subscribe on factory automation use cases. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 1071–1074. IEEE, 2018. 12

[13] Estuary. 10 Event-driven Architecture Examples: Real-World Use Cases. *https://estuary.dev/event-driven-architecture-examples/*. 1

[14] Fernandez-Cortizas, Miguel and Molina, Martin and Arias-Perez, Pedro and Perez-Segui, Rafael and Perez-Saura, David and Campoy, Pascual. Aerostack2: A software framework for developing multi-robot aerial systems. *arXiv preprint arXiv:2303.18237*, 2023. 46, 48, 84, 94, 95

[15] Fiterau-Brostean, Paul and Jonsson, Bengt and Sagonas, Konstantinos and Tåquist, Fredrik. Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations. 2023. 80

[16] Gog, Ionel and Kalra, Sukrit and Schafhalter, Peter and Gonzalez, Joseph E and Stoica, Ion. D3: a dynamic deadline-driven approach for building autonomous vehicles. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 453–471, 2022. 13

[17] Google. Pub-Sub Systems in Google Cloud. *https://cloud.google.com/pubsub/docs/pubsub-basics*. 13

[18] Happ, Daniel and Karowski, Niels and Menzel, Thomas and Handziski, Vlado and Wolisz, Adam. Meeting IoT platform requirements with open pub/sub solutions. *Annals of Telecommunications*, 72:41–52, 2017. 12

[19] Harun Teper and et al. End-to-end timing analysis in ROS2. *IEEE Real-Time Systems Symposium (RTSS)*, 2022. 13

[20] Himanshu Gupta. Real-Time Use Cases in Capital Markets Part 3: Pre-Trade Order Processing. *https://solace.com/blog/capital-markets-use-case-pre-trade-order-processing/*, 2022. 12

[21] IBM. Pub-Sub Systems in IBM Cloud. *https://www.ibm.com/docs/en/integration-bus/10.0?topic=applications-publishsubscribe-overview*. 13

[22] Jaseemuddin, Muhammad and Alam, Abrar and Gawhar, Nuzhat. MQTT pub-sub service for connected vehicles. In *2021 IEEE 18th International Conference on Smart Communities: Improving Quality of Life Using ICT, IoT and AI (HONET)*, pages 167–172. IEEE, 2021. 12

[23] Jo, Wonse and Kim, Jaeeun and Wang, Ruiqi and Pan, Jeremy and Senthilkumaran, Revanth Krishna and Min, Byung-Cheol. SMARTmBOT: A ROS2-based low-cost and open-source mobile robot platform. *arXiv preprint arXiv:2203.08903*, 2022. 13

[24] Jose-Luis Blanco-Claraco et al. mrpt-navigation Package. *https://github.com/mrpt-ros-pkg/mrpt_navigation*, 2024. 84, 97

[25] Kang, Eun-Young and Choudhary, Gaurav and Campusano, Miguel and Kühnrich, Morten and Pedersen, Anders. Enhancing Dependability of Industrial Robots: Security and Safety Assessments Based on Model-Driven Engineering. pages 106–113, 2024. 80

[26] KG Larsen and et al. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1997. 16, 25

[27] Krichen, Moez. Timed automata-based strategy for controlling drone access to critical zones: A UPPAAL modeling approach. *Electronics*, 13(13):2609, 2024. 80

[28] Li, Ao and Wang, Jinwen and Baruah, Sanjoy and Sinopoli, Bruno and Zhang, Ning. An empirical study of performance interference: Timing violation patterns and impacts. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'24)*. IEEE Computer Society Press, 2024. 13, 105

[29] Lukas Johannes Dust and et al. Experimental evaluation of callback behavior in ROS 2 executors. *IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023. 13

[30] Lukas Johannes Dust and et al. Pattern-Based Verification of ROS 2 Nodes Using UPPAAL. *International Conference on Formal Methods for Industrial Critical Systems. Cham: Springer Nature Switzerland*, 2023. 13, 80

[31] Macenski, Steven and Martin, Francisco and White, Ruffin and Ginés Clavero, Jonatan. The Marathon 2: A Navigation System. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020. 13, 38

[32] Marius Mikučionis. Special Nested Property UPPAAL. *https://stackoverflow.com/a/63988340/9913310*. 25

[33] Mateus S. Meneses. Axebot. *https://github.com/mateusmenezes95/axebot/*, 2024. 84, 96

[34] Open Robotics. Executors in ROS2. *https://docs.ros.org/en/humble/Concepts/Intermediate/About-Executors.html*. 35

[35] OpenRobotics. ROS2 Control. *https://control.ros.org/rolling/index.html*, . 13

[36] OpenRobotics. ROS 2: Documentation. *https://docs.ros.org/en/humble*, . 13, 16, 33

[37] OpenRobotics. Pub-Sub Systems in ROS2. *https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Cpp-Publisher-And-Subscriber.html*, . 13

[38] OpenRobotics. Understanding actions ROS2. *https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html*, . 33

[39] OpenRobotics. ROS 2: Quality Of Service settings. *https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html*, . 33

[40] OpenRobotics. Understanding services ROS2. *https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html*, . 33

[41] PickNik Robotics. MoveIt 2. *https://moveit.picknik.ai/main/index.html*, 2024. 13, 38

[42] Rajeev Alur and et al. A theory of Timed Automata. *Theoretical computer science 126.2*, 1994. 16

[43] Rathee, Geetanjali and Kerrache, Chaker Abdelaziz and Calafate, Carlos T. An Ambient Intelligence approach to provide secure and trusted Pub/Sub messaging systems in IoT environments. *Computer Networks*, 218:109401, 2022. 12

[44] Roy, Rajarshi and Neider, Daniel. Inferring properties in computation tree logic. *arXiv preprint arXiv:2310.13778*, 2023. 19

[45] Ryohei Sasaki. Lidarslam Package. *https://github.com/rsasaki0109/lidarslam_ros2*, 2024. vi, 13, 38, 40, 42, 46, 69, 71, 73, 84, 85, 88, 90, 91

[46] Ryze Robotics. Tello Package. *https://github.com/clydemcqueen/tello_ros*, 2018. 84, 100

[47] Shabani, Peyman and Ghassemi, Fatemeh and Kargahi, Mehdi. Estimating Energy Wastage in Embedded Systems Using Model Checking of Timed Automata. pages 1–8, 2024. 80

[48] Shrikanth Rajgopalan. How PubSub+ Platform Distributes Trade Processing Events in Post-Trade Systems. *https://solace.com/blog/pubsub-platform-distributes-trade-processing-events/*, 2021. 12

[49] Slomp, GH. Reducing UPPAAL models through control flow analysis. Master's thesis, University of Twente, 2010. 26, 28

[50] Solace. Results from the Industry's First Event-driven Architecture Survey. *https://solace.com/event-driven-architecture-statistics/*, 2021. 1

[51] Solace. Patterns in EDA. *https://docs.solace.com/Get-Started/message-exchange-patterns.htm*, 2021. 1

[52] Steve Buchko. Building a Better "Connected Car" with MQTT 5.0. *https://solace.com/blog/connected-car-with-mqtt-5-0/*, 2021. 12

[53] Steve Macenski et al. Navigation2 Package. *https://github.com/ros-navigation/navigation2*, 2024. 47, 49, 50, 84, 98, 99

[54] Tian, Yuan and Song, Biao and Hassan, Mohammad Mehedi and Huh, Eui Nam. The distributed pub-sub system with privacy protection in smart home environments. *Information (Japan)*, 16(1 B):663–668, 2013. 12

[55] Tilkov, Stefan and Vinoski, Steve. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, 2010. 1

[56] Tobias Blaß and et al. A ROS 2 response-time analysis exploiting starvation freedom and execution-time variance. *IEEE Real-Time Systems Symposium (RTSS)*, 2021. 13

[57] Vizzo, Ignacio and Guadagnino, Tiziano and Mersch, Benedikt and Wiesmann, Louis and Behley, Jens and Stachniss, Cyrill. Kiss-icp: In defense of point-to-point icp–simple, accurate, and robust registration if done the right way. *IEEE Robotics and Automation Letters*, 8(2):1029–1036, 2023. 84, 97

[58] Wikipedia. Pub-Sub Systems in GUI. *https://en.wikipedia.org/wiki/Publish-subscribe_pattern#Topologies*. 13

[59] Wilson, Kurt and Arafat, Abdullah Al and Baugh, John and Yu, Ruozhou and Liu, Xue and Guo, Zhishan. Soteria: A formal digital-twin-enabled framework for safety-assurance of latency-aware cyber-physical systems. In *Proceedings of the 28th ACM International Conference on Hybrid Systems: Computation and Control*, pages 1–11, 2025. 104

[60] Xu Jiang and et al. Real-time scheduling and analysis of processing chains on multi-threaded executor in ROS 2. *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2022. 13