Timing is Key - A WCET Analysis of RTLola

Saarland University

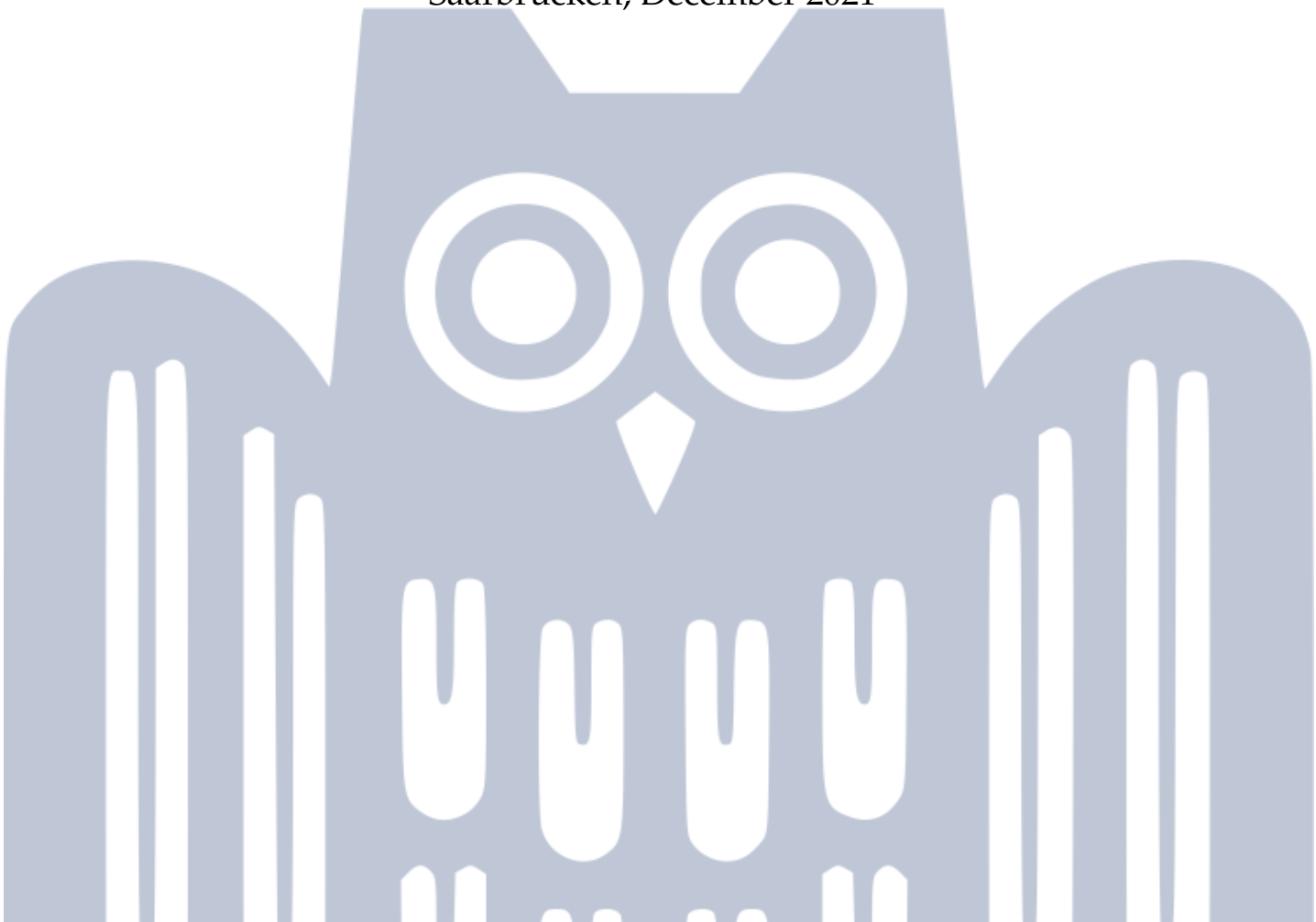
Department of Computer Science

Bachelor's Thesis

submitted by

Julia Laichner

Saarbrücken, December 2021



Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jan Baumeister, M.Sc.
Reviewer:	Prof. Bernd Finkbeiner, Ph.D.
	Rayna Dimitrova, Ph.D.
Submission:	30 December, 2021

Abstract

Cyber-physical systems (CPS) are safety-critical, and monitoring these systems is essential to avoid dangerous situations. Therefore, the time needed between two updates in CPS must be below a specified upper bound to ensure that these systems do not produce any flaws.

RTLola is a stream-based specification language for real-time constraints and is used as a monitoring language for cyber-physical systems. Worst-case execution time (WCET) analyses determine the upper bound of all possible executions on specific hardware. So far, WCET tools exist for many programming languages like Java, C++, and many more. Estimating the upper bound of one update of the monitor with such an existing tool leads to imprecise results as none of them consider the specification being monitored.

This thesis presents two approaches, which aim to compute an upper bound of the monitor using the RTLola specification and its underlying dependencies of the streams. The first approach focuses on the front-end of RTLola , and the second approach focuses on the analysis of the assembly code of the RTLola interpreter using the given specification.

Acknowledgements

First of all, I want to thank Prof. Bernd Finkbeiner for supporting my work and giving me the chance to write this thesis.

Moreover, I am grateful for the help of my advisor Jan Baumeister, who always supported me during my thesis. He always had an open ear and encouraged me to never give up, especially during the writing process.

Further, I thank Prof. Finkbeiner and Rayna Dimitrova for reviewing this thesis.

Last but not least, a special thanks to my family and friends for their support throughout the process of this thesis.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 30 December, 2021

Contents

Chapter 1

Introduction

Cyber-physical systems (CPS) interact with their environment, i.e., by gathering data through sensors. CPS are found in different aspects of life, reaching from unmanned aircraft systems to medical implants. They must guarantee specific correctness properties to avoid critical situations. These critical situations may include physical damage or injuries of humans in case of an accident with the CPS. Such cases must be prevented under any circumstances, especially as people's lives are at stake. Even if no one is harmed, the costs of the damage of the CPS are not negligible. In the case of a drone, the necessary properties are to avoid any crashes and guarantee that while the drone is flying, the power source is never empty. These requirements cover the desired correctness and safety of CPS. On the one hand, the drone must always correctly handle new input, e.g., from its sensors, and not change directions suddenly.

On the other hand, if the CPS notices an obstacle, it should initiate a countermeasure protocol that keeps the aircraft on the planned trajectory but flies around the obstacle. In addition to the safety and correctness requirements of CPS, another desirable requirement is the timing aspect. To illustrate this aspect, consider the following example of a drone interacting with its environment on a mission to get to a destination. The drone has to detect obstacles on its trajectory. If an obstacle is found in the distance of two meters, the drone enters a critical state. Afterward, the drone has to initiate a countermeasure protocol in time depending on the current velocity and distance of the drone to avoid any crashes. Suppose runtime monitoring does not process the data leading to the critical state fast enough. In that case, there may not be enough time for the countermeasure resulting in a crash with the obstacle. Thus, CPS are safety-critical and time-critical since seconds can decide whether the drone is involved in an accident or not if the monitor computes the output of the input data too slow. Therefore, an upper bound of the execution of the monitor receiving inputs is desired. This bound allows the CPS to incorporate it into the computation.

To ensure safety-critical properties, a possible approach is testing. Test cases for the CPS only cover a subset of all possible executions. Especially in a real-time setting, every possible time and place configuration - the arrival of inputs from the sensors and the drone's location at a particular time point - cannot be covered by testing. Therefore, testing does not guarantee the safety of a CPS since the created test cases may not cover a critical situation leading to an accident.

Another method of covering the strict guarantees of CPS is model checking, a static verification method analyzing the system before its execution. In this approach, a system model is generated, and every possible execution run is checked against the system's safety properties. This problem is specified as the language inclusion test. If all execution paths of the system satisfy the correctness properties, the system guarantees all correctness requirements. Since the complexity of CPS increases drastically due to its real-time behavior, the complexity of generating such a model also increases drastically. Model-checking suffers from the state-space explosion problem caused by easily exceeding the computational memory amount needed to build the model for the most complex practical, relevant systems. Although this approach offers highly desirable safety and correctness guarantees, it is often not applicable in practice.

Considering the problems of the previous approaches, we search for a method fulfilling the correctness, safety, and timing requirements desired for a CPS, which is also applicable in practice. Runtime monitoring combines these properties and provides the guarantee of a reliable system. Monitoring is unlike model checking, a dynamic verification method where the system is analyzed during its runtime. Like model checking, runtime monitoring checks if the CPS satisfies the given specification of system requirements. The monitor checks only one execution at once and analyzes the given specification for its correctness. If violations occur, the monitor outputs feedback, either interpreted by the CPS or a human outside the CPS by initiating a countermeasure. This analysis is also referred to as the membership test. The feedback output may be boolean or quantitive, where the latter helps to improve the system.

To specify the correctness properties of the system, a specification language is needed, like temporal logics. All temporal logics share a similar structure. They express the state of the system by atomic propositions as well as connecting formulas by boolean and temporal operators. Using temporal logics for runtime verification is limited in the feedback of the monitor as it only decides if the CPS fulfills the specification.

In the setting of CPS, boolean feedback, stating whether a formula is satisfied or not, is not always enough. We aim for quantitive feedback. Consider the following example where two drones detect obstacles and change their trajectory. In more detail, a formal guarantee of the two drones is that they should keep a distance of three meters to any obstacles. If a drone violates this property, the alarm being raised should contain the boolean feedback and state that the distance to the obstacle is, for example, two meters at the time the alarm is raised.

This thesis emphasizes the quantitive and stream-based specification language RTLola. The incoming data is interpreted as input streams, building the output streams' foundation, representing the system's desired properties. Finally, the trigger streams raise the alarm if the specification is violated or not. So far, RTLola already offers the desired correctness and safety guarantees.

Further, an upper bound of the maximal memory consumption is computed, which benefits CPS running on devices with limited resources. The computation of the memory-bound provides a static check for the monitor meaning prior to its execution. The timing guarantees have yet to be implemented and included into the RTLola framework. This thesis presents an approach to compute the upper bound by a WCET analysis of the execution of an RTLola monitor. We present two approaches for static checks. First, the RTLola specification and its underlying intermediate representation are analyzed. The second approach maps each function to its assembly code of the RTLola interpreter and counts the instructions given a specification.

1.1 Motivation

Analyzing the code of the monitor itself with an RTLola specification narrows down which code in the monitor is considered for the WCET. Consequently, we know which parts of the code are executed and which parts are ruled out by the assumptions of the specification. The approaches presented in this thesis differ from other WCET tools like [1, 2, 3, 4]. Even tools with annotations like the loop bound and maximal recursion depth like the aiT tool [1] have to choose the same bound for all functions. They cannot decide that for a given specification, we only have synchronous lookups. Therefore, when evaluating the expression, we would only match the stream access case and neglect other cases for the WCET.

Example 1.1.1. Consider the following Rust code of the function `insert` of the `bit_set` library[1].

```rust
/// Adds a value to the set. Returns `true` if the value was not already
/// present in the set.
pub fn insert(&mut self, value: usize) -> bool {
    if self.contains(value) {
        return false;
    }

    // Ensure we have enough space to hold the new element
    let len = self.bit_vec.len();
    if value >= len {
        self.bit_vec.grow(value - len + 1, false)
    }
```

[1] https://docs.rs/bit-set/latest/bit_set/

```
    self.bit_vec.set(value, true);
    return true;
}
```

This function is part of the standard libraries from Rust. Within the RTLola interpreter, the function is called to add new outputs to the set of recently computed outputs. From the intermediate representation, we know the exact number of outputs that have yet to be evaluated. Consequently, we conclude that the call `grow` is never executed, and therefore this part of the code is not relevant for the WCET analysis. △

Chapter 2

Related Work

To ensure the strict timing and safety properties of cyber-physical systems (CPS), we must know the upper bound of the worst-case execution time (WCET) beforehand.

There are two main categories for different methods for performing a WCET analysis presented by [5].

The first method is the measurement-based approach, which measures the maximal execution time given a set of input data and the used hardware. This approach cannot guarantee that the input data set covers a program's WCET path since only a representation of possible WCET scenarios is used for the analysis. To guarantee that the WCET is overestimated, a safe margin is typically added to the measured time, making the estimate not always usable in practice since the result is not a tight upper bound. We seek a safe upper bound because timing is essential for the safety of CPS. Therefore, the measurement-based approach is not ideal in our setting.

The second method is the static WCET analysis which determines the WCET without actually executing the program by creating a program model and abstracting all possible inputs guaranteeing safe upper bounds. Usually, a static WCET analysis tool consists of three phases: flow analysis, low-level analysis, and final calculation. This structure is, for example, used by the aiT tool[1], the Bound-T tool [4], and the WCET tool used for Java [6]. The WCC tool [3] differs from this structure as a WCET analysis is connected to a C compiler and uses the aiT tool for obtaining flow facts. Another tool that deviates from the structure is the SWEET tool [7] since it only has a focus on the flow analysis and lacks a low-level analysis. Therefore, this tool is often used as an add-on to analyze more detailed flow facts.

The three phases are implemented in WCET tools with various methods. The two main techniques are the control-flow analysis and integer-linear programming explained in the following.

The first technique is the control-flow analysis which extracts information from the program code over all possible execution paths represented mainly by a control-flow

graph. It can be divided into a value analysis, a loop-bound analysis, and an analysis to determine the maximal recursion depth. The latter two do not always lead to a result without any further user annotations. However, we will adapt this technique to our approach and combine the flow analysis of the monitor with the analysis of the specification of our CPS. This combination of analyses and the fact that we will use a stream-based specification language facilitates the overall flow analysis. We know the exact loop bound and call history in our specification language.

A widespread technique to finally compute the worst-case execution path is Integer-Linear Programming (ILP), as it is also used by aiT[1], where paths of the program are represented by a system of linear constraints over integer variables. The Implicit Path Enumeration Technique (IPET) uses ILP and is originally introduced by [8] where the execution time of a program is maximized under some constraints formulated as linear constraints. Unlike this popular approach, we will determine the maximal execution path differently. As we analyze our specification beforehand and combine it with the monitor's analysis, we can determine which assembly code will be executed in our monitor in the end. So we do not have to find our maximal execution path as we already know it.

Several approaches for WCET tools are for specific programming languages like C [3, 2] or Java programs [6]. However, there also exist approaches for different types of programming languages like Hume, a domain-specific high-level programming language based on functional programming [9]. Besides analyses performed for specific higher-level languages, some tools operate on binary code like the tool aiT by AbsInt [1]. Further, there has been introduced a new code format, the ALF language [10], to formalize flow facts independent from the code level or programming language. The ALF language is, for example, used by the tool SWEET [7].

Several specification languages can be used for runtime verification. The first approach for defining specification languages is using temporal logics [11]. These consist of atomic propositions, boolean formulas, and temporal operators. Runtime verification with linear temporal logic (LTL) [12] lacks in expressiveness since LTL is limited to describing discrete-time properties. To increase the expressiveness, real-time temporal logics are introduced like the metric temporal logic (MTL) [13].

Typically the feedback produced by monitoring CPS with the presented temporal logics is only boolean. The CPS receives feedback on whether the formula is satisfied or not. While monitoring CPS, quantitive feedback is desired to capture how often violations occur and possibly the last value of the formula.

An approach that incorporates quantitative feedback is using stream-based specification languages like Lola [14, 15] and Lustre [16, 17]. These languages consist of input and output streams where offset lookups can express properties about the future and the past. Since the data received by the input stream is assumed to be received synchronously, new approaches like RTLola are introduced, which can receive input data without a fixed rate. RTLola is based on Lola, and output streams can have a frequency

that denotes the time this output stream should be evaluated. In this thesis, we will use this specification language, introduced by Schwenger [18], as the monitoring language for CPS to perform the WCET analysis.

Chapter 3

Background

This chapter gives a short introduction to the specification language RTLola in Sect. 3.1 followed by an overview over worst-case execution time analyses in Sect. 3.2. Further, the tool cargo asm is explained in Sect. 3.3.

3.1 RTLola

This section first describes the concrete syntax of RTLola represented by an abstract syntax tree (AST) in Sect. 3.1.1. Then, we use the AST to build the dependency graph in Sect. 3.1.2 followed by the evaluation model in Sect. 3.1.3. In the end, the type system of RTLola is introduced in Sect. 3.1.5.

All definitions, lemmas, and propositions are introduced and proven by Schwenger [18].

3.1.1 Concrete Syntax

AST

For the abstract representation of the concrete syntax, the abstract syntax tree (AST) is used. Later on, the semantics of RTLola are defined. There are three types of nodes in the AST representing the different categories of streams: output streams ($s^{\uparrow}$), input streams ($s^{\downarrow}$), and trigger ($s^{!}$). To denote that a stream can be either an input stream or an output stream, we use s^{-}. We define the concrete and abstract syntax of input and output streams and triggers in the following.

Concrete and Abstract Syntax

The i^{th} input stream of the form:

```
input a : T
```

corresponds to the AST $s_i^{\downarrow} = (a, AST(T))$. The first component is the name of the input stream and is defined as $s_i^{\downarrow}.name := a$. The second component is the AST object of the respective type of the stream and is also defined as $T_i^{\downarrow} := AST(T)$.

The j^{th} output stream of the form:

```
output a : T @nHz := e
```

corresponds to the AST $s^{\uparrow}_j = (a, AST(T), n, AST(e))$. Compared to the input stream, the output stream has two additional components, first the evaluation frequency `@nHz` and then the expression `e`. Similar to the input stream we have an additional definition of all components $s^{\uparrow}_j.name := a$, $T^{\uparrow}_j := AST(T)$, $s^{\uparrow}_j.ext := n$, and $s^{\uparrow}_j.expr := AST(e)$.

We restrict the evaluation frequency to a natural number for simplicity reasons. As the frequency is optional in the syntax of the output stream, the AST for an output stream of the form:

```
output a : T := e
```

is defined as $s^{\uparrow}_j = (a, AST(T), \bot, AST(e))$.

Finally, the k^{th} trigger of the form:

```
output a "msg"
```

consists of the name `a` and the trigger message `msg`. If the name corresponds to a name of some input or output stream, it is replaced by the stream reference else, it will result in $\bot$ yielding in the AST

$$s^{!}_k = (s, msg) = \begin{cases} (s^{-}_j, msg) & \text{if } s^{-}_j.name = a \\ (\bot, msg) & \text{otherwise} \end{cases}$$

Incoming data is interpreted as input streams. These values are then used for the computation of the output streams. Depending on the input and output streams, a trigger can be raised.

Expressions:

RTLola specifications contain expressions in the output streams. To compute the abstract syntax of an output stream, we define the computation rules of the AST function for expressions. We differ between synchronous lookup expressions and asynchronous lookup expressions.

Synchronous Lookup A stream accesses another stream synchronously whenever the accessee stream produces a new value.

$$\text{AST}(s) := \begin{cases} Sync(s^{-}_i) & \text{if } s^{-}_i.name = s \\ Sync(\bot) & \text{otherwise} \end{cases}$$

Offset Lookup A stream can access the n-th previous value of another stream synchronously. The accessor stream has to wait for the accessee stream to produce a new value in order to evaluate its expression.

$$\text{AST}(s\texttt{.offset(by: -}n\texttt{)}) := \begin{cases} Offset(s^{-}_i, n) & \text{if } s^{-}_i.name = s \\ Offset(\bot, n) & \text{otherwise} \end{cases}$$

Sample & Hold Lookup This stream lookup is asynchronous. It always accesses the latest computed value of the accessee stream. The accessor stream gets the default value if there is no value for the accessee stream yet.

$$\text{AST}(\texttt{s.hold()}) := \begin{cases} \mathit{Hold}(s_i^-) & \text{if } s_i^-.\mathit{name} = s \\ \mathit{Hold}(\bot) & \text{otherwise} \end{cases}$$

Sliding Window Lookup A stream can access all values of another stream over a specific time interval asynchronously. These values are used with an aggregation function to compute the output.

$$\text{AST}(\texttt{s.aggregate(over: } \delta\texttt{, using: } \gamma\texttt{)}) := \begin{cases} \mathit{Window}(s_i^-, \delta, \gamma) & \text{if } s_i^-.\mathit{name} = s \\ \mathit{Window}(\bot, \delta, \gamma) & \text{otherwise} \end{cases}$$

Default Expression Whenever a lookup fails because so far, no value is produced, the default value gets accessed. Therefore, the sample & hold, the offset, and the sliding window lookup have a default expression that is used in case the lookup fails.

$$\text{AST}(\texttt{e.defaults(to: d)}) := \mathit{Default}(\text{AST}(\texttt{d}), \text{AST}(e'))$$

Function Expression Function expressions `f(e_1,...,e_n)` are used to compute a value for incoming data. General operators as constants, arithmetic operators, or conditionals support infix notation.

$$\text{AST}(\texttt{f(}e_1\texttt{,...,}e_n\texttt{)}) := \mathit{Func}(\texttt{f}, \text{AST}(e_1), ..., \text{AST}(e_n))$$

We have described all syntactical components of the specification. In the following, we define if a given specification is valid:

Definition 1 (Syntactic Validity [18])

An RTLola specification is *syntactically valid* iff

Def. Syntactic Validity

- all stream and trigger definitions conform to the concrete syntax stated above.
- $\mathit{AST}(s)$ can be computed without violating a condition. This especially includes that all stream names can be resolved, such that no $\bot$ value is contained in the AST.
- all names of streams are unique, i.e., $\forall i, j : s_j^-.\mathit{name} = s_i^-.\mathit{name} \implies i = j$.

Example 3.1.1. RTLola specification:

```
input dist : Float64
input height : Float64

output max_dist : Bool := dist > 10000.0
trigger max_dist "The drone has exceeded the maximal distance."

output avg_height : Float64 @ 1Hz := height.aggregate(over: 2min, using: avg)
output too_low : Bool := height < avg_height.hold().defaults(to: 3.0)
trigger too_low "The current height of the drone is below the average height"

output count : Int64 := count.offset(by: -1).defaults(to: 0)
                        + if too_low then 1 else 0
trigger count > 4 "The drone was too low more than 4 times"
```

This example is a simplified version of a specification for monitoring a drone. The properties we monitor are the maximal travel distance of at most 10000 meters before the need of recharging. Then, the drone should ideally not make too big jumps in height.

The first incoming input stream is `dist`, which describes the distance traveled so far by the drone in meters. The second input stream `height` states the current height of the drone, also given in meters.

The first property is realized in the output stream `max_dist`, where we check if the current input stream `dist` is larger than 10000 meters or not. If this is the case, the first trigger is raised.

The average height is computed by the output stream `avg_height` for the second property. This stream is a sliding window and is evaluated periodically every second. It computes the average over the input data of `height` over the last two minutes. Then, the output stream `too_low` does a sample & hold lookup where the last value of the stream `avg_height` is compared to the current value of the drone's height. The second trigger is raised if the current height is lower than the average. Further, the output stream `count` counts the violations of `too_low` and raises the alarm with the third trigger if the violation occurred more than four times. △

3.1.2 Dependency Graph

In the previous section, the abstract representation and the syntactic validity were defined. The next step is to clarify the semantic validity. That's why we introduce the *dependency graph (DG)*, which captures the dependencies between streams.

dependency graph (DG)

Formally the DG is defined in the following:

Definition 2 (Dependency Graph [18])

The *dependency graph* of a specification is a directed multi-graph $DG = (V, E)$ with weighted edges. Its vertices are streams and the edges reflect dependencies between streams:

Def. Dependency Graph

$$V := \mathit{Stream}$$

$$E := \bigcup_{1 \leqslant i \leqslant n^{\uparrow}} \mathit{dep}_{s_i^{\uparrow}}(s_i^{\uparrow}.\mathit{expr}) \cup \bigcup_{1 \leqslant i \leqslant n^{!}} \{(s_i^{!}, 0, s_i^{!}.\mathit{tar})\}$$

The function $\mathit{dep}_{s_i^{\uparrow}}$ is defined as:

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Offset}(s_j^{-}, n)) := \{(s_i^{\uparrow}, n, s_j^{-})\}$$

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Default}(e, e')) := \mathit{dep}(e) \cup \mathit{dep}(e')$$

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Func}(f, a_1, ..., a_n)) := \bigcup_{1 < i \leqslant n^{\uparrow}} \mathit{dep}(a_i)$$

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Sync}(s_j^{-})) := \{(s_i^{\uparrow}, 0, s_i^{-})\}$$

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Hold}(s_j^{-})) := \{(s_i^{\uparrow}, 0, s_j^{-})\}$$

$$\mathit{dep}_{s_i^{\uparrow}}(\mathit{Window}(s_i^{-}, \delta, \gamma)) := \{(s_i^{\uparrow}, (\delta, \gamma), s_j^{-})\}$$

Intuitively the DG is built from the AST by creating a node for every stream and adding an edge for every dependency in the specification. Whenever an output stream accesses another stream, an edge is added to this stream with its weight or a tuple for a sliding window with the aggregation function and the duration.

Example 3.1.2. Consider the RTLola specification from Example 3.1.1. We build the dependency graph in Fig. 3.1 first by creating a node for every stream. Input streams are always sink nodes as they do not depend on any other streams. On the other hand, triggers are always leave-nodes as no other streams can access them. In this example, the output stream `count` ($s_4^{\uparrow}$) does an offset lookup with 1 to itself as well as a synchronous lookup to the stream `too_low` ($s_3^{\uparrow}$). Therefore, we add in the DG a self-loop to the node $s_4^{\uparrow}$ with weight 1 and an edge with weight 0 from $s_4^{\uparrow}$ to $s_3^{\uparrow}$. The output stream `avg_height` ($s_2^{\uparrow}$) is a sliding window, and therefore the edge to $s_2^{\downarrow}$ is denoted with the pair of duration and the respective aggregation function. All other stream lookups are 0-weighted as they are either synchronous lookups or sample & hold lookups. △

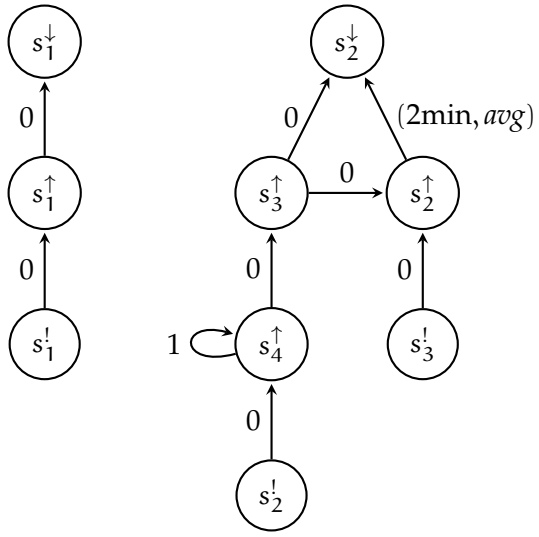


Figure 3.1: Dependency Graph for RTLola specification in Example 3.1.1.

3.1.3 Evaluation Order

We determine the evaluation order and layers of an RTLola specification from the DG. Especially the evaluation layers will be a key component for the WCET analysis of RTLola. To get an intuition for the evaluation order and which difficulties may arise, consider the following examples:

Example 3.1.3. RTLola specification:

```
input a : Int64
output b : Int64 := a + c.offset(by: -1).defaults(to: 0)
output c : Int64 := a + 1
```

The input stream `a` is accessed by both streams `b` and `c` synchronously, so `a` has to be evaluated first. Output stream `b` has an offset lookup by 1 to `c`, so we access the previous value of `c`. For this reason, we have to evaluate `c` before `b`. As output stream `c` only depends on `a`, we can evaluate `b` after `c` and get a correct evaluation.

This results in the evaluation order $\prec$ with $a \prec c \prec b$. △

Example 3.1.4. RTLola specification:

```
input a : Int64
output b : Int64 := a + c.offset(by: -1).defaults(to: 0)
output c : Int64 := b.offset(by: -1).defaults(to: 0) - 1
```

If we compare this example to the previous one, only output stream `c` has changed to an offset lookup with 1 to `b`, so we access the previous value of `b`. Applying the same approach to this example as in the previous one, difficulties arise. Since `b` accesses `c` and `c` accesses `b` we get a cyclic dependency and therefore cannot define a total order

for this specification. For evaluating output stream b we can use the default value of c and evaluate b before c. So, in theory, we can evaluate b first and then c. However, this results in a new problem as c has not been evaluated yet. So there does not exist a current value of c. Therefore, b gets the second-last value and not the previous one as intended by the specification. △

pseudo evaluation phase

To overcome this difficulty, we introduce a new evaluation phase, called *pseudo evaluation phase*, where each evaluated stream gets a new pseudo value. This pseudo value is replaced with the actual computed value when a new value is computed.

→ Exm. 3.1.3, p. 14

To sum up the results from the two examples, Example 3.1.3 and Example 3.1.4, we can give the formal definition of the evaluation order.

Definition 3 (Evaluation Order [18])

Def. Evaluation Order

The *evaluation order* $\prec$ is a partial order on streams, reflecting the structure of the dependency graph $DG = (V, E)$. The evaluation order is the transitive closure of a relation satisfying the following rules:

1. $\forall i, j : s_i^{\downarrow} \prec s_j^{\uparrow}$
2. $(s_i^{\uparrow}, x, s_j^{-}) \in E \wedge (x = 0 \vee x = (\delta, \gamma)) \wedge s_i^{\uparrow} \neq s_j^{-} \implies s_j^{-} \prec s_i^{\uparrow}$

Note that triggers are not considered in the definition of the evaluation order as they do not get accessed by any other stream. Therefore, triggers can always be evaluated after all other streams. The evaluation order can also be represented as evaluation layers described in the following definition.

Definition 4 (Evaluation Layer [18])

Def. Evaluation Layer

The *evaluation layer* is an equivalent representation of $\preceq$. If $Layer(s_i^{-}) = k$, then there is a strictly decreasing sequence of k streams w.r.t. $\prec$ starting in $s_i^{\uparrow}$.

The maximal evaluation layer λ^{max} is described as:

$$\lambda^{max} := \max\{\lambda \mid \exists s_j^{-} : \lambda = Layer(s_j^{-})\}$$

Example 3.1.5 (Evaluation Layer)**.** The specification from Example 3.1.1 results in the following evaluation layers:

→ Exm. 3.1.1, p. 12

$$Layer(s_1^{\downarrow}) = 0 \qquad Layer(s_2^{\uparrow}) = 1 \qquad Layer(s_1^{!}) = 4$$
$$Layer(s_2^{\downarrow}) = 0 \qquad Layer(s_3^{\uparrow}) = 2 \qquad Layer(s_2^{!}) = 4$$
$$Layer(s_1^{\uparrow}) = 1 \qquad Layer(s_4^{\uparrow}) = 3 \qquad Layer(s_3^{!}) = 4$$

Input streams are always in layer zero as they do not depend on any other streams. The output streams `max_dist`($s_1^{\uparrow}$) and `avg_height`($s_2^{\uparrow}$) are on the first layer as they both only depend on input streams. The output stream `too_low` does not only depend on the input stream `height` but also on the output stream `avg_height`, so `too_low` has to be on layer 2. The output stream `count` depends on itself and on `too_low`, which results in layer 3. All three triggers only access one stream, so they are all on layer 4. △

3.1.4 Memory Analysis

A first static check on an RTLola specification is made by determining the maximal memory-bound of a stream by considering the dependency graph. This bound will be later on used in the memory handling of the WCET algorithm in Sect. 4.2.2

→ Sect. 4.2.2, p. 28

Definition 5 (Memory Bound of a Stream)

Def. Memory Bound

Let $G = (V, E)$ be a dependency graph. The *memory bound* of a stream s_i^{-} is

$$mem^{\max}(s_i^{-}) = \max\{n \mid \exists s_j^{\uparrow} : (s_j^{\uparrow}, n, s_i^{-}) \in E \wedge n \in \mathbb{N}\} + 1$$

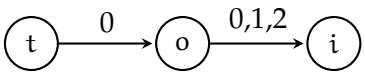
Intuitively the memory-bound is determined with the dependency graph by comparing the weights of all incoming edges of a certain stream and filtering the maximal offset. Incremented by one, this offset is equivalent to the memory bound of this stream.

Example 3.1.6. RTLola specification:

```
input execTime : Float64
output lastThree : Bool := execTime.offset(by -2).defaults(to: 5.0) > 5.0
                         ∧ execTime.offset(by -1).defaults(to: 5.0) > 5.0
                         ∧ execTime > 5.0
trigger lastThree "The last three tasks each took longer than 5ms"
```

This example checks whether three consecutive values of the input stream `execTime` are greater than 5ms and release a trigger if the property is satisfied.

The corresponding dependency graph where `t` is the trigger, `o` is output stream `lastThree`, and `i` is input stream `execTime`:



For each stream, we compute the memorization bound as defined in Def. 5, resulting in:

1. $mem^{\max}(\texttt{execTime}) = 3$
2. $mem^{\max}(\texttt{lastThree}) = 1$

Input stream `execTime` is only accessed by the output stream `lastThree` where the maximal offset is two. Consequently, the maximal memorization bound of `execTime` is three. `lastThree` is only accessed by the trigger stream with one synchronous stream lookup leading to the maximal memorization bound of 1. △

3.1.5 Types

Value Type/Pacing Type

In RTLola, types are represented by a pair consisting of the *value type* and the *pacing type*. The value type defines the size and the domain of the resulting value of a stream. These types range from boolean values, signed and unsigned integers, to floating-point numbers. Except for the boolean value type, all value types are annotated with the corresponding bit length. The pacing type indicates when a particular stream has to be evaluated, either periodically or after arbitrary time points. Therefore, the pacing type of a stream has two categories. If the stream has an optional value after the symbol `@`, this value indicates the fixed frequency of the stream. The pacing type of this stream is periodic. Otherwise, the pacing type of a stream is event-based and is represented by a positive boolean formula φ overall input streams. In detail, the event-based type represents the dependencies of output streams to input streams. Note that input streams cannot be of periodic type since we have no control over the arrival of the input streams. Thus, every input stream has an event-based type as its only dependency is on itself.

To get more intuition for the pacing type, consider the following example:

Example 3.1.7. A fragment of Example 3.1.1: → Exm. 3.1.1, p. 12

```
input height : Float64
output avg_height : Float64 @ 1Hz := height.aggregate(over: 2min, using: avg)
output too_low : Bool := height < avg_height.hold().defaults(to: 3.0)
output count : Int64 := count.offset(by: -1).defaults(to: 0)
                        + if too_low then 1 else 0
```

The input stream `height` is event-based and has the type of the input stream itself. The output stream `avg_height` is periodic and has the pacing type 1Hz, and is evaluated every second. The output stream of `too_low` is event-based, and the pacing type is only bound to the input stream `height` as the hold lookup to `avg_height` can be omitted since we always take the latest value of `avg_height`. The output stream `count` is again event-based, and the pacing type is `count` $\wedge$ `too_low`. △

3.2 Worst-Case Execution Time Analysis (WCET)

This section introduces a WCET analysis first by a short introduction to control flow representations and definitions used to define a control flow graph in Sect. 3.2.1 and Sect. 3.2.2. The foundation of these definitions is built up by Stattelmann [19]. Then, we define the worst-case execution path, the worst-case execution time of a routine in Sect. 3.2.4.

As we will perform the WCET analysis on assembly code in Sect. 5.4 the definitions and examples in the following sections are restricted to assembly code instructions.

3.2.1 Basic Blocks

A basic block is a sequence of instructions with no branches except for the end of the basic block. The definition of a basic block is given in the following.

Definition 6 (Basic Block [19])

Def. Basic Block

Let B be a finite set of *basic blocks*. Every basic block $b \in B$ has at the start an instruction with an unique address $a \in A \subset \mathbb{N}$ computed by the function $address : B \to A$. The respective basic block b for a given address a can be computed by $address^{-1} : A \to B$.

Example 3.2.1. Fig. 3.2 shows a snippet of x86 code and its basic blocks, which computes the gaussian sum of the given input parameter with the usual calling convention. The input parameter is expected in the `edi`-register, and the result is stored in the `eax`-register. This code snippet consists of four basic blocks where the last instruction is always a branching instruction. The first unconditional jump instruction forms the first basic block. The second basic block consists of the next three instructions where the conditional jump at address `0x800c` closes the basic block. The third basic block contains a move instruction followed by a return instruction. The last two instructions of the code snippet form the last basic block. △

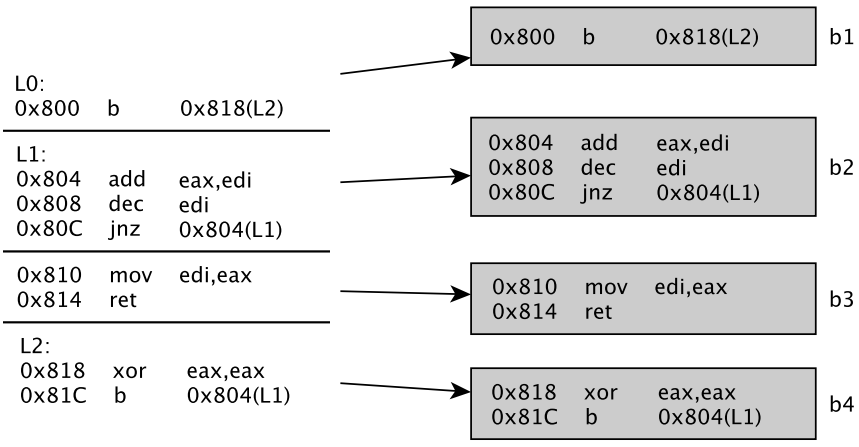


Figure 3.2: Assembly code for computing the gaussian sum and its basic blocks

L3:			
0x100	b	0x118(L5)	b5
L4:			
0x104	mul	edi	
0x108	dec	edi	b6
0x10C	jnz	0x104(L4)	
0x810	mov	edi,eax	b7
0x814	ret		
L5:			
0x118	mov	eax,1	b8
0x11C	b	0x104(L4)	

Figure 3.3: Routine $r_{faculty}$ computing the faculty

3.2.2 Routines

In this thesis, we will analyze more complex programs than the code snippet from Example 3.2.1. Most programs are divided into different subroutines, which usually handle one part of the program. In our context, such subroutines are functions and procedures of a program to which we will refer to as routines from this point on. The definition of a routine is given in the following:

Definition 7 (Routine [19])

Let $\mathtt{R}$ be the finite set of all routines for a given program. Every *routine* $r \in \mathtt{R}$ consists of a fixed number n of basic blocks $b_1, ..., b_n \in B_r$.

Def. Routine

To guarantee that routines have unique addresses and to access some routine through their entry block and last block, we define the following properties for routines.

Definition 8 (Properties of Routines [19])

There exists a function that assigns a basic block b to a routine r *routine* : $\mathtt{B} \rightarrow \mathtt{R}$ and for every $r_1, r_2 \in \mathtt{R} : r_1 \neq r_2 \Rightarrow B_{r_1} \cap B_{r_2} = \emptyset$.

The set of all entry blocks of a given program is given by *Starts* $\subseteq \mathtt{B}$, and the function *start* : $\mathtt{R} \rightarrow$ *Starts* maps each routine to its entry block.

The set of all end blocks of a given program is given by *Ends* $\subseteq \mathtt{B}$, and the function *end* : $\mathtt{R} \rightarrow$ *Ends* maps each routine to the last block before the routine ends.

Def. Properties of Routines

Example 3.2.2. We compare the routine $r_{faculty}$ in Fig. 3.3 with the routine r_{gauss} in Fig. 3.2. $r_{faculty}$ computes faculty, and r_{gauss} computes the gaussian sum explained in Example 3.2.1.

The faculty routine consists of four basic blocks. Similar to the code of the gaussian sum, the structure of the basic blocks is the same. The blocks only differ in the computation. In basic block `b6`, we change the addition of the gaussian sum to a multiplication. Note that the `mul`-instruction only takes one register, as the result register is defined

to be `eax`. Since the result register has to be initially 1 in block `b8` to perform the multiplication, we move 1 to the result register. The input parameter is expected in the `edi`-register, and the result is stored in the `eax`-register. Each basic block ends with a branching instruction. △

3.2.3 Control Flow Graph

In the following, we define the control flow graph (CFG), which describes all possible control flow actions for a routine and its basic blocks.

Definition 9 (Control Flow Graph [19])

Def. Control Flow Graph

For some routine $r \in R$, the *CFG* is denoted by: $CFG_r := (B_r, E_r)$ where B_r consists of all basic blocks of the routine r and the set of edges $E_r \subseteq B_r \times B_r$ that represents the control flow for the given routine r.

Example 3.2.3. According to Def. 9, we build the CFG for routine r_{gauss} from Example 3.2.2 in the following. The CFG is shown in Fig. 3.4. The branches of the basic blocks `b1` and `b4` result in a clear edge. Only the branch's address has to be resolved into an edge to its corresponding basic block. The `ret`-instruction from `b3` is transformed into an edge to the caller function. The conditional jump of `b2` is split into two edges where we have a self-loop if `edi`$\neq 0$ and an edge to `b3` if `edi`$= 0$. △

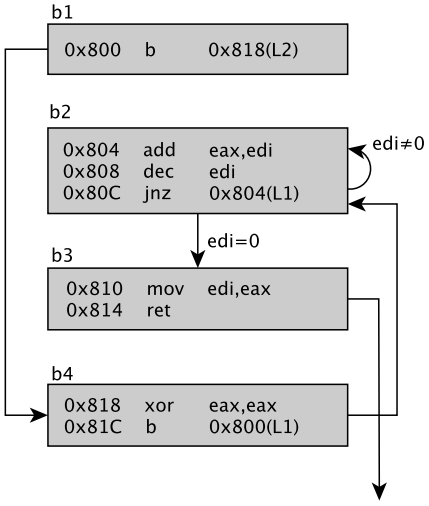


Figure 3.4: CFG for routine r_{gauss}

3.2.4 Worst-Case Execution Path

The definition of the WCET of a routine is divided into different levels of abstraction, where the lowest level is the execution time of a single assembly code instruction. We give the first definition for the next level of basic blocks using the lowest level.

Definition 10 (Execution Time of a Basic Block)

Let $b \in B_r$ be a basic block of the routine $r \in R$ and $A_b \subset \mathbb{N}$ the set of all addresses $a_1, ..., a_n \in A_b$ of the basic block b. The function $exec_{instr} : A_b \to \mathbb{R}$ assigns the execution time to every instruction of A_b. The *execution time of the basic block* b is defined as:

Def. Execution Time of a Basic Block

$$exec_{block}(b) = \sum_{a \in A_b} exec_{instr}(a).$$

The next level after the basic blocks is the abstraction level of paths of a routine. In the following, we define the execution time of a path.

Definition 11 (Execution Time of a Path)

Let $r \in R$ be a routine of a program, $p \in Paths(r)$ be a possible path of the routine r where $Paths(r)$ is the set of all possible paths of the routine $r \in R$. The path p consists of a sequence of length i of basic blocks $b_1, ..., b_i \in B_r$. The *execution time of the path* p is defined as:

Def. Execution Time of a Path

$$exec_{path}(p) = \sum_{b \in B_r} exec_{block}(b).$$

With the execution time for every path of a routine, the last step is defining the worst-case execution path (WCEP) of a routine.

Definition 12 (Worst-Case Execution Path)

Let $r \in R$ be a routine of a program and $ExecPaths(r)$ the set of all execution times of the routine $r \in R$. The WCET of routine r also called the *Worst-Case Execution Path (WCEP)* of routine r is defined as:

Def. Worst-Case Execution Path

$$wcet(r) = \max_{p \in ExecPaths(r)} (exec_{path}(p))$$

Example 3.2.4. Consider the CFG from Fig. 3.4. Since there is a loop in basic block b_2 we assume that the basic block is only visited once. With this restriction, the only path possible in the CFG is $p = b_1, b_4, b_2, b_3$.

The first step is computing the execution time of all basic blocks by combining the execution times of all instructions of the corresponding block. For simplicity, we assume

that the execution time of all instructions is 1. The resulting execution times of the basic blocks are:

$$exec_{block}(\mathrm{b1}) = exec_{instr}(\mathrm{0x800}) = 1$$

$$exec_{block}(\mathrm{b2}) = exec_{instr}(\mathrm{0x804}) + exec_{instr}(\mathrm{0x808}) + exec_{instr}(\mathrm{0x80C}) = 3$$

$$exec_{block}(\mathrm{b3}) = exec_{instr}(\mathrm{0x810}) + exec_{instr}(\mathrm{0x814}) = 2$$

$$exec_{block}(\mathrm{b4}) = exec_{instr}(\mathrm{0x818}) + exec_{instr}(\mathrm{0x81C}) = 2$$

Since we only have one possible path p this path is the WCEP. The result of the execution time of p is equal to the WCET of the entire routine:

$$\begin{aligned} wcet(\mathrm{r}_{gauss}) &= exec_{path}(\mathrm{p}) \\ &= exec_{block}(\mathrm{b1}) + exec_{block}(\mathrm{b2}) + exec_{block}(\mathrm{b3}) + exec_{block}(\mathrm{b4}) = 8 \end{aligned}$$

△

3.3 Cargo asm

The RTLola framework is implemented in Rust. As we build up a CFG for an RTLola program with assembly code instructions, we must obtain the assembly code for the used functions.

We will use an extension for `cargo` called *cargo asm*. This tool allows us to use the `cargo asm` command with a proper path to the function to obtain the assembly code.

Before using this extension we have to install it:

```
cargo install cargo-asm
```

To obtain the assembly code from a function, we need the path of the function. To see the rust annotations, we have to set the parameter `--rust`. The resulting command is:

```
cargo asm "<path_to_function>" --rust
```

Example 3.3.1. With the command

```
cargo asm "bit_set::BitSet<B>::contains" --rust
```

we get the assembly code for the function `contains` of the `bit_set` library:

```
pub fn contains(&self, value: usize) -> bool {
    push rbp
    mov rbp, rsp
    sub rsp, 64
    mov qword, ptr, [rbp, -, 56], rsi
    mov qword, ptr, [rbp, -, 24], rdi
    mov qword, ptr, [rbp, -, 16], rsi
```

```
    mov qword, ptr, [rbp, -, 48], rdi
let bit_vec = &self.bit_vec;
    mov qword, ptr, [rbp, -, 8], rdi
value < bit_vec.len() && bit_vec[value]
    call bit_vec::BitVec<B>::len
    mov qword, ptr, [rbp, -, 40], rax
    jmp LBB7269_4
LBB7269_1:
 value < bit_vec.len() && bit_vec[value]
    mov byte, ptr, [rbp, -, 25], 0
    jmp LBB7269_3
LBB7269_2:
    mov rsi, qword, ptr, [rbp, -, 56]
    mov rdi, qword, ptr, [rbp, -, 48]
    value < bit_vec.len() && bit_vec[value]
    lea rdx, [rip, +, l___unnamed_383]
    call <bit_vec::BitVec<B> as core::ops::index::Index<usize>>::index
    mov qword, ptr, [rbp, -, 64], rax
    jmp LBB7269_5
    LBB7269_3:
}
    mov al, byte, ptr, [rbp, -, 25]
    and al, 1
    movzx eax, al
    add rsp, 64
    pop rbp
    ret
LBB7269_4:
    mov rax, qword, ptr, [rbp, -, 56]
    mov rcx, qword, ptr, [rbp, -, 40]
    value < bit_vec.len() && bit_vec[value]
    cmp rax, rcx
    jb LBB7269_2
    jmp LBB7269_1
LBB7269_5:
    mov rax, qword, ptr, [rbp, -, 64]
    value < bit_vec.len() && bit_vec[value]
    mov al, byte, ptr, [rax]
    value < bit_vec.len() && bit_vec[value]
    and al, 1
    mov byte, ptr, [rbp, -, 25], al
    jmp LBB7269_3
```

△

Chapter 4

Worst-Case Execution Time Analysis of RTLola Specifications

This chapter presents a WCET algorithm of an RTLola specification. First, some notation in Sect. 4.1 is introduced, followed by defining the WCET on an RTLola specification in Sect. 4.2.

The type of WCET analysis used in this chapter is based on a path analysis covered in Sect. 3.2. The computation of the WCET uses an abstract time metric analyzing all streams. We will cover two versions for executing the specification: first, a parallel model where all streams of one layer are evaluated simultaneously, followed by a sequential one where the streams of one layer are processed one after another. In both cases, the analysis of each stream is equal.

→ Sect. 3.2, p. 17

4.1 Notation

Before we introduce the computation of the WCET of a specification in more detail, we define the set of all evaluation layers with the maximal layer λ^{max} defined in Sect. 3.1.3 and the respective streams of each layer.

→ Sect. 3.1.3, p. 14

Definition 13 (Streams of a Layer)

Let $l \in \{0, 1, ..., \lambda^{max}\}$ and $s_1^-, ..., s_i^- \in \mathit{Streams}$. We define $\mathit{streams} : \{0, 1, ..., \lambda^{max}\} \to \mathit{Streams}$ as:

$$\mathit{streams}(l) = \{s_i^- \mid l = \mathit{Layer}(s_i^-)\}$$

Every layer has a fixed number of streams where each stream is executed. The inverse function is introduced in Sect. 3.1.3.

→ Sect. 3.1.3, p. 14

→ Exm. 3.1.1, p. 12
→ Exm. 3.1.5, p. 15

Example 4.1.1. Consider the RTLola specification from Example 3.1.1 and the corresponding layers of the streams from Example 3.1.5. The maximal layer of this specification is $\lambda^{\max} = 4$. We can determine the corresponding streams to the five layers with the function introduced in Def. 13.

$$streams(0) = \{s_1^{\downarrow}, s_2^{\downarrow}\}$$

$$streams(1) = \{s_1^{\uparrow}, s_2^{\uparrow}\}$$

$$streams(2) = \{s_3^{\uparrow}\}$$

$$streams(3) = \{s_4^{\uparrow}\}$$

$$streams(4) = \{s_1^{!}, s_2^{!}, s_3^{!}\}$$

△

4.2 WCET Analysis

To compute the WCET of a stream, we divide the computation into several stages. We start with the calculation of the WCET of an expression, then define the influence of the memory handling on the WCET. These two analyses correspond to the execution time of a basic block introduced in Sect. 3.2.4. Finally, we combine the results to the final WCET of input streams, output streams, and triggers. The WCET of one stream can be interpreted as an abbreviation of the determination of the execution time of a path defined in Sect. 3.2.4. A path corresponds to a combination of a stream and its expression.

→ Sect. 3.2.4, p. 21

→ Sect. 3.2.4, p. 21

4.2.1 Expression Analysis

Reconsider that output streams and triggers contain arbitrary expressions that affect the WCET of a stream. Adding two terms is more costly than loading a constant. Therefore, we have to define the WCET of an expression before defining the WCET of a stream.

Definition 14 (WCET of an Expression)

Def. WCET Expression

The *WCET of an expression* e is defined as:

$$wcet_{expr}(Default(\mathrm{d}, expr)) = \max(wcet_{expr}(\mathrm{d}), wcet_{expr}(expr)) + wcet_{op}(def)$$

$$wcet_{expr}(Sync(\mathrm{s})) = streamAccess$$

$$wcet_{expr}(Hold(\mathrm{s})) = streamAccess$$

$$wcet_{expr}(Offset(s, n)) = offsetAccess$$

$$wcet_{expr}(Func(ite, c, t, e)) = wcet_{op}(ite) + wcet_{expr}(c) + \max(wcet_{expr}(t), wcet_{expr}(e))$$

$$wcet_{expr}(Func(f, e_1, ..., e_n)) = wcet_{op}(f) + \sum_{i \in \{1,...,n\}} wcet_{expr}(e_i)$$

with *streamAccess*, *offsetAccess*, and $wcet_{op}$.

Consequently, the WCET of an expression is composed of the overhead caused by an operator's execution time and the WCET of all operands. In the case of a default expression, we must check whether the term's value is accessible, which we interpret as an if-then-else operation. Then, we determine the WCEP of the default expression by comparing the WCET of the default expression and the expression. Further, a synchronous and hold lookup is represented by *streamAccess* and an offset lookup by *offsetAccess*. Both variables state the time needed to perform a lookup. We do not differentiate between synchronous lookups and hold lookups because both access the latest value of their stream. They only differ in their pacing type. However, offset lookups are handled differently since we have an additional overhead for looking up the value with the correct offset. So an offset lookup might access the first value of the stream. Depending on the underlying memory model of the CPS, accessing the first value or the latest value result in different access times, which is discussed in more detail in Sect. 4.2.2.

→ Sect. 4.2.2, p. 28

Remark 4.2.1. *For the remainder of this thesis, we assign an operator of an expression to its execution time.*

$$wcet_{op}(comp) = exec_{comp} \qquad when\ comp \in \{=, \neq, <, \leqslant, >, \geqslant, \wedge, \vee, ...\}$$

$$wcet_{op}(arith) = exec_{arith} \qquad when\ arith \in \{+, -, *, /\}$$

$$wcet_{op}(ite) = exec_{ite} \qquad when\ ite \in \{ite\ or\ def\}$$

$$wcet_{op}(const) = exec_{const} \qquad when\ const\ is\ a\ constant$$

We assume that the if-statement and the default expression share the same execution time because the implementation handles these two cases equivalently.

Example 4.2.1. RTLola specification:

```
input execTime : Float64
output δexecTime : Float64 := execTime
                            - execTime.offset(by: -1).defaults(to: 15.0)
trigger δexecTime > 5.0 "The difference between the last two tasks is greater than
    5.0"
```

In this example, the output stream `δexecTime` computes the difference between the last two processed tasks and releases a trigger if the difference is larger than 5ms, meaning

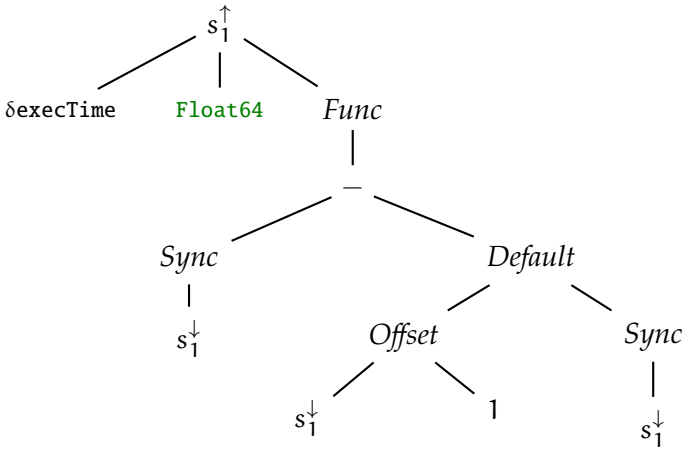


Figure 4.1: AST of δexecTime

that the last processed task had an execution time longer than 5ms. We do not need the absolute amount of δ`execTime` as the input stream `execTime` states the execution time of each processed task in milliseconds from a starting point where the monitor can process only one task at once. So the inputs for this stream are continuously increasing.

We compute the WCET of the output stream δ`execTime`'s expression. The first step is to extract the part of the AST representing the expression of the stream. Then, we apply the function $wcet_{expr}$ from Def. 14.

→ Def. 14, p. 26

Fig. 4.1 shows the AST of the output stream δ`execTime`. The root element is the subtraction which is translated into $exec_{arith}$. In the next step, we compute the WCET of both operands. The right-hand side is a synchronous lookup which results in the constant *streamAccess*. The offset lookup of the left-hand side is computed by determining the maximum of the default expression and the offset lookup resulting in $\max(offsetAccess, streamAccess)$. The final result of the WCET of this expression is:

$$wcet_{expr}(\delta\texttt{execTime}) = exec_{arith} + streamAccess + \max(offsetAccess, streamAccess)$$

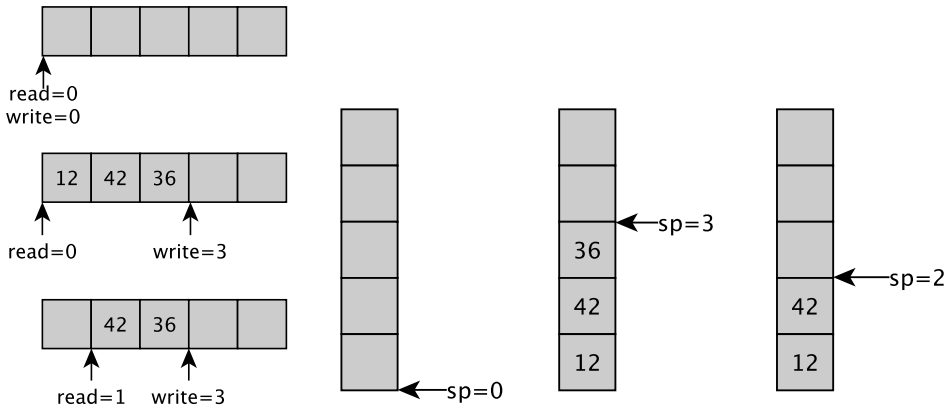
△

4.2.2 Memory Handling

→ Exm. 4.2.1, p. 27

As Example 4.2.1 indicates, accessing different values of the streams causes different accessing times depending on the memory model. This section will compare two memory models and explain how these different access times arise.

Example 4.2.2. We compare two different memory management systems, a ring buffer and a stack displayed in Fig. 4.2. A ring buffer has two pointers `read` and `write`. `read` points to the first element being stored in the buffer, whereas `write` points after the last element stored in the memory. If an element is read, the `read` pointer increases by one.



(a) Ring buffer (b) Stack

Figure 4.2: Overview of different memory handling

Similarly, the `write` increases if an element is put into the buffer. In Fig. 4.2a, we store the incoming values of an input stream being $12, 42, 36$. Reading value 12 is an atomic operation, whereas reading value 36 requires reading all values before increasing the memory access time.

In contrast to the ring buffer, the values are stored linearly in a stack, with a stack pointer (sp) pointing to the last element stored in the memory. If a value is read, the sp decreases. In Fig. 4.2b, we also store the incoming values of an input stream being $12, 42, 36$. The difference to the ring buffer is that only the last stored value is accessible. Consequently, the memory access of value 12 requires first reading both values 42 and 36, resulting in a linear runtime depending on the size of the stack. In contrast to the ring buffer, accessing value 36 is atomic.

Consequently, the ring buffer works with the first-in-first-out (FIFO) protocol and the stack with the last-in-first-out (LIFO) protocol. △

From the results of the previous example, we conclude that using, for example, a ring buffer for a synchronous lookup of a stream means that the latest stored value has to be read. This access is of linear time, whereas it is atomic in a stack.

Remark 4.2.2. *To monitor an RTLola specification, the number of required values of a stream is its memory-bound* $\text{mem}^{\max}(s_i^-)$ *defined in Sect. 3.1.4. As Example 4.2.2 shows, different memory management systems result in different memory access times. Thus, we introduce* exec_{mem} *stating the memory access time.*

→ Sect. 3.1.4, p. 16

The next step of the WCET analysis of RTLola is to define the memory analysis.

Definition 15 (WCET of Memory Handling)

Def. WCET Memory Handling

Let $mem^{\max}(s_i^-)$ be the memory bound of s_i^-. The *WCET of memory handling* of a stream s_i^- is defined as:

$$wcet_{mem}(s_i^-) = mem^{\max}_{s_i^-} * exec_{mem}$$

with $exec_{mem}$.

Intuitively, the WCET is reached for accessing a stream if we have to go through all values to access the wanted value. For example, accessing the latest value in a stack is the worst-case. The maximal number of values that must be read beforehand is, in the worst-case, the maximal memory bound. In consequence, memory handling is composed of the access time $exec_{mem}$ and the memory-bound of the respective stream. The memory handling is only defined for input and output streams since a trigger has no values that must be stored.

4.2.3 WCET of RTLola Streams

With the memory analysis from Sect. 4.2.2 and the expression analysis from Sect. 4.2.1, we define the WCET of a stream.

Definition 16 (WCET of a Stream)

Def. WCET Stream

The *WCET of a stream* is defined as:

$$wcet_{stream}(s_i^{\downarrow}) = wcet_{mem}(s_i^{\downarrow})$$

$$wcet_{stream}(s_i^{\uparrow}) = wcet_{expr}(s_i^{\uparrow}) + wcet_{mem}(s_i^{\uparrow})$$

$$wcet_{stream}(s_i^{!}) = wcet_{expr}(s_i^{!})$$

→ Exm. 4.2.1, p. 27

Example 4.2.3. Consider the RTLola specification from Example 4.2.1 before. Now we want to compute the WCET of all streams. The WCET of input stream `execTime` only consists of the memory handling time. First, the memory bound of `execTime` is computed, which is $mem^{\max}(s_1^{\downarrow}) = 2$. Then, the factor $exec_{mem}$ is multiplied, resulting in the WCET

$$\begin{aligned} wcet_{stream}(s_1^{\downarrow}) &= wcet_{mem}(s_1^{\downarrow}) \\ &= 2 * exec_{mem} \end{aligned}$$

The WCET of output stream δ`execTime` consists of the memory handling time and the WCET of its expression. The memory handling time is computed analogously to `execTime`. Finally, the WCET of the expression we computed in Example 4.2.1 is added.

$$wcet_{stream}(s_1^{\uparrow}) = wcet_{expr}(s_1^{\uparrow}) + wcet_{mem}(s_1^{\uparrow})$$

$$= exec_{arith} + streamAccess + \max(offsetAccess, exec_{const}) + exec_{mem}$$

$$= exec_{arith} + streamAccess + offsetAccess + exec_{mem}$$

$$(\text{assuming } offsetAccess > exec_{const})$$

To compute the WCET for the trigger stream, we simply compute the WCET of its expression resulting in:

$$wcet_{stream}(s^{!}_{1}) = wcet_{expr}(s^{!}_{1})$$

$$= exec_{comp} + streamAccess + exec_{const}$$

△

4.2.4 Parallel Model

An RTLola specification has an evaluation model with different layers shown in Sect. 3.1.3. The streams within a layer are all independent from each other in their execution and only depend on the streams from the layers before. Therefore, we introduce a parallel model by dividing the specification into its evaluation layers, simultaneously executing all streams of one specific layer. The WCET in the parallel model is defined in the following.

→ Sect. 3.1.3, p. 14

Definition 17 (WCET of Parallel Execution)

The *WCET of a parallel execution* is defined as:

Def. WCET of Parallel Execution

$$wcet^{parallel}_{spec} = \sum_{l \in \{0,1,\ldots,\lambda^{max}\}} \left(\max_{s^{-}_{i} \in streams(l)} wcet_{stream}(s^{-}_{i}) \right)$$

Intuitively, we determine the WCEP of one layer and sum up the WCET of each layer resulting in the WCET of the whole specification.

Example 4.2.4. RTLola specification:

```
input execTime : Float64
output δexecTime : Float64 := execTime
                            - execTime.offset(by: -1).defaults(to: 15.0)
trigger δexecTime > 5.0 "The difference between the last two tasks is greater than
    5.0"

output maxTime : Bool := 15.0 < execTime
trigger maxTime "The last task exceeded the maximal time"

output lastThree : Bool := 5.0 < execTime.offset(by -2).defaults(to: 5.0)
                         ∧ 5.0 < execTime.offset(by -1).defaults(to: 5.0)
                         ∧ 5.0 < execTime
trigger lastThree "The last three tasks each took longer than 5ms"
```

→ Exm. 4.2.1, p. 27

This specification extends Example 4.2.1 from before. We add two more properties to our specification. Output stream `maxTime` checks whether a task exceeds the maximal time of 15ms and releases a trigger if this is the case. The second property is modeled by the stream `lastThree` and checks if the last three tasks individually take longer than 5ms. If this check is satisfied, a trigger is released.

→ Def. 17, p. 31
→ Exm. 4.2.3, p. 30

The WCET of this specification is computed by applying Def. 17. The WCET of the first output and trigger are computed in Example 4.2.3. Analogously, the remaining streams are handled:

$$
\begin{aligned}
wcet_{stream}(s_1^{\downarrow}) &= 3 * exec_{mem} \\
wcet_{stream}(s_2^{\uparrow}) &= exec_{comp} + streamAccess + exec_{const} + exec_{mem} \\
wcet_{stream}(s_3^{\uparrow}) &= 5 * exec_{comp} + streamAccess + \max(offsetAccess, exec_{const}) \\
&\quad + \max(offsetAccess, exec_{const}) + 3 * exec_{const} + exec_{mem} \\
&= 5 * exec_{comp} + 2 * offsetAccess + streamAccess \\
&\quad + 3 * exec_{const} + exec_{mem} \text{ (assuming } offsetAccess > exec_{const}) \\
wcet_{stream}(s_2^{!}) &= streamAccess \\
wcet_{stream}(s_3^{!}) &= streamAccess
\end{aligned}
$$

Since we compute the WCET of the parallel execution, we have to determine the WCEP of each layer:

$$
\begin{aligned}
streams(0) &= \{s_1^{\downarrow}\} \Rightarrow wcet_{stream}(s_1^{\downarrow}) = 3 * exec_{mem} \\
streams(1) &= \{s_1^{\uparrow}, s_2^{\uparrow}, s_3^{\uparrow}\} \Rightarrow wcet_{stream}(s_3^{\uparrow}) \\
&= 5 * exec_{comp} + streamAccess + 2 * offsetAccess + 3 * exec_{const} + exec_{mem} \\
streams(2) &= \{s_1^{!}, s_2^{!}, s_3^{!}\} \Rightarrow wcet_{stream}(s_1^{!}) = exec_{comp} + streamAccess + exec_{const}
\end{aligned}
$$

The final WCET of the parallel execution of the specification is:

$$
wcet_{spec}^{parallel} = wcet_{stream}(s_1^{\downarrow}) + wcet_{stream}(s_3^{\uparrow}) + wcet_{stream}(s_1^{!})
$$

△

4.2.5 Sequential Model

The parallel model is not applicable in practice since the RTLola interpreter executes all streams sequentially. That is why we introduce an approach for analyzing the sequential execution of the specification. The WCET in the sequential model is defined as:

Definition 18 (WCET of Sequential Execution)

The *WCET of a sequential execution* is defined as:

Def. WCET of Sequential Execution

$$wcet_{spec}^{sequential} = \sum_{l \in \{0,1,\ldots,\lambda^{max}\}} \left(\sum_{s_i^- \in streams(l)} wcet_{stream}(s_i^-) \right)$$

This approach differs from the parallel approach only in summing up all execution times of the streams of a layer instead of determining the maximal execution time of a layer.

Example 4.2.5. Consider the RTLola specification from Example 4.2.4. We determine the WCET of this specification for a sequential execution first by summing all WCET of a layer:

$$streams(0) = \{s_1^{\downarrow}\} \Rightarrow wcet_{stream}(s_1^{\downarrow})$$

$$streams(1) = \{s_1^{\uparrow}, s_2^{\uparrow}, s_3^{\uparrow}\} \Rightarrow wcet_{stream}(s_1^{\uparrow}) + wcet_{stream}(s_2^{\uparrow}) + wcet_{stream}(s_3^{\uparrow})$$

$$streams(2) = \{s_1^{!}, s_2^{!}, s_3^{!}\} \Rightarrow wcet_{stream}(s_1^{!}) + wcet_{stream}(s_2^{!}) + wcet_{stream}(s_3^{!})$$

The final WCET of the sequential execution of the specification is:

$$\begin{aligned} wcet_{spec}^{sequential} &= wcet_{stream}(s_1^{\downarrow}) \\ &\quad + wcet_{stream}(s_1^{\uparrow}) + wcet_{stream}(s_2^{\uparrow}) + wcet_{stream}(s_3^{\uparrow}) \\ &\quad + wcet_{stream}(s_1^{!}) + wcet_{stream}(s_2^{!}) + wcet_{stream}(s_3^{!}) \end{aligned}$$

△

Remark 4.2.3. *As Example 4.2.4 and Example 4.2.5 show, the WCET analysis of a parallel model outperforms the analysis of a sequential model if the given specification has several streams in one layer. In fact,* $wcet_{spec}^{parallel} \leqslant wcet_{spec}^{sequential}$ *since the analysis of the parallel execution only results in the same outcome as the sequential one if all streams are on a unique layer.*

Chapter 5

Implementation

This chapter introduces two approaches for computing a WCET analysis for an RTLola specification. The first approach focuses its analysis on the intermediate representation of the RTLola framework. The second approach considers the front-end of the framework and the underlying assembly code of the RTLola interpreter.

5.1 Setup

The WCET analyses described in the following sections are implemented in Rust, and the implementation is published on *GitHub*[1]. The first analysis is based on the intermediate representation, which is created by the RTLola front-end [2]. The second one analyzes the assembly code of the RTLola interpreter [3]. For this thesis, the project was executed on an Intel processor using the x86 instruction set architecture with the Intel syntax. Therefore, we only consider the x86 code.

5.2 RTLola Framework

The RTLola framework is written in Rust and is divided into a front-end and several back-ends. Fig. 5.1 shows the structure of the framework. We have to provide an input file with the textual representation of an RTLola specification and an input file with the corresponding CSV file or the input from std-in. Each input stream is annotated with the time it gets updated with a value. Before the specification of a CPS is monitored, the front-end creates the Mid-level Intermediate Representation (MIR). Then, the specification is monitored and provides feedback to the system by releasing triggers.

[1] https://projects.cispa.saarland/c01jaba/rtlola-wcet
[2] https://docs.rs/rtlola-frontend/0.3.3/rtlola_frontend/
[3] https://docs.rs/rtlola-interpreter/0.7.0/rtlola_interpreter/

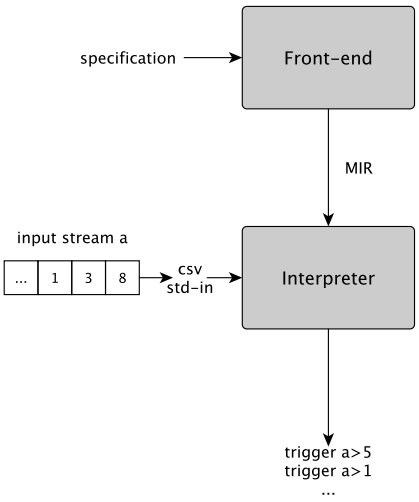


Figure 5.1: Overview of the connection between parts of the framework

5.2.1 RTLola Front-end

The front-end parses the RTLola specification and creates the MIR containing all input, output, and trigger streams. It also provides additional information about each stream like the induced type, the evaluation layer, the memorization bound, and which stream is accessed. The MIR forms the base for the first implementation of a WCET algorithm discussed in more detail in Sect. 5.3.

5.2.2 RTLola Interpreter

The interpreter implements a monitor, which analyzes the specification with the incoming data of the system as a CSV file or from the std-in. The interpreter gets the MIR from the front-end and interprets the specification based on the incoming data. In the scope of this thesis, we focus on the implementation of the online monitoring of event-based streams shown in Fig. 5.2.

The online evaluation process starts by fetching new events meaning that new values are received from the input streams. These events are divided into items interpreted by `eval_event`. Before the streams are evaluated, the function `clear_freshness` ensures that all streams are cleared before being evaluated. Then, the event must be incorporated into the input streams by accepting all values of the input streams the interpreter needs to evaluate the specification. Afterward, the function `eval_all_event_driven_streams` iterates over all output streams of each layer and computes the result of each stream if the event invokes its activation condition.

The execution of the function `eval_event` is the starting point of the interpreter analysis explained in Sect. 5.4

5.3 Frontend Analysis

To gain a more accurate timing constraint of the specification, the front-end analysis obtains all information and parameters encoded in the MIR and interprets them into a time value indicating the complexity of the evaluation of a specification. In this section, we implement the algorithm presented in Chapter 4 for a sequential and parallel model.

→ Chapter 4, p. 25

5.3.1 Implementation Details

Remark 5.3.1. *To reason about time, every operation is assumed to be atomic, and we assign the following variables all to one.*

$$streamAccess = 1 \qquad exec_{comp} = 1$$

$$offsetAccess = 1 \qquad exec_{arith} = 1$$

$$exec_{mem} = 1 \qquad exec_{ite} = 1$$

We split the analysis into an input and an output analysis. We iterate over all inputs and obtain the execution time for an input stream by computing the memory handling overhead with the memorization bound. To evaluate each output stream, we iterate over all evaluation layers and compute the execution time for the memory handling and the expression of an output stream. We recursively construct the result by mapping every operation and stream lookup to one and adding a possible overhead to compute the expression's result. Moreover, we analyze the output streams with the same structure the interpreter has. For the resulting WCET analysis for the parallel model, we compute the maximal result of all output streams of each layer. Considering a sequential execution, we sum up all results.

5.4 Interpreter Analysis

To obtain a more precise WCET estimation, we analyze the interpreter and the MIR. The time instance for this analysis is the number of assembly instructions needed to monitor an RTLola specification. Note that we are not interested in the initialization and compilation of the monitor but only in evaluating the given specification. Moreover, we do not distinguish between different assembly instructions.

Before analyzing any assembly code, the assembly code has to be generated. We analyze the compiled binary with the cargo asm tool explained in Sect. 3.3 to extract the unoptimized assembly code with the corresponding Rust code annotations. This tool only allows us to extract the assembly code for a specific function. Thus, we have to

→ Sect. 3.3, p. 22

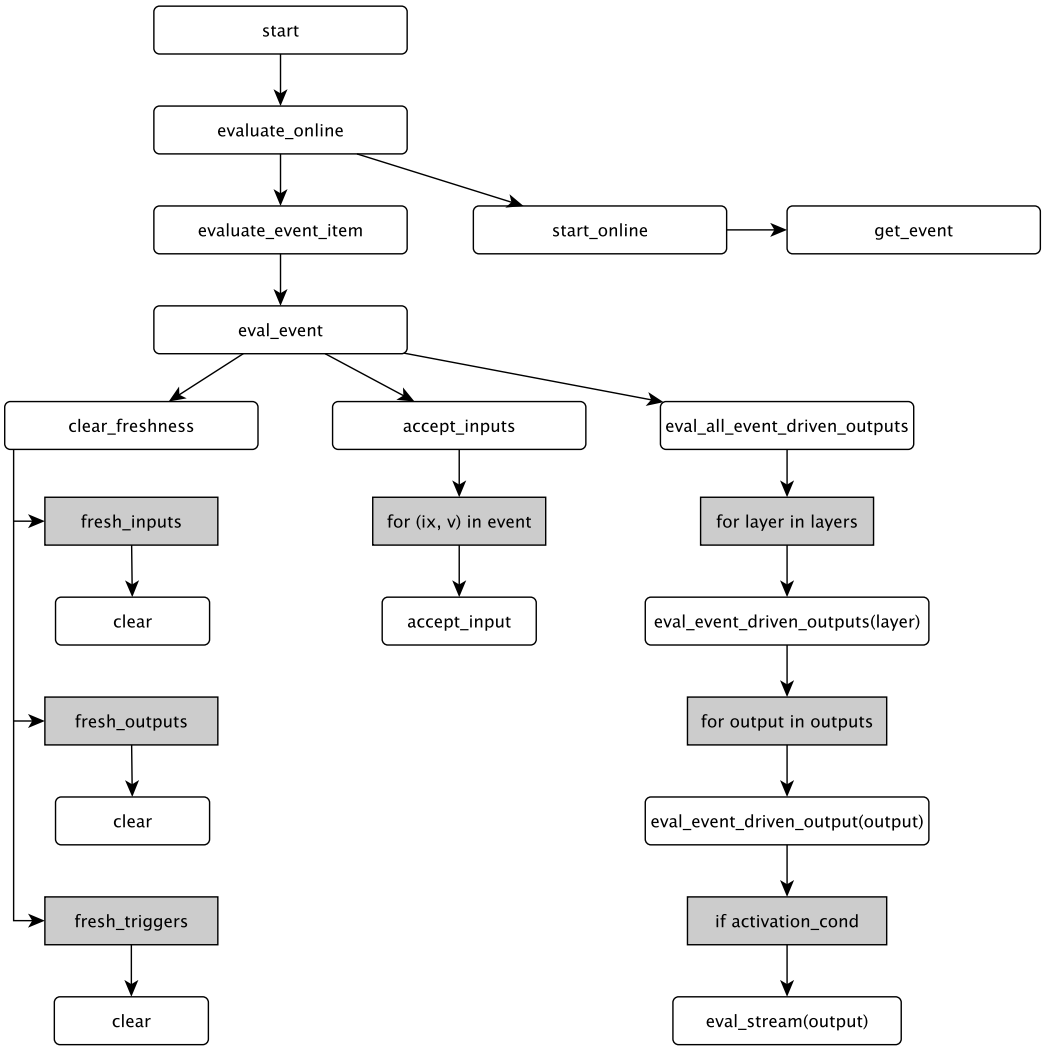


Figure 5.2: Partial CFG of the interpreter

build the control-flow graph for the interpreter as outlined in Fig. 5.2 to know which functions are relevant.

Remark 5.4.1. *We set the starting point of the analysis to the function* `eval_event` *since the evaluation of the specification starts after receiving an event.*

5.4.1 Assembly Code Analysis

We analyze the assembly code top-down by extracting the assembly code of the function `eval_event` and splitting the further analysis into three blocks. Each block represents the function being called in `eval_event`, which are `clear_freshness`, `accept_inputs`, and `eval_all_event_driven_outputs`. Then, we dissolve every function being called and extract their underlying assembly code. To keep track of the connection of the functions and which function we already processed, we depict the complete call history starting from `eval_event`.

We start bottom-up, annotating each function with its instruction number with the complete assembly code. Each assembly file must be analyzed for any loops, if-statements, or dead code. If there are any loops or conditionals, we annotate the instruction number by the corresponding parameter. If a function is called within a loop, we note how often the function is called. Note that all these parameters are known from the MIR, and in the WCET analysis, we replace them with the actual parameters from the specification.

Remark 5.4.2. *Annotating these parameters is not limited to the RTLola framework alone but is also applied to the standard libraries of Rust, as we have seen in Sect. 1.1. However, there are some limitations of the cargo asm tool, and not every function can be dissolved into its assembly code like for example,* `_memcpy` *or* `___rust_dealloc`*. All unresolved functions are handled as atomic parameters in the final calculation.*

→ *Sect. 1.1, p. 3*

5.4.2 Implementation Details

We only focus the WCET computation on a subset of the RTLola language consisting of stream accesses like synchronous lookups, hold lookups, past offset lookups, and loading constants. Further, we support simple arithmetic and boolean operations like addition, negation, conjunction, and if-statements. The only possible value types for streams are boolean values, unsigned and signed integer values, and floating values.

The evaluation of the WCET considers the three blocks we created during the assembly code analysis. The final result is composed of the instruction count of these blocks and the overhead from the function `eval_event`. All parameters like the number of inputs or evaluation layers annotated during the assembly code analysis are now replaced by the values placed in the MIR. While evaluating a stream, we can match all possible expressions constructions and compute an accurate WCET estimation for these streams.

5.4.3 Example

To gain more intuition on how the interpreter analysis works we we will discuss the WCET algorithm in more detail using the function `eval_event_driven_outputs`:

```
fn eval_event_driven_outputs(&mut self, outputs: &[OutputReference], ts: Time) {
    for output in outputs {
        self.eval_event_driven_output(*output, ts);
    }
}
```

The Rust code shows that we loop over all output streams and call the function evaluating this output stream. Consequently, we must annotate the number of instructions with the number of output streams in the assembly code analysis.

We extract the assembly code with cargo asm and get the resulting assembly code shown in Fig. 5.3. For simplicity, the move instructions in the beginning are omitted. The loop is formed between the labels `LBB8254_2` and `LBB8254_6`. By the jump instruction in line 18, the condition to enter the loop is checked whether all outputs have been processed yet. If not, the code within the loop is executed by jumping to the assembly code of label `LBB8254_6`. Analyzing the assembly code, we conclude that there are 32 instructions until the first output is evaluated. To evaluate a single output, 20 instructions have to be executed. Three instructions are needed to return to the caller function. In consequence, the number of instructions for `eval_event_driven_outputs` with *out* outputs is: $35 + out * 20$.

Moreover, the function has three calls in the assembly code that have to be resolved into their complete call history. To show such a call history, consider the call history of the iterator's `next` function:

```
ox <core::slice::iter::Iter<T> as core::iter::traits::iterator::Iterator>::next = 11
    + 3 + 44 // out*148 // out = number of outputs of a layer
  4x core::ptr::non_null::NonNull<T>::as_ptr = 8
  2x core::ptr::mut_ptr::<impl *mut T>::is_null = 16
     core::ptr::mut_ptr::<impl *mut T>::guaranteed_eq = 17
  core::ptr::non_null::NonNull<T>::new_unchecked = 9
```

Every function is annotated with the instruction number where the first two appear multiple times in the code. Additionally, we know that `next` is called within the loop *out* times. Therefore, the final instruction count for `next` is composed of the overhead of `next` itself and of all calls from the call history: $out * 148$.

To sum up all results, the instruction count of `next` has to be added to the overhead of the caller function as well as the two remaining function calls. Assuming that the number of `into_iter`=i and `eval_event_driven_output`=e the instruction number for `eval_event_driven_outputs` is: $35 + out * 20 + out * 148 + i + e = 35 + out * 168 + i + e$.

```
fn eval_event_driven_outputs(&mut self, outputs: &[OutputReference], ts: Time) {
    push rbp
    mov rbp, rsp
    sub rsp, 144
    mov qword, ptr, [rbp, -, 144], rdx ... (13 more mov-instructions)
for output in outputs {
    call core::slice::iter::<impl core::iter::traits::collect::IntoIterator for
        &[T]>::into_iter
    mov qword, ptr, [rbp, -, 104], rax ... (5 more mov-instructions)
LBB8254_2:
    lea rdi, [rbp, -, 88]
    call <core::slice::iter::Iter<T> as core::iter::traits::iterator::Iterator>::next
    mov qword, ptr, [rbp, -, 72], rax
for output in outputs {
    mov rax, qword, ptr, [rbp, -, 72]
    test rax, rax
    setne al
    movzx eax, al
    je LBB8254_4
    jmp LBB8254_8
LBB8254_8:
    jmp LBB8254_6
LBB8254_4:
}
    add rsp, 144
    pop rbp
    ret
for output in outputs {
    ud2
LBB8254_6:
    mov ecx, dword, ptr, [rbp, -, 108]
    mov rdx, qword, ptr, [rbp, -, 120]
    mov rdi, qword, ptr, [rbp, -, 128]
for output in outputs {
    mov rax, qword, ptr, [rbp, -, 72]
    mov qword, ptr, [rbp, -, 24], rax
    mov qword, ptr, [rbp, -, 16], rax
for output in outputs {
    mov qword, ptr, [rbp, -, 8], rax
self.eval_event_driven_output(*output, ts);
    mov rsi, qword, ptr, [rax]
self.eval_event_driven_output(*output, ts);
    call rtlola_interpreter::evaluator::Evaluator::eval_event_driven_output
for output in outputs {
    jmp LBB8254_2
```

Figure 5.3: Assembly code with rust annotations of `eval_event_driven_outputs`

Chapter 6

Case Study

This chapter discusses the results of the two proposed WCET approaches, the front-end analysis in Sect. 6.1 and the interpreter analysis in Sect. 6.2. Note that interpreting the abstract time metrics into real-time was not in the scope of this thesis. Therefore, the results presented in this section cannot be compared to real-time results.

6.1 Front-end Analysis

In this section, we perform the front-end analysis on several RTLola specifications. Most of these specifications are used as statistical input. Fig. 6.1a describes a specification where all output streams are in a unique layer and a specification where all output streams are in the same layer, as shown in Fig. 6.1b. These two specifications are then extended by the same pattern with 10 and 20 output streams. The results of the parallel and sequential model are:

Specification	Sequential Model	Parallel Model
linear dep 5	11	11
linear dep 10	21	21
linear dep 20	41	41
parallel dep 5	11	3
parallel dep 10	21	3
parallel dep 20	41	3

For the specification with linear dependencies, the results of both models do not differ. However, the difference for the highly parallel specifications increases the more output

```
input a: Int64

output b := a
output c := b
output d := c
output e := d
output f := e
```

(a) RTLola specification linear dep 5

```
input a: Int64

output b := a
output c := a
output d := a
output e := a
output f := a
```

(b) RTLola specification parallel dep 5

Figure 6.1: Statistical specifications

streams the specification has. Since in the specifications with parallel dependencies, the additional output streams have the same structure, the time needed to evaluate these streams is equal. Therefore, no matter how many output streams we add to the specification with parallel dependencies, the result will be equal to three.

In the following, we analyze the specifications from Fig. 6.1, but extend every synchronous lookup of each output stream with an offset lookup by one. For example, the output stream `output b := a` from Fig. 6.1a is transformed into `output b := a + a.offset(by: -1).defaults(to: 0)`. Analogously, all output streams are transformed. Again, we consider the specifications with five, 10 and, 20 output streams resulting in:

Specification	Sequential Model	Parallel Model
linear dep 5 with offset	26	26
linear dep 10 with offset	51	51
linear dep 20 with offset	101	101
parallel dep 5 with offset	22	6
parallel dep 10 with offset	42	6
parallel dep 20 with offset	82	6

The relation between the results of the parallel and sequential model is equal to the results before. However, the specifications covering the parallel and linear dependencies do not lead to the same result for the sequential analysis. The specifications with parallel dependencies do not need to store the offset values for the output streams since only the input stream `a` is accessed with an offset. Therefore the memory needed for the specifications with linear dependencies is higher since we have to store more values overall.

Throughout this thesis, we discussed two examples in more detail. Since the analysis does not support aggregations, we change from Example 3.1.1 the two output streams

→ Exm. 3.1.1, p. 12

involving the aggregations to `output too_low : Bool := height < 5.0`. The front-end analysis for these two examples results in:

Specification	Sequential Model	Parallel Model
Example 4.2.4	31	17
Example 3.1.1	27	18

→ Exm. 4.2.4, p. 31

→ Exm. 3.1.1, p. 12

Both specifications have a similar complexity and share a similar result.

6.2 Interpreter Analysis

In this section, we discuss the results of the interpreter analysis for the specifications covered in the last section.

Specification	Sequential Model
linear dep 20	110723
parallel dep 20	98126
linear dep 20 with offset	231078
parallel dep 20 with offset	214586
Example 4.2.4	62511
Example 3.1.1	57210

The second specification has a lower instruction number than the first one. This result differs from the front-end analysis, where the results were equal. This is due to the fact that in the assembly code, accessing an input stream is less costly than accessing an output stream. Since only synchronous lookups to the input stream are made in the second specification, the instruction count is smaller than the first one. The same observation is made from the specifications extended with an offset lookup. The two simple examples discussed in this thesis have a similar instruction count, as already indicated by the front-end analysis.

6.3 Conclusion

The computed results from both implementations represent the expected behavior of the specifications. In the worst-case, the results of the parallel model are equal to the results of the sequential model. This worst-case are specifications where all output streams are in a unique layer. The main result of these experiments is that the instruction

count increases with increasing complexity. Here, the front-end analysis serves as an indicator for the complexity and indicates if the specification is highly parallel or consists of output streams with linear dependencies. Moreover, the interpreter analysis refines the result of the front-end analysis by considering the RTLola interpreter.

Chapter 7

Conclusion and Future Work

In this thesis, we presented a WCET algorithm for RTLola with two different implementations: the front-end analysis and the interpreter analysis. The first one is based on the intermediate representation of RTLola and analyzes the parallel and the sequential model. The analyses of both models analyze the specification for its dependencies resulting in a time metric expressing the specification's complexity.

The second approach is based on the assembly code of the RTLola interpreter considering the sequential model. First, the complete assembly code is unrolled by mapping each function to its assembly code. Then, based on the intermediate representation, the assembly code is chosen, which is executed for the specification. This analysis provides a result for the first step of a more sophisticated WCET analysis.

The WCET algorithm could be extended by the full RTLola language in future work. Supporting sliding windows, aggregations, and real-time streams would significantly increase the expressiveness of RTLola. This way, more practical relevant requirements of CPS can be expressed and analyzed.

Further, the abstract time metric used by the two approaches could be translated into a real-time context. This could be done by mapping the instructions to its WCET on specific hardware where the monitor is run.

Moreover, one could estimate the WCET with a measurement-based analysis by measuring the time for each function before it is called and after the function is executed. These results can be merged similar to the interpreter analysis into a final result.

Bibliography

[1] Reinhold Heckmann and Christian Ferdinand. 2004. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004, pages 26–30. IEEE Computer Society*.

[2] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia* (OASICS). Jan Reineke, editor. Volume 57. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:12. DOI: 10.4230/OASIcs.WCET.2017.8. https://doi.org/10.4230/OASIcs.WCET.2017.8.

[3] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46, (October 2010), 251–300. DOI: 10.1007/s11241-010-9101-x.

[4] Niklas Holsti and Sami Saarinen. 2002. Status of the bound-t wcet tool, (January 2002).

[5] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7, (January 2008).

[6] E.Y.-S Hu, Guillem Bernat, and Andy Wellings. 2002. A static timing analysis environment using java architecture for safety critical real-time systems. In (February 2002), 77–84. ISBN: 0-7695-1576-2. DOI: 10.1109/WORDS.2002.1000039.

[7] Björn Lisper. 2014. SWEET - A tool for WCET flow analysis (extended abstract). In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part II* (Lecture Notes in Computer Science). Tiziana Margaria and Bernhard Steffen, editors. Volume 8803. Springer,

482–485. DOI: 10.1007/978-3-662-45231-8_38. https://doi.org/10.1007/978-3-662-45231-8%5C_38.

[8] Yau-Tsun Steven Li and Sharad Malik. 1997. Performance analysis of embedded software using implicit path enumeration. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 16, 12, 1477–1487. DOI: 10.1109/43.664229. https://doi.org/10.1109/43.664229.

[9] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. 2006. Towards formally verifiable WCET analysis for a functional programming language. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany* (OASICS). Frank Mueller, editor. Volume 4. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2006/677.

[10] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. 2009. ALF - A language for WCET flow analysis. In *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009* (OASICS). Niklas Holsti, editor. Volume 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/2279.

[11] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 46–57. DOI: 10.1109/SFCS.1977.32.

[12] Andreas Bauer, Martin Leucker, and Christian Schallhart. 2011. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20, 4, 14:1–14:64. DOI: 10.1145/2000799.2000800. https://doi.org/10.1145/2000799.2000800.

[13] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2, 4, 255–299. DOI: 10.1007/BF01995674. https://doi.org/10.1007/BF01995674.

[14] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 166–174. DOI: 10.1109/TIME.2005.26. https://doi.org/10.1109/TIME.2005.26.

[15] Sebastian Schirmer. 2016. *Runtime Monitoring with Lola*. Master's Thesis. Saarland University.

[16] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. 1987. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '87). Association for Computing Machinery, Munich, West Germany, 178–188. ISBN: 0897912152. DOI: 10.1145/41625.41641. https://doi.org/10.1145/41625.41641.

[17] N. Halbwachs. 2005. A synchronous language at work: the story of lustre. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.* 3–11. DOI: 10.1109/MEMCOD.2005.1487884.

[18] Maximilian Schwenger. 2019. *Let's not Trust Experience Blindly: Formal Monitoring of Humans and other CPS*. Master Thesis. Saarland University.

[19] Stefan Stattelmann. 2009. *Precise measurement-based worst-case execution time estimation*. eng. Universität des Saarlandes, Saarbrücken.