UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Parallel Analysis of Large Traces with Stream Based Specifications

Master Thesis

for obtaining the Master of Science (M. Sc.) in
Computer Science
at the University of Lübeck

written by
René Kremer

supervised by
Prof. Dr. Martin Leucker

assisted by
Torben Scheffel

Lübeck, 19. December 2019

Preface

Before you lies the thesis *Parallel Analysis of Large Traces with Stream Based Specifications*, which has been written to fulfil the graduation requirements of the Masters Curriculum of Computer Science at the University of Lübeck and states my thoughts on the question: How can parallel programming approaches be applied to Runtime Verification engines?

My interest for Software-Verification started while I was studying for my bachelor's degree and I became more interested in the field as I was confronted with the problems raised as part of the Model Checking and Runtime Verification courses in my Master's Program. Therefore I would like to thank Prof. Martin Leucker for introducing me to a more in depth view of the field.

The research question was formulated by Malte Schmitz and I would like to thank him, Torben Scheffel and Sebastian Hungerecker for their guidance throughout the writing of the whole thesis, their helpful thoughts but also support on getting my teeth into *TeSSLa*, which originated at the Institute for Software Engineering and Programming Languages of the University of Lübeck.

Before you dig into this analysis I would like to thank my friend Kilian Cohrs for his time to debate about this topic while not being deep into this specific field and also my girlfriend, Franziska Gläser, and parents, Uwe and Barbara Kremer, for being a big support and motivation during this whole process.

I hope you enjoy reading and that you might gain helpful insights out of this work.

René Kremer

Lübeck, 19. December 2019

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(René Kremer)
Lübeck, 19. December 2019

Kurzfassung Runtime Verification (RV) ist in der Lage den Lauf eines Programmes über seinen Trace zu überwachen. Genau wie RV ist Stream Runtime Verification in der Lage nicht nur logische Eigenschaften sondern auch statistische Metriken zu überprüfen. Dafür wird der Trace als Eingabestrom behandelt und daraus Zwischen- und Ausgabeströme abgeleitet.

Diese Arbeit widmet sich der Frage, wie Parallelverarbeitung große Traces von strombasierten Spezifikationen verarbeiten kann. Der vorgestellte Ansatz basiert auf der Temporal Stream-based Specification Language (TeSSLa) und analysiert verschiedene Fälle von Spezifikationen und Algorithmen.

Die Ergebnisse dieser Algorithmen werden anschließend im Vergleich zu einem sequenziellen Ansatz evaluiert und zeigen eine Steigerung hinsichtlich Verarbeitungsmenge von Ereignissen und Leistung. Während der Evaluation der Parallelverarbeitung wurden auch einige Limitationen sowie die Signifikanz der Größe von Chunks und Heap hinsichtlich der Leistungssteigerung und der Verarbeitungsmenge von Ereignissen beobachtet. Die Korrektheit des parallelen Ansatzes wird über einen Abgleich der Ergebnisse mit dem sequentiellen Ansatz bewiesen.

Abstract Runtime Verification (RV) is able to monitor a program by traces of its run. Just like RV, Stream Runtime Verification (SRV) is able to check logical properties from the trace but also statistical metrics. This is done by handling the trace as input streams and derive intermediate as well as output streams.

This work is concerned with the question of how parallel computation can process large traces of stream based specifications. The presented approach is based on the Temporal Stream-based Specification Language (TeSSLa) and analyses different cases of specifications and algorithms to handle them.

The results of those algorithms are then evaluated in comparison to a sequential approach and show improvement in terms of speed-up and event throughput. While evaluating parallel computation of large traces some limitations were observed but also the importance of chunk and heap size for improving speed-up and event throughput. The proof of correctness of the parallel approach is provided by an equality check with the sequential trace result.

Contents

1 Introduction

Computer systems are more ubiquitous than ever in our everyday life. With computers surrounding us everywhere, it is easier than ever to stay in touch with friends and family, organize the daily life or participate in a community while being mobile. But not only consumer electronics contain a vast amount of software, but also transport machinery and medical devices for example.

In the age of Internet of Things (IoT) [AIM10] devices, starting with lamps to televisions and robots for cleaning or mowing the lawn, are getting more and more aware of their surroundings. Sensors in IoT devices and other machinery gather large amounts of data every second and a system or machine needs to filter, store and evaluate the data to act upon their environment. With increasing network bandwidth and coverage of internet connectivity more data than ever is gathered and processed via distributed systems and used to improve software or enable a system to make the right decisions. In case of transportation machines (like automobiles, ships or airplanes) the data could be used to automate the driving process to a certain degree by using an autopilot or at least give an assistance to the driver via surveillance and warnings.

Such a transport machinery and its subsystems (e.g. a braking or airbag system inside of a car) are safety-critical systems and a failure in those systems could lead to severe damage on the environment or people's health. But also systems that are directly affecting the health of a person like medical, life supporting systems (e.g. heart-lung machines or pacemaker) are safety-critical. For these kinds of systems standards exist (in case of automotive ISO 26262[1] and IEC 62304[2] for medical devices) that describe an approach to carefully develop, inspect, document, test, verify and analyse these systems, so that the risks of failure and their severity are known and minimized. Therefore verification techniques are necessary to provide formal proofs that these designed systems are correct and functioning.

Safety-critical systems and their sensors benefit from the trend of higher computational power via computer clusters, computing cloud engines and lower cost for data storage. On the other hand this means that safety-critical systems are more often designed as distributed systems, which lead to having increased complexity

[1] see `https://www.beuth.de/en/standard/iso-26262-6/300423967`
[2] see `https://webstore.iec.ch/publication/22794`

and also higher demand for data. This results in a higher risk of software failures due to the increasing complexity, but also provides more resources for testing and analysing the behaviour of the system. Different verification techniques can provide, in comparison to *testing*, a proof to a correctness of theorems and software systems. Runtime Verification (RV) is one of the techniques used at runtime and checks a run of a system against specified properties, therefore detecting incorrect states of the system and unexpected behaviours. It also allows to act whenever an incorrect state of the system is detected by using design patterns like runtime reflection. [LS09]

Temporal Stream-based Specification Language (TeSSLa) is a stream based specification language used for runtime verification that takes input streams of a given input trace and, while checking on logical properties of a given specification, derives output streams. Due to TeSSLas nature it can generate intermediate streams that allow for computing temporal metrics and statistics. [CHL+18] As a runtime verification engine TeSSLa is able to process large batches of input data, but due to its current implementation is limited to a single thread for computation.

Offline analyses of large amounts of data (e.g. as logs or data streams in size of tera- or petabytes) need high computational power. Traces, like logs or data streams in this context, can contain for example recorded values of sensors or computation results and are basically the output a program generates. Processing these traces can be done with single-threaded programs, but as described in [TSBR18] parallel programs might speed up the time needed to solve problems if one can design parallel algorithms for them. By designing such parallel algorithms one can use the benefit of multi-core CPUs or computing nodes in a distributed system. One example for such an algorithm is the *MapReduce* algorithm. [DG04]

Based on the TeSSLa specification language and engine, this work will analyse the question: *How can parallel programming approaches be applied to such a runtime verification engine?* The insight gained from this work might enable parallel programming on the topic of large stream-based runtime verification and therefore make it more applicable to the verification process of safety-critical systems and increase the reliability of especially such software systems in general by checking against correctness properties like real-time constraints or heat values of a temperature sensor.

1.1 Related Work

This work is based on TeSSLa as described in [CHL+18] – in comparison to LOLA in [DSS+05] – because it allows for asynchronous Stream Runtime Verification (SRV) while granting (un)bounded data types which allow for low memory consumption.

This is especially helpful in embedded systems due to limitations of the used hardware. Another benefit of such a specification language like TeSSLa or LOLA is that it is easier to use and write compared to logics like Linear Temporal Logic (LTL) introduced in [Pnu77].

Regarding parallel programming [Kum02] introduces different kinds of parallelism like task and data parallelism and their problems as well as limitations. Furthermore decomposition techniques on how to split tasks and data for parallelization are described. This work introduces *task parallelism* as an alternative to *data parallelism* but will focus more on *data parallelism*. Decomposition techniques are restricted to *data decomposition* and *speculative decomposition*. Techniques such as *exploratory* and *recursive decomposition* are not relevant in this work and will not be part of it.

In [DG04] the MapReduce algorithm is introduced. As part of the main question of how such a parallel algorithm can be used to process large traces the MapReduce algorithm is a reference on handling large files on single computing nodes but also on clusters of nodes. Therefore the MapReduce algorithm will be shortly introduced in Subsection 3.4.5 but for further insight one should read [DG04]. The implementation presented in this work will be a proof of concept. It will relate to the MapReduce algorithm but not use a framework like *Apache Hadoop* to allow for a flexible customization of the algorithms needed to solve the problem of processing large traces.

[BCE+14] shows different *slicing strategies* for metric first-order temporal logic (MFOTL) and how to apply them on MapReduce frameworks. To apply those slicing strategies on the MapReduce approach different *configurations* containing *slicing functions* were necessary. It was also discovered that most slices were between 61 and 135 MB but also that nested operators substantially increased the time needed to process. Compared to [BCE+14] this work will focus on slicing functions for different SRV cases based on TeSSLa specifications.

In [SBB+18] the presented slicing framework of [BCE+14] was applied to online rather than offline monitoring using sub-monitors and the stream processing framework *Apache Flink*. The base for the splitting strategy was the *hypercube algorithm* [Ost87]. It was shown that the sequential Splitter was a bottleneck. The proposed solution was a parallel splitter to further improve the event throughput. A restriction for the usage of a parallel splitter is the need to chronologically order incoming events. It is necessary to have the split events ordered for processing because of online monitoring and the verification of the system under scrutiny at runtime. This work will not use sub-monitors for offline monitoring but the full capability of TeSSLa. Nonetheless the bottleneck of a sequential splitter, it's influence on the performance and possible solutions will also be discussed in this work.

The author of this work is aware of [HKG17] and its implementation of a multi-threaded approach for *BeepBeep*. The difference is that [HKG17] discusses an implementation around pipelining streams on multiple CPU processor instances while this work covers different slicing functions for TeSSLa specifications based on a MapReduce like approach on a local machine with possible portability to a MapReduce or similar distributed processing framework like *Apache Hadoop*.

1.2 Structure of this Work

Besides this introduction and the conclusion at the end, the thesis will consist of the following chapters:

Chapter 2 will compare existing verification techniques and motivate the parallelization of a runtime verification engine based on a TeSSLa example.

Chapter 3 will clarify concepts, terms and basics which are necessary to follow this thesis. The focus will be on an overview of verification techniques and parallelization concepts as well as TeSSLa.

Chapter 4 will present examples of specifications and traces as different cases of the parallelization question. These cases will be investigated and possible approaches derived and discussed.

Chapter 5 describes the prototypical implementation of the approaches mentioned in Chapter 4 in Scala for TeSSLa and shows how the concepts for parallelization might be put to practical use.

Chapter 6 compares each parallel approach with the single-threaded solution and thus shows how effective a parallelization in different scenarios is as well as the limitations.

2 Motivation

As mentioned in Chapter 1 a failure in a safety critical system (like an automobile or medical device) could lead to severe damage in the environment or to people's health. Therefore it is essential to prove the correctness of such systems to minimize risk of a failure.

In the context of software there are different techniques to verify that a system works as intended. One such technique is *software testing*. Testing is an activity that examines the software under development and is a primary method in the industry. Based on the specifications and requirements of the software under development tests are designed to be executed against the software. The results of this execution are then reviewed by software testers or software developers to determine faults in the software. [AO08]

One of the goals of software testing is to find incorrect behaviour and ways to fix their origin. There are different test activities for each software development phase as shown in Figure 2.1. This will not be explored in detail, but shows the depth of software testing. Coverage criteria are introduced to determine which input is used and most likely to find failures, as software testing is not exhaustive and ranges from large to effectively infinite. [AO08].

On the downside, as the limitation stated in [AO08] and pointed out by the famous citation of Edsger W. Dijkstra in [Dij72]: "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence." suggest, finding all failures in a program with tests alone is no easy task and impossible.

A more formal way to check the correctness of a program is by using a mathematical proof. [LS09] To show the correctness the mathematical steps are broken into logical steps, which are checked all the way back to the axioms of mathematics. This is manually done and can be assisted by theorem provers. In the last decades many theorem provers use higher-order logic (HOL) to assist mathematicians and computer scientists in their work of *theorem proving*. [Hal08]

In comparison to *theorem proving* and *software testing*, *model checking* (MC) is a model-based verification technique that describes the systems behaviours in a model of the system, which represents the states of the system in a mathematically precise way. [BK08] The modelling of the system itself might lead to the discovery

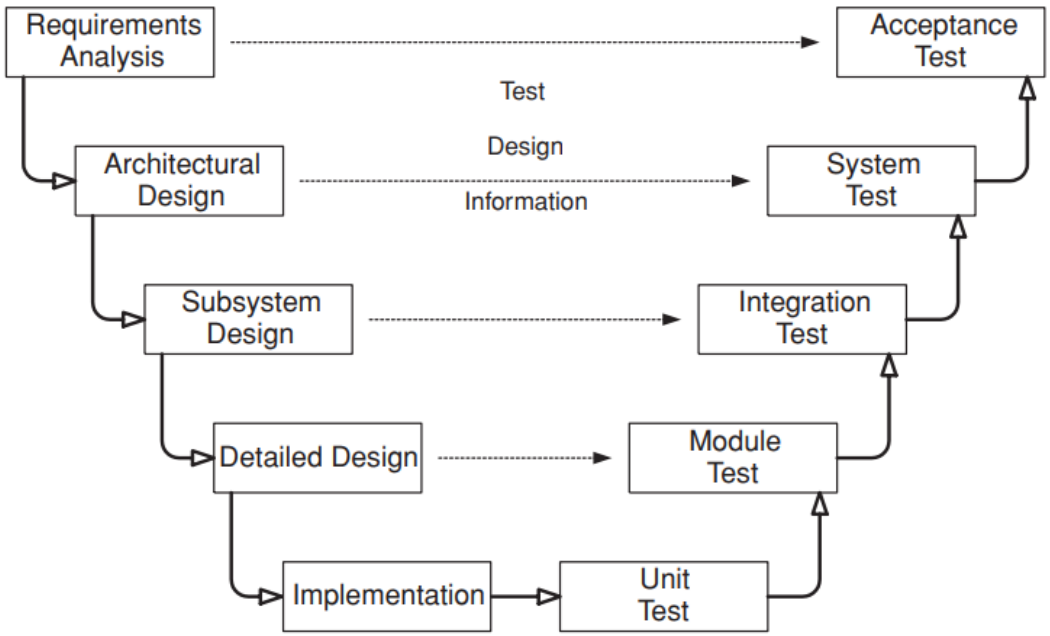

Figure 2.1: Software development activities and testing levels - the "V-Model" as shown in [AO08, p. 6]

of incompleteness of the system or specification, because the person designing the model might find insufficient correctness properties or functions of the system fulfilling those properties. MC explores all possible system states of the model in a brute-force manner. [BK08] Therefore MC finds incorrect states regarding the specification, if those states exist. Properties of a modelled system can also be checked for qualities like *is the result ok? Could a state lead to a deadlock? Does the system responds within 8 minutes of a request?*

The problem is that a very realistic model of the system is needed as the verification of the *system under scrutiny* is only as good as the model of it. The benefit of model checking is that it is easy to examine states that violate a property. As the states are checked against different variations of the states of the system, if a state violates a property, a counterexample to the expected behaviour is found. [BK08]

As the verification is just as good as the model of the system complementary techniques are necessary to find hardware faults or errors in the model and therefore software. MC is also in general not effectively computable as it is subject to decidability issues due to infinite-state systems or deciding about abstract data types with undecidable or semi-decidable logic. Another big issue is the *state-space explosion* problem, meaning that the number of states needed to model the system requires more space than the computer memory might provide. [BK08]

A more *lightweight* verification technique is *runtime verification* (RV) which detects violations or satisfactions of correctness properties by observing the execution of the software. In RV a *monitor* is used to read a finite trace of an execution of the program. The monitor results in three different truth values (true, false, inconclusive) to indicate whether or not a correctness property is violated. RV will be described in more detail in Section 3.1.

Proving the correctness of a safety-critical system is possible with the aforementioned techniques. Some might be used in combination and model checking in particular provides a heavy analysis of the models and thus the systems behaviour. In terms of hardware MC is limited. RV is a compromise between software testing as well as theorem proving and checking all states contrary to MC only a trace of an execution of the program is checked against the specification. Using a design pattern like *Fault-Detection, Fault-Isolation and Recovery Techniques* (FDIR) the RV monitor can be used as a watchdog and as it is not part of the program it can react based on incorrect behaviours. This reaction can be for example a restart of the system or a warning message to the user. [LS09]

The correct behaviour of a program or system does not need to be defined only based on a single property being true or false but could also be based on some statistical measurement. One such example could be a freezer that mustn't have an inner temperature above a certain degree Celsius over time. A simple check if the temperature hits that threshold might not be sufficient as due to the fine regulation of the freezer it might sometimes hit that threshold for some milliseconds. One could measure the average temperature over time and determine whether or not the inner temperature of the freezer was in average sufficient or just use the average temperature as a reference value for further processing. Regardless, the possibility of aggregating statistical measures as part of RV allows for more detailed verification of correctness properties. SRV allows to aggregate these measurements by using streams consisting of multiple events to a certain time. SRV will be described in more detail in Section 3.2.

TeSSLa is a SRV language able to build an SRV monitor based on a given specification. Due to its current implementation the processing capability is limited and therefore large traces can't be processed efficiently. Many sensor values are gathered in tests and need to get analysed. In case of automobiles this might lead to hundreds of sensors gathering data every millisecond, so the size of data is easily in the range of terabytes. Analysing this amount of data in a sequential way takes a huge amount of time (weeks to maybe months), which could be reduced by using parallel methods to analyse the data and in consequence the *system under scrutiny*.

The question for this work is how a parallel algorithm such as MapReduce could be applied to a runtime verification engine. The goal is to decrease the time needed to

compute a qualitative and quantitative verdict based on a given specification and input trace.

Parallelization, as later shown in Section 3.4, has its own limitations and problems. It is often not an easy task to split input or operations to process them concurrently. The task of finding suitable cut points in the input trace to chunk it, is further called *intelligent chopping* in this work. In terms of automobile data, lets imagine the test data is gathered while driving, in which the data is stored dynamically on multiple hard disks. The data is split naturally due to the fact that at some point a hard disk is full and the next one is used for writing data. This means that the input trace could not only be classified by size, but also by number of files.

The size of traces could be *in-memory size*, *single-hard-disk size* and all above a size of terabyte(s) usually leads to *multiple-hard-disk size*. Regardless of the size there could be reasons why multiple trace files have to be written at runtime of a program. The size of a file might indicate why the file is chunked in the first place, but might not be the only reason. That means that a trace can be a *single-file trace* or *multi-file trace* regardless of size.

If a trace is stored on a single hard disk, the data is usually ordered and sorted inside of the file. Therefore *intelligent chopping* is needed to figure out where a cut is possible to get all the necessary dependencies for the parallelization task. In case of multiple files (on even multiple hard disks) one could assume that there is a natural ordering due to the writing process. Even in that case *intelligent chopping* is needed to find points to cut, because the natural ordering does not necessarily mean that each end of a disk is a good point to split the input trace on. It heavily depends on the data and its dependencies as later shown in Section 3.4.4.

The level of parallelization can reach from one machine with a multi-core CPU to multiple computation nodes in a network. This work will focus on an approach one could think of as a local MapReduce first and then show based on a prototype how this approach can be distributed on multiple machines.

MapReduce partitions the input data to schedule the execution of tasks, which contain those data partitions, on multiple machines. Parallelization is therefore done via data parallelism and data decomposition, as later described in Subsection 3.4.1 and Subsection 3.4.4.

The first problem is to find a partition function that splits the input files into pieces that can be used in the mapping and reducing step. Another problem is that based on the specification the map and reduce function needs to be defined. A *recursive equation*, as later described in Section 3.3, in a specification might require a different approach than a non *recursive equation* or reference to the past via the *last* operator.

In Section 4.3 different cases will be shown and explained but to clarify the previous statement two examples of Section 3.3 and Section 3.1 will be explained here. Based on the task at hand – checking a temperature in TeSSLa as shown in Figure 2.2 and calculating the average temperature shown in Listing 2.1 – one can see that those tasks are different and might conclude that parallel processing is also different for both cases.

$$low := temperature < 3$$
$$high := temperature > 8$$
$$unsafe := low \vee high$$

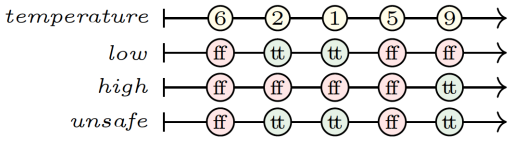

Figure 2.2: A basic example of derived streams in TeSSLa as described in [CHL+18]

To get the average temperature it is necessary to compute all values and at the end divide them by the number of events. Splitting the input values can be done freely, but mapping and reducing needs to ensure that each chunk of input results in a sum of temperatures and count of events. Thus the mapping function needs to know which values are needed in the reducing step and therefore need to be stored. In the reducing step the sum and count of events for each chunk have to be added to the previous sums and counts to get to the final average temperature of the whole input trace. To get the final average temperature the reducing step could run the result of each chunk against a sub-specification which computes the average temperature.

```
1 in temperature: Events[Int]
2
3 def count: Events[Int] := merge(last(count, temperature) + 1, 1)
4 def sum: Events[Int] := merge(last(sum, count) + temperature, temperature)
5 def avg: Events[Int] := if (count > 0) then sum / count else 0
6
7 out sum
8 out avg
```

Listing 2.1: A TeSSLa specification to get the average temperature

In comparison the check whether a temperature is lower than 3 or higher than 8 (Figure 2.2) does not need special chopping algorithms and neither a complex mapping or reducing phase. Each time step is independent thus each chunk is independent. The mapping just needs to run the input chunk against the specification while the reducing will put all results together. The mapping and reducing phase do not need to store meta information as in the previous example for calculating

an average temperature. Similar cases are shown in Section 4.3 as specifications without dependencies and self-dependency.

Another question could be how one needs to chop if each chunk needs the results of the previous one. This is the main problem of Subsection 4.3.4. In that specific case multiple streams are needed to compute the result. The problem is that based on the cutting point for the chunks a chunk may not have a start event for each stream. In Subsection 4.3.4 two approaches will be discussed. One of them is *speculative decomposition*, as it will be described in Section 3.4.4. Speculative decomposition will be used to compute the result of the next chunk based on multiple possible pre-results. After receiving the previous chunks result all wrong possibilities will be discarded.

These problem descriptions imply that in this work we have a look at each identified case individually. The cases which are part of this work are examples and might not be exhaustive.

The general problem can be summarized into these sub-problems:

1. Identify the case of specification
2. *Intelligent Chopping:* Partition the input trace according to partition function for that specification case
3. Apply map and reduce algorithms for the identified specification case

In step 3, the application of map and reduce algorithms does not have to happen on the same machine, but could also be done on different computation nodes in a bigger distributed system.

3 Preliminaries

This chapter introduces concepts and terminology necessary for the following approach described in Chapter 4 and its implementation in Chapter 5. The beginning of the chapter will start with a short overview of runtime verification and stream runtime verification as well as specifics of the used specification language and RV engine called *TeSSLa*. This chapter also introduces programming approaches as well as the MapReduce algorithm which are used and discussed in the following chapters.

3.1 Runtime Verification

This section references mostly [LS09] and will otherwise explicitly cite another work.

Runtime Verification is a verification technique although it is a bit more *lightweight* than MC. It itself only detects violations or satisfactions of correctness properties. This means that runtime verification does not influence the execution of the software, but observes the software only.

In this work the following definitions are used in the context of RV:

Definition 3.1 (Runtime Verification)**.** Runtime verification is the discipline of computer science that deals with the study, development and application of those verification techniques that allow checking whether a run of a system under scrutiny satisfies or violates a given correctness property.

Definition 3.2 (Run of a System)**.** A run of a system is a possibly infinite sequence of the system's state, which are formed by current variable assignments, or as the sequence of actions a system is emitting or performing. Formally, a run may be considered as a possibly infinite word or *trace*.

Definition 3.3 (Execution of a System)**.** An execution of a system is a finite prefix of a run and, formally, it is a finite trace.

Theorem Proving and Model Checking – and thus static verification methods – are interested in answering the question if all (infinite) runs of a system satisfy the correctness properties and therefore the specification. On the other hand, RV is interested in answering the question if a single run (execution) of the system satisfies the specification. In that regard RV can be viewed as an extension of testing with more powerful specification languages. [DSS+05] To accomplish the task of checking an execution of the system against a given specification a *monitor* is used which observes the execution and decides if the correctness properties are satisfied.

A *monitor* is defined as a device that reads a finite trace and yields a certain *verdict*. A verdict is a truth value from some truth domain e.g. the standard two-valued truth domain $\mathbb{B} = \{true, false\}$. If a monitor is checking the current execution of a system it is called *online monitoring*. The monitor is incrementally fed with the statements of the execution. On the other hand, if a monitor works on (finite) recorded executions, e.g. logs as traces, it is called *offline monitoring*.

A monitor should consider *impartiality* and *anticipation*. *Impartiality* means that a finite trace is not evaluated to true or false if there still exists an (infinite) continuation that might change the verdict. *Anticipation* on the other hand means that once every (infinite) continuation of a finite trace leads to the same verdict, then the finite trace leads to that verdict as well.

Therefore a monitor should have three different truth values: *true*, *false*, *inconclusive* with inconclusive being used if neither an answer to impartiality nor anticipation is found yet.

3.2 Stream Runtime Verification

As shown, RV uses a Monitor to yield a verdict over a programs execution checked against correctness properties noted in a specification. SRV and languages like *LOLA* and *TeSSLa* focus on enrichment of the monitors functionality while writing specifications within this language is easier for engineers compared to using logics like LTL as described in [Pnu77]. [DSS+05, CHL+18]

The enrichment of the monitors is achieved by allowing another type of property. In RV correctness properties are specified while in SRV one can also specify statistical measure properties. [DSS+05] Specifications are accepted as sets of stream expressions and the verification engine is run on a set of input streams. A stream in this context is a finite sequence of values with each value being mapped with a timestamp. While running the monitor, multiple intermediate streams might be generated to obtain the output streams. These output streams contain the desired information about satisfying correctness properties and quantitative measures. This

allows for other usages besides bug-finding, namely profiling, coverage and other analyses based on the statistical measurements like counting, minima, maxima etc. [DSS+05]

An example shall clarify what kind of enrichment is achieved by SRV. Consider a specification with *temperature* being an input stream of integers. *low* is checking whether the value of *temperature* is below 3 while *high* checks whether the value is above 8. A temperature is *unsafe* in the specification if either *low* or *high* are true. This example is shown in detail in Figure 3.1. The specification can validate a given run of a program and state if there are times where the temperature is unsafe or not. SRV can enrich these statements by measuring more statistical means. Without going into further details regarding the syntax, Listing 3.1 shows that the number of temperature events can be counted alongside the sum of all temperatures to get the average temperature. In some cases one might want to check a correctness property not only against the current value but statistical values.

```
1 in temperature: Events[Int]
2
3 def count: Events[Int] := merge(last(count, temperature) + 1, 1)
4 def sum: Events[Int] := merge(last(sum, count) + temperature, temperature)
5 def avg: Events[Int] := if (count > 0) then sum / count else 0
6
7 out sum
8 out avg
```

Listing 3.1: A TeSSLa specification to get the average temperature

Monitoring in this context is distinguished between *online* – monitoring that happens at runtime – and *offline* – monitoring that happens on a recorded trace on a hard drive. [DSS+05]

This numeric data provided by evaluating streams of the past and the future allows for, amongst other things, *context-free* properties like checking if after memory allocation memory is freed exactly once. [DSS+05]

This work focuses on offline monitoring and the parallelization of it by using TeSSLas language specifics as an example.

3.3 TeSSLa

This section references mostly [CHL+18] and will otherwise explicitly cite another work.

As mentioned in Section 3.2 RV can be used offline on previously recorded traces or online to evaluate correctness properties at runtime of the system under scrutiny. As also mentioned in Section 3.2 SRV also allows for quantitative measures.

TeSSLa is described as a language tailored for SRV of cyber-physical systems where timing is critical. For that purpose TeSSLa supports timestamped events natively as well as recursive definitions.

TeSSLa has six elementary operators that are used to provide more advanced operations like *merge*. Namely those operators are *time*, *last*, *lift*, *delay*, *nil* and *unit*. *nil* and *unit* are used internally by the TeSSLa compiler and not relevant in this work. While the *time* operator allows to get the timestamp of an event the *last* operator allows access to the previous value of the stream. New timestamps can be created by the *delay* operator which periodically creates an event unless a reset event occurred. The *lift* operation lifts a function f from values to streams. An example of such a function f could be *merge* or typical mathematical functions and operators. In Figure 3.1 the greater or lesser operators for example are lifted internally so that *low* or *high* checks the *temperature* stream against the condition and not only the value. The same applies to the *or* operator in *unsafe*.

$$
\begin{aligned}
low &:= temperature < 3 \\
high &:= temperature > 8 \\
unsafe &:= low \vee high
\end{aligned}
$$

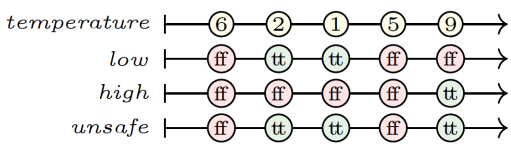

Figure 3.1: A basic example of derived streams in TeSSLa as described in [CHL+18]

Input measurements are generated streams based on the given specification but new (intermediate) streams could also be derived while computing. A stream in TeSSLa may consist of multiple events which each have a value and timestamp. In Figure 3.1 the *temperature* is checked. *low* returns true or false if the temperature is below 3 and *high* returns true or false if the temperature is over 8. Those two streams are generated based on the input measurement, the temperature. *unsafe* on the other hand is a derived stream of the streams *low* and *high*.

TeSSLa can process *asynchronous streams*. It is required that the events of all streams are in a global order but not all streams need to have synchronous events. For this purpose each event has a timestamp in the time domain $\mathbb{T}$ which contains positive numbers in $\mathbb{R}$. Therefore a stream may contain infinite events. Asynchronous streams allow that at a timestamp t only one stream has a new event and causes not all streams to respond with an event. This asynchronous behaviour is more suitable for cyber-physical systems that receive or raise signals at unstable

frequencies. On receiving a new event on a stream a new signal is generated. If a property needs to evaluate a stream and there is no new signal on that stream at the latest timestamp the last known value of the stream is used. The signal is therefore held until a new signal arrives just like a rising or falling edge in electronics. This is called *signal semantics*.

Like in [DSS+05] TeSSLa relates a set of input streams to a set of output streams. This is done via mutually *recursive equations* and allows self-references to the past. The *last* operator *(last(parameter, trigger))* references the last known value of a given parameter at a given trigger. A trigger in this context is an event of a stream. The combination of recursion and the last operator allows for *recursive equations*. As *last* only refers to events in the past those equations can be computed incrementally. But to use *last* it might be necessary to provide a starting point. The *merge* operation *(merge(event a, event b))* fills that gap. By providing two streams, merge combines the events of those streams. If the user wants to count events of stream x he might call *count := map(last(count, x) + 1, 0)* to merge a start stream with value 0 (as the starting point) with the last value of *count* (when event x happens) while incrementing the count value and assigning it to count again.

In TeSSLa each event has a timestamp to establish a global order without having synchronous events. This timestamp is accessible via the *time* operator *(time(event))*. The global order is based on a global clock and no two same timestamps are allowed for an event. Thus a timestamp is unique for its stream. By accessing the timestamp of an event it is possible to determine ordering but also perform computations based on the information about the timestamp.

The *delay* function *(delay(period, event))* allows to create new timestamps. For this purpose the first argument is a value for a timeout and the second argument relates to the event that resets the timeout. In Figure 3.2 it is shown how this function is used to report errors as fast as possible. The *const* function *(const(value)(event))* in that example maps the constant value 5 to the write stream. The result of that *const* function is the period for *delay* so that delay expects a write event every 5 time units. At timestamp 12 the last write event was seen at 7 and an error is thrown because delay triggers and creates an event at timestamp 12. The behaviour of *time*, *delay*, *last* and the global ordering via timestamps support the statement of TeSSLa that *Time is a First-Class Citizen*.

TeSSLa is designed to run on FPGAs which are more restricted by their hardware than desktop computers or servers. Therefore TeSSLa follows two principles to ensure *Efficient Parallel Evaluation* on those systems. The first principle is *explicit memory usage*. Streams in TeSSLa can operate on unbounded and bounded datatypes. The latter however are used in practice because they have a constant size. The operators in general only need finite memory as only one data value is stored by an operator. The second principle is *local operator composition*. Semantics in

$$timeout := \mathbf{const}(5)(write)$$
$$error := \mathbf{delay}(timeout, write)$$

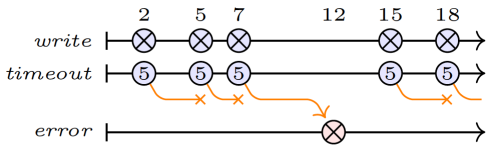

Figure 3.2: A basic example of creating new timestamps with the delay operator in [CHL+18]

TeSSLa are defined so that individual operators can be locally composed. That allows message passing without global synchronization. The progress of a stream is known so that passing this information is sufficient for recursive TeSSLa equations.

Two additional terms are defined regarding time in TeSSLa: *timestamp conservative* and *future independence*. A function is called *timestamp conservative* if and only if it does not introduce new timestamps. On the other hand, a function is *future independent* if and only if output events only depend on current or previous events.

Specifications in TeSSLa without *delay* operators are *timestamp conservative* as only delay can introduce new timestamps. Furthermore every TeSSLa specification is *future independent* because the only operators referring to events with different timestamps are *last* and *delay* and they refer to the past.

3.4 Parallel Programming

Nowadays computers, even in embedded environments, are benefiting from multi-core CPUs. Problems which need high computation power can be solved by using parallel programming. Those problems could range from sorting algorithms and physics simulations to non-deterministic state machine simulations.

But even simpler sequential programs might benefit from parallelization. Redesigning a program's execution flow might lead to a significant speed-up compared to the sequential code according to [Kum02]. But this does not mean that programming a parallel algorithm or even software is easy as it introduces a new dimension of problems and limitations.

The following sections focus on the topic of parallel programming its different approaches and techniques but also problems and limitations in this field so that it is easier to understand the proposed solution in Chapter 4 and Chapter 5.

3.4.1 Data Parallelism

In data parallelism similar operations are performed on different sets of data. The data is *decomposed* into *tasks* and onto computing nodes which perform those operations.

For further clarification the definitions of [Kum02] are used for decomposition and tasks:

Definition 3.4 (Decomposition)**.** The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called *decomposition*.

Definition 3.5 (Task)**.** A *task* is a programmer-defined unit of computation into which the main computation is subdivided by means of decomposition.

Depending on the location of the nodes those might be on a local computer, for example on different CPUs or CPU cores, or even different computers connected via a network. The decomposition of data is mostly done by partitioning the data which needs to consider dependencies. Those dependencies can be expressed in a *data-flow-graph*. Therefore for a node to compute the correct result it needs all necessary dependencies. In fact the computation does not need to be in a single phase but could consist of several phases of computation to lessen the burden on each node or use specialized nodes for certain computations, for example based on the hardware configuration. [Kum02]

3.4.2 Task Parallelism

In contrast to data parallelism, task parallelism concerns itself with the objective to execute different tasks on computation nodes. So instead of partitioning data the execution of a program is decomposed into different tasks which can be expressed in a *task-dependency-graph*. Like in data parallelism, one problem may be the computation based on the task dependencies. A node performs a task on the same or different data in relation to the other nodes. [Kum02]

3.4.3 Problems and Limitations

The performance and effective speed-up ($\frac{\text{sequential processing time}}{\text{parallel processing time}}$) of a parallel program is not only determined by the CPU but also the memory size and speed which feeds data to the CPU. In [Kum02] the difference between *latency* – when does the data flow – and *bandwidth* – the speed of the data flow – are discussed. These are

important for the hardware as differences in speed between memory and processor lead to times where the processor might not compute as it waits for data. In these times the processor can focus on other tasks.

Another important part is the size of the memory. The interesting question is, what happens when the size of the data exceeds the size of the memory and in consequence does not fit in it. In [DG04] the MapReduce algorithm is proposed that deals with that issue for large files. In short it chunks the data and provides those chunks to computation nodes that can perform tasks on them and finally map the results together. This could provide a solution for operations that cannot be effectively done in constant memory space like computations with a large data dependency.

3.4.4 Decomposition Techniques

One of the biggest challenges of parallelization can be the split of necessary computations into tasks that can be executed in parallel. This is was defined in Definition 3.4 as *decomposition*. There are different techniques to decompose a problem into smaller portions that can be handled simultaneously. Some of those techniques are introduced in the following sections.

Data Decomposition

One way of decomposing is based on possibly large data structures. This technique is called *data decomposition* and does the decomposition in two steps. First the data is partitioned and in a second step these partitions are used to get a partitioning of the computations. This results in tasks as defined in Definition 3.5 which operate on those different data partitions. [Kum02]

Partitioning the output data can be done independently as a function of the input. In this case the partition is a decomposition of a certain problem into tasks. Because of this the function can be executed in parallel already. [Kum02] shows this on the example of dense matrix-vector multiplication where a dense $n \times n$ matrix A is multiplied with a vector b. Each task represents one of n rows which is multiplied with b.

A downside is that partitioning the output data can only be performed if the output can be computed as a function of the input. Not every algorithm is capable of doing that. But it might be possible to partition the input data instead. For this to happen the input data is partitioned into a task. In consequence each partition of input data is a task used for computation. It might be necessary to combine the results as those partial results do not solve the original problem. In that case,

the results are combined to solve the original problem in a following computation. [Kum02]

An example algorithm for data decomposition is the *MapReduce* framework described in Subsection 3.4.5.

Speculative Decomposition

Another way is to use *speculative decomposition*. This technique is used when depending on a preceding computation a certain (following) branch needs to be processed. Before the currently computed output is known following branches are processed in parallel. When the output is ready and known the following branch is chosen and all other non-relevant branches are discarded.

As this wastes computation power one could think about a formulation of this speculative decomposition to only take the most promising branch. In case the anticipated branch is not the right one the computation is rolled back and the correct branch is taken instead. [Kum02]

One prominent example for its usage and effectiveness are processor architectures which use speculative branching to speed-up the execution of instructions. [MG13]

3.4.5 Map and Reduce

This section references mostly [DG04] and will otherwise explicitly cite another work.

The map and reduce framework allows parallel execution on large data sets (many terabytes and even petabytes) and thousands of machines. Based on the *map* and *reduce* functions of Lisp and other functional programming languages, users define those functions while the framework handles execution of these functions on possibly large clusters of machines, partitions the input data, schedules the execution and handles inter-machine communication as well as failures.

The whole computation model takes a set of input value and key pairs and produces a set of output value and key pairs. The map function takes an input pair and produces a set of intermediate key/value pairs. All intermediate values will be associated to the same intermediate key. Those pairs are passed to the reduce function which is another user written function. It accepts an intermediate key and a set of values of that key. It merges these values together to form a smaller set of values, typically just zero or one output.

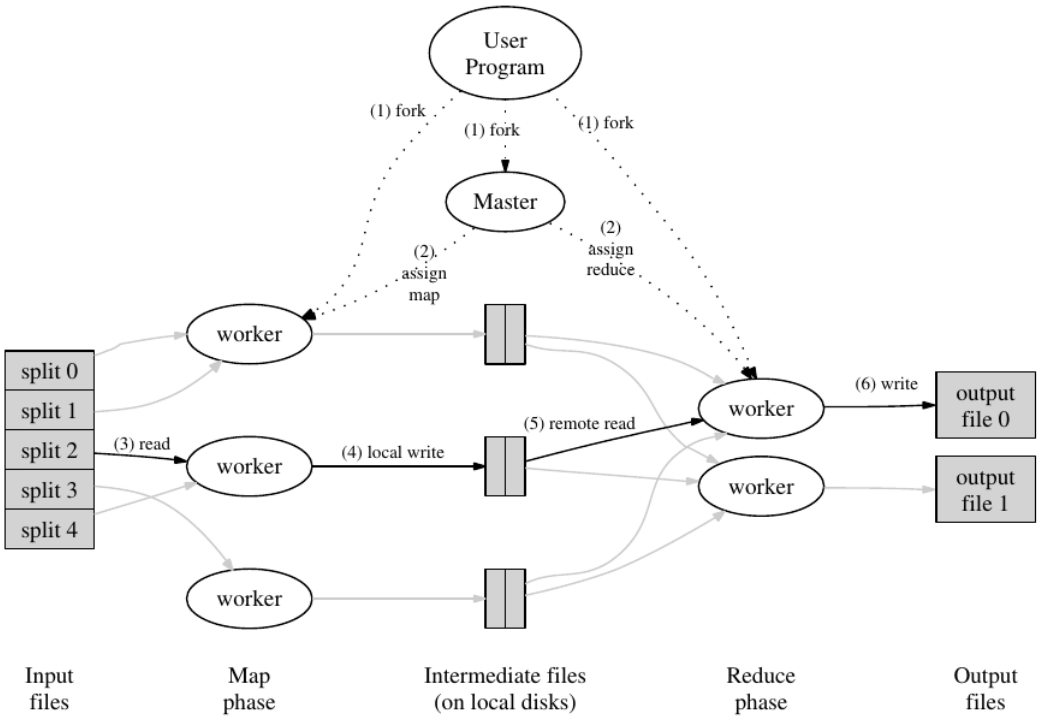

Figure 3.3: The execution overview of the MapReduce Framework shown in [DG04]

In Figure 3.3 the whole execution and its components is described. First the framework splits the input files into a number of pieces. Those pieces are typically 16 to 64 MB in size.

The next step is to start up many copies of the program on the cluster of machines. One of those copies is special as it is the master. The rest are workers. The master picks idle workers and assigns a map or reduce task to them.

A worker with a mapping task reads the contents of its corresponding input split, parses the key/value pairs out of it and passes it to the user written *Map* function. As already mentioned the map function generates intermediate pairs. Those pairs are buffered in memory.

Periodically the buffered pairs are written to a local disk. But before that they are partitioned into regions by a framework specific partitioning function. The location of each file is passed to the master who will pass a location with a reduce task to an idle worker in the reduce phase.

A reduce worker uses remote procedure calls to read the buffered data. It sorts the data by their intermediate keys. If the data is too large for the memory an external sort is used. The worker iterates over the set of intermediate keys and passes key

and the corresponding set of values to the *Reduce* function which is also defined by the user. The final result is output to a final output file for this reduce partition. Therefore for each reduce task a separate result file is written.

The master writes periodic checkpoints of its data structures. In the case that the master dies, the master is restarted from the last checkpoint.

It may also be noted that the network communication may lead to errors. Broken connections, unreachable or otherwise frozen computation nodes might lead to faulty results which might impede the whole MapReduce process. Because of this fault tolerance has to be graceful. The master pings every worker periodically. If no response is received after a certain amount of time the worker is marked as failed and all map tasks the worker completed or currently works on are reset. Completed tasks are re-executed because the intermediate results are saved on the local disk of the worker and inaccessible if he cannot be reached.

3.5 Abstract Syntax Tree

An abstract syntax tree (AST) is a directed graph $G = (V, A)$ with V being a set of nodes and A being a set of arrows. An arrow is a directed edge in the form of $a = (head, tail)$ with $a \in A$ and head, tail $\in V$, therefore going from *head* node to the *tail* node.

Every important token in an AST has a node and uses operators as subtree roots. An AST is used for implementing source code analysers, translators and interpreters. [Par09]

```
1 in x: Events[Int]
2
3 def sum: Events[Int] := merge(last(sum, x) + x, 0)
4
5 out sum
```

Listing 3.2: Summation Specification showing a recursion in TeSSLa specifications

To show an AST of TeSSLa in this section, the example of Section 4.1 is duplicated. Consider the specification shown in Listing 3.2. The AST generated by TeSSLa can be seen in Figure 3.4.

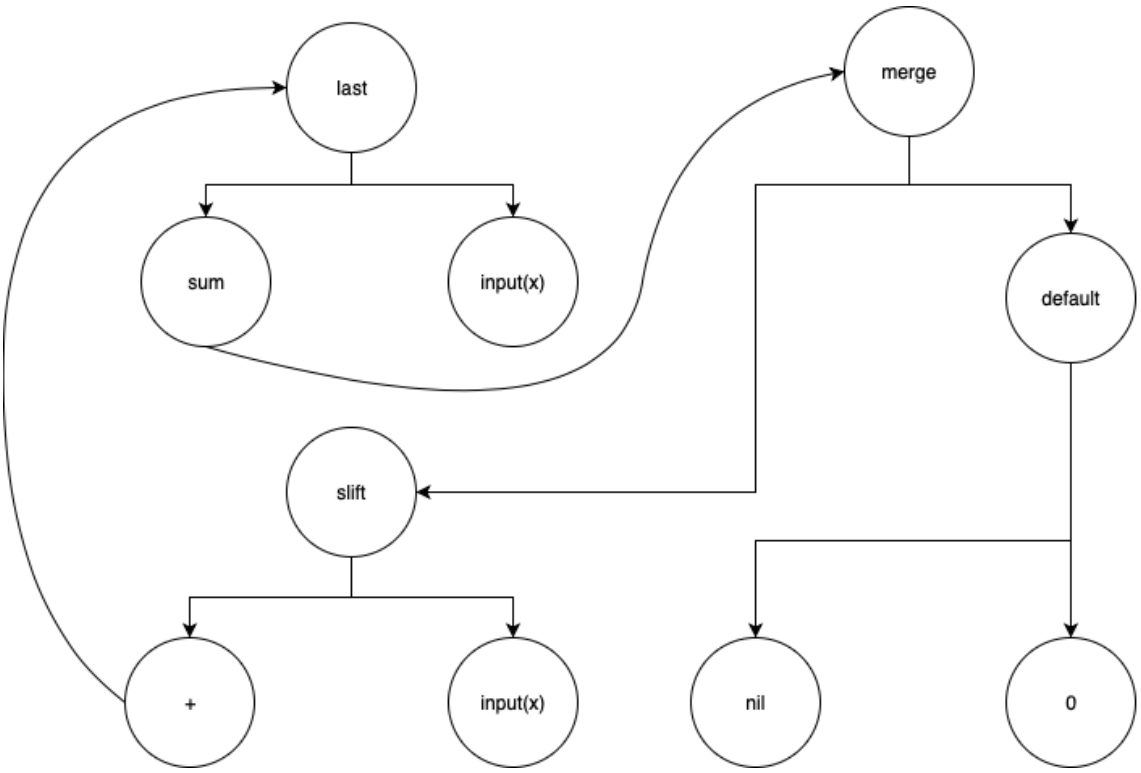

Figure 3.4: AST of a recursion example in TeSSLa specification

3.6 Symbolic Computation

All mathematical objects and their representations in computers can be symbolic objects. This thought gave rise to two research topics, namely *computational algebra* or symbolic computation and *computational logic*. Symbolic computation is, among others, used to manipulate mathematical expressions and objects. [Buc85]

The term *symbolic computation* in this work restricts itself to the manipulation and calculation of formulas that have no specific values. Those values are therefore substituted with an algebraic expression as long as the value is unknown. This allows for computations without knowing values and the usage of techniques such as *speculative decomposition* as described in Section 3.4.4.

3.7 Sub-Specification

A sub-specification φ_{sub} in the context of this work is a partial of a specification φ and contains a subset of operations and functions of φ.

For example lets consider the specification in Listing 3.3. One can see that φ_{sub} only consists of the summation of the event values of x while φ also consists of an error check.

```
1  # parent specification φ
2  in x: Events[Int]
3
4  def sum: Events[Int] := merge(last(sum, x) + x, 0)
5
6  def error: Events[Bool] := x > 5
7
8  out sum
9  out error
10
11 # sub−specification φ_sub
12
13 in x: Events[Int]
14
15 def sum: Events[Int] := merge(last(sum, x) + x, 0)
16
17 out sum
```

Listing 3.3: Summation Specification with examplary Sub-Specification

3.8 Deterministic and Nondeterministic Finite Automata

Some specifications of TeSSLa could be written so that those specifications resemble an automaton. In this work an *deterministic finite automaton* (DFA) is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, consisting of a finite set of states Q, an finite set of input symbols Σ, a transition function $\delta : Q \times \Sigma \to Q$, an initial state $q_0 \in Q$ and accepting states $F \subseteq Q$.

A *nondeterministic finite automaton* (NFA) is defined in this work as a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ consisting of a finite set of states Q, an finite set of input symbols Σ, a transition function $\Delta : Q \times \Sigma \to P(Q)$ (with $P(Q)$ being a power set of Q), an initial state $q_0 \in Q$ and accepting states $F \subseteq Q$.

4 Concept

This chapter starts with an approach to detect recursions in runtime verification engines and will explain how specific problems in the task of parallelization of a SRV engine like TeSSLas can be solved and how distributed approaches could be designed.

4.1 Detecting Recursion

As shortly noted in Chapter 2, one needs to identify the specific case that is put into the TeSSLa engine. One important information is whether or not the specification contains a recursion.

TeSSLa's compiler generates an AST to interpret the input specification. This AST can be used to analyse the structure of the specification and therefore check all tokens that are part of the specification, e.g. operators, input and output streams.

Observation 4.1 (Recursions are cycles in an AST)**.** *To check whether a TeSSLa specification contains a recursive call, one can look for a cycle in the AST generated by TeSSLa.*

By traversing the AST and its subtrees one can store the root node and check whether the current node is the root node. If it is not, traversing the current tree is continued. If it is the root node there is a recursion. After traversing all nodes without encountering the root node again the algorithm can conclude that there is no recursion in the specification.

The specification in Listing 4.1 intends to sum up all values of the events of x. These events are part of the input trace. *sum* is initialized with 0 and gets, when an event of x occurs, the last value of x (sum) and adds it to the current x. The result is assigned to *sum* again. Lastly sum is output as an output-stream. As one can see, the result which is calculated by using sum is reassigned to *sum*. Thus the recursion is part of the calculation of *sum*. An example of the output is shown in Figure 4.1.

```
1 in x: Events[Int]
2
3 def sum: Events[Int] := merge(last(sum, x) + x, 0)
4
5 out sum
```

Listing 4.1: Summation Specification showing a recursion in TeSSLa specifications

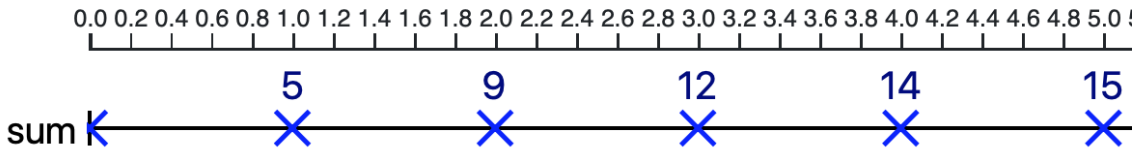

Figure 4.1: Output streams of summation example by TeSSLa playground visualization

The Figure 4.2 shows the corresponding AST of Listing 4.1 generated by TeSSLa. The root is *last* and contains *sum* and *input(x)*. One can see that *sum* is a merge of *slift* and *default*. *default* is an initial stream of 0 for the merge here, as mentioned in Section 3.3. *slift* contains an addition of *input(x)* indicated by the + operator. Because *last* is the root node one can deduct that *last* contains *sum* which contains *last*. Therefore the recursion led to a cycle in the AST.

4.2 Decomposing TeSSLa Input Data

As mentioned in Section 3.2 SRV evaluates a program's execution against correctness properties. The program's execution can be evaluated online or offline. In either case, in this works context it is the *trace* of the program's execution which is evaluated against the *specification*.

Decomposing the input data of TeSSLa means partitioning the given trace and specification. It might be necessary to partition only the trace or only the specification or even both, based on the specific case of specification.

Due to TeSSLa's signal semantics TeSSLa computes a verdict of an event when a new event happens. The principle of explicit memory usage allows TeSSLa to work with a small footprint which is necessary for TeSSLa to work on low memory systems such as embedded ones. This means that in the mapping phase the evaluation of sub-traces and specification will be done just-in-time and the reducing step will focus on merging the results.

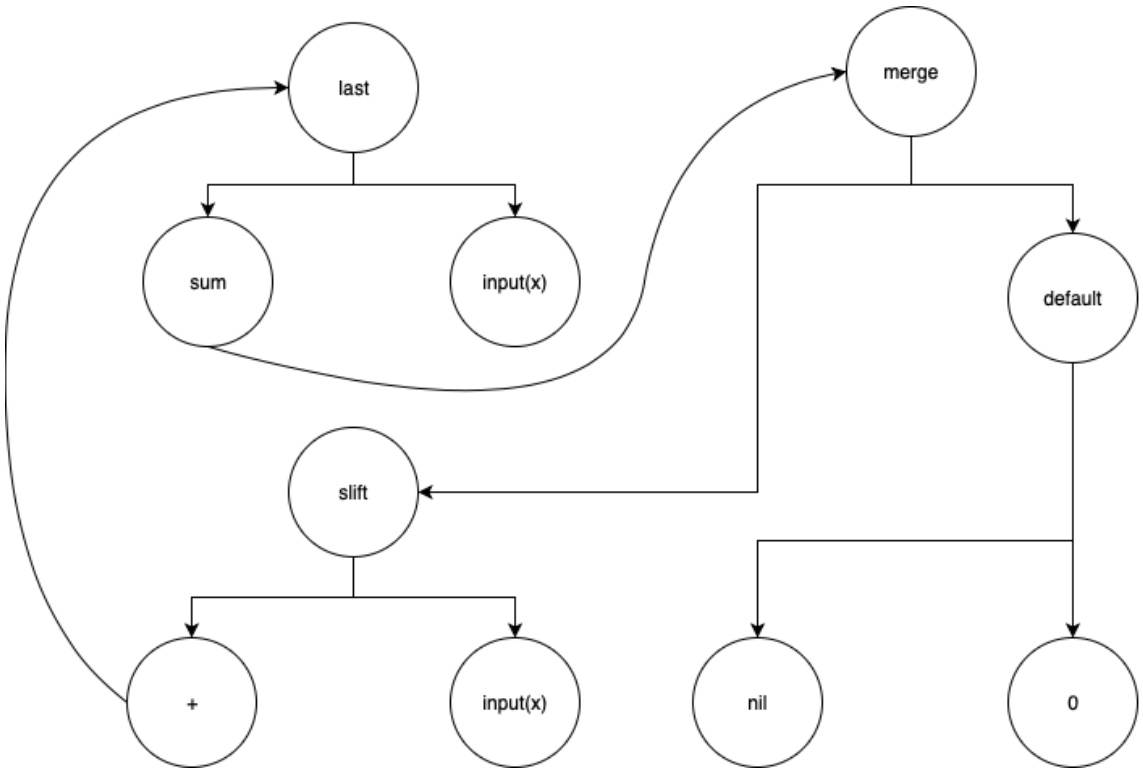

Figure 4.2: AST of a recursion example in TeSSLa specification

In the reducing phase the result is combined by applying the reduce function to the set of mapping results. To combine each result of the workers of the mapping phase, it might be necessary to run a sub-specification against those results.

The example shown in Listing 4.1 would generate sums for each sub-trace given to a worker in the mapping phase. The final result needs to combine all previous calculated results. So the specification needs to be run again in the reducer phase with the results of each mapper as new input trace.

Partitioning the specification means that one needs to know the AST nodes and therefore TeSSLa operations to write a new sub-specification with only the operations needed in the reduce step.

A *task* for TeSSLa, in this work, is a partial trace acquired by partitioning the trace which is related to a specific (sub-)specification. This task can then be evaluated by a computation node (in a distributed system or locally). That means that in each phase, mapping or reducing, it might be necessary to evaluate different tasks not only divided by the trace but also specification. Which map and reduce algorithm is used and how the resulting *tasks* are mapped and reduced will be described in Section 4.3 for each specification case separately.

4.3 Specific Problem Cases

In this section a number of examples for specific problem cases in the task of parallelization of SRV engines are given. The theoretical approach for each case is discussed here and the implementation of those cases using TeSSLa will be discussed in Chapter 5.

4.3.1 Trace Data without Dependencies

The simplest case one might think of is an input trace without dependencies between the single input streams. A look at Figure 3.1 shows that the input stream *temperature* is used to determine the value of *low* and *high* to get to the verdict whether the temperature is *unsafe* or not. That case can be identified by checking the AST generated out of the specification for references on input streams. If input streams are not used in operations with other input streams or values of other time stamps then there are no dependencies.

Figure 4.3 shows abstractly how *intelligent chopping* can be applied to this example case. Because the values are not dependent on anything, chopping the input trace into smaller pieces can be done freely. The resulting chunks are then used as input to the TeSSLa specification φ. In the reduce step for each chunk a result will be produced. If one wants to have a combined result it can be done by appending the results in ascending order.

The *map* function simply runs the trace against the specification, while the *reduce* function has no further steps to take other than outputting the result of each chunk of the input trace.

4.3.2 Trace Data with Self-Dependency

Another example is to compute a verdict or other statistical measurements based on previous values. To give an example one could think of a summation like $\sum_{t=0}^{10} a$. As one can see it sums up the value of a. Instead of using a loop one could use a recursion to solve this summation. This leads to $sum_t = sum_{t-1} + a_t$ and therefore *sum* being dependent on the previous value of *sum*.

As shown in Figure 4.4 every point cuts off a dependency. Therefore the *intelligent chopping* can focus again on creating chunks with the best possible size. It is important, though, to analyse the specification φ for a recursion as explained in

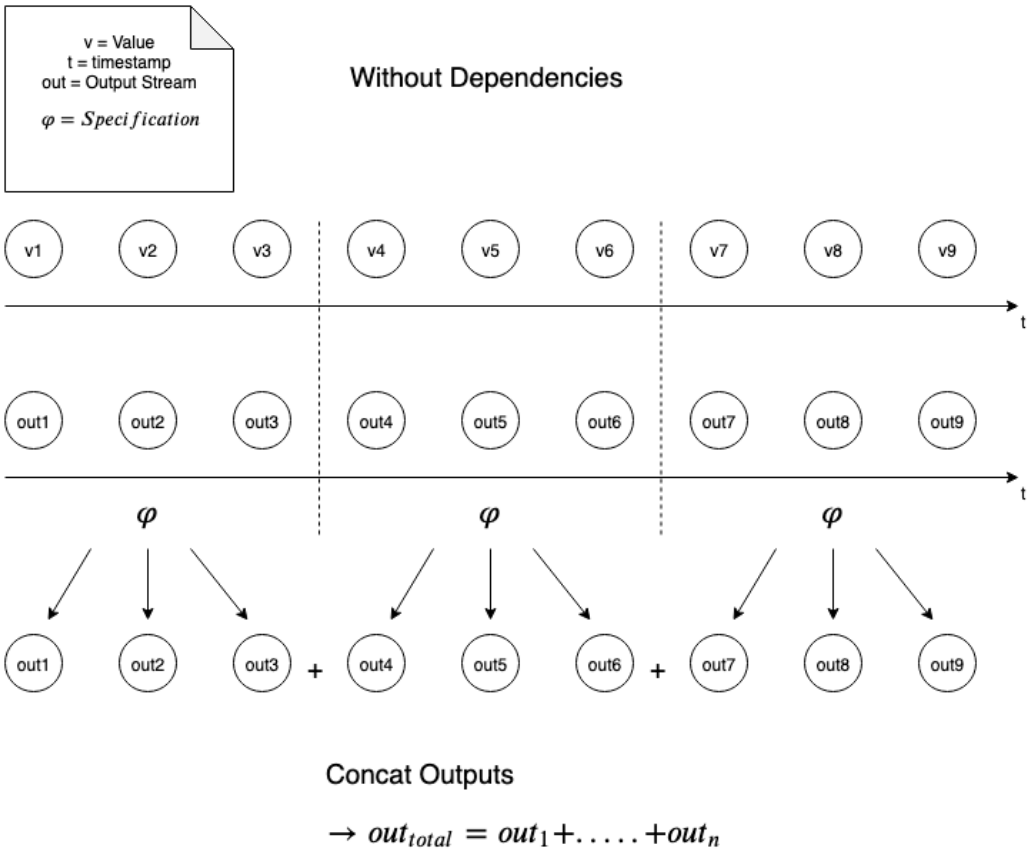

Figure 4.3: Abstract way to handle Trace Data without Dependencies

Section 4.1. The lines of the recursion are necessary to build the sub-specification(s) φ_{sub} which are needed in the *reduce* function.

The *map* function is responsible for computing the last values for each output stream. The result of each of those streams is used by the *reduce* function as an input to the sub-specification φ_{sub}. This way a final verdict is gained from computing each partial result.

In this case there is no need to use *symbolic computation* (and thus simulate the unknown value at the start of each chunk with an algebraic variable) on each step. There is a need to define a default value as the start value for each chunk. The reason for that is that the computation has no dependency to other variables besides itself and therefore the chunk results starting with a default value can be combined in the reducing step to gain the final result of each chunk. That is also the reason for the aforementioned need to create a sub-specification φ_{sub} based on the operations of the recursion.

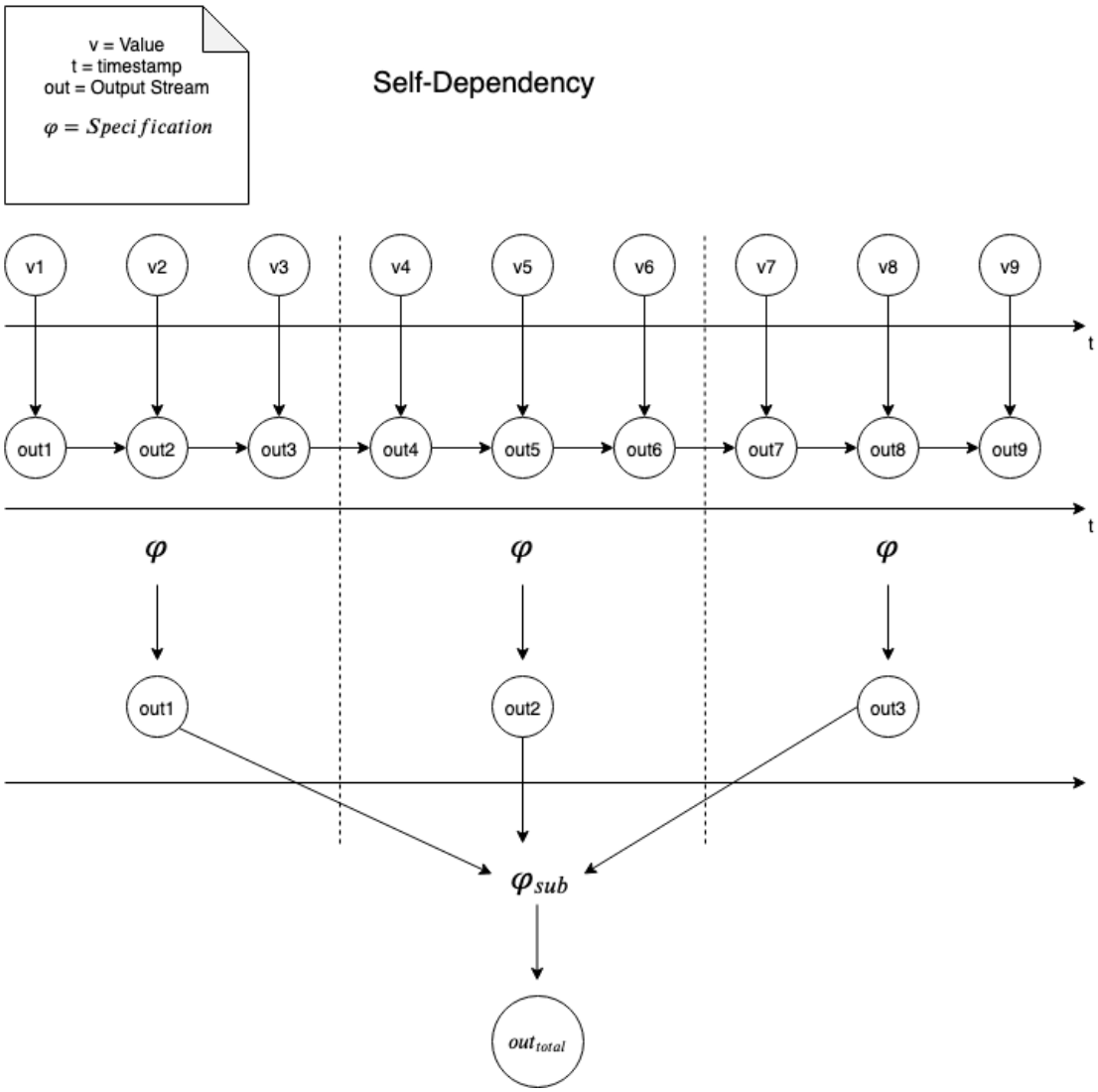

Figure 4.4: Abstract way to handle Trace Data with Self-Dependency

4.3.3 Trace Data with Reset-Variable

If the trace data has a recognizable reset variable *intelligent chopping* is done natively. On each reset event the trace will be chopped. In this case the input of the trace is not analysed further and each chunk will be run against the specification φ and a separate output will be produced for each chunk. The mapping and reducing phase is similar to the ones in Subsection 4.3.1.

The algorithm used to process this case would need to read the trace and look for reset points, chop at those reset points, run every chunk against the specification

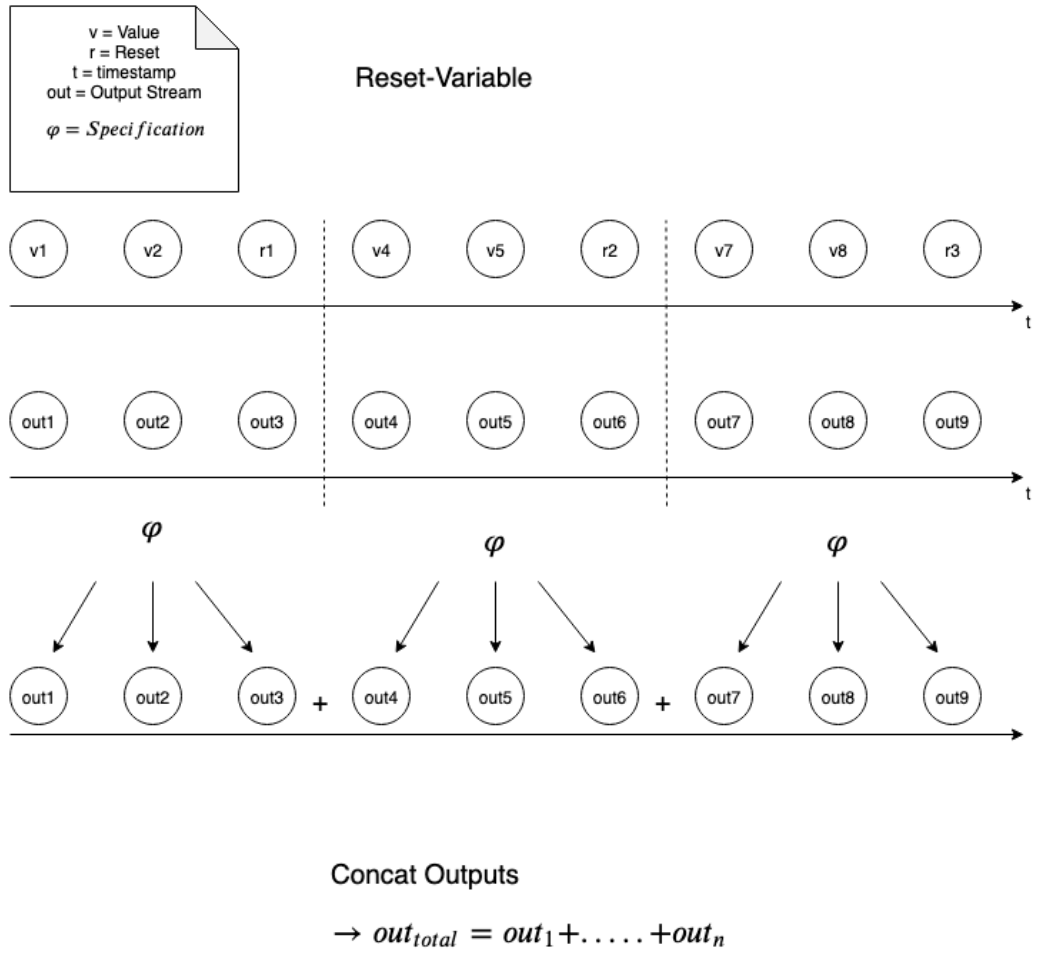

Figure 4.5: Abstract way to handle Trace Data with Reset-Variable

and output a result for each chunk. This procedure is shown in Figure 4.5.

Just like in the MapReduce approach defined in [DG04] each chunk has its own output. Therefore a second specification $\varphi_{combine}$ could be run after the parallel computation to combine the chunked results. For this to happen the output of the chunked traces needs to be combined into a single output which is the input trace for the combining specification.

A specification for this case is shown in Listing 4.2. At some time t an input event *E1* or *E2* happens and increments the *sum1* or *sum2* accordingly. These two sums are also the output streams.

```
1 in E1: Events[Int]
2 in E2: Events[Int]
3
```

```
4 def sum1: Events[Int] := merge(last(sum1, E1) + E1, 0)
5 def sum2: Events[Int] := merge(last(sum2, E2) + E2, 0)
6
7 out sum1
8 out sum2
```

Listing 4.2: Example specification for Trace Data with Reset-Variable case

A trace for this specification is shown in Table 4.1. The Event R is the reset variable. After analysing the trace each event R could be used as a cut point to chunk the trace. In the example shown in the table there would be 4 chunks of the trace. A chop would happen after timestamp 10, 20, 25 and 40.

Table 4.1: Example trace for Trace Data with Reset-Variable

Timestamp	Events	Value	Timestamp	Events	Value
5	E1	1	25	E1 R	3 0
10	E2 R	1 0	30	E1 E2	4 4
15	E1 E2	2 2	35	E2	5
20	E2 R	3 0	40	R	0

Each chunk will be checked against the specification shown in Listing 4.2 and result in Table 4.2. As seen the total result would be *sum1*: 10 and *sum2*: 15. But as already mentioned this needs to be computed in a second specification with Table 4.2's results as input trace.

Table 4.2: Result for each chunk of the example trace for Trace Data with Reset-Variable

Chunk	Result	Chunk	Result
1 (t = 5 to 10)	$sum1 : 1$ $sum2 : 1$	3 (t = 25)	$sum1 : 6$ $sum2 : 6$
2 (t = 15 to 20)	$sum1 : 3$ $sum2 : 6$	4 (t = 30 to 40)	$sum1 : 10$ $sum2 : 15$

4.3.4 Trace Data with N-Dependencies

Previously shown were cases of traces and their respective specifications without dependencies (Subsection 4.3.1) and with self-dependency (Subsection 4.3.2) but

one could easily think of a propositional logic formula with multiple variables. In that case a verdict might need to compute multiple variables in a single time step.

An example could be an error check. Let G be the gear of a car, R the revolutions per minute and V the velocity. A condition could be $G \geq 0 \wedge G \leq 6 \wedge R \geq 0 \wedge R \leq 8000 \wedge V \geq 0 \wedge V \leq 200$. This formula has multiple dependencies. On a each time step t there is a need to know the value of G, R and V.

The problem is that each chunk needs to have those information about G, R and V after the cut point. One solution would be to use *speculative decomposition* as shown in Section 3.4.4. Based on the input range a branch for each possible input value could be computed. In the worst case this leads to computing a branch for each possible system state and to the problem of *Model Checking* as described in Chapter 2.

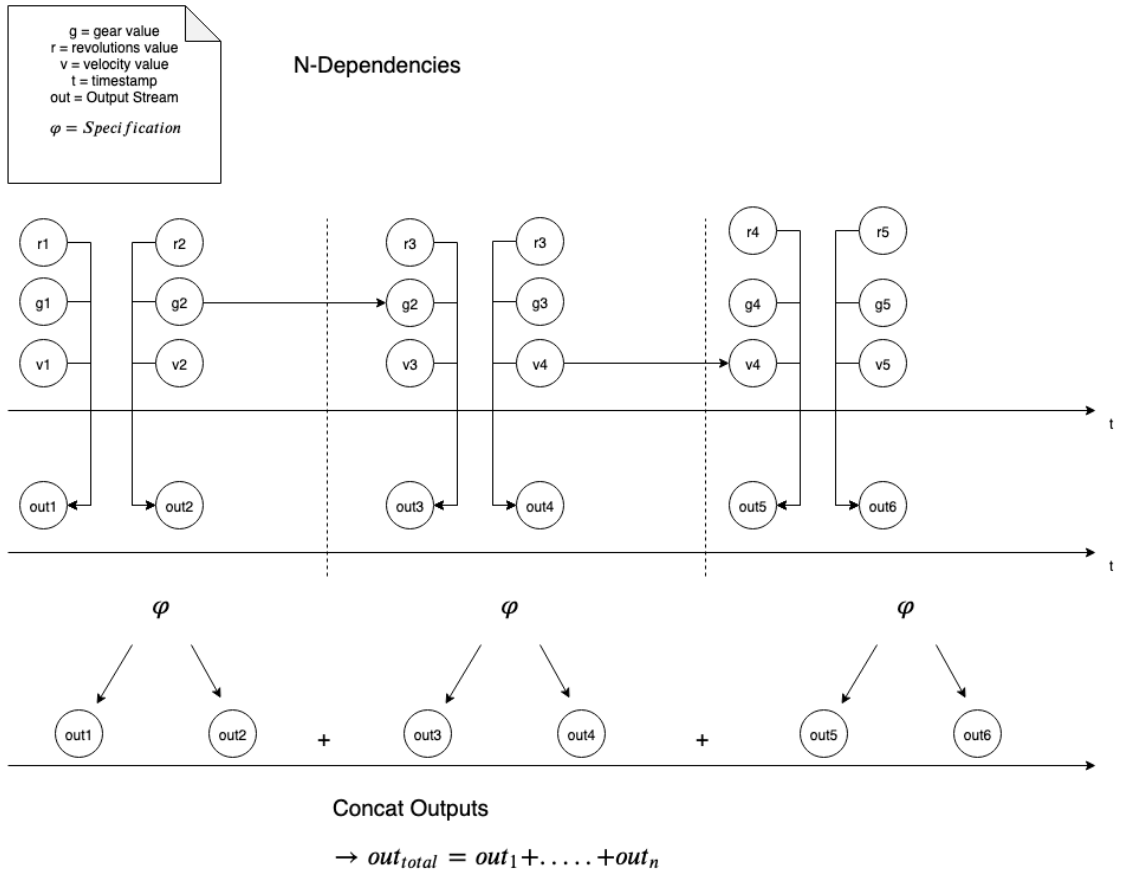

Figure 4.6: Error Check example showing overlapping of event values between chunks

In the simple example of getting a truth value as a verdict one could reduce the necessary branches by checking the conditions. The number of branches of the error

check example of this section could be reduced to branches that are within the ranges and those that are not.

Table 4.3: Speculative Branches for error check example

Branch	G	R	V
1	4	3250	125
2	8	3250	125
3	4	13250	125
4	4	3250	-125

The branches 2, 3 and 4 shown in Table 4.3 could be reduced to one branch as it does not make a difference in this example whether and which single value or even all values turn the formula to *false*. This is the case because the formula only computes a truth verdict and there are only two cases to cover: All values are in the specified ranges and thus there is no error or one or more values do not satisfy the formula and in consequence an error is detected. Having 3 branches to cover that one error case could be reduced to only one formula covering the error situation.

However this changes if the specification is not only about getting a truth verdict but also about statistical measurements. If one wants to get the average velocity (V) and corresponding revolutions per minute (R) one needs the values of the previous chunk. Therefore a branch for each possible value needs to be computed and after the previous chunk returns a result all branches that do not fit the result are discarded.

Another approach would be to overlap events between chunks. This means that the first and last few time steps around the cutting point are added to the chunk before and after the cutting point. In case of the error check example it is possible to only include the last few time steps of the previous chunk because it is not necessary to process the input events in any other way than checking against another value. This is shown in Figure 4.6. There the event *g2* is overlapping to the next chunk as well as *v4* from the second chunk to the third. If the input events are used to compute other intermediate events this approach might not work.

4.3.5 Specification with Automata

A specification that resembles an automaton has the same problem that the error check example with a simple truth verdict in Subsection 4.3.4 has. Each transition from one state to the other requires a specific set of values of the input alphabet. Each chunk after the first needs to know the current position of the input. Thus speculative decomposition can be used to create different branches based on each possible state as it is unknown which state was the last in the previous chunk.

For DFA each state has only one transition to follow. The number of branches is therefore limited due to the size of the set of states of the DFA. For a NFA branching is not limited on the size of the set of states as each state might have multiple transitions. In that case branches for each state and each possible transition have to be created. Based on the condition for the specific transition even branching for different values on which the correct transition is chosen is needed. After getting the result of the previous chunk all but the right branch are discarded.

4.3.6 Specification with Reset-Condition

If the specification has a reset condition – a condition that results in setting a value to a constant – the problem is based on the condition itself. The error check condition in Subsection 4.3.4 could be a reset condition and needs to be handled with an approach like previously shown with speculative decomposition. The same applies to the cases shown in Subsection 4.3.1, Subsection 4.3.2 and Subsection 4.3.3. If the specification has no deep dependencies using overlapping is also a possibility. Examples for these cases as reset conditions are shown in Listing 4.3.

```
1  in G: Events[Int]
2  in D: Events[Int]
3  in V: Events[Int]
4  in Reset: Events[Bool]
5
6  # Reset Condition with N-Dependencies
7  def count: Events[Int] := if (G >= 0 && G <= 6 && D >= 0 && D >= 8000
       && V >= 0 && V <= 200) then 0 else count + 1
8
9  # Reset Condition without dependency
10 def count: Events[Int] := if (G >= 0 && G <= 6) then 0 else count + 1
11
12 # Reset Condition with self-dependency (last and current G between 0 and 6)
13 def count: Events[Int] := if (merge(last(G, G), 0) >= 0 && merge(last(G, G), 0)
       <= 6 && G >= 0 && G <= 0 ) then 0 else count + 1
14
15 # Reset Condition with reset variable
16 def count: Events[Int] := if (Reset == true) then 0 else count + 1
17
18 out count
```

Listing 4.3: Example specification for Trace Data with Reset-Condition case

4.4 Distributed Computation of SRV

It is shown in Subsection 3.4.5 how to design a distributed system which orchestrates multiple map- and reduce-workers. The approach described in this chapter is close to the idea of MapReduce. Therefore the distributed approach is in most parts the same. A master will split and assign chunks to mapper workers. The result of each mapper is returned to the master. The mapping results are then assigned to reducer workers. Those workers output a result for each output of the mapping step. This approach is shown in Figure 3.3 in Subsection 3.4.5.

The approach is nearly the same as on a single computer with a multi-core CPU. The difference is that network communication needs to be modelled and implemented. By adding the dimension of network communication one also includes problems such as fault tolerance which might be a more complex problem due to the chosen strategy on how to handle broken connections or computation nodes.

5 Implementation

The implementation is using TeSSLa 0.76-Snapshot and implements a *parallel engine* which takes the input trace and specification and will determine whether parallelization is possible (based on the known and implemented cases). In case that it is possible to use parallel computation the engine will then run the TeSSLa engine in parallel to compute the result of the task at hand. In this chapter the architecture of this *parallel engine* is described as well as the specific algorithms used for the cases described in Section 4.3.

5.1 Architecture

The architecture consists of multiple components that implement analysing algorithms but also the mapping and reducing logic for the specific cases described in Section 4.3. An overview is shown in Figure 5.1.

A more technical overview can be seen in Figure 5.2.

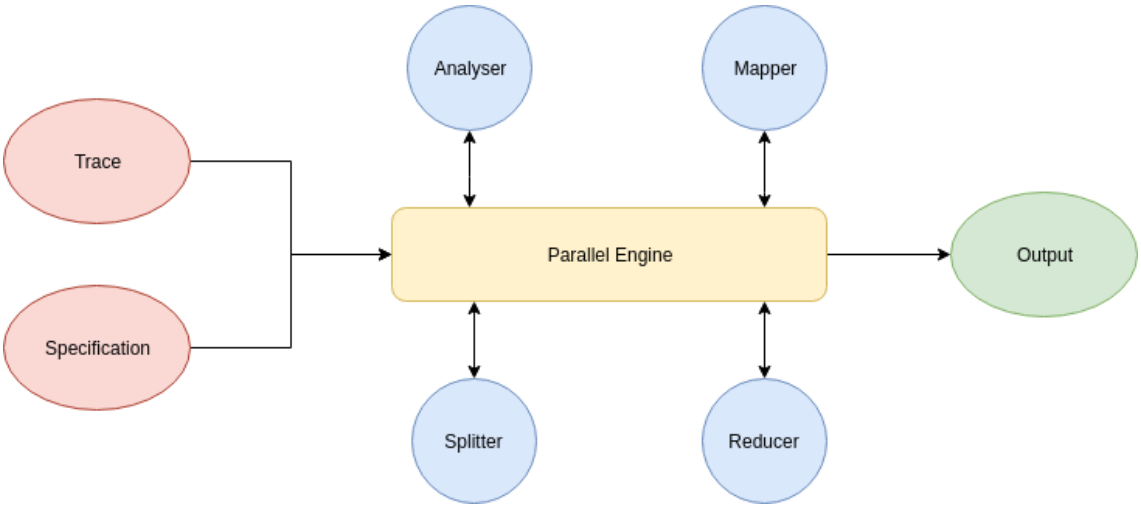

Figure 5.1: Graphic showing the architecture of TeSSLa Parallel

The *parallel engine* is responsible for starting the process of analysing the input trace and specification. If a possible parallel processing and corresponding case of

parallel computation is found the engine hands the results of each component to the next. It acts in that way as a coordinator. In case no known parallel computation is found the engine will start the TeSSLa Interpreter in sequential mode with the given specification and trace. The result of that is simply returned to the user.

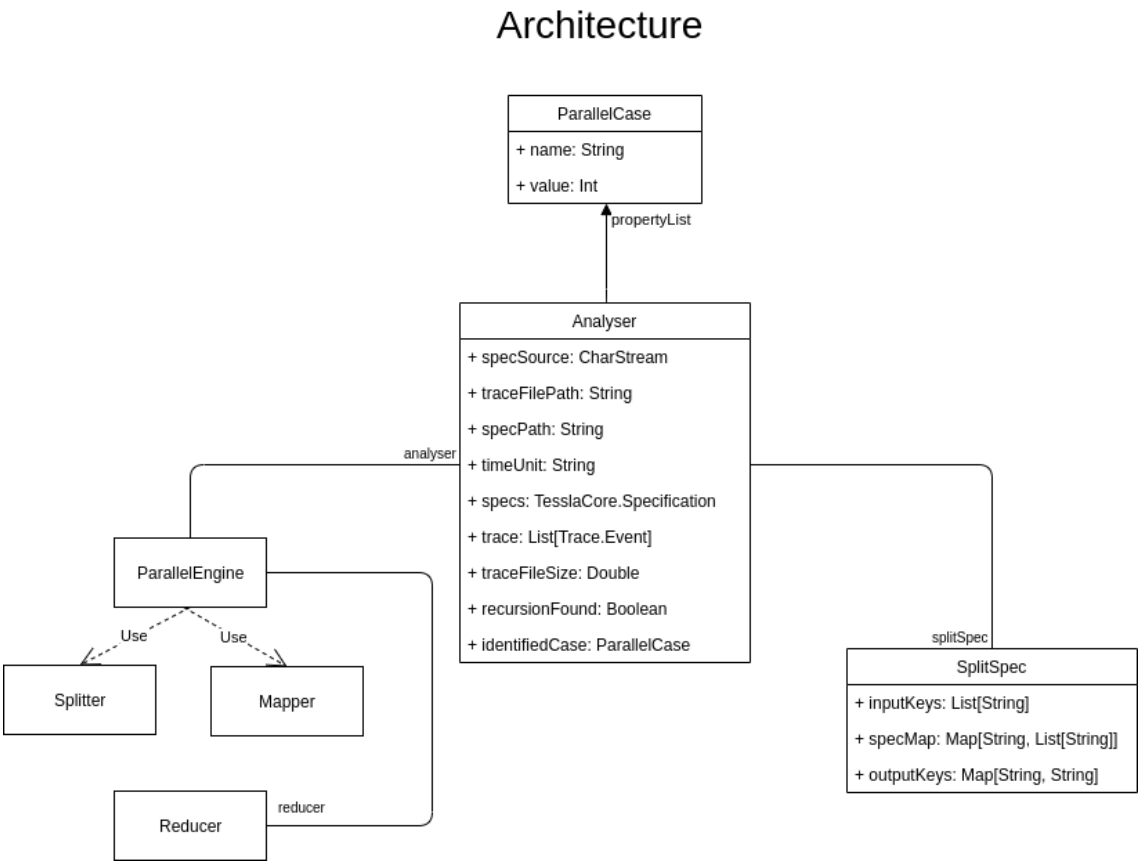

Figure 5.2: Diagram showing the technical architecture of the core parallel engine for TeSSLa

The *analyser* checks whether the specification contains a recursion using the approach described in Section 4.1. It also stores the input and output streams as well as the reference of where to find these streams inside the specification as shown in Listing 5.1.

This is necessary for some cases to process sub-specifications to gain the final result of the computation of input trace and specification. The final verdict of the *analyser* is the parallelization case that states which case of Section 4.3 is identified. The *propertyList* contains for this task a list of properties found during the analyse process. One such example of properties would be the information whether the specification contains a *last* operator or a recursion. All properties together lead to the unique case which will be stored in the variable *identifiedCase* for later references.

The *parallel engine* branches based on the identified case and uses the specific algorithms for splitting, mapping and reducing according to the identified case. The first step after the analyser is to split the input trace.

Chunking the input trace is done inside of the *splitter*. The splitter cuts the trace based on the identified case in parts of equal size and stores them in chunk files. The size of files is based on the number of processors and size of the heap. This is a more technical problem because of garbage collection. The garbage collection triggers at a certain percentage of heap usage. In single-core TeSSLa mode it is not problematic because TeSSLa computes and stores only a single value at a time. This is because of TeSSLa's feature of *explicit memory usage* (described in Section 3.3). When using multiple cores the memory usage is multiplied as well. The problem here is that the parallel computation needs the intermediate results in later phases, e.g. the result of the mapping phase is needed in the reducing phase. Because of this the implementation of the splitter splits the files according to the hardware configuration so that the garbage collection is not using too much CPU power and the program is not running into memory shortage. The partial files are stored to the hard disk and the filenames are put into a list for later reference. The basic splitter algorithm is shown in Listing 5.2.

```
1  val variableName = stream.asInstanceOf[TesslaCore.Stream].id.
       nameOpt.get
2  var subSpec: List[String] = List()
3
4  val inputKey = getStreamLocation(specs, inputKey)
5  val lastStreamLocation = getStreamLocation(specs, variableName)
6
7  val src = Source.fromFile(specPath)
8  val srcLines = src.getLines.toStream
9  subSpec = createSubSpec(inputStreamLocation, lastStreamLocation,
       variableName)
10 src.close
11
12 // map sub-spec to input key and store in spec map
13 splitSpec.specMap = splitSpec.specMap + (inputKey -> subSpec)
14 // store input key in list
15 splitSpec.inputKeys = inputKey :: splitSpec.inputKeys
16 // map output key to input key
17 splitSpec.outputKeys = splitSpec.outputKeys + (variableName ->
       inputKey)
```

Listing 5.1: Algorithm to store input streams and locations in the specification

After splitting the input trace into chunks those chunks are passed to the *mapper*. Here the specific algorithm for the detected case is used and each chunk is computed

in parallel. The result of each partial input file is stored in a map with the name of the partial file being the key and the result being the value.

The *reducer* now applies the reducing step onto the result of each file chunk the mapper produced. The reducing function is also case specific. The result of the reducing phase is returned to the parallel engine and from there returned to the user.

The specific implementations for each case are described in Section 5.2.

```
1  var fileIndex = 0
2  var beginIndex = 0
3  var endIndex = 0
4  for (i <- 1 to numberOfSplits) {
5    fileIndex = i
6    endIndex += traceSize / numberOfSplits
7    val fileName = writeChunkFile(fileIndex, traceFilePath,
       beginIndex, endIndex)
8    beginIndex = endIndex
9    // add chunk file name to return list
10   retList = fileName :: retList
11 }
12 // if after that iteration the beginIndex is not the traceSize
     there are still leftovers
13 if (traceSize != beginIndex) {
14   fileIndex += 1
15   val fileName = writeChunkFile(fileIndex, traceFilePath,
       beginIndex, traceSize)
16   retList = fileName :: retList
17 }
```

Listing 5.2: Basic Splitter Algorithm

5.2 Implemented Cases

This section shows the specific implementation for the cases named in Section 4.3. Some code fragments will be used to show in listings how certain algorithms are implemented.

The analyser identifies the case by using a bit mask. Each entry in the already mentioned *propertyList* references a numeric value. Those values are converted to bits. This is shown in Listing 5.3 but for simplicity only the cases *LastWithRecursion* (Trace-Data with Self-Dependency) and *NoLastNoRecursion* (Trace-Data without

Dependencies) are shown. The other case is *MultipleDependencies* (Trace-Data with N-Dependencies).

```
1  val specs = getSpecs()
2  if (specs == null) return false
3  val mask = getBitMask(analyseToken(specs))
4  mask match {
5    case LastWithRecursion.property => {
6      identifiedCase = ParallelCase(LastWithRecursion.name,
           LastWithRecursion.property)
7      true
8    }
9    case LastNoRecursion.property => {
10     identifiedCase = ParallelCase(LastNoRecursion.name,
           LastNoRecursion.property)
11     true
12   }
13   ...
14   case _ => false
15 }
```

Listing 5.3: Bitmask and Identified Cases

5.2.1 Trace Data without Dependencies

For this case a TeSSLa specification and a trace are given. According to the specification the input streams do not hold any dependencies with themselves or other input data. Each input stream is therefore independent. The temperature example of Figure 3.1 shows that independence. *temperature* is the input stream and used to check the condition $unsafe := temperature < 3 \vee temperature > 8$. *temperature* is used by *unsafe* and necessary to check that condition but is not dependent on any other data of the trace nor itself.

As mentioned in Subsection 4.3.1 we can split at random and in regards to chunk size. The splitter uses the algorithm as shown in Listing 5.2.

The bit mask for this identified case is 0. The strategy for this case will be used *iff* no *last* operator and no recursion is found. If there is a *last* operator or recursion in the specification but no other matching case is found and therefore no multiple dependencies, recursions, reset variables etc. exist, the analyser will return that parallelization is not possible and the parallel engine will proceed with starting a single thread with TeSSLa's interpreter.

The mapper uses the algorithm shown in Listing 5.4. The mapper simply loads each chunk and the specification file and runs the interpreter of TeSSLa to get the result of the input trace (chunk) corresponding to the given specification. Afterwards the chunk file is deleted and the result returned. *Futures* (as seen on line 2 of Listing 5.4) are Scalas way of running multiple threads simultaneously and a major tool used in each implemented case.

The reducer simply returns the result of the mapper. The mapper returns a stream of iterators of trace events and so does the reducer. Depending on the next processing step of the user those results can be output in a parallel multi threaded fashion or sorted and flattened into one stream that is sorted by the events timestamps.

One problem with the implementation – which is a problem of parallelization in general – is memory usage. Iterators in Scala are consumed on iteration. This is necessary to process large files or chunks of traces. The stream of iterators contains the reference to the start of those iterators of each result of a chunk. If those are compared the memory is depleted as sorting the timestamps is done in memory and depending on the trace size might need too much memory for the garbage collector to handle.

```
1 def noLastNoRecursion(sourcePaths: Stream[String],
    specSourceFilePath: String, timeUnit: String): Stream[
    Iterator[Trace.Event]] = {
2   val f = Future.traverse(sourcePaths)(path => Future {
3         val result = Interpreter.run(Unwrapper.unwrapResult(
            Compiler.compile(CharStreams.fromFileName(
            specSourceFilePath), Some(timeUnit))), Trace.fromFile
            (path), None)
4         new File(path).delete()
5         result
6   })
7   val result = Await.result(f, Duration.Inf)
8   result
9 }
```

Listing 5.4: Trace Data without Dependencies Mapper Algorithm

5.2.2 Trace Data with Self-Dependency

In this case the given TeSSLa specification defines a stream with a reference to itself. One simple example would be an addition such as $a = a + b$. As seen in this simple example and as stated in Subsection 4.3.2 it is necessary for this case to identify the recursion that leads to the self-dependency. For this purpose, as mentioned earlier

the *analyser* whose core logic is shown in Listing 5.5 analyses the AST of the given specification.

The algorithm starts at a stream and checks recursively whether a reference to the start stream exists somewhere in the child nodes of the AST of said start stream. A flag for recursion is added to the bit mask that specifies the identified case by the analyser if a recursion is found in the specification.

In this particular case we need to know what the input and output variables are. This is also done by the analyser as shown in Listing 5.1 (lines 13, 15, 17). The analyser not only stores the input variable and its occurrence in the specification but also the resulting output variable. This reference is used by the reducer to find the specific output variable and create a sub-specification with the value of the output key of the previous chunk.

The splitter works the same way as for Subsection 5.2.1 and therefore the splitter chunks the trace according to a certain size.

```
1  def recursionDetection(specs: TeslaCore.Specification,
       currentStream: TeslaCore.StreamDescription, startStream:
       TeslaCore.StreamDescription): Boolean = {
2
3    def resolveStream(s: TeslaCore.StreamRef, uid: TeslaCore.
       Identifier): Boolean = s match {
4      case _: TeslaCore.Nil | _: TeslaCore.InputStream =>
         false
5      case stream: TeslaCore.Stream => {
6        if (isNotStartStream(stream.id)
7          && existsInSpecStreams(stream.id.uid)) {
8            recursionDetection(specs, specs.streams.find(s =>
               s.id.uid == stream.id.uid).get, startStream)
9        } else {
10         true
11       }
12     }
13   }
14 ...
15 }
```

Listing 5.5: Recursion Detection by Analyser

The mapper creates a thread for each trace chunk and runs that chunk against the specification. Afterwards the result is filtered according to the output variables which the analyser stored at the analysing step before. If the filtered result is not empty the last event is stored in an output list. The *output list* contains tuples

which itself contain the name of the output stream, the value and timestamp of the output event. The complete output list of each chunk is stored in a map with the key being the name of the chunk file and the value the output list. This result map is needed by the reducer. An overview of this structure is shown in Figure 5.3.

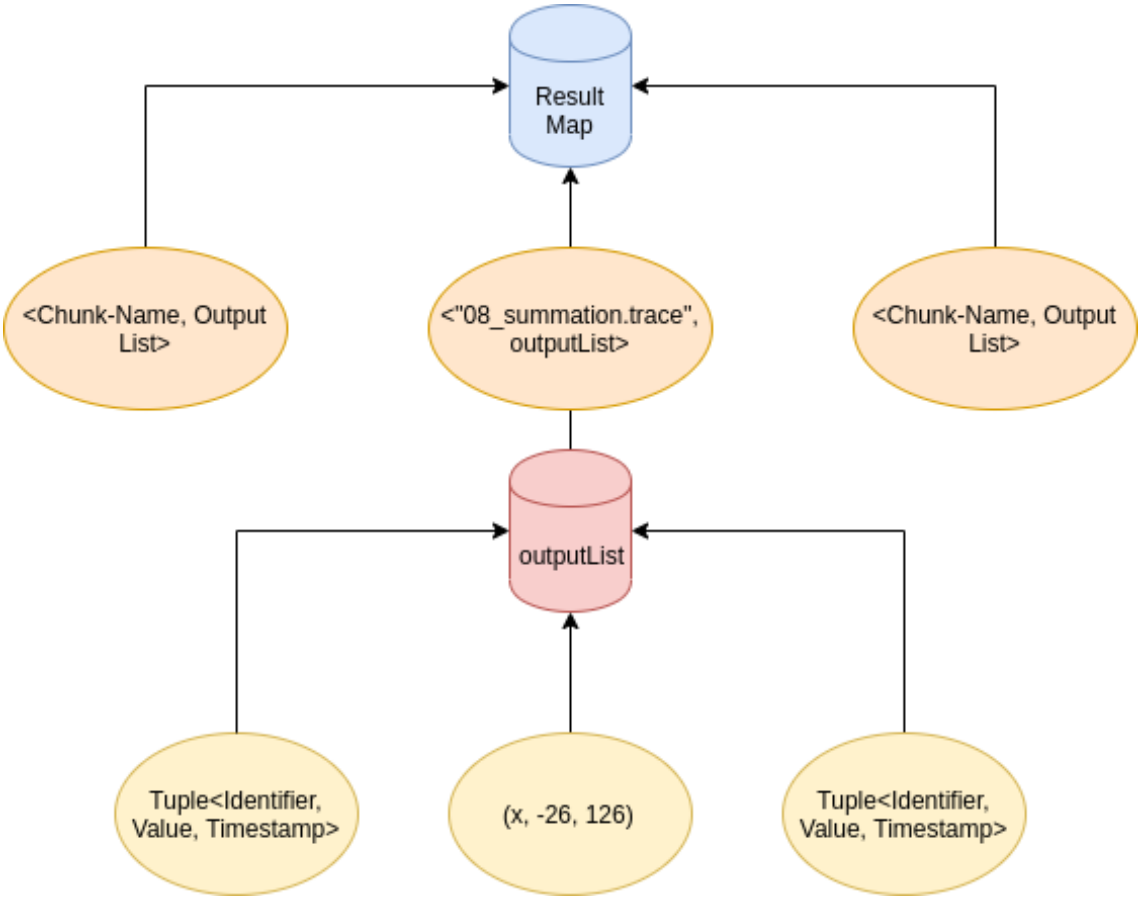

Figure 5.3: An overview of the output list and result map structure of the mapper for Trace Data with Self-Dependency

The reducer creates new traces for each output value of the result map and a subspecification based on the information of the analyser. For each output variable the result of the previous chunk is stored in a map. If no end result of a previous chunk exists a default value based on the type of value (string, boolean or integer) is used instead.

These new traces and specification are then processed by the TeSSLa interpreter with the result being stored in a final result list. The core algorithm is seen in Listing 5.6.

This approach has a limitation, though. Processing a specification that counts events

will lead to a faulty result. This approach calculates a result for each chunk. This result is then used again for the operation defined in the specification leading to the final result. For specifications that calculate something with the stream values this works fine.

To count events, though, there needs to be a number of events. This approach has only one event per chunk, as all other results are processed to get the result of a chunk. Applying the sub-specification of a "count event specification" therefore leads to counting the chunk files as each file contains a single event.

```
1  var tmpList: List[Trace.Event] = List()
2  var lastValues: Map[String, TesslaCore.Value] = Map()
3  result.foreach(resultFile => {
4    resultFile._2.foreach(outputValue => {
5          splitSpec.specMap.foreach(entry => {
6            val traceValue = outputValue._2
7            val traceTimestamp = outputValue._3
8            val inputKey = entry._1
9            val outputKey = getOutputKey(splitSpec, inputKey)
10           val specStrings = entry._2
11           if (outputValue._1.name == outputKey) {
12                 val defaultValue = getDefaultValue(lastValues,
                       outputKey)
13             val traceString = createTraceString(traceTimeStamp,
                   inputKey, defaultValue, traceValue)
14             val fileName = writeTraceTempFile(traceString)
15             val result = Interpreter.run(Unwrapper.unwrapResult(
                   Compiler.compile(CharStreams.fromString(
                   specStrings.mkString("\n")), Some(timeUnit))),
                   Trace.fromFile(fileName), None).toStream.last
16             lastValues = updateMap(outputKey, result.value)
17             // add result event for filtering later on
18             tmpList = result :: tmpList
19           }
20         })
21   })
22 })
```

Listing 5.6: Reducer Self-Dependency

It is not an unsolvable problem, though. One possible solution to handle this situation could be to introduce a keyword to symbolize such a case. This would be similar to the introduced reset variable in Subsection 5.2.3. Another solution would be to distinguish such specifications from ordinary self-dependency cases with special checks based on the expressions of the AST. The core logic of *Trace Data with*

Self-Dependency would still hold but would need to be extended to a differentiation between counting and non counting cases in the reducer. The mapper would sum up and count the events for each chunk. The reducer on the other hand would need to use a specification to sum up not simply each event last seen but the result of each chunk.

5.2.3 Trace Data with Reset-Variable

The last cases analysed the given TeSSLa specification but this particular case adds the ability to enhance the chopping logic natively by analysing the trace. A reset variable will be introduced to mark the point of chopping. The analyser will search for that variable and chop at that point. In Subsection 4.3.3 an example is shown in which the input values of two streams are summed up.

The analyser identifies the case of the specification as normal. Thus this *Trace Data with Reset-Variable* case is an addition to the existing and already mentioned cases. The analyser also processes the trace and looks for a variable named *ResetVariable* as TeSSLa does not support a keyword for reset points at the time of writing. A list of numbers called *resetVariableCutPoints* will store the line number of the reset variable in the trace. A flag is also set inside the analyser to signal that a reset variable was found and therefore splitting shall proceed accordingly.

The information (reset variable found and on which line numbers the cut points are) is given by the parallel engine to the splitter.

The splitter thus uses the number of reset variables as the number of iterations. Two index variables are introduced to mark the last end index and the begin index of the next chunk. In consequence the beginning is the end of the previous chunk. The end of a chunk is the line number of the reset variable stored in *resetVariableCutPoints*. This is shown in Listing 5.7.

After splitting the trace into chunks the algorithm for mapping and reducing is used according to the identified case. As an example, if the analyser detected self-dependent streams, the mapper and reducer of that case are used.

The *Trace Data with Reset-Variable* case shows in that regard that the core algorithm for the specific cases can be modified by changing parameters of the splitting process.

Due to the implementation details at this point it is not possible to enable *online splitting*. The idea behind online splitting is that in certain cases one might be able to map and reduce while splitting of the trace is not done yet. For this to happen splitter, mapper and reducer each need to run in separate threads. A communication needs to be established so that the splitter might inform the mapper that new chunks

are available. A coordinator might be the smartest choice as it would not be intrusive to the mapper/reducer process and could also start new mapper/reducer threads. This is the approach of [DG04] but on a local machine.

```
1  var beginIndex = 0
2  var endIndex = 0
3  for (i <- 1 to resetVariableCutPoints.size) {
4    val traceSrc = Source.fromFile(traceFilePath)
5    val traceIt = traceSrc.getLines
6    endIndex = resetVariableCutPoints.head
7    val fileName = createFileName(i, originFileName)
8    // add chunk file name to return list
9    retList = writeFile(fileName, traceIt.slice(beginIndex,
       endIndex)) :: retList
10   // drop head after processing to lessen burden on memory
11   resetVariableCutPoints = resetVariableCutPoints.drop(1)
12   // move pointer
13   beginIndex = endIndex
14   traceSrc.close
15 }
```

Listing 5.7: Splitting on cut points of reset variables

5.2.4 Trace Data with N-Dependencies

This case revolves around the question what happens when a stream has multiple dependencies. An example is given in Subsection 4.3.4. In that example is a stream that returns a boolean stating whether or not an error was detected. To detect errors, the values of G (gear), R (revolutions per minute) and V (velocity) of a car are checked.

Mentioned in Chapter 4 was the idea to use *speculative decomposition* to branch in a speculative manner. Over the course of the implementation that idea was discarded for another solution – *overlapping*.

Similar to checking for a recursion in the specification of self-dependency for n-dependencies a check is needed regarding multiple dependencies. For multiple dependencies the analyser stores those dependencies in a map. The key of that map is the stream and the value is a list of other streams that are necessary for that key stream. That is the map of dependencies. Therefore the analyser builds that map and if that map holds an entry the n-dependency case is detected. Additionally the analyser stores all input streams in a map. This map has the name of the stream as key and the value is a list of tuples. A tuple consists of a primitive operator which

is a TeSSLa specific class to implement infix operators like +, −, /, ∗, <, <=, >, >=, &&, || etc.

The splitter – just like in Subsection 5.2.1 – splits regarding size. Every split cuts off dependencies necessary for the computation of an intermediate or output stream. Because of this the splitter does not need any kind of special logic. The mapper on the other hand holds the core logic to solve the problem of cutting off dependencies. The chunks of the trace and the specification file are given to the mapper but also the map of input streams. In a parallel fashion each chunk is checked for missing input streams. The idea behind that is that if there are multiple dependencies we need all input variables to ensure that the necessary input streams are available. While checking, the missing input streams are added and a temporary chunk is written to disk. The crucial part is adding the missing values.

```
1  while (trace.hasNext && isFirstTimeStamp) {
2    val event = trace.next
3    if (firstTimestamp == -1) {
4          firstTimestamp = event.timeStamp.time.bigInteger.
               intValue
5    }
6    // store seen variables to list if it's still the first
         timestamp
7    if (firstTimestamp == event.timeStamp.time.bigInteger.intValue
         ) {
8          val variableName = event.stream.name
9          if (!seenInputVariables.contains(variableName)) {
10           seenInputVariables = variableName ::
                 seenInputVariables
11         }
12   } else {
13         isFirstTimeStamp = false
14   }
15 }
```

Listing 5.8: Adding the seen Input Streams of a chunk for the case of N-Dependencies

For each chunk file a list of seen input variables (*seenInputVariables*) is filled. This is shown in Listing 5.8. TeSSLa uses timestamps to distinguish events on the same stream. The mapper therefore checks the first timestamp of a chunk for its input streams. All seen streams are written into *seenInputVariables*. Afterwards the previous chunk is read and all events of that chunk are read backwards starting with the latest timestamp. This is shown in Listing 5.9. The reason for that is that the missing input streams are gathered from the last timestamps of the previous chunk. All input streams gathered this way are stored in a separate list (*addInputVariableList*)

to add later. Each tuple in that list contains the name of the stream but also its value and timestamp. Lastly, it is ensured that in case one input stream was not found in the previous chunk the remaining not seen variables are initialized with a default value at least (mostly this will be the case for the first chunk as there is no chunk before that). Afterwards the temporary file is written with the addition of the elements of *addInputVariableList*.

```
1  val previousChunkFileName = getPreviousFileName(traceFileName)
2  var events = Trace.fromFile(previousChunkFileName).toStream.
       reverse
3  var isDone = false
4  lastTimestampPrevious = events.head.timeStamp.time.bigInteger.
       intValue
5  // add missing input variables of the last events of the
       previous chunk
6  while (events.nonEmpty && !isDone) {
7    val event = events.head
8    val variableName = event.stream.name
9    if (!seenInputVariables.contains(variableName)) {
10         addInputVariableList = (variableName, event.value,
               lastTimestampPrevious) :: addInputVariableList
11         seenInputVariables = variableName :: seenInputVariables
12   } else {
13         isDone = true
14   }
15   events = events.drop(1)
16 }
17 // add still missing input values with initial value
18 if (seenInputVariables.length != inputMap.size) {
19   inputMap.foreach(entry => {
20         if (!seenInputVariables.contains(entry._1)) {
21           addInputVariableList = (entry._1, TesslaCore.IntValue(
                 new BigInt(new BigInteger("0")), Location.unknown),
                   lastTimestampPrevious) :: addInputVariableList
22           seenInputVariables = entry._1 :: seenInputVariables
23     }
24   })
25 }
```

Listing 5.9: Adding the missing input streams for the case of N-Dependencies

This procedure adds the missing input streams in a fashion that resembles the whole trace file. The gap that is created by splitting the trace into chunks destroys the chain of dependency as cutting might split up variables that are necessary for the first calculation of a chunk. Those missing parts are added as if the split did not

happen. Because of this the timestamps of the previous chunk are written into the new (temporary) chunk file. Basically the mapper is rewriting the chunks to include all variables at the first timestamp. Missing variables are gathered from the chunk before and added to the first timestamp of the temporary chunk. In cases that a variable is rarely written it might even be possible that a missing variable is not present in the previous chunk. In such cases it is necessary to search in previous chunks for the missing variable and if it could not be found, a default value is necessary to initialize that variable.

After adding all input streams that were missing and writing a new temporary file, the old chunks are deleted and the specification is applied in parallel to each new chunk. The result is a stream of iterators of trace events like in Subsection 5.2.1.

The reducer – just like in Subsection 5.2.1 – returns the result of the mapper. The same limitations of that previous case apply here. The order of timestamps might be incorrect due to parallelism. Based on the sorting algorithm, it might be necessary to process the whole output in a single thread again which would nullify the speed-up. It might not be necessary to order the timestamps and otherwise the user can still choose to do so and decide how.

5.2.5 Specification with Automaton

The implementation for specifications with automaton was not done in time for the deadline of this thesis. Therefore in regards to finishing up the remaining chapters and evaluating the done cases the implementation was left open for a later time.

5.2.6 Specification with Reset-Condition

The implementation for specifications with reset-conditions was not done in time for the deadline of this thesis. Therefore in regards to finishing up the remaining chapters and evaluating the done cases the implementation was left open for a later time.

6 Evaluation

In this chapter the aforementioned cases are evaluated. The main interest is to compare the runtime of the single- and multi-threaded approach of each case. The mentioned cases are not exhaustive and this section is meant to determine whether the implementation of other cases is worth the effort. Another constraint for comparing the multi-threaded approach to the already existing processing of TeSSLa specification and traces with a single thread is that the result is still the same. Even when using multiple threads, it must not return an incorrect result.

6.1 Hardware used

All tests are run on an Ubuntu 18.04.3 LTS with 16 GB RAM DDR3 at 1.600 MHz, a 256 GB SSD with SATA III (6 Gbps; reading speed of 520 MB/s and writing speed of 420 MB/s) and an AMD Fx-8120 eight-core processor at 3.100 MHz.

6.2 Test Cases

In the following section all tested cases and their results are described. A subsection will explain the limits of reproducibility and how to accomplish a reproduction of the test cases.

Each test case will have its own subsection and also describe the used specification to test the specific case. The sizes of the traces of the test sets are 100, 250, 500, 1,000 and 2,000 MB. These sizes might not be as large as mentioned in Chapter 2 but are sufficient to show the difference for traces with millions of events in terms of speed-up. After determining whether or not a higher performance can be achieved using multiple CPU cores the second question is for which cases and configurations it is most beneficial.

The results of each test case are gathered over the course of running the test with a given configuration three times to lessen the possibility of outliers.

6.2.1 Reproducibility

Each of the following cases needs a specification and trace that TeSSLa can interpret. The trace is then used by TeSSLa as input to check against the given specification.

To generate traces as part of these test cases NodeJS scripts were written and can be found in Appendix A. Those scripts generate random values for the traces to prevent that the implementation only works on input traces with specific values.

The Parallel Engine discussed in Chapter 5 accepts four arguments. The first argument is the trace file. The second argument is the tessla file while the third and forth argument check if those arguments are the strings *debug* and *diagnostic*. These strings will tell the parallel engine to start in debug and/or diagnostic mode. Not providing the third and fourth argument or writing those words (*debug* and *diagnostic*) wrong will disable the respective debug and diagnostic mode. The user can decide by providing a wrong input to the third or forth argument to start either the debug, diagnostic mode or none at all. In debug and/or diagnostic mode additional evaluations are gathered. In those modes the engine will provide a result based on a single thread and multi thread computation. If neither of those modes are enabled the parallel engine will try to use a multi-threaded approach if possible and will use as fallback single-threaded.

6.2.2 Trace Data without Dependencies

As previously mentioned, this case has input streams that are not used in any dependency with other input streams.

To test this case the specification shown in Listing 6.1 is used.

```
1  in value: Events[Int]
2
3  def lowerBound := value > 0
4  def upperBound := value < 10
5
6  def inBound := lowerBound && upperBound
7
8  out lowerBound
9  out upperBound
10 out inBound
```

Listing 6.1: TeSSLa specification for the test case of Trace Data without Dependencies

The script for generating an input trace (shown in Listing A.1) will set *value* on each timestamp with a value between -12 and 12.

The correctness of the multi-threaded approach is tested by checking the output events one by one against the single thread result. It is assumed that the single thread result is correct. Testing large traces might lead to *out of memory* exceptions. This is because the whole result will be read into memory for comparing. Testing correctness was in consequence limited to files less than 100 MB because files larger than that threw out of memory exceptions while comparing the TeSSLa results. As no faults in correctness were detected with 100 MB sized files however, it can be reasonably assumed that larger files were correct too.

Table 6.1: Evaluation of Time needed in case of Trace Data without Dependencies

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	4.45	1.10	4.05
Events/Second	18,728	75,914	
250 MB	11.87	2.89	4.10
Events/Second	17,528	71,936	
500 MB	23.63	5.42	4.36
Events/Second	17,260	75,215	
1,000 MB	47.07	10.97	4.29
Events/Second	17,686	75,866	
2,000 MB	77.81	18.36	4.24
Events/Second	20,329	86,172	
10,000 MB	371.99	142.15	2.62
Events/Second	20,589	53,880	

The corresponding numbers of input events are 4,995,000 (100 MB), 12,487,500 (250 MB), 24,475,000 (500 MB), 49,950,000 (1,000 MB), 94,905,000 (2,000 MB) and 459,540,000 (10,000 MB). In this test set a trace of roughly 10 GB was added to analyse the speed-up trend of the system using a bigger size step.

As shown in Table 6.1, the speed-up ranges from 2.62 to 4.36 with an average of 3.94. The speed-up is reduced by larger traces. In relation to the sizes up to 2,000 MB though it isn't crucial. The step to 10,000 MB suggest that bigger files in this configuration will have lower speed-up rates. The reason for the loss of speed-up is based on the splitter implementation and chunk sizes calculated by the software. The importance of chunk sizes in relation to performance is further analysed in Subsection 6.2.4 while the reason is discussed in Subsection 6.2.6.

It should be noted that the result is in random order due to multiple threads processing the trace. Asserting the correctness of the result is done by sorting the result

which is not part of the speed-up. Sorting was a sequential process and therefore needed additional time. But it is up to the user if the order of events is necessary and if it can be done in a subsequent processing step. Another possible solution could be to write the results into separate files and provide them to the user if necessary.

6.2.3 Trace Data with Self-Dependency

As described in Subsection 5.2.2, this case calculates input streams that depend on themselves by creating a sub-specification based on the input and output streams but also the expression of those streams in the specification.

To test this case the specification shown in Listing 6.2 is used.

The script for generating an input trace (shown in Listing A.2) will randomly generate an event for x or y with values ranging from -2 to 2.

The specification will then sum up the values of all x and y events. This tests also the behaviour of multiple input streams having a self-dependency. Asserting the correctness is done by comparing the final results and their timestamps.

The number of events of the test sets for *Trace Data with Self-Dependency* are 6,993,000 (100 MB), 14,985,000 (250 MB), 29,970,000 (500 MB), 59,940,000 (1,000 MB) and 121,878,000 (2,000 MB). The results are shown in Table 6.2. It is shown that the multi-threaded approach is faster than the single-threaded one. The speed-up in this case is ranging from 3.91 to 4.72 with an average of 4.33.

```
1 in x: Events[Int]
2 in y: Events[Int]
3
4 def sum: Events[Int] := merge(last(sum, x) + x, 0)
5 def summation: Events[Int] := merge(last(summation, y) + y, 0)
6
7 out sum
8 out summation
```

Listing 6.2: TeSSLa specification for the test case of Trace Data with Self-Dependency

This test set has more events than the one used in Subsection 6.2.2 and the implementation (see Subsection 5.2.2) shows the additional steps which are necessary to gather the result in a parallel processing. Still the speed-up is higher.

This indicates that more complex specifications benefit more from parallelization. The multi CPU core approach splits the task of calculating the result of each trace chunk into multiple jobs. These are done in parallel and use a fraction of the

Table 6.2: Evaluation of Time needed in case of Trace Data with Self-Dependency

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	5.79	1.43	4.06
Events/Second	20,135	81,688	
250 MB	12.73	2.70	4.72
Events/Second	19,624	92,549	
500 MB	23.76	6.07	3.91
Events/Second	21,027	82,269	
1,000 MB	53.39	11.46	4.66
Events/Second	18,712	87,135	
2,000 MB	104.36	24.13	4.32
Events/Second	19,464	84,177	

time the single core approach would need. As shown in the implementation (see Subsection 5.2.2) some overhead is generated by using parallelization – namely the initialization of threads but also filtering the mapper result according to output variables. Even with larger traces that overhead seems not to have a heavy impact on the speed-up. Rather smaller files are less performant because of that additional overhead. In consequence this means that the simultaneous processing enables the use of the full computation power of the CPU at the cost of a small overhead.

6.2.4 Trace Data with Reset-Variable

This case adds a new input stream called *ResetVariable*. If the *analyser* reads that stream it marks a cut point for the *splitter* later on. More details are described in Subsection 5.2.3.

The specification shown in Listing 6.3 is used for this test case. The specification is basically the same as the one used in Subsection 6.2.3.

```
1 in E1: Events[Int]
2 in E2: Events[Int]
3
4 def count1: Events[Int] := merge(last(count1, E1) + E1, 0)
5 def count2: Events[Int] := merge(last(count2, E2) + E2, 0)
6
7 out count1
8 out count2
```

Listing 6.3: TeSSLa specification for the test case of Trace Data with Reset-Variable

The input trace generated with Listing A.3 adds a reset variable every one million events. Thus, the chunks each count a million events.

Table 6.3: Evaluation of Time needed in case of Trace Data with Reset-Variable

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	5.14	1.32	3.90
Events/Second	19,445	75,898	
250 MB	14.91	3.97	3.75
Events/Second	16,752	62,891	
500 MB	24.12	6.48	3.72
Events/Second	19,328	71,914	
1,000 MB	49.30	13.50	3.65
Events/Second	18,912	69,050	
2,000 MB	94.38	34.98	2.70
Events/Second	20,288	54,741	

In this configuration, Table 6.3 shows a speed-up of 2.70 to 3.90. The average is 3.54. The number of events are 5,994,006 for 100 MB, 14,985,015 for 250 MB, 27,972,028 for 500 MB, 55,944,056 for 1,000 MB and 114,885,115 for 2,000 MB.

This case also shows properly the importance of chunk size. The cases before used the configuration (number of CPU cores and 15 % of the maximal heap size of the Java Virtual Machine (JVM)) of the computer to calculate the number of chunks to prevent exceeding the memory while processing the TeSSLa specification.

Using the *Reset Variable* as input stream, but also manually set cut points, the choice of how large chunks will be is in the hands of the person writing the trace. The speed-up differs based on total trace size and frequency of cut points. This is shown by two additional configurations of the test set.

Previously the test set used a reset variable cutting the trace into chunks of one million events. To show the importance of chunk size the test set is cut into chunks of half a million events and two million events respectively.

Starting with half a million events the results are shown in Table 6.4. Compared to one million event chunks the processing speed of half million event chunks is always lower. At a trace size of 100 and 250 MB the chunk size of half a million events seems to be on par with one million event chunks. But in general a chunk size of one million events has a higher speed-up on the chosen chunk sizes. The highest difference is at a trace size of 1,000 MB or roughly 55 - 61 million events in total. At that size chunks with half a million events are processed at a speed-up of 2.66 while processing chunks with one million events achieve a speed-up of 3.65.

Table 6.4: Evaluation of Time needed in case of Trace Data with Reset-Variable cutting half million event chunks

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	5.37	1.41	3.82
Events/Second	18,617	71,035	
250 MB	13.99	3.68	3.80
Events/Second	17,262	65,536	
500 MB	24.69	7.83	3.15
Events/Second	19,217	60,626	
1,000 MB	55.49	20.88	2.66
Events/Second	18,604	49,431	
2,000 MB	94.27	45.88	2.05
Events/Second	19,782	40,644	

When increasing the chunk size to two million events the opposite is observed. The results are shown in Table 6.5. At a trace size of 100 MB the speed-up is less than in case of half or one million event chunks. Trace sizes above 100 MB (mostly) result in higher speed-ups regarding the processing time of those traces. The biggest difference is here at 250 MB or roughly 14,5 - 15 million events in total. The speed-up for one million event chunks is 3.75 while two million event chunks have a speed-up of 4.36.

Comparing two million event chunks to half a million event chunks leads to the highest differences at a trace size of 100 MB and 2,000 MB or roughly 6 and 112 million events respectively. For a trace size of 100 MB the half million event chunks achieve a speed-up of 3.82 while the two million event chunks processing speed-up is at 2.53. In case of 2,000 MB the half million event configuration achieves a speed-up of 2.05 and the two million event one a speed-up of 3.20.

The conclusion of this observation is that choosing a chunk size is critical to the performance of the parallel approach. While the calculation of chunk size based on the number of CPU cores and available memory enforces a safe execution, it is possible – by carefully placing a number of reset variables – to achieve a higher performance. The problem of finding the right amount of cut points and therefore a most efficient size for each chunk is an *optimization problem* and not part of this work.

Table 6.5: Evaluation of Time needed in case of Trace Data with Reset-Variable cutting two million event chunks

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	5.14	2.03	2.53
Events/Second	19,428	49,247	
250 MB	13.32	3.06	4.36
Events/Second	18,123	79,011	
500 MB	25.31	5.92	4.27
Events/Second	18,421	78,743	
1,000 MB	44.20	12.27	3.60
Events/Second	21,096	75,981	
2,000 MB	90.47	28.26	3.20
Events/Second	20,613	65,991	

6.2.5 Trace Data with N-Dependencies

Another implemented and already discussed case is a specification including streams with multiple dependencies. In Subsection 5.2.4 this is shown in detail with a specification checking multiple conditions to determine whether or not the system entered an error state. This specification is also used to evaluate the performance of *Trace Data with N-Dependencies* and is shown in Listing 6.4.

The trace for this test set was generated with the script shown in Listing A.4. The values of G are within -8 and 8 while the values of R lie in the range of -9000 to 9000. Lastly the values of V are within -300 and 300.

Compared to the other cases Table 6.6 shows that the speed-up is less for N-Dependencies. This is likely because of a higher complexity in processing such specifications.

```
1 in G: Events[Int]
2 in R: Events[Int]
3 in V: Events[Int]
4
5 def error: Events[Bool] := if (G < 0 || G > 6 || R < 0 || R > 8000 || V < 0 || V >
      200) then true else false
6
7 out error
```

Listing 6.4: TeSSLa specification for the test case of Trace Data with N-Dependencies

The speed-up in this case is ranging from 2.79 to 3.00 with an average of 2.88. The number of events processed are 5,994,000 (100 MB), 13,986,000 (250 MB), 27,472,500 (500 MB), 54,945,000 (1,000 MB), 109,890,000 (2,000 MB). The same trend of speed-up rates – as in previous cases – can be observed.

Table 6.6: Evaluation of Time needed in case of Trace Data with N-Dependencies

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	7.22	2.59	2.79
Events/Second	13,843	38,589	
250 MB	16.55	5.71	2.90
Events/Second	14,086	40,855	
500 MB	34.72	11.57	3.00
Events/Second	13,186	39,586	
1,000 MB	61.12	21.39	2.86
Events/Second	14,983	42,817	
2,000 MB	133.21	46.90	2.84
Events/Second	13,749	39,049	

6.2.6 Trace Data on smaller Heap

After mentioning the importance of chunk size in Subsection 6.2.4, this section shows the performance on a different size of the heap. The JVM uses a default maximum heap size of 25 % of the RAM. This was observed in this thesis but is also stated in [Ata18].

Thus the size of the heap – given the hardware configuration presented in Section 6.1 – is roughly 4 GB. Every used trace size in the test sets is within the bounds of that heap and because of that *in-memory size*. To test the behaviour of the implemented algorithms and parallel engine in case the input is larger than the available memory – and as a consequence at least *single-hard-disk size* instead of *in-memory size* – the size of the available heap space of the JVM is reduced in this test set. It is worth mentioning that the implementation of this work does not use the whole heap of the JVM for computation. In some cases the heap is sized at only around 1 - 2 GB of the assigned 4 GB. But as shown in the evaluation in this section there is a correlation between speed-up and available heap size for the computation. It can be observed that the *Garbage Collector* has to do a lot more deconstructing and deallocating of objects based on the available heap size. In consequence this decreases the speed-up because of the CPU (cores) being busy with tasks of the Garbage Collector instead of computing the result of the trace at hand.

The TeSSLa specification shown in Listing 6.2 was used as well as the trace of Subsection 6.2.3. The heap size was limited to 64 MB by adding the JVM Option *-Xmx64m*.

Table 6.7: Evaluation of Time needed in case of using a Heap smaller than the Trace

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
100 MB	6.80	2.60	2.62
Events/Second	17,143	44,836	
250 MB	14.31	6.61	2.17
Events/Second	17,447	37,809	
500 MB	25.70	18.93	1.36
Events/Second	19,443	26,389	
1,000 MB	53.56	58.62	0.91
Events/Second	18,653	17,041	
2,000 MB	106.60	214.67	0.50
Events/Second	19,055	9,462	

The memory consumption using parallel processing is higher than processing sequential. Multiple collections or iterators are created, consumed and referenced while using multiple threads. Introducing a *Reset Variable* showed the difference in processing speed based on the chunk size. It was mentioned that finding the best chunk size for the hardware configuration can improve the performance. Regarding heap size the statement is a lot easier. If more memory is available less CPU power is used for garbage collection but also input/output operations like reading and writing chunk files. But it should also be mentioned, as a reminder, that the default chunk size is calculated by the parallel engine based on available heap space. In consequence this means that more space results in larger chunk sizes.

Comparing the results of Subsection 6.2.3 with the same run on a smaller heap (as shown in Table 6.7) it is observed that the achieved speed-up is lower. In case of the 100 MB trace its speed-up is 2.62 instead of the previous 4.06 and it steadily decreases with larger traces. The average for this smaller heap situation is 1.51 while processing *Trace Data with Self-Dependency* with a heap size of 4 GB has an average of 4.33.

The worst case is a speed-up below 1 and consequently a lower performance with multiple threads than with a single thread. The core problem is the splitter as the default algorithm of calculation currently is based on the maximum heap size and splitting is done sequentially. Setting the maximum heap size to 64 MB reduces the chunk size. Thus more chunks are generated and the amount of I/O operations increases as well. Less time is used on computation of the specification than for

overhead tasks like creating threads and allocating memory. It needs to be tested and evaluated but in an upscaled environment this should create the same issue. This observation indicates that with a large discrepancy between heap and trace size the speed-up declines more rapidly. In Subsection 6.2.3 the splitter needed 68.76 - 100.17 seconds to create chunks for the 2,000 MB trace. With a smaller heap the time of the splitter rose to 10,000 - 12,357 seconds. Because of this, the aforementioned *online splitting* (see Subsection 5.2.3) should be further reviewed as a possible countermeasure. Another solution is the use of a *Reset Variable* and thus manually setting the cut points of the chunks.

6.2.7 A Real-World Example: Idle Engine

A company recorded sensor data of an engine. A TeSSLa specification is used to determine whether or not that engine is in an idle state. The idle state is defined as an engine speed below 100 rpm. The specification is shown in Listing 6.5.

```
1  include "inputs.tessla"
2
3  def EngSpeedThreshold := 100
4
5  def idle := EngSpeed <= EngSpeedThreshold
6  out pureBool(idle) as idle
7
8  def t := time(EngSpeed) / 10s
9  out pureInt(t) as t
10
11 def pureInt(x: Events[Int]) := if default(x != last(x,x), true) then x else 0
12
13 def pureBool(x: Events[Bool]) := if default(x != last(x,x), true) then x else true
```

Listing 6.5: TeSSLa specification for a real-word example of an idle engine

Only a single trace is given. That trace has the size of 18.7 MB and roughly 591,000 events. As shown in Table 6.8 the speed-up is at 2.42.

Table 6.8: Evaluation of Time needed in case of Real-World Example: Idle Engine

Size	Single Thread Time (Minutes)	Multi Thread Time (Minutes)	Speed-Up
18.7 MB	0.27	0.14	1.92
Events/Second	36,777	70,631	

This test case also introduced a new case to the parallel engine. Until then only the case using the *last* operator with a recursion existed. The specification shown in Listing 6.5 introduced the case of using the *last* operator without leading to a recursion. As already mentioned, *Trace Data with Self-Dependency* uses the *last* operator but in combination with recursion. *Trace Data without Dependencies* neither uses the *last* operator nor a recursion. *Last* without a recursion behaves the same as *Trace Data without Dependency* though because only the last value is used. Thus no dependencies to other streams exist in this case, too.

7 Conclusion and Outlook

This chapter concludes the work, discusses the results and possible limitations as well as problems but also improvements to the concept and design. The outlook will focus on topics for future works.

7.1 Conclusion

Different cases of specifications (see Section 4.3) and their problems were identified. Algorithms were developed to compute the result of a trace and specification in parallel. This list of cases and algorithms is not exhaustive and future works might present unhandled cases as well as algorithms for them. The problem of processing the input trace was decomposed into smaller tasks, computed in parallel and evaluated in comparison to a sequential approach. The discussed algorithms (see Section 5.2) provided specific solutions for parallel processing of traces for specifications based on the MapReduce approach presented in [DG04].

In Subsection 4.3.4 the use of *speculative decomposing* was discussed for solving specifications with multiple dependencies. This method was dropped later for a much simpler solution of *overlapping* input streams to adjacent chunks of the trace. Nonetheless speculative decomposition could be a promising approach for the unhandled cases of specifications with automata (see Subsection 4.3.5) as transitions could be modelled as branches. This situation is similar to the behaviour of some CPUs choosing a branch for execution which is in fact a currently used method to improve the performance of a CPU [MG13].

The evaluation in Chapter 6 showed an improved performance in comparison to a single-threaded computation. The speed-up ranged from 2.05 to 4.72 on an eight-core CPU. Besides measuring an increased performance, some limitations were also observed.

The test cases in Subsection 6.2.4 presented the importance of chunk sizes in regards to the performance by using a *Reset Variable* instead of letting the system calculate the chunk size based on the available heap size. Using a manually set *Reset Variable* in the trace to force a cut at the *Reset Variable* resulted in higher speed-ups at larger input traces if the chunk size increased as well. The opposite was observed if there

were less chunks than CPU cores or if the chunks were too small. The reason for the latter is that the used *Splitter* is working sequentially. Cutting a large trace into smaller chunks can lead to an overhead that is larger than the computation of the chunk itself. *Online Splitting* was discussed in Subsection 5.2.3 and Subsection 6.2.6 as a possible solution to reduce the overhead of the Splitter. If this approach is used in a productive environment the architecture of the parallel approach should be switched to a more event based processing architecture. Running the parallel engine as coordinator while listening for events of incoming traces/chunks and results of the submodules (Splitter, Mapper, Reducer) as well as informing those submodules via events might allow for a higher level of parallelization.

As described in Section 3.3, TeSSLa is designed for *explicit memory usage*. The JVM still needs to store references and objects used by TeSSLa. In consequence the garbage collector will at some points will destruct objects and deallocate memory on the heap. Therefore based on the available heap and trace size but also the level of parallelization the garbage collector will be called more often and thus using more CPU time. Testing on smaller heap sizes (see Subsection 6.2.6) showed this behaviour and subsequently the reduced speed-up compared to the other test cases. For larger traces, an even lower performance than with a sequential computation was observed. The test case for showing this behaviour was constructed and might not happen too much in the field. Nevertheless optimizing the point of cutting a trace into chunks has a big impact on the performance and might lead to such behaviour. Further works should focus on that part too so that the drop of performance is lessened on larger traces (see Table 6.1). One possibility to handle such cases is the already mentioned Online Splitter or exploring a distributed solution using a MapReduce or similar parallel computation framework that allows to inject customizable algorithms.

No solution for combining different cases of specifications is presented in this work. Each case is handled in singularity. This is a limitation one needs to be aware of. On the other hand, in its current state the usage of SRV in the industry is sparse and thus more real world examples are needed to show if there is a necessity for combining multiple cases such as those presented in this work or if those cases are rare or can be computed in smaller but multiple steps instead of one large combined computation. The aforementioned injection of customizable algorithms forces that problem onto the engineer that wants to apply SRV on a specification that uses multiple cases or is not handled by the algorithms designed for generic specifications.

This work serves as a proof of concept for the topic of parallel computation of large traces with stream based specifications. To be as flexible as possible, this work did not use any MapReduce frameworks so that the necessary algorithms could be freely customized. Nevertheless the implemented and evaluated approaches for the specific cases are based on MapReduce and because of that it should be possible to port

these algorithms on frameworks like *Apache Hadoop*. This would enable the usage of multiple computation nodes and thus distributed parallel processing and even lessen limitations based on hardware configuration and in consequence the heap size problem.

7.2 Outlook

The improved performance due to parallelization suggests that other works should dive further into the topic of making SRV more applicable for real world use cases. TeSSLa enables a more fluent way of defining specifications and handles the part of transforming those specifications into a RV monitor to check the logical properties based on an input trace. On the other hand, the impact of chunk and heap size was shown in this work and impedes the performance of an applied algorithm. The problems and limitations of the parallel approach were described and should be subject of future research. Namely in the field of engineering, improvements to the architecture and discussed processing algorithms in form of using *Online Splitting*, *Event-based Parallel-Architectures* and distributed parallelization frameworks in general should be explored as an infrastructure to deploy the described algorithms and those that follow in the future. A more logical approach would be to solve the problem of finding the best chunk size based on a trace and given hardware configuration.

Nevertheless, this proof of concept was successful in showcasing novel ways of parallelizing large traces of stream-based specifications. The benefit of parallelization in real world cases seems crucial based on the size of a trace. Depending on the complexity of the specification it might be necessary to design a parallel algorithm for that specification. A groundwork is described in this work. Future works not only in science but in the industry as well will tell if those solutions presented here can be used on established frameworks like *Apache Hadoop* or *Apache Spark* or will be used to write a specific SRV framework around languages like TeSSLa. Providing examples or implementations of algorithms for generic specifications but also being able to add custom algorithms might be crucial for practical use of parallelization for large traces of stream-based specifications.

A Appendix

A.1 Test-Case Trace Scripts

In this section the NodeJS scripts used for the test-cases of Section 6.2 are listed. These scripts were run with NodeJS v10.16.3.

```
1  const fs = require('fs');
2  var filename = process.argv[2] || "output.trace";
3  var iterations = process.argv[3] || 1000 * 1000 * 1;
4  var str = '';
5
6  fs.writeFileSync(filename, str);
7
8  for (let i = 0; i < iterations; i++) {
9    if (i % 1000 === 0) {
10         console.log(`Writing lines ${i} to ${i + 1000}`);
11         fs.appendFileSync(filename, str);
12         str = ''
13   } else {
14         str += `${i}: value = ${getRandomInt(2) > 0 ? '-' : ''}$
               {getRandomInt(12)} \n`;
15   }
16 }
17
18 if (str) {
19   fs.appendFileSync(filename, str);
20 }
21
22 function getRandomInt(max) {
23   return Math.floor(Math.random() * Math.floor(max));
24 }
```

Listing A.1: Trace Generation Script for Trace Data without Dependencies

The scripts accept two parameters, the output file name and the number of lines to create. An example to create a file called *inbound.trace* for the specification shown in Figure 3.1 with 80.000.000 input events would be:

node create-no-last-no-recursion-trace-file.js inbound.trace 80000000

The *create-no-last-no-recursion-trace-file.js* in this case is the script shown in Listing A.1.

```
1  const fs = require('fs');
2  var filename = process.argv[2] || "output.trace";
3  var iterations = process.argv[3] || 1000 * 1000 * 1;
4  var str = '';
5
6  fs.writeFileSync(filename, str);
7
8  for (let i = 0; i < iterations; i++) {
9    if (i % 1000 === 0) {
10         console.log(`Writing lines ${i} to ${i + 1000}`);
11         fs.appendFileSync(filename, str);
12         str = ''
13   } else {
14         str += `${i}: ${getRandomInt(2) > 0 ? 'y' : 'x'} = ${
               getRandomInt(2) > 0 ? '-' : ''}${getRandomInt(2)} \n`
15   }
16 }
17
18 if (str) {
19   fs.appendFileSync(filename, str);
20 }
21
22 function getRandomInt(max) {
23   return Math.floor(Math.random() * Math.floor(max));
24 }
```

Listing A.2: Trace Generation Script for Trace Data with Self-Dependency

```
const fs = require('fs');
let filename = process.argv[2] || "output.trace";
let iterations = process.argv[3] || 1000 * 1000 * 1;
let resetVariableAfter = 1000 * 1000;
let str = '';

fs.writeFileSync(filename, str);
let i;

for (i = 0; i < iterations; i++) {
  if (i % resetVariableAfter === 0 && i !== 0) str += `${i}: ResetVariable = 0 \n`;
  if (i % 1000 === 0) {
      console.log(`Writing lines ${i} to ${i + 1000}`);
      fs.appendFileSync(filename, str);
      str = ''
  } else {
      str += `${i}: ${getRandomInt(2) > 0 ? 'E2' : 'E1'} = ${getRandomInt(2) > 0 ? '-' : ''}${getRandomInt(2)} \n`
  }
}

if (str) {
  str += `${i}: ResetVariable = 0 \n`;
  fs.appendFileSync(filename, str);
}

function getRandomInt(max) {
  return Math.floor(Math.random() * Math.floor(max));
}
```

Listing A.3: Trace Generation Script for Trace Data with Reset-Variable

```
const fs = require('fs');
let filename = process.argv[2] || "output.trace";
let iterations = process.argv[3] || 1000 * 1000 * 1;
let str = '';

fs.writeFileSync(filename, str);
let i;

for (i = 0; i < iterations; i++) {
  if (i % 1000 === 0) {
        console.log(`Writing lines ${i} to ${i + 1000}`);
        fs.appendFileSync(filename, str);
        str = ''
  } else {
        switch (getRandomInt(3)) {
          case 0:
                str += `${i}: G = ${getRandomInt(2) > 0 ? '-' :
                    ''}${getRandomInt(8)} \n`
                break;
          case 1:
                str += `${i}: R = ${getRandomInt(2) > 0 ? '-' :
                    ''}${getRandomInt(9000)} \n`
                break;
          case 2:
                str += `${i}: V = ${getRandomInt(2) > 0 ? '-' :
                    ''}${getRandomInt(300)} \n`
                break;
        }
  }
}

if (str) {
  fs.appendFileSync(filename, str);
}

function getRandomInt(max) {
  return Math.floor(Math.random() * Math.floor(max));
}
```

Listing A.4: Trace Generation Script for Trace Data with N-Dependencies

List of Figures

List of Tables

List of Theorems and Definitions

Table of Listings

Abbreviations

IoT	*I*nternet *o*f *T*hings
RV	*R*untime *V*erification
TeSSLa	*Te*mporal *S*tream-based *S*pecification *La*nguage
SRV	*S*tream *R*untime *V*erification
MFOTL	*M*etric *F*irst-*O*rder *L*ogic
LTL	*L*inear *T*emporal *Logic*
HOL	*H*igher-*O*rder *L*ogic
AST	*A*bstract *S*yntax *T*ree
DFA	*D*eterministic *F*inite *A*utomaton
NFA	*N*ondeterministic *F*inite *A*utomaton
JVM	*J*ava *V*irtual *M*achine

Bibliography

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54, 2010. Elsevier North-Holland, Inc.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[Ata18] Roman Ataman. JVM Memory Settings in a Container Environment, 2018.

[BCE+14] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. Scalable offline monitoring. In *Runtime Verification*. Springer LNCS, 2014.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[Buc85] Bruno Buchberger. *Symbolic Computation (An Editorial)*. Academic Press Inc., Harcourt Brace Jovanovich Publishers, London, 1985.

[CHL+18] Lukas Convent, Sebastian Hungerecker, Martin Leucker, Torben Scheffel, Malte Schmitz, and Daniel Thoma. Tessla: Temporal stream-based specification language. In *Formal Methods: Foundations and Applications*. Springer LNCS, 2018. Conference: SBMF 2018.

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004. USENIX Association.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15, 1972.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. Lola: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. IEEE Computer Society, 2005.

[Hal08] Thomas C Hales. Formal proof. *Notices of the AMS*, 55, 2008. American Mathematical Society.

[HKG17] Sylvain Hallé, Raphaël Khoury, and Sébastien Gaboury. Event stream processing with multiple threads. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*. Springer International Publishing, 2017.

[Kum02] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78, 2009. Elsevier BV.

[MG13] Pece Mitrevski and Marjan Gusev. On the performance potential of speculative execution based on branch and value prediction. *FACTA UNIVERSITATIS Series Electronics and Energetics*, 16, 2013.

[Ost87] G. Ostrouchov. Parallel computing on a hypercube: An overview of the architecture and some applications. In *19th Symposium on the Interface of Computer Science and Statistics*. American Statistical Association, 1987.

[Par09] T. Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 2009.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE Computer Society, 1977.

[SBB+18] Joshua Schneider, David Basin, Frederik Brix, Srđan Krstić, and Dmitriy Traytel. Scalable online first-order monitoring. In *Runtime Verification*. Springer International Publishing, 2018.

[TSBR18] Roman Trobec, Bostjan Slivnik, Patricio Bulic, and Borut Robic. Introduction to parallel computing. In *Undergraduate Topics in Computer Science*. Springer International Publishing, 2018.