DYNAMIC ANALYSIS FOR COMPLEX EVENT PROCESSING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

MUHAMMET OĞUZ ÖZCAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

DECEMBER 2015

Approval of the thesis:

DYNAMIC ANALYSIS FOR COMPLEX EVENT PROCESSING

submitted by **MUHAMMET OĞUZ ÖZCAN** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver Dean, Graduate School of **Natural and Applied Sciences**	______________
Prof. Dr. Gönül Turhan Sayan Head of Department, **Electrical and Electronics Engineering**	______________
Assoc. Prof. Dr. Ece Guran Schmidt Supervisor, **Electrical and Electronics Eng. Dept., METU**	______________
Prof. Dr. Ali Hikmet Doğru Co-supervisor, **Computer Engineering Dept., METU**	______________

Examining Committee Members:

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı Electrical and Electronics Department, METU	______________
Assoc. Prof. Dr. Ece Guran Schmidt Electrical and Electronics Department, METU	______________
Prof. Dr. Ahmet Coşar Computer Engineering Department, METU	______________
Prof. Dr. Halit Oğuztüzün Computer Engineering Department, METU	______________
Prof. Dr. Semih Bilgen Computer Engineering Department, Yeditepe University	______________

Date: ______________

ABSTRACT

DYNAMIC ANALYSIS FOR COMPLEX EVENT PROCESSING

Özcan, Muhammet Oğuz
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Ece Guran Schmidt
Co-Supervisor : Prof. Dr. Ali Hikmet Doğru

December 2015, 93 pages

Analysis facilities are developed in the course of this thesis for a domain-specific real-time and rule-based language along with a supporting tool. Such analysis facilities are required due to the need for investigating the functional correctness and stringent timing properties expected to take place in the software developed through this language. An early version of this language was developed during a Ph.D. study for the domain of fault management in mission critical systems. Five program analysis facilities are proposed and tested with randomly generated numbers of events and rules. Also, discussions about static and dynamic analysis in the event processing domain are presented along with a comparison of related existing tools. The comparisons of existing tools include the two different implementations of the similar design for interpreters for the language. The different implementations involved the languages C++ and Python.

Keywords: Real-Time, Rule-Based Languages, Complex Event Processing, Dynamic Analysis, Fault Management Systems, Domain Specific Languages

ÖZ

KARMAŞIK OLAY İŞLEME İÇİN DİNAMİK ANALİZ

Özcan, Muhammet Oğuz
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Ece Guran Schmidt
Ortak Tez Yöneticisi : Prof. Dr. Ali Hikmet Doğru

Aralık 2015 , 93 sayfa

Bu tezde gerçek zamanlı ve kural tabanlı bir alana özel dil için analiz kabiliyetleri ve bir yorumlayıcı geliştirilmiştir. Analiz kabiliyetleri, dilin kullanılması ile geliştirilecek yazılımlarda gerekecek olan fonksiyonel doğruluk ve katı zaman kısıtlarının irdelenmesi icin önerilmiştir. Bu dil ilk olarak görev kritik sistemlerin hata yonetimi icin gerekmiştir ve bir başka doktora çalışmasında geliştirilmiştir. Beş tip analiz kabiliyeti geliştirilmiş ve rastgele sayıda kural ve olay ile denenmiştir. Olay işleme alanında statik ve dinamik analiz seçenekleri incelenmiştir ve benzer araçlarla karşılaştırma da yapılmıştır. Ayrıca karşılaştırmalar, aynı tasarıma tabi olan dil yorumlayıcısının C++ ve Python ile iki farklı uygulamalarını da içermektedir.

Anahtar Kelimeler: Gerçek Zamanlılık, Kural Tabanlı Diller, Karmaşık Olay İşleme, Dinamik Analiz, Hata Yönetim Sistemi, Alana Özgü Diller

To my parents and girlfriend

Hasan Özcan, Ayşe Özcan, Hilal Özcan, Mehtap Safi

ACKNOWLEDGMENTS

I would like to thank my supervisors Associate Professor Ece Guran Schmidt and Professor Ali Hikmet Doğru for their constant support and guidance. It was a great honor to work with them for the last two years and our cooperation influenced my academical and world view highly. I also would like to thank Dr. Selma Süloğlu for her support and guidance on my studies.

My family has always provided invaluable support to me. I would like to thank specially to my father, mother and sister. They always make me feel loved and cared. I am also thankful for all love and support from my girlfriend and future wife Mehtap Safi.

My friends, Mehmet Şerif Öncü, Ekin Kaya, Nuri Armağan, Furkan İlbahar, Elif Solmaz who have been with me over 10 years have very special place in my life. I would like to thank them all for their constant support and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
AOF	Activation Order Flowgraph
CEP	Complex Event Processing
EQL	Equational Logic Language
GAA	General Analysis Algorithm
LHS	Left Hand Side
MRL	Macro Rule-based Language
PM	Production Memory
RHS	Right Hand Side
RPN	Reverse Polish Notation
SSG	State Space Graph
WM	Working Memory

CHAPTER 1

INTRODUCTION

In this thesis study, a simulator is developed to analyze various aspects of the programs written using a domain specific language. This language has the following properties: it is rule based, it can process events with real time responsiveness, and it can process complex events. Five analysis facilities are proposed to support this language. The initial version of the language has been developed before, during a PhD study for utilization in the development of mission critical fault management systems [20]. It's requirements have originated in a project. Referred to as KOTAY in some project documentation, unfortunately the language does not have a well published or documented name. The need for analysis facilities for the language surfaced before using it in development, to assess the reliability of the systems to be developed using this language and to simulate the system beforehand. Various adaptations of KOTAY is used in real-time mission-critical systems for fault management through complex event processing (CEP).

KOTAY name is an abbreviation in Turkish that includes the 'rule-based, event-based, and fault management' terminologies (Kural ve Olay Tabanlı Arıza Yönetimi). The facilities behind such terminology are explained in the following paragraphs. In general, CEP languages are implementing some kind of event algebra. There has been some event algebras introduced and supported with programming languages. Most of those however, are very complex and not suitable for real-time processing. A simple and efficient to implement such language has been the Generalized Event Monitoring (GEM) language [38]. KOTAY has largely utilized the concepts in GEM. Basically, GEM was adapted with some simplifications and some additional definitions for optimization towards time performance efficiency. Besides the language supporting run-time performance, the language processors (compiler or interpreter as in our case) efficiency becomes a critical factor. Part of the work conducted for this thesis includes the performance improvement for the interpreter.

Previously existing interpreter was implemented in the Python language. Although it was fast enough for the specific domain and the requirements for the originating project, Python is interpretive and hence, slow. This thesis aims at providing a wider application opportunity for the language and so making use of any chance to further improve the speed performance. A faster version was implemented for this thesis, utilizing the faster run-time code, first of all as a result of being a compiled executable. Also, the data structures that are heavily based on tables in Python, were mostly converted to various data structure making use of pointers. One principle in the new interpreter is, using as much memory as required since there is no space constraint, for the benefit of faster run-time response. The results related to speed-up are provided in the late sections as time responses collected during the tests run using random input events and randomly generated rules. Although, there is no direct measure to compare the speed-up a general idea can be perceived looking at the measurements conducted in the previous case (Python) and this one.

Rule-based languages consist of sets of rules and are usually used in expert systems. Most of them comprise "if Condition, then Action" structures. When the condition part is satisfied, the action part executes. An advantage of rule-based languages is the high abstraction level they offer: Every rule can be coded independent of the others, without considering an execution order. Real-time systems are decision systems which respond to external events and conduct decisions based on input and state information in a pre-determined time period. Usually this response time is very short. CEP is a technology used to analyze and track streams of data from multiple sources and identify meaningful complex events to be able to respond to them quickly. Complex events are the composition of events with logical, causal and temporal operators. Domain specific languages, in contrast to general purpose languages such as Java, C, or Pascal, are defined for development in a special application field, easily. The field associated with the language involved in this research is fault management.

The analysis facilities are specific to KOTAY. However, they can be adapted to other languages with similar properties. These facilities are mentioned in a language independent manner, and they should be applicable to the languages that include the corresponding capability for each analysis type. This thesis work includes the development of the facilities on the interpreter of KOTAY, as a supporting tool for the language. For other languages, a similar implementation effort will be required. Most of the analysis functions would be applicable to any event-processing language.

Real-time systems are important for many military and business related technologies. There are two main requirements for these systems; functional correctness and stringent timing constraints where exceeding the upper bound could cause fatal consequences. These two main requirements need to be analyzed very carefully before using the systems in their real applications to avoid undesirable consequences.

Aim of timing requirement analysis studies is finding whether the system has a bounded response time and if there is, what the response time is. Existing timing analysis studies are all related to formal or static analysis and none of them are defined for event processing languages [1], [2], [3], [4], [5], [6], [26], [27], [28], [29], [30], [31]. Although static analysis is a very important subject, it does not give the real system response results and can be considered as inadequate for some environments. Therefore the importance of dynamic analysis for the software that are part of such systems emerges. Difference between static and dynamic analysis will be explained in the following paragraphs.

Functional correctness analysis relates to validation and verification (V&V) of software [24]. Validation is determining the correctness of a software according to the requirements and verification is determining the completeness and consistency aspects. Consistency analysis includes determination of redundant, conflicting, subsumed and circular rules, and unnecessary if conditions. Completeness analysis includes determination of the dead-end, missing and unreachable rules [24]. Some simulators exist in the literature that provide the aforementioned analysis facilities. The tools and the existing studies cover only static analyses and do not have CEP capability [24], [25], [34], [35], [36], [37]. Therefore, a tool that provides dynamic functional correctness analysis of event processing rule-based programs is required.

It is stated that many of rule-based languages are not suitable for real-time applications in [13]. Representing knowledge in the real-time domain can be problematic [20]. This opinion supports the need for the created language and implicitly shows that analysis of these kind of languages are also important. Besides, in [12] it is directly stated that most of the CEP studies do not include real-time processing, they focus on improving the performance and handling large volumes of streaming data. It is also stated that instead of the real-time properties, they are more focused on accomplishing maximum throughput and minimum end-to-end latency.

In [8] the analysis of event based systems are divided into two parts; static analysis and dynamic analysis. Static analysis considers analysis before run-time. It is mostly used to validate the design of the application and it shows possible errors that might occur when the system is actually working. To be completely certain about the results, dynamic analysis must be conducted. Dynamic analysis is conducted at run-time. It might provide some results that cannot be understood in static analysis. It is done based on the observation of runtime execution of the application. Because of these advantages, we have added dynamic analysis facilities into our simulator. We have also added one static analysis facility which is the static cycle warning.

The advantages of dynamic analysis over static analysis can be combined in three main topics. Firstly, dynamic analysis enables to detect if the system terminates at some point or in other words, complete its execution. This is called the termination problem detection. The termination problem might occur because of a loop involving some rules, resulting with infinitely executing cycles. Rather than predicting it through static analysis, dynamic analysis gives the real consequences [8]. Secondly, some of the rules in the existing rule set may never fire during the system execution. These rules are called unreachable rules and the situation is called the reachability issue. Those rules are redundant and must be eliminated from the rule set. Detection of these kind of rules with static analysis is not conclusive and not possible for all different situations. Some of the rules that are stated as reachable after a static analysis may never fire during the system execution. Therefore, dynamic analysis is required in order to be certain about the results of the reachability issue detection [8]. Thirdly, after condition parts have completed, action parts of the rules are executed and they produce some outputs which are called generated events in the event processing domain. In some cases these outputs may not be used by the other rules in the system. Static analysis is not suitable for this analysis, since the consumption of the generated outputs depends on the current situation of the system. Static analysis may only provide a checking mechanism for the existence of generated outputs in the condition parts of the rules. With dynamic analysis this situation can be detected easily with a guarantee. Therefore, preferring dynamic analysis seems to be a better choice since it provides real results rather than predictions.

There are two types of dynamic analysis: simulation and tracing [8]. Creation of random test data, using these data in the system and observing the results of scenarios are defined as simulation. Taking the traces of execution and analyzing them is called tracing. In this study, simulation type of a dynamic analysis approach is preferred. The simulator and created analysis facilities were tested with randomly created event and rule sets.

In this thesis, we created a dynamic simulator and five analysis facilities to be able to analyze the programs developed using the mentioned language. The analysis facilities are required to test the software applications before using them in mission critical systems. Existing studies in this area either do not include a comprehensive simulator, or included simulators do not provide dynamic analysis and complex event processing facilities.

Contributions can be summarized in two fields: 1- Improving the existing language interpreter for performance and 2- developing analysis facilities for the

CEP field. The 2nd contribution is an innovation that differentiates this research.

The thesis continues as follows: In Chapter 2, some foundation technologies and literature survey in the following subcategories: Real-Time Systems, Rule-Based Languages, Complex Event Processing, Rete, and Structure of the Language are introduced. Chapter 3, defines the work done in the process of simulator development. Creation of analysis facilities and experimentations are presented in Chapter 4. Chapter 5 includes conclusion

CHAPTER 2

BACKGROUND AND LITERATURE SURVEY

2.1 Real-Time Systems

Real-time systems are computer controlled systems that respond to external events and decide based on inputs and state information in a pre-determined and usually very short time. Real-time systems range over several domains in computer science. Defense systems in "Command and Control Information Systems (C2IS)", space systems in space stations, hospital systems in intensive care units and emergencies, automobile networking systems in "Anti-lock Braking Systems (ABS)" are some of the very critical systems where real-time systems are being widely used. Real-time systems generally are critical systems, where a failure might cause catastrophic consequences. In real-time systems the correctness of the system depends not only on producing the effects of logical computation and result of that computation, but also on the physical time that these results are produced and also how fast they are produced [18], [19].

There are many definitions of real-time in the literature, all of them are somewhat similar to each other. A system is defined as real-time if it is "fast" or "faster than human" in a very basic way [12]. In [13] real-time systems are defined as the systems that have operational deadlines for the processing of events. Their processing periods start with the occurring of the events and ends with the response of the system. A real-time term is defined as a system that responds to incoming data at a rate fast or faster than it is arriving [13]. Lastly, real-time is defined as a feature, which is the ability of a system that can guarantee a response after some time has elapsed. The crucial point in that definition is that the elapsed time must be provided as a part of problem definition [13]. From all of these definitions it can be stated that a real-time system needs to respond to inputs that are entering the system quickly with some limitations on system resources.

Real-time systems comprise two sub-systems: controlling system and controlled system. As its name implies a controlled system is the environment that is being directed by something else, the controlling system. Whereas the controlling system leads the environment, (i.e. the controlled system), with information gathered from various sensors [19].

Timing constraints of real-time systems may also vary depending on the system requirements. Basic types of real–time systems can be stated as being "periodic" or "aperiodic. "Periodic" means events happen continuously with some time between them which is generally defined as "T" time units. Whereas, "aperiodic" means events that have beginning and ending times with no strict time between their occurrences. More complex timing requirements can also be defined in real-time systems, for example in a case where an event having to occur before some other one.

If timing requirements are not fulfilled, the result will be failure and fatal consequences might happen. Therefore, designing and creating a real-time decision system is an important and a critical job. It requires extra attention and detailed calculation when analyzing the time bounds.

Finally, the relation between real-time systems and rule-based languages should be mentioned. Actually these two concepts are orthogonal. Trying to support the real-time environment with a rule-based language – that consequently, has to perform fast and more deterministically also, is due to the expressive power of the rules. Especially in situations where a domain specific program needs to be developed, such as medical diagnosis or fault management, the domain related concepts can directly take place in the code rather than lower-level statements like those of C or Java. Besides, it is expected to support the rapid development of new fault management programs through coding by rules.

2.2 Rule-Based Languages

A rule is a statement that describes a policy or a procedure. Rules can be atomic or complex. Atomic rules cannot be broken down any further, whereas complex rules include many rules and they can be broken down to lower levels. Some event processing applications are based on rule-based languages. Therefore, a rule can be defined as the basic processing primitive of event processing in some areas.

Rule-based languages are mainly studied in "Artificial Intelligence", "Knowledge Based Systems", and "Expert Systems". They are used in a variety of practical systems through storing and manipulating data for making decisions intelligently. Most of the rule-based systems use "If-Then" structured rules to represent knowledge and to make decisions. The number of if-then structured rules can vary from system to system and can be very big depending on the system's capacity [14].

Making an inference corresponds to firing a rule in most rule-based expert systems. If all the necessary conditions take place then rules fires, in other words action part of the rule generates the system an input [15].

In [8] these types of languages are referred to as "Rule Oriented Languages". Then, three types of rules are defined: production rules, active (event-condition-action) rules, and rules based on logic programming.

Production Rules:

These types of rules comprise "if-condition then-action" forms. When the condition part is satisfied in a rule, its action part is fired. This type of rules operate in a forward chaining way. They are mainly used in expert systems. Two types of operational processing exist for these rules; declarative production and procedural production.

Declarative production rules:

Since these rules can fire in no pre-determined sequences, a different kind of the Rete algorithm can be applied for interpreting these rules. This algorithm contains three main cycles; match, select, and act cycles. When a condition in a rule set matches one of the patterns, that rule is selected and the state of the rule changes. In that match cycle the history is also stored in an internal state, so that there is no information loss and evaluation is done easily.

Procedural production rules:

These rules are executed in a sequential manner. They contain a series of execution steps to be carried out and any compiled rule can be executed at any time during the execution, by other rules or by itself.

Actually production rules are not based on events, they are based on state changes. To support event processing capability some languages made events as an explicit part of the model, so event occurrences are used in the condition parts of the rules and thus events can be used for invoking a rule.

Active Rules:

These types of rules are also known as "event-condition-action" (ECA) rules. They work as follows: When an event occurs, (this event can be primitive or composite), if that event is found in the condition part, then condition part is evaluated. After evaluation there are two cases, if condition part is fully satisfied then action part is fired. If the condition part is premature, then waiting continues for occurrences of further events.

Logic Programming Languages:

Logic programming is based on the notion of logical deductions in symbolic logic. The goal of logic programming is to state what needs to be done, not how to do it. It aims to separate logic from control. Prolog (PROgramming in LOGic), is the first logic programming language. It is not a functional programming language, but rather it is a relational programming language. Therefore, it is better to think about these types of languages as in the case of working with a data base. Three types of statements exist in prolog: Facts, Rules, and Queries. They also can be stated as Hypotheses, Conditions, and Goals. The language is looking for all answers that satisfy the query. Therefore, the language is thought as non-procedural or non-deterministic.

2.3 Complex Event Processing

To be able to fully understand CEP, firstly the meaning of "event" needs to be clarifed. There are different definitions of "events" that are stated in different studies. The event concept is explained with two meanings in [8]. Basic definition of event is "something that has happened". This definition is based solely on the real world, in other words this definition is given by only taking the real world occurrences in a particular system or domain into account. Event is also defined as "contemplated as having happened" in a domain or system. What is meant by the 'contemplated' word is that, things that have not actually occurred in reality, instead some mimicking of the incident took place. Here the domain and system words deserve extra explanation. Event processing concerns two domains and systems: mainly it deals with real world incidents, things like gun fire, plane crash or doorbell ringing are the real world incidents. Additionally event processing also deals with artificial domains. Incidents in virtual worlds like those created in training simulations are also considered as events [9]. The event word is also used to represent an occurrence in computing environments via a programming entity. It can be referred to as "event object", however is not necessarily an object as in object-oriented programming concepts. A record in a database, a structure in programming language C or a message sent and received between systems can be an event [8]. An event is defined as anything that happens, occurs or changes the current state of affair in [10]. Composition or derivation of events from other events using different event operators and temporal relationships creates complex events. The operators mentioned here can be mathematical, logical, and bitwise operators that perform some operations on event sets. Temporal operators can be used to represent some time related operations. Waiting, occurrence priority, periodicity, aperiodicity can be specified as temporal relationships.

Event Processing is defined as operations on events to perform some computations [9]. It is concerned with timely detection of compound events within streams of simple events [10]. Detection of complex events from a cloud of events in real-time is called Complex Event Processing (CEP). Difference between event processing and CEP is that, event processing deals with only one type of events from one source, whereas CEP deals with event from multiple sources. A complex event is a combination of more than one events, possibly from different sources, and connected by logical operators such as AND and OR, and also sequencing operators. An example complex event, constructed from the simple events of a, b, and c is: complex-event$_1$ = a AND (b ; c). The meaning of this complex event expression is that c should happen after b and a should also happen. A CEP software will accept the complex-event$_1$ as happened, if a event is received before or after the reception of the sequence "b;c". For this example, the reception of a could happen before b, between b and c, or after c for a successful acceptance of the complex event. CEP is done by matching complex event patterns against event instance sequences and it is required to be able to define and trigger reactions to the complex events [11]. CEP is also defined as extracting meaningful and actionable information from event streams in [12].

CEP is a technology which allows finding real-time relationships between different events using elements such as timing, causality, and membership in a stream of data in order to extract relevant information [9]. With the help of CEP, applications have the ability to detect and report meaningful patterns in the condition part of events with respect to the incoming events and thus they can react to these detections by executing their action parts [12].

2.4 Rete

Rete is a pattern matching algorithm, invented by Dr. Charles L. Forgy of Carnegie Mellon University. Rete is a Latin word that means "net" [21]. To increase the efficiency of the simulator, a pattern matching approach inspired from Rete is used. Thus, Rete algorithm will be mentioned in this section. The approach in Rete is to keep the condition of evaluations in the memory similar to recording the state, hence, decrease the need for continuous calculations. As a result, pattern-matching intensive rules can be evaluated much faster, especially as new inputs enter the system. A cycle comprises three phases (Match-Select-Act) that needs to be executed in every production system by an interpreter [22]. The match phase is repeated in every cycle of this process and it consumes up to ninety percent of the execution process [32]. Therefore, improvement of matching process is crucial and it is done by using certain matching algorithms as Rete. This cycle is also called “Recognize-Act Cycle” or “Inference Cycle”.

Having improved the rule-based environment’s performance, there has been a desire to incorporate the Rete approach to real-time systems, but such attempts have not been very successful or widely accepted yet. Our approach achieves similar performance gains through “dynamic subscription” to events hence avoiding the broadcasting of an incoming event. Initial rule-based systems employed a black-board algorithm that may be implemented through broadcasting. Rete on the other hand, based on the immediate expectations of the rules, drives the system to its current status as the working memory changes. Only required actions are processed instead of attempting to activate every rule.

2.5 Structure of the KOTAY

In this section, the structure and the grammar of the subject language are explained. The language is created and defined in a previous PhD study [20]. It includes temporal, logical, and arithmetic operations. The main structure of the language is given in the following.

expression = condition ‘->’ action;

The rules are composed of two parts: condition and action. The separator symbol ’->’ is used between them, in this study. There are three keywords for defining the events: “ariza”, “belirti”, and “kaynak”, their English translations are “fault”, “symptom”, and “source”. The events and operators combine and create the condition part. The action part may or may not include events. An example event is “belirti(kaynak(6,9),7)”, which means that there is a sign of malfunction, located at source defined as (6,9) and type of belirti event is 7. "ariza" and "belirti" event types are defined with numbers in the range of 0 to 99. The events in an action part are separated with semicolon ";". The events in the condition part are separated with operators and parentheses. "belirti" typed events are integer valued events, whereas "ariza" type events are Boolean valued events. The value of "ariza" events can be "arizali" which means faulty or "arizasiz" which means working without a fault. These two values are defined as true or false correspondingly.

An example rule is given below:

(belirti(kaynak(4,2),8)||belirti(kaynak(3,9),4))

->ariza(kaynak(3,1),6),7;ariza(kaynak(9,4),2),1

There are two “symptom events” in the above example, connected through an ‘or’ operator in the condition part of the rule. If any of these events is received with ‘true’ value the rule fires by creating two “fault events". The numbers accompanying the event type names indicate the source and the types of the symptom or fault.

The three operators are explained in the following subsections. They are the sequence operator, abort operator, and timer operator, where sequence and abort operators are treated similarly in the temporal blocks section.

2.5.1 Temporal Blocks

There are two sequential operators, first one is the sequence operator “;” and second one is the abort operator “~”. The sequence operator accepts two operands and abort operator accepts three operands. For these operators ordered reception of events is important, in other words one needs to be before or after the other. An operand can also be a parenthesis statement which may or may not include another sequential statement, yet the whole statement creates a temporal block. In abort operator if the sequence of the events is not as expected then there is a roll-back mechanism which will reset the sub tree under the corresponding tree node. At run-time there can be many active temporal blocks in the same tree. In Figure 2.1, an example Abstract Syntax Tree (AST) is shown for the sequence operator ";". The “A” event needs to come into the system before the “C” event. Also, the “B” event needs to arrive before “D”. In our tool, we manage this by not activating nodes that are not supposed to come first. This situation is explained in the Semantic Issues section. So, in the example in Figure 2.1, initially only “A” and “B” nodes are active which means if those events enter the system than they will be processed. Then, "C" and "D" tree nodes will be active and start listening to those events.

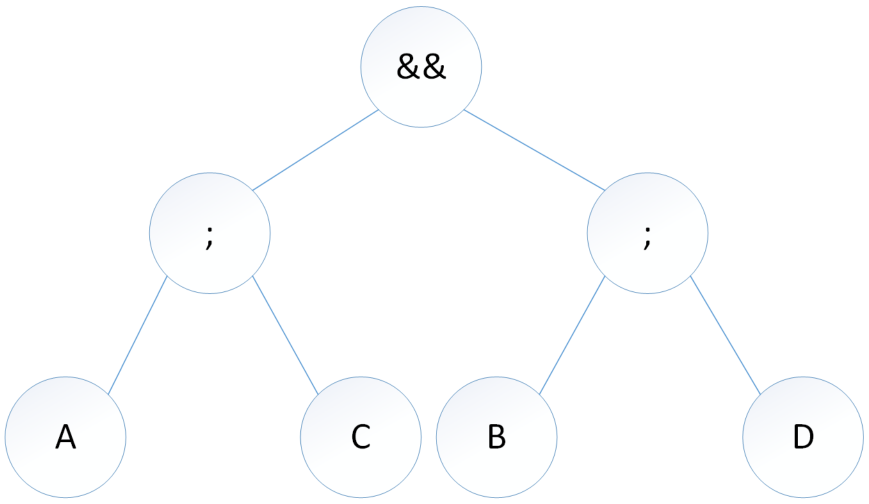


Figure 2.1: Sequence Operator “;”

In Figure 2.2, the abort operator is introduced. This operator takes three operands and works as follows: Nothing happens before the left child becomes mature. A node being mature is defined as, either it is a leaf node and it has received its event, or it is an operator node that has evaluated to true. Once left child is mature, depending on what other child matures, the abort operation makes a different decision. If center child matures first, then the “literal abort” action is taken: the whole sub-tree under the abort node will be rolled back. Alternatively, if the right child matures before the center, the operation, although it is called abort, behaves like any other operator, reporting a successful maturation to its parent. In other words, a “left; right” sequence is success but a “left; center” sequence is a reset. This operation is used in the cases where a sequence of two events is desired but without a third one happening in between.

Specifically for the example in Figure 2.2, initially, only "A" event node is active and all the other nodes are passive, which means they are not listened. After event "A" is received, B event become "active", it starts to listen to the incoming events. After event "B" is received, the most left child of the abort operator, which is the sequence operator node, completes its evaluation (it matures). Then "C", "D", and "E" event nodes become "active": now that the left is mature, we will wait to see if center or right is the next to mature. If C comes before D and E, the abort mechanism is activated and the sub tree under the abort operator node will be rolled back. Tree returns to its initial state, which is the state that only A event node is active and all the others are passive – as in the beginning of this paragraph. If D and E events come before the C event, "and" operation results with true value (matures), then the abort operator node becomes "mature", meaning that result of the abort operation will be true and event C is no longer active, its node becomes passive.

If there is a parent node for the abort operation, the result of the abort operation will act like the result of a complex event. It would be offering its result to its parent which should be an operator. Unless the abort operator node matures, it acts like providing the false value to its parent operation such as an “and operator” or so. An analogy could be made to an and gate, that receives on one of its inputs, the output of the abort operation: that is always false until abort node matures. Then, always true after the abort node matures.

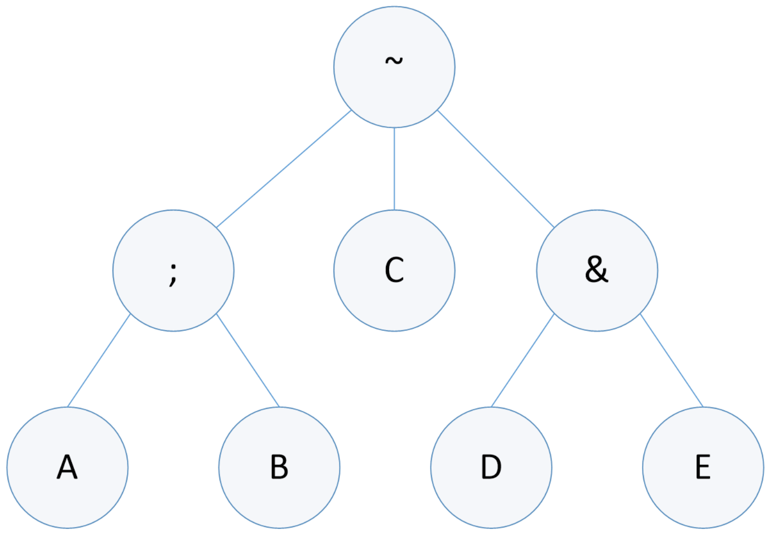


Figure 2.2: Abort Operator “~”

2.5.2 Timer Blocks

Apart from operator, event and constant related nodes, we also have timer nodes. Time can be defined in “hours”, “minutes”, “seconds”, “milliseconds”, “microseconds”, and “nanoseconds” units in these nodes. Those nodes are used to hold the system for specified time. Time nodes are used in temporal blocks. Figure 2.3 can be used to understand the usage of these nodes. When the tree shown in Figure 2.3 is initialized with Activation Order Flow-graph (AOF), which will be explained in the following sections, initially only “A” event is listened. When A event enters the system, the sequence operator then activates right child which is a timer node. This right child holds that node for 3 milliseconds and after that the “;” node is successfully finishes its operation and it sends message to the nodes that are going to be activated next, which are in this case “B” and “4s” nodes. Now, “B” event and “4s” nodes are listened. Actually “4s” node is not listened but it is activated and after 4 seconds, that node will be evaluated and sends completion message to its parent. In this case if “B” event enters the system in 4 seconds than “~” operator node will fail and roll back. If B event does not occur within 4 seconds, than the “~” operator node will be successful and evaluated as true.

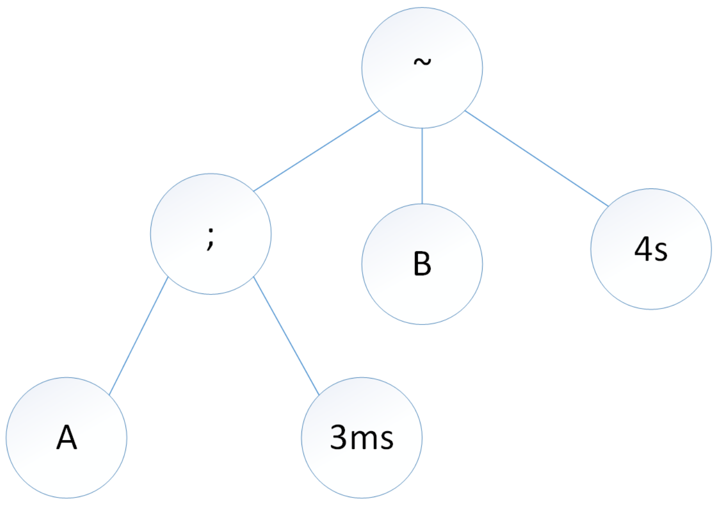


Figure 2.3: Timer Nodes

2.6 Literature Survey

This chapter includes the related studies that take place in the literature. The relation to this thesis work is defined with respect to how close an article is to the dynamic analysis facilities in CEP, being the main contribution of this thesis. The articles could be of interest because of the probability to be utilized in developing necessary concepts, providing a baseline to improve for CEP related enhancements, or providing related definitions for similar facilities even if not addressing the CEP field.

There are two kinds of analyses in rule-based systems: timing requirements and functional correctness. Aim of timing requirement analysis studies is finding whether the system has a bounded response time and if there is what the response time is. Functional correctness analysis is the validation and verification (V&V) of a rule-based system. Validation is determining the correctness of systems according to the requirements. Whereas verification is determining the consistency and completeness phases of systems. Consistency analysis is finding "Redundant", "Conflicting", "Subsumed", and "Circular" rules and "Unnecessary If" conditions in the rule set. Completeness analysis is finding "Dead-end", "Missing", and "Unreachable" rules [37]. There are some simulators existing in the literature that have some analysis facilities. The tools and the existing studies provide only the static analysis.

In [1], [2], [3], [4], [5] and [6], response time analysis of rule-based systems which uses Equational Rule-Based Language (EQL) and EQL based languages like Macro Rule-based Language (MRL) is studied statically using State Space Graph (SSG) representation. A method called fixed point convergence is used for response time analysis. In [1], the response time is optimized by the constructed reduced cycle-free finite SSG in addition to bounded response time analysis. With this optimization response time is reduced and therefore execution time is reduced. A formal analysis strategy is presented in [2] to guide other studies. In [3] a fault tolerance mechanism is presented additionally. Timing analysis and refinement of OPS5 language is studied in [6]. Although SSG is a technique for timing analysis, it is not applicable as the rule set gets bigger. The memory constraints and performance problems occur as the SSG gets bigger. These articles clearly assert the use of performance analyses that we also provide. Alternative studies for timing analysis are presented in the next paragraph.

In [26], [27], [28], [29], [30], and [31] timing analysis of the systems which use OPS5, EQL, and Estella are investigated. Those studies differ from the ones mentioned in the previous paragraph because they do not apply the SSG technique. Instead a set of behavioral assertions are proposed and if a rule set obeys one of the four proposed sets of conditions, then it is stated that the rule set has a bounded response time. The proposed set is called "Special Forms". There are four special forms (A, B, C and D) presented. Also, three compatibility relations (CR1, CR2 and CR3) are needed to be checked before considering special forms. These articles were used as reference in investigation of some of the introduced analysis facilities, especially the one that provided supplementary static analysis.

In [24], [25], [34], [35], and [36], tools for verification and validation (V&V) of rule base systems are created. In [24] a program called 'CHECK' is created to verify consistency and completeness of a rule-based expert system called Lockheed Expert System (LES). The program checks for redundant rules, conflicting rules, subsumed rules, missing rules, circular rules, unreachable rules, and dead-end clauses. A 'dependency chart' is also created to show dependencies among rules and the goals and to detect circular chains. The created program is used to detect errors before rule base testing phase. CHECK does not perform any syntax checking, it statically analyzes the rule base. Among seven criteria, four are concerned with potential problems and the last three are concerned with gaps in knowledge bases. These articles have been instrumental in directing our analysis efforts toward defined V&V problems. Potential Problems:

Potential Problems:

Redundant rules: two rules succeed in the same situation and have the same results.

Conflicting rules: two rules succeed in the same situation but with conflicting results.

Subsumed rules: two rules have the same results, but one contains additional constraints on the situation in which it will succeed.

Circular rules: a set of rules is a circular rule set if the chaining of those rules in the set forms a cycle.

Missing rules: a situation in which some values in the set of possible values of an object's attributes are not covered by any rules.

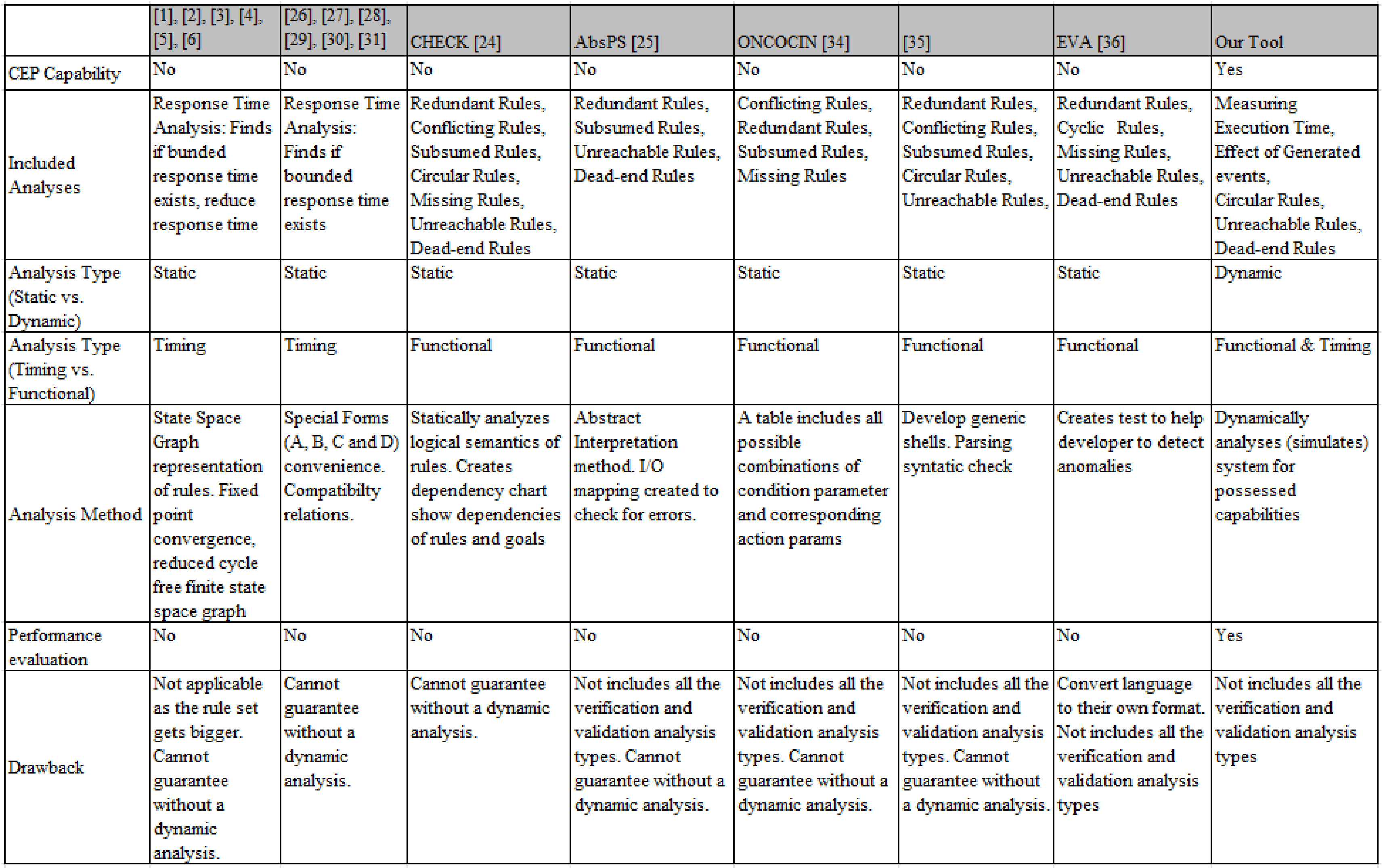
Unreachable rules: These rules are not invoked by any of the other rule or input event in the system. They generally reduce system performance.

Dead-end rules: Action part of these rules do not affect the other rules in the system, their results have no impact on generating a solution.

In [25] analysis of forward chaining, rule-based systems is done by modelling the procedural semantics of such languages rather than declarative semantics. 'Abstract Interpretation' process is defined and used to map input and output of rules through a program called AbsPS. The program is implemented to analyze the effect of conflict resolution, closed-world negation, and retraction of facts. It is claimed that the previous approaches only consider declarative semantics and this is adequate if procedural semantics are not an issue. Some of the errors occurring in the systems may reduce efficiency of rule-based systems while the others may result with erroneous inferences. To improve reliability and efficiency of forward chaining rule-based systems these errors must be fixed. Four efficiency reducing features are defined and AbsPS can detect these rule types: redundant rules, subsumed rules, unreachable rules, and dead-end rules. This article can be utilized as a source for selecting problems related with efficiency, in an effort to decide what analysis facilities to include.

'ONCOCIN' is a system which is a rule-based consultant for clinical oncology [34]. The authors suggest a set of mechanisms to correct problems for rule base consistency and completeness before they cause problems. Some problems that occurred during knowledge acquisition and debugging are described, then automated assistant for checking completeness and consistency of the rule base system is created. In this study conflicting rules, redundant rules, subsumed rules, and missing rules are analyzed. In [35] to guarantee a certain degree of reliability in rule base programs a tool is implemented to use during the system development process. Five consistency issues; redundant rules, conflicting rules, subsumed rules, circular rules, and unnecessary conditions are examined using the tool. EVA is another well-known verification tool, it can detect unreachable, cyclic, missing, redundant, and dead-end rules through analysis. For developers to determine anomalies the program also creates tests [36]. The work reported in the sources [34-36] provide ideas about practical use of analysis facilities in the application fields.

The comparison of the existing studies and our tool is given in Figure 2.4.



	[1], [2], [3], [4], [5], [6]	[26], [27], [28], [29], [30], [31]	CHECK [24]	AbsPS [25]	ONCOCIN [34]	[35]	EVA [36]	Our Tool
CEP Capability	No	No	No	No	No	No	No	Yes
Included Analyses	Response Time Analysis: Finds if bunded response time exists, reduce response time	Response Time Analysis: Finds if bounded response time exists	Redundant Rules, Conflicting Rules, Subsumed Rules, Circular Rules, Missing Rules, Unreachable Rules, Dead-end Rules	Redundant Rules, Subsumed Rules, Unreachable Rules, Dead-end Rules	Conflicting Rules, Redundant Rules, Subsumed Rules, Missing Rules	Redundant Rules, Conflicting Rules, Subsumed Rules, Circular Rules, Unreachable Rules,	Redundant Rules, Cyclic Rules, Missing Rules, Unreachable Rules, Dead-end Rules	Measuring Execution Time, Effect of Generated events, Circular Rules, Unreachable Rules, Dead-end Rules
Analysis Type (Static vs. Dynamic)	Static	Static	Static	Static	Static	Static	Static	Dynamic
Analysis Type (Timing vs. Functional)	Timing	Timing	Functional	Functional	Functional	Functional	Functional	Functional & Timing
Analysis Method	State Space Graph representation of rules. Fixed point convergence, reduced cycle free finite state space graph	Special Forms (A, B, C and D) convenience. Compatibilty relations.	Statically analyzes logical semantics of rules. Creates dependency chart show dependencies of rules and goals	Abstract Interpretation method. I/O mapping created to check for errors.	A table includes all possible combinations of condition parameter and corresponding action params	Develop generic shells. Parsing syntatic check	Creates test to help developer to detect anomalies	Dynamically analyses (simulates) system for possessed capabilities
Performance evaluation	No	No	No	No	No	No	No	Yes
Drawback	Not applicable as the rule set gets bigger. Cannot guarantee without a dynamic analysis.	Cannot guarantee without a dynamic analysis.	Cannot guarantee without a dynamic analysis.	Not includes all the verification and validation analysis types. Cannot guarantee without a dynamic analysis.	Not includes all the verification and validation analysis types. Cannot guarantee without a dynamic analysis.	Not includes all the verification and validation analysis types. Cannot guarantee without a dynamic analysis.	Convert language to their own format. Not includes all the verification and validation analysis types	Not includes all the verification and validation analysis types

Figure 2.4: A comparison of existing studies

CHAPTER 3

IMPLEMENTATION OF SIMULATOR

In this chapter, implementation details of the created simulator are explained. Four main components of the tool are presented, then initialization and run-time algorithms are presented in detail. Figure 3.1 shows the relationships of the main components with each other. Event and rule generator parts are excluded from the figure to avoid complications. Four main components in the tool are: Parser Control, AST, AOF and Subscriptions.

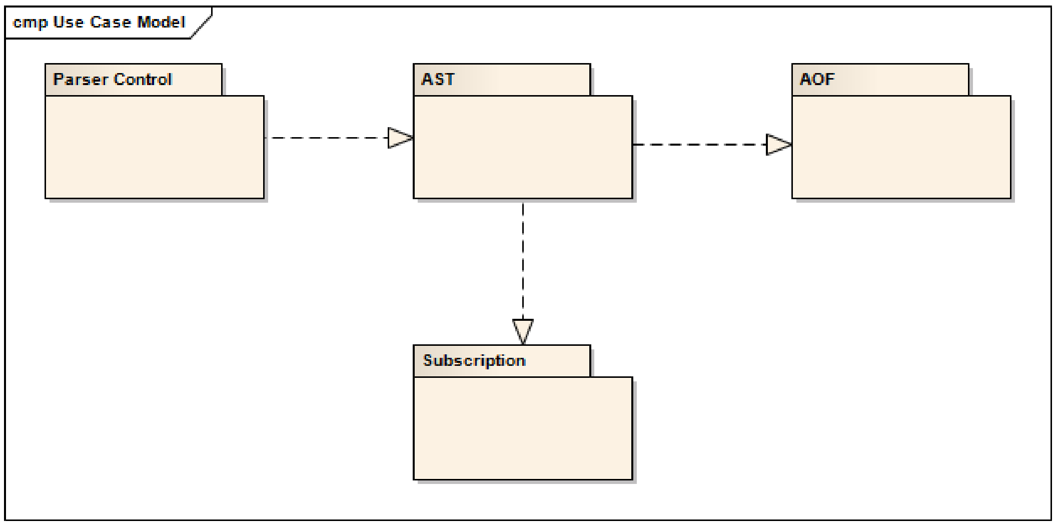


Figure 3.1: Initialization Architectural Components related with program initialization

3.1 Initialization Subsystems at Initialization

The data structures are constructed when the program is initialized, during early run-time. Their explanations are given in the following subsections in detail. Figure 3.2 shows the class diagram for the simulator.

3.1.1 Parser Control

Parser Control component is responsible for reading and parsing the event based rules, then creating an AST for condition parts of each rule. The rules are in infix format. In the parser class, they are converted into postfix notation to be able to create ASTs for these rules [33].

Conversion from Infix Notation to Postfix:

Edsger Dijkstra invented the Shunting-Yard Algorithm to convert infix expressions to postfix form (RPN), the name comes from the operation resembles to a rail road shunting yard [33].

For the operators that have only two operands, an AST is created using Shunting-Yard Algorithm will be a binary tree. However, in our case operators can take one or three operands in addition to those that take two operands. Therefore care is needed to implement the algorithm with different numbers of operands rather than expecting always two operands per operator: most sources reporting the algorithm may mention only the two operand case.

Our operators' priorities and their numbers of operands needed to be defined. To be able to utilize these parameters, we created a class named "Operators.cpp", in this class we defined a static array to store the "Operator", "Priority" and "Number of Operands". This class has two static methods "getNOperands()" and "getPriority()", where they accept an operator name as a parameter and return that operators number of operands or priority. The Operator Priority Table is shown in Table 4.1.

In table, MathNeg and MathMinus operators are shown differently. MathMinus

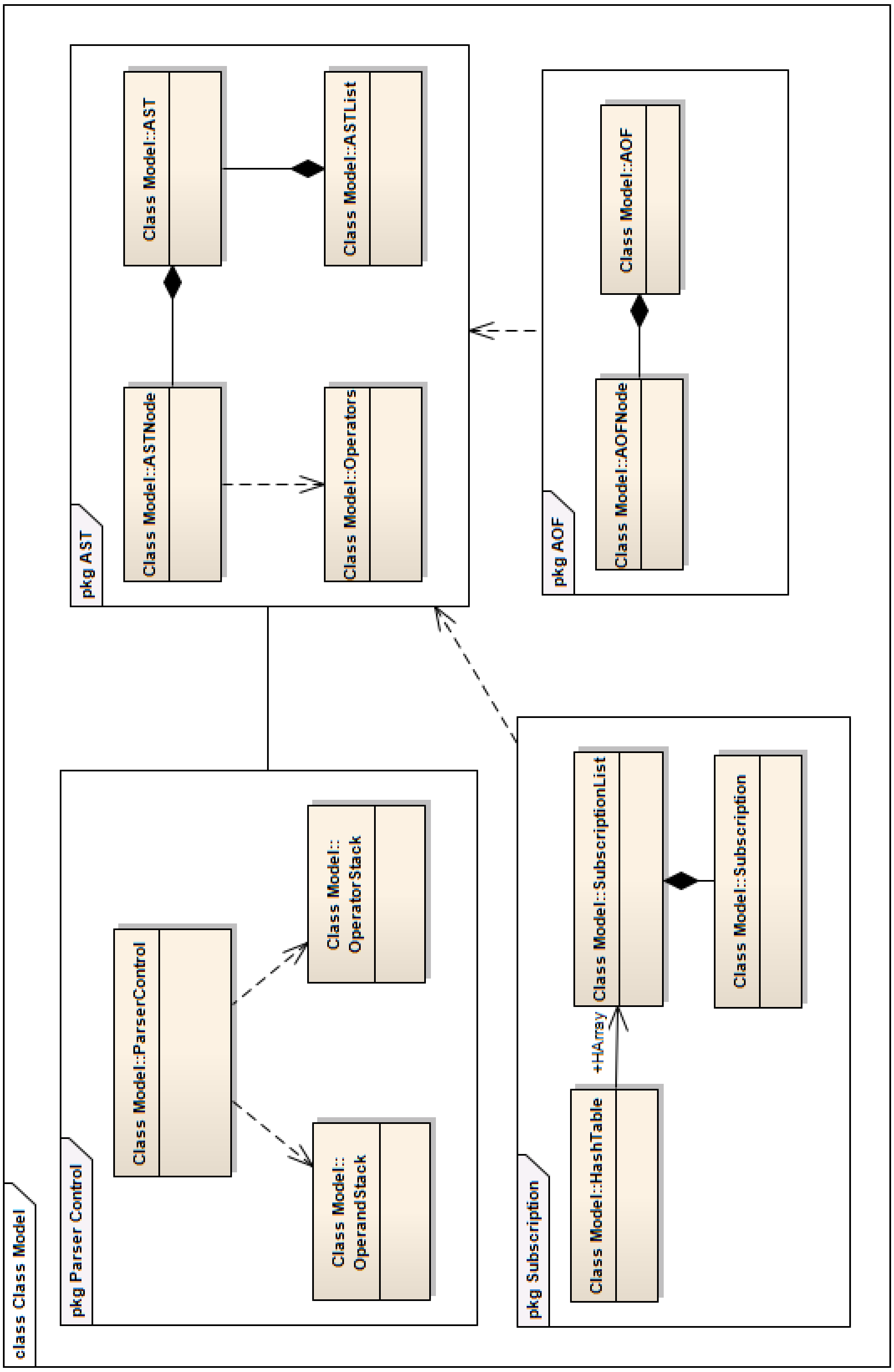


Figure 3.2: Class Diagram of the Simulator

operator is shown as "-" and MathNeg operators is show with two consecutive minus operators as "- -". This difference occurred because in the parsing, the AST creator method could not differentiate which one is the minus and which one is the negate operation.

Table3.1: Operator Priority Table

Priority List			
Operator Name	Operator Sign	Priority	Number of Operands
LogNot	!	5	1
MathNeg	- -	5	1
MathDiv	/	4	2
MathTimes	*	4	2
BitAnd	&	4	2
MathAdd	+	3	2
MathMinus	-	3	2
BitOr	|	3	2
Smaller	<	2	2
Greater	>	2	2
SmallerEqual	<=	2	2
GreaterEqual	>=	2	2
Equal	==	2	2
NotEqual	!=	2	2
LogAnd	&&	2	2
LogOr	||	1	2
Sequence	;	0	2
Abort	~	0	3

3.1.2 Abstract Syntax Tree

An Abstract Syntax Tree is created for every condition part of every rule. Leaves of an AST correspond to events and intermediate nodes correspond to operators. The root node is responsible for the final evaluation and returns the result of its evaluation.

When the analysis application is started the parser scans the rules and creates AST's for every rule. In run-time as the events enter the system and are received by the nodes, evaluation is made conducted from leaves to the root of every AST. The "active" state nodes are allowed to receive events through subscription whereas the "sleep" state nodes are not. Determination of such states of the nodes, hence controlling the sequencing is managed by a structure called Activation Order Flow-graph, which will be mentioned in the following sections. Every AST node has three states: Sleep, Active, and Mature. Their explanations are given in the following.

States of AST Nodes:

SLEEP: Since the node is not in the sequence currently, nodes at this state are insensitive to the incoming events.

ACTIVE: The node starts listening to its event. The node waits for its event to be mature.

MATURE: When the expected event received by the node, the nodes transits to its 'mature' state, its parent node in AST is informed about the case for a possible evaluation (if all operands have arrived in expected sequence). The node stays at mature state until a re-initialization after the firing of the whole AST.

AST Class: This class represents a single tree and includes some methods that a tree requires and AST Node does not. After parsing is complete, the root of the ASTNode is equated to root attribute of the AST class.

ASTList Class: This class is used to store the AST's that are generated from the rule set. This list is used in the main class to perform analysis on the created ASTs.

3.1.3 Activation Order Flowgraph

AOF is designed to activate the AST nodes when their time comes. A node in the AST that performs its transaction successfully sends a message to its corresponding AOF node. This message states that the current node in the AST has finished its job; consequently AOF triggers the next nodes. To do so, the node receiving the message in the AOF passes the control to its neighbour node. Newly activated node in AOF, in turn, activates its corresponding AST nodes, that could be more than one to activate.

Creation of an AOF structure requires traversing of the corresponding AST. This process is done at the initialization-time, thus it does not affect time efficiency and a recursive algorithm can be used for that.

Next, the methods used to create AOF and AOF Nodes are mentioned. There is one base method which creates the basic structure of AOFs. Rest of the four methods create the AOF types corresponding to the operators.

•makeAtomicFlow(): This method creates the smallest structure, atomic flow, to be used in AOF construction. It takes an AST node pointer as a parameter and creates three nodes that are connected to each other. The first one is a start node, it is used to indicate the start of an AOF. Start node is connected to an event node, which has a pointer to the corresponding node in the AST, and corresponding AST node has a pointer to the AOF node. By this way, a two way connection is created and transaction between AST and AOF can be done quickly. Atomic flow structure ends with an end node. Figure 3.3 shows an example atomic flow structure.

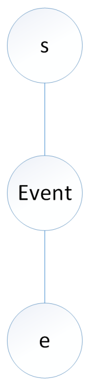


Figure 3.3: Atomic flow structure

•makeSingleFlow(): This method is used for MathNeg and LogNot operators. Since, these operators take only one operand, their corresponding AOF will be in the form of "single flow". This function takes only one AOF node as parameter. Then it connects this AOF node with a start and end node as shown in Figure 3.4-a.

•***makeSerialFlow():*** This method is used for the sequential operator ";". The activation order is important in sequential operations. One event needs to be active before the other. This function takes two AOF nodes as parameters. Then it connects this AOF nodes serially, the first one in the parameters, occurs first in the AOF. Then they are connected with a start and end node as shown in Figure 3.4-b.

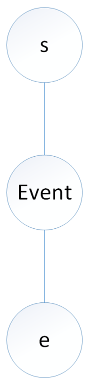
•***makeDiamondFlow():*** This method is used for abort operator "~". It creates a single flow followed by a parallel flow. This function takes three AOF nodes as parameters. Then it creates a parallel flow with the second and third nodes in the function parameter, which is shown in Figure 3.4-d. Then, the first AOF node is connected to the Parallel flow. Lastly, they are connected with a start and end node as shown in Figure 3.4-c.

•***makeParallelFlow():*** This method is used for all other operators. It creates start and end nodes and two nodes between them in parallel configuration. Parallel nodes represent that two nodes (in the AST) will be actived at the same time. This is shown in Figure 3.4-d.

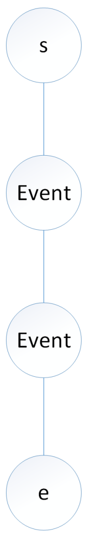
Figure 3.4 shows the visual representation of the AOF structures. These are only example figures. Note that, there can be many start and end nodes in an AOF structure which point to each other. To reach an event node, many start nodes must be passed in bigger AOFs.

3.1.4 Subscriptions: (Incoming Events)

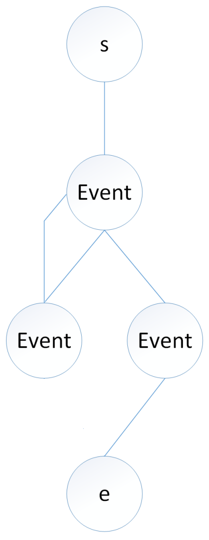
This subsystem is responsible for receiving the incoming events and directs them to the related parts at run-time. During the initialization time, after AST and AOF structures are generated, subscriptions are conducted for related nodes using AST and AOF. This is done by constructing the Hash Table and inserting pointers to related AST nodes into its lists. Hash Table is the most important data structure in the Subscriptions subsystem. The Hash function is one-to one: any incoming events name is directed to one entry for a pre-specified event name in the hash table.



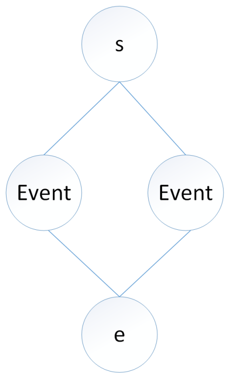
(a) Single Flow



(b) Serial Flow



(c) Diamond Flow



(d) Parallel Flow

Figure 3.4: AOF Representations

An approach to check whether the incoming events exist in our AST list or not, is developed to avoid iterating over all the AST's event nodes. This approach increases time performance whereas it increases memory usage. Having Rete capability in real time systems and engines is not very common. The aim of this approach is to achieve advantages of Rete while avoiding its implementation difficulties by the following two mechanisms:

• Subscription records are stored that utilize pointers between the hash table and the requestor nodes in ASTs.

• For AST's the activation order for the subscribing nodes is stored in a data structure (AOF). This way only the nodes that can be active are sensitive to the incoming events, the others are not covered in the search. Then, subscription is conducted: A node in the AST becomes active and starts listening to an incoming event. This approach is called "Dynamic Subscription". Its dynamicity is due to the fact that the subscription can be continuously formed and destroyed based on when a node is activated to listen to an event.

Hash Table: A hash table is used to store event names taking place in all AST's to be able to control whether the incoming events exist in the rule set. If an incoming event exists in the hash table, it performs the necessary operation which is sending event notification to corresponding AST nodes. With the help of the hash table, sending input events to corresponding nodes operation is performed in O(1).

Hash Function: A one-to-one hash function is created to determine location of the events stored in the hash table. The hash function takes the event name as input parameter and returns the location for the event name if it exists in the hash table. The hash function makes use of the special naming for events. Numbers used in the event naming are directly converted to an index, avoiding a longer mathematical calculation.

There are three numbers that are separated with comma or parenthesis in event names. The first two numbers determine the source of the event and the last number defines the type of event. An example event is "ariza(kaynak(2,9),6)". In that event, 2 and 9 defines the source of the event and 6 defines the event type. Source of the events is defined with two digits between 0-9. Whereas, type of events are defined with a number between 0-99, since there are 100 event types. Therefore, there can be twenty thousand different event names. An array to store all possible events are created, whose size is 20000. Event names and a forward list which stores all the AST nodes that have the same event are stored in the array. There are two type of events in our system, "ariza" and "belirti" typed events. The location between 0 to 9999 in the array is reserved for “ariza” typed events and 10000 to 19999 is stored for "belirti" typed events. For example, if ariza(kaynak(2,9),6) event is given to the hash function, returning value is 296. Which means, this event is stored in 296th location in the hash table. If belirti(kaynak(2,9),6) event is given to the hash function then returning value is $296 + 10000 = 10296$. So, that event is stored in 10296th location in the hash table. By this way, a hash function which guarantees no collision is created.

In hash table event name, and a list which holds a pointer to corresponding AST node and state of that node is stored. The state information is different than the state in an AST node. The state in hash table only has two values: “ACTIVE” and “PASSIVE”.

ACTIVE: If the corresponding AST node’s state is active than state in hash table is also active. Here, the active state indicates that corresponding AST node is ready to receive the incoming event. If the event enters the hash table and state of a node is active, then incoming event is sent to the corresponding AST node (event notification).

PASSIVE: If the corresponding AST node's state is passive or mature, than state in hash table is passive. Here the passive state indicates that corresponding AST nodes do not listen to incoming events. The AST node might be in sleep or mature state, but this case does not need to be known by the hash table.

3.2 Run Time Subsystems

In run-time an event that enters the system, is firstly welcomed in hash table. Here, the event follows the pointers that indicate corresponding AST nodes, which carry that event name in their nodes and are currently active, e.g. waiting for an event. Important thing here is that, the event is only sent to the active nodes, not the passive ones. After sending the event, the AST node pointers in the hash table are unsubscribed, which means hash table becomes insensitive to these types of events. Then, all the AST nodes make their evaluations and send message to their parents and also to the corresponding AOF nodes. AOF nodes are leaders to decide which nodes will be subscribed and listened next.

In Figure 3.5 three main data structures of the system are shown with their connection to each other.

3.3 Initialization Time Algorithms

In this part algorithms that are used at the beginning of the run time are explained. There are four main activities for the initialization time as stated before: Parsing the rules, AST creation, AOF creation, and Hash Table generation.

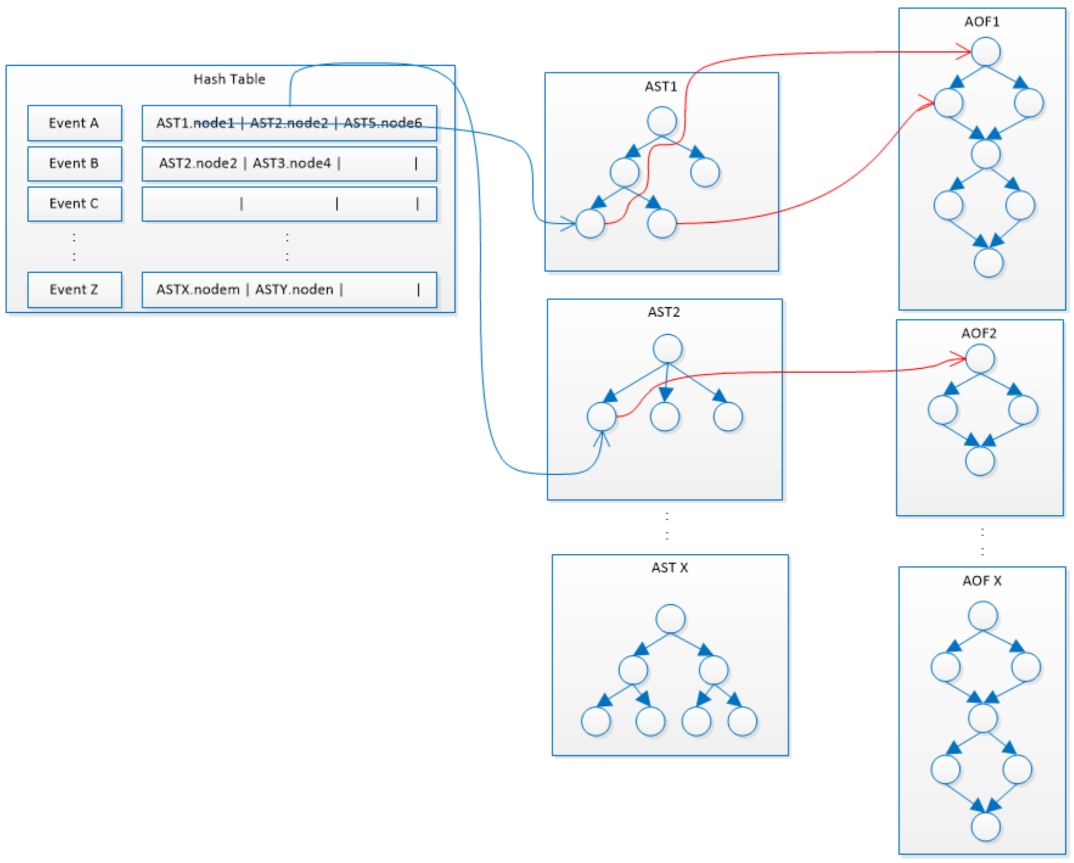

Hash Table
Event A
Event B
Event C
Event Z
AST1.node1 | AST2.node2 | AST5.node6
AST2.node2 | AST3.node4 |
ASTX.nodem | ASTY.noden |
AST1
AST2
AST X
AOF1
AOF2
AOF X

Figure 3.5: Main data structures and their connections

3.3.1 Parsing Rules:

The process of parsing requires other subsystems and includes the creation of other data structures. After parsing, AST is created in postfix form. By traversing the created ASTs, AOFs are created. While parsing the rules, hash table is also filled concurrently. As the new events are encountered a new place is filled and existing events are added to the end of list in the hash table. Parsing is basically given in the following process definition. The below processes are explained further in the upcoming sections.

For every rule:

1. Make Tree

2. Create AOF

3. Fill Hash Table

4. Initialize Tree

3.3.2 AST Creation:

There is an AST for each condition part of the rules. The rule language is in “infix” form and rules are converted into “postfix” form. For this conversion “Shunting Yard” algorithm is used. In this algorithm, AST can also be created when the Reverse Polish Notation’s are generated. Algorithm is based on binary trees; however AST’s in this work can have one, two or three child nodes. Therefore, the algorithm is modified accordingly to adopt this difference. The algorithm is presented in the following:

```
1. While there is more token to be read:
    1.1. Get the next token
    1.2. If token is an operand, put it into OperandStack
    1.3. If token is an operator called (o1):
        1.3.1. While there is an operator (o2) in OperatorStack and its priority is higher than (o1)
            1.3.1.1. Take (o2) from OperatorStack and take n operands from OperandStack where n is the number of required operands of (o1) and make a tree with them, then push this tree back into OperandStack
        1.3.2. Put (o1) into OperatorStack
    1.4. If token is a left parenthesis, put it into OperatorStack
    1.5. If token is a right parenthesis:
        1.5.1. Until left parenthesis comes from OperatorStack:
            1.5.1.1. Pop all the operands and operators, make trees and push them back to OperandStack
        1.5.2. Drop left paranthesis.
```

2. After no more tokens to be read:

    2.1. While there are operators in OperatorStack:

        2.1.1. Take operator and operands, make tree with them and put it into OperandStack

This algorithm was traced manually to see if it creates correct AST representations given some test expressions. Also, a test code was written to draw trees and inspect the AST diagrams. Hence the correct operation of the algorithm was verified.

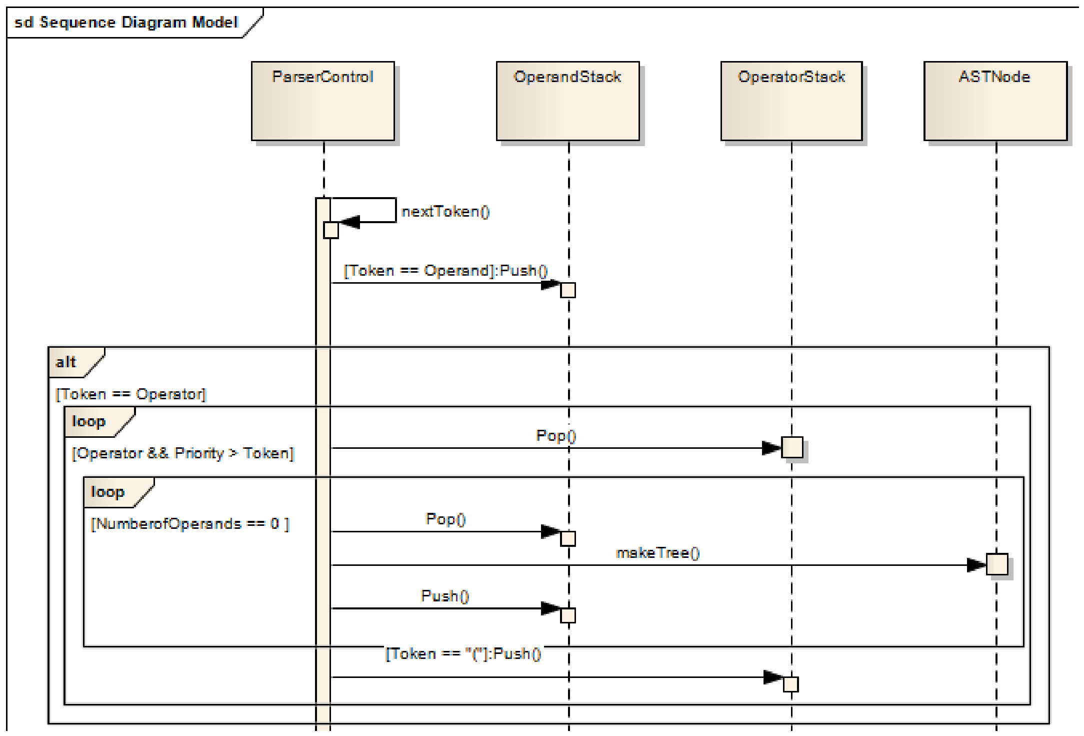


Figure 3.6: AST Creation Sequence Diagram

In Figure 3.6 and Figure 3.7 sequence diagram of AST creation is given:

Frame-alt (IF) Token is an operator

sdFrame1 (Loop): There is an operator at the top of the OperatorStack and priority of this operator is bigger than or equal to the operator that is hold at token

Frame-alt (IF) Token is a right parenthesis

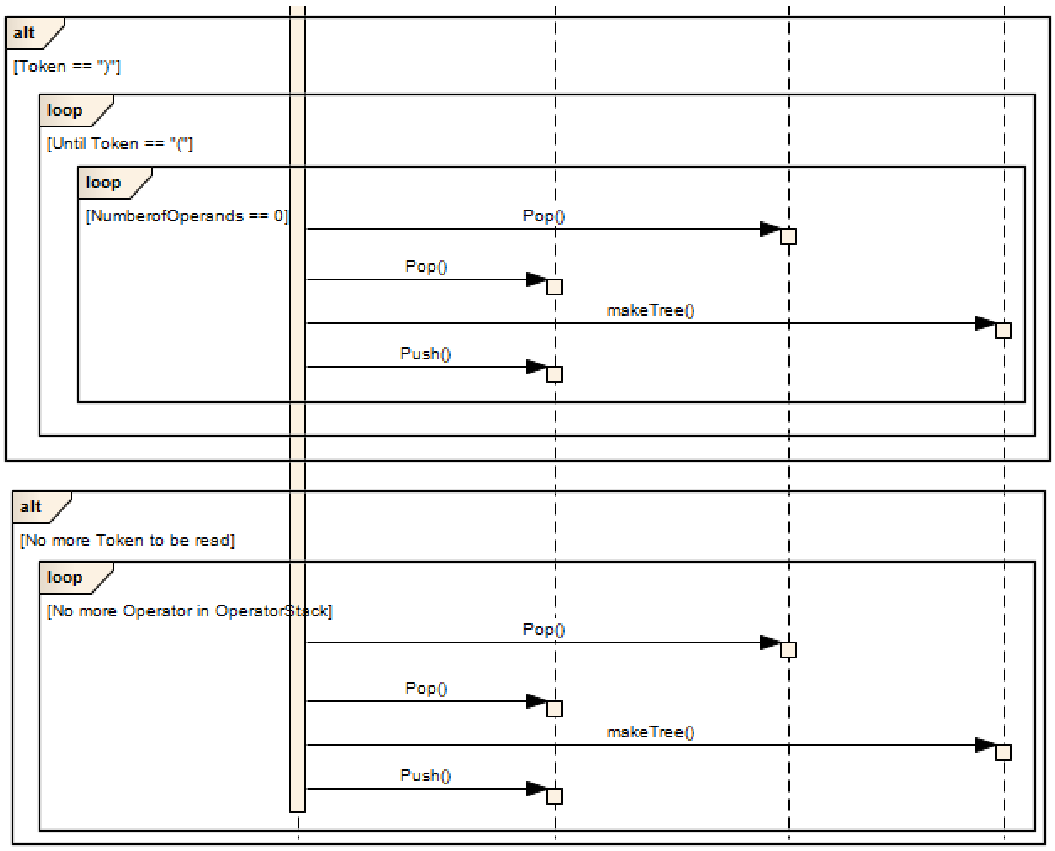


Figure 3.7: AST Creation Sequence Diagram

sdFrame2 (Loop): Until left paranthesis occurs

Frame-alt (IF) there is no more token to be read

sdFrame3 (Loop): There is no more operator in OperatorStack

While creating ASTs, two stacks are used: Operator Stack and Operand Stack. As their names imply operator stack is used to store operators e.g. Math, Comparison, Logical, Bitwise, Sequential. The operand stack is used to store operands e.g. Events, Constants and Time operands. Operators and operands are pushed in and popped out of these stacks during the process. During the AST creation scenario, ASTs that may have one to three child nodes are created with operators and operands, then these ASTs are pushed back into the operand stack. As the algorithm being applied, these little trees will be added to the sub parts of bigger ASTs and they all create only one AST at the end. This scenario is conducted during the conversion of infix to postfix notation. Therefore, creation of AST is actually a sub part of parsing the rules and it is done using the makeTree() method in the implementation.

Make Tree method requests can only be generated in a situation that an operator and required number of operands exist in the stacks. After that the created tree will be a child tree in the composed tree. The makeTree() method algorithm is explained below.

1. Inputs of the method are one operator and number of operands depending on the operator.

2. Create nodes for the operands.

3. Bind the child or children nodes according to the structure of the given operator.

3.3.3 AOF Creation:

The algorithm to create AOFs is given below. MakeAOF() method will take the root of an AST and as a result it returns the start node of the corresponding AOF after creating it, which is defined as "s" in the following. Make Serial-Flow, Make Parallel-Flow, and Make Diamond-Flow methods are explained in section 4.3.1.

```
s = MakeAOF (ASTnode n) // Start node of AOF is returned
  if (n is leaf )
     s = Make Atomic-Flow (n)
  Else if (n is unary operator) // MathNeg and LogNot operators
    S = MakeSingleFlow(n)
  Else if (n is combinatorial-operator) // All two operand operators
    s1 = MakeAOF (n.LeftChild)
    s2 = MakeAOF (n.RightChild)
```

```
        s = Make Parallel-Flow (s1, s2)
    Else if (n is “;”) //Sequence operator
        s1 = MakeAOF(n.LeftChild)
        s2 = MakeAOF(n.RightChild)
        s = Make Serial-Flow(s1, s2)
    Else if (n is “~”) //Abort operator
        s1 = MakeAOF(n.LeftChild)
        s2 = MakeAOF(n.MiddleChild)
        s3 = MakeAOF(n.RightChild)
        s = Make Diamond-Flow(s1, s2, s3)
```

In the algorithm above, there are steps that construct different types of AOF parts. These steps receive atomic flows or more complex flows, and combines those received flow types under one of the patterns: single, serial, parallel, or diamond flows. In the process, the input nodes for starting and ending the input flow, may be deleted: the resultant flow has only one start (s) and one end (e) nodes.

s = Make Serial-Flow(s1, s2) This step receives two input flows, connects s2 after s1 and adjusts ‘s’ and ‘e’ nodes: the resultant flow-graph will start with the ‘s’ node of s1 and ends with the ‘e’ node of s2. The s1’s end is connected to the s2’s start as the internal structures of s1 and s2 require.

s = Make Parallel-Flow (s1, s2): This step receives two input flows, connects them as to the two out-flow edges of the resultant ‘s’ node. Also the end (e) nodes of the input flows are combined as one e node for the resultant flow – for this, the edges coming to the e nodes of the two input flow graphs, are connected to one e node and the other e node is discarded.

s = Make Diamond-Flow(s1, s2, s3) This step receives three input flows, makes a parallel flow using the s2 and s3: let us call this intermediate result as s4, and makes another serial flow using s1 and s4.

Sequence diagram for AOF creation is given in Figure 3.8.

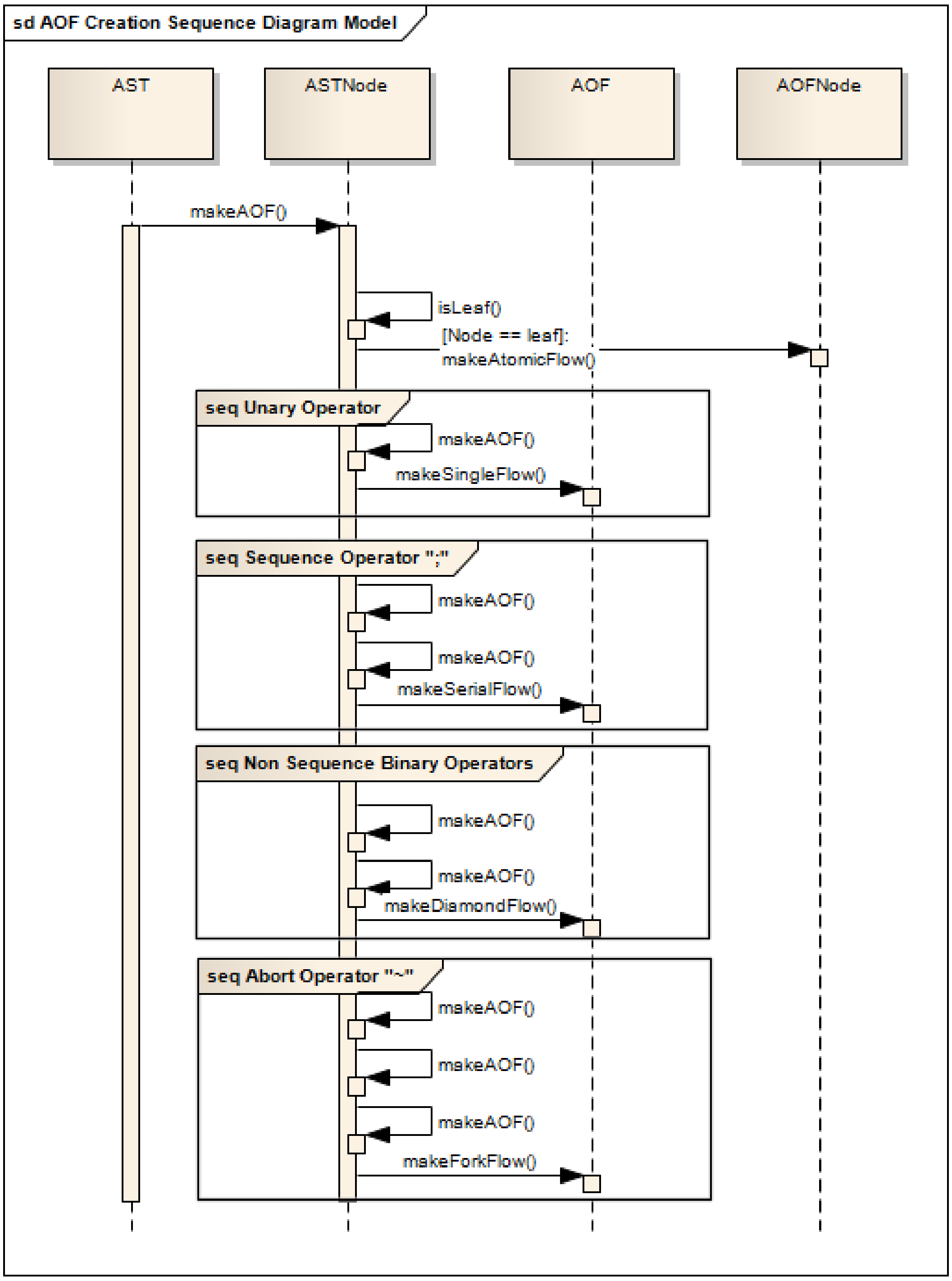


Figure 3.8: AOF Creation Sequence Diagram

3.3.4 Hash Table:

During run time, Hash Table is the first system that an incoming event encounters, then that event is redirected to corresponding ASTs. In initialization time, the hash table is created during parsing. Then, with the information in the ASTs and the AOFs, the initial subscriptions are processed in the hash table.

Hash Table algorithm is basically as follows: When AST node is created, the event name contained in it is sent to the hash table by calling related function. In that function, hash function is used and location for that event is found. This location in the hash table holds a list of subscriptions for one event type (name). The information about the AST node is added to the end of this list. In Figure 3.9 hash table creation sequence diagram is given.

3.4 Run-Time Algorithms

3.4.1 Sending events to subscription:

When an event enters the system, initially it is received by the hash table. Then, using hash function a location is found. Next, that location is searched in the hash table. If that location does not carry any subscription (all the nodes that carry this event are in sleep or mature states), the event will be discarded. If found location carries one or more subscriptions, then that event will be sent to all subscribed AST nodes. Figure 3.10 shows the sequence diagram of this event reception process. Since the hash function returns the exact location of the events, complexity of this algorithm is O(1).

3.4.2 Tree evaluation:

Tree evaluation is started after an event arrives. Only leaf nodes can receive incoming events. After the event is received by the node, that node triggers parent nodes for evaluation. Different evaluation types exist for leaf, middle and root nodes. After evaluation is complete, the result will be saved in the

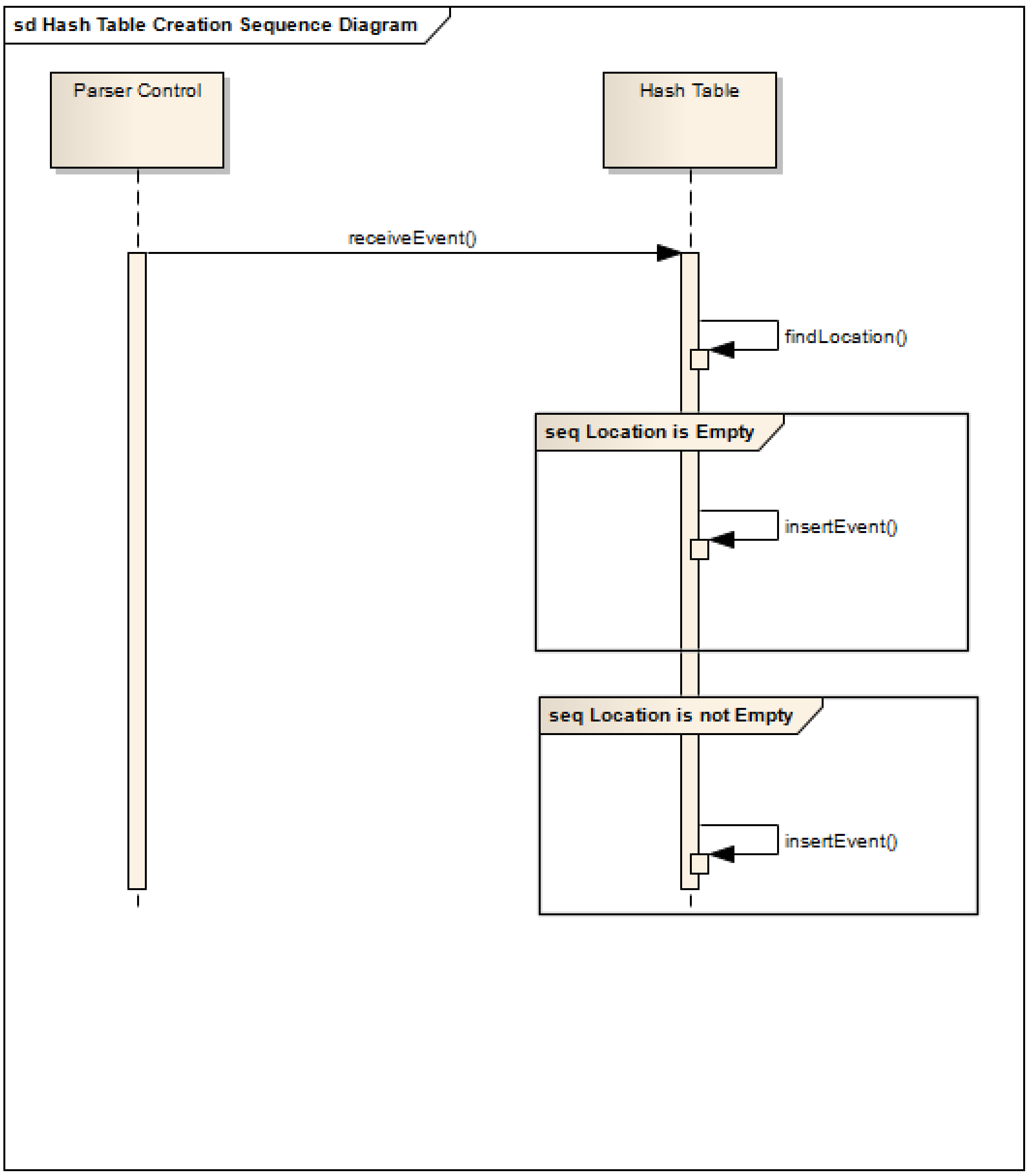


Figure 3.9: Hash Table Creation Sequence Diagram

corresponding node. Events without a value are considered as Boolean valued and if that event is received by the node it is interpreted that result is true. The tree evaluation algorithm is given in the following:

For leaf nodes:

1. receiveEvent() method

2. unsubscribe

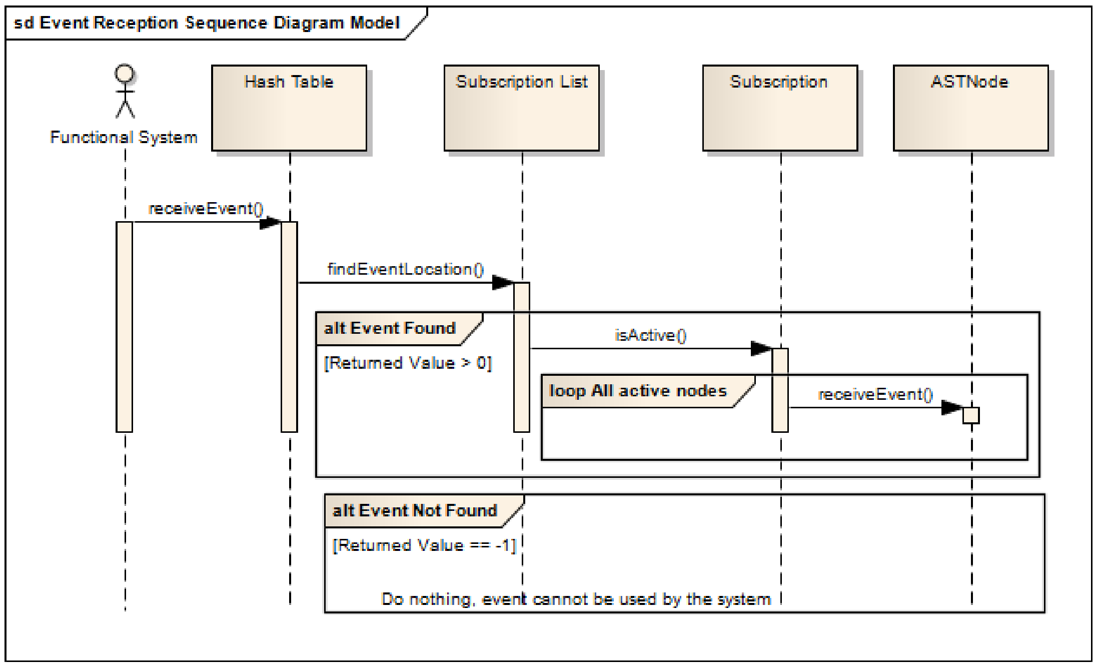


Figure 3.10: Sequence Diagram of Event Reception

3. state = mature
4. call parent.evaluate();

For middle nodes:

1. If right and left child state = mature
   a. State = mature
   b. Conduct operation: Result will depend on operands and operator
   c. Call parent.evaluate()

For root node:

1. If right and left child state = mature
   a. Conduct operation: Result will depend on operands and operator
   b. If result is true fire action part
2. Initialize tree to start from the beginning

Complexity of evaluate() method, which is used in tree evaluation algorithm, is

O(log(n)), because the maximum number of evaluation methods can be equal to depth of the tree, where n is the number of operands in the condition part of a rule. However, initializing a tree requires a traversing on the tree, which has the complexity of O(n). Moreover, finding the nodes that are going to be activated next also has O(n) complexity. There are two O(n) complexity messages and one O(log(n)) message, so the worst case for the tree evaluation algorithm is O(n).

3.5 Semantic Issues

During the design and implementation of the simulator, many changes have been made to create a better simulator and to have better results. In this section some of these design and implementation decision changes are mentioned.

Listening Nodes in Sequence Operators:

There were some issues about incoming event listening in temporal operators. The requirement of sequence operator is that, left child has to occur before the right child, and if not then this operator will not conduct its evaluation. There were two options when designing that requirement; first one was to listen to both of the child nodes that are without account for whether the node is the left or right child of the sequence operator node. Second choice was not to listen to the right side of the node before the left child matures since listening to the right sub tree list will be a waste. After careful thoughts, it is decided not to listen to the right child before left matures. Because when the right node matures before the left one, the operator results with a false value and it immediately re-initializes and starts to listen to incoming events until left child node event occurs before right one for this process to continue. Since there is no abortion or cancel operation, listening to both child is not logical. Therefore, we suppress the right child (possibly a sub-tree) for incoming events until left child is evaluated, therefore it increases the efficiency of our system and system performance.

Listening Nodes in Abort Operators:

A design issue existed to listen abort nodes. Should all the nodes need to be listened for incoming events or not? If all the nodes are listened concurrently, then there will be many wasted operations. Figure 3.11 shows the selected structure for the abort node. For this rule to be true firstly, A event should occur in the system. If it never comes, then no matter how many times B and C events occur, this operator will not be evaluated. Thus, it is decided that, until A event occurs, B and C event nodes are closed for incoming events and it is decided not to listen to all the nodes at the same time. Only A node is listened and after it occurs, B and C operators are activated concurrently. If B matures before C, the operator node and its sub tree are re-initialized. If C matures before B then the abort operator node evaluates to true. At this point another issue surfaces: After abortion, previously evaluated nodes must change their values to initial values. Therefore, a structure called subtreelist is created: The list contains the list of nodes to change their states quickly.

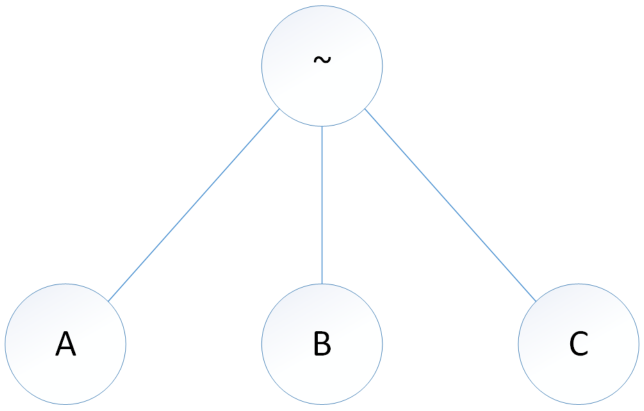


Figure 3.11: Abort operator structure

Subtree list storage

If an abort operation occurs in run-time, the nodes that become mature beforehand need to be initialized and the process needs to start from the beginning. There can be many nodes under abort operator nodes. In run-time to re-initialize these nodes, AST must be traversed and some controls are needed to confirm location of the nodes. This process takes too much of computation resource. To avoid this computation, a structure is created in each abort node to store pointers to its subtree nodes. This change uses more storage to gain speed in run time.

Decreasing AOF usage

There was also a design issue about the usage of AOF. The first usage of AOF was to activate AST nodes that are in turn next for evaluation. For that AOF requires a message from the AST. When an expected event is received by an AST, that receiving node becomes “mature” and sends a message to its corresponding AOF node. Then AOF advances its state to render control to its new set of nodes. Those nodes send activation message to their nodes to activate their corresponding AST nodes via pointers. This process is done repeatedly for every AST node evaluation to find which nodes to activate next and this is done for every rule in the rule set. It is obvious that traversing AOF and sending activation requests for every evaluation are expensive operations. Therefore, another approach is chosen and successfully applied.

A vector data structure is created to hold list of AST nodes that are going to be activated next and that structure is called “nextActivationList”. This is only necessary for the sequential operations, in other words for the "sequence" and "abort" operators. Because only in those operators, AOF goes vertical down, and system needs to know the next activated nodes. Other than these operators, there is no need for resetting some subtree. Other operators will be evaluated by calling the evaluate method of parent node. After the left children of sequential operators are matured, what to activate next is found with the help of this data structure. Thus, with this approach system gains speed by sacrificing memory.

Short Circuit Evaluation

In some cases, the result of "AND" and "OR" operations can be decided only by having one of the operand values. This process is called short circuit evaluation. After that the nodes below these operator nodes should be deactivated without waiting for the other operand to mature, to prevent unnecessary computation and to increase system efficiency. The cases that short circuit evaluation can take place are given below.

a. In "||" (Or) operation if one of the operands matures with the “true” value

b. In "&&" (And) operator if one of the operands matures with the “false” value

This approach is chosen to speed up execution time and prevent unnecessary evaluations. In "OR" operator if one of the children appears as true, then there is no need to wait for the other child. Therefore, after the reception of this decisive value, the subtree corresponding to the other operand is closed for incoming events. For the "AND" operator, if one of the child nodes is false valued, then whatever the value of other child is, the process will end with false. Therefore, waiting for other child to mature is a waste. These two approaches increase the speed of our execution and prevent unnecessary resource consumption.

Creation of Operator and Operand Classes

In run-time, when the system is working all the trees are evaluated based on incoming events and related operator. There are many types of operators and the evaluation will change depend on the current operator. To accomplish the evaluation based on our initial approach, all the operator types were checked and evaluation was completed based on the result of checking. For example, if the operator is mathematical add operator then two numbers would be added. However, while re-factoring our code, it is realized that polymorphism can be utilized to avoid type checking. Subclasses of ASTNode are created for all the operator types. The evaluate() method is created as a virtual method in the "ASTNode.h" class and it is implemented differently in all the different operator classes. The "evaluate" method is one of the most used methods at run time. With this change, type checking is avoided which would slow down the processing at run-time. The detail of this approach is given in the following. To have a more generic structure, the operand classes were also classified using specific sub classes.

Operators and Operands are the main distinction in our system. In the operands, we might have three different types, event, variable and constants. Among them only the event class has subclasses, which are Boolean and Integer. An event can be of one of the two types, “ariza” type events are Boolean valued and “belirti” type events are Integer valued. There are no more subclasses of variable or constant classes.

In operators, the main distinction is “Sequential” and “Instant”. Sequential block and subclasses are related to time related operators and issues. There are two sequential operators in our language: the “;” sequence and the “~” abort operators. Therefore, the "sequential" class has two subclasses called sequence and abort. In the “Instant” class, there are four alternatives. Our language supports Comparison, Mathematical, Bitwise and Logical operations. The supported operations are listed below and this class structure is shown in 3.12.

Math: MathPlus, MathMinus, MathTimes, MathDiv, MathNeg, MathMod

Comparison: Greater, GreaterEqual, Smaller, SmallerEqual, Equal, NotEqual

Logical: LogAnd, LogOr, LogNot, LogExor

BitWise: BitAnd, BitOr, BitComplement

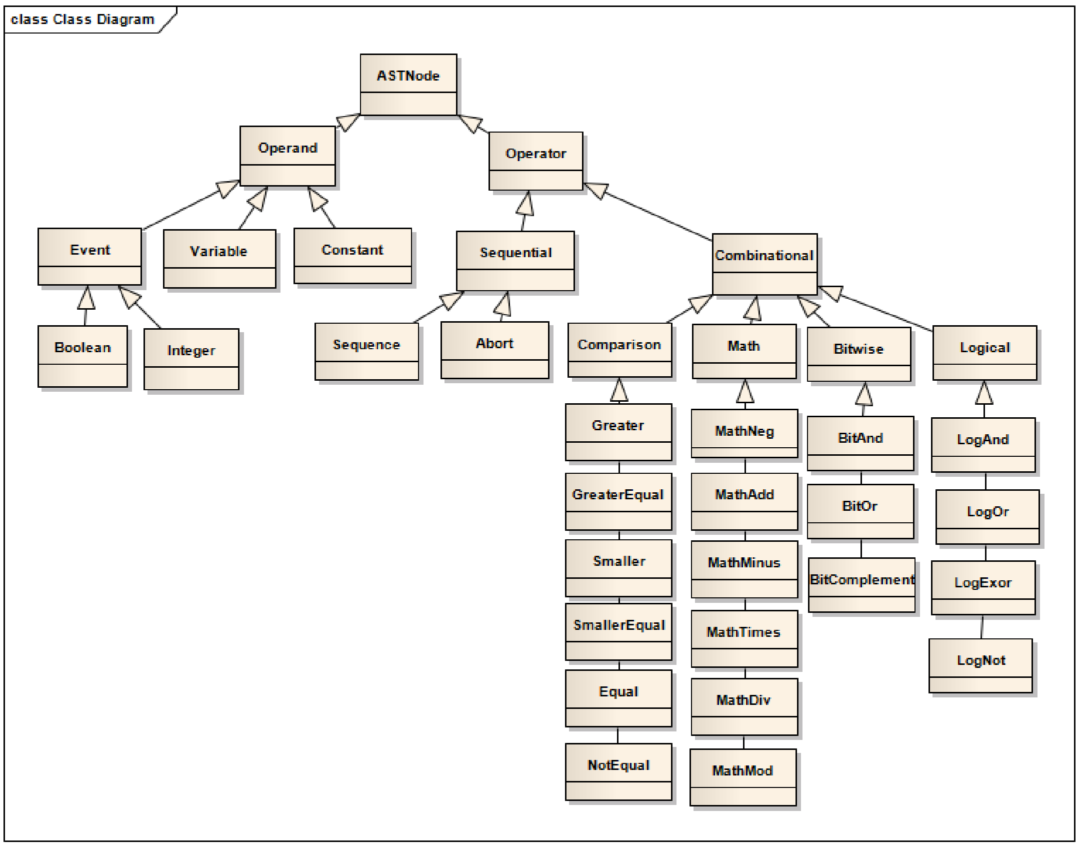


Figure 3.12: Polymorphism - Class Diagram for AST Classes

The main aim of this change was to use power of polymorphism for the evaluation functions. As a result, deciding based on the type of the operator and type of the operands to select appropriate operations is eliminated. Any operation is handled similarly and consistently. The evaluate() method is the most used method in our system and decreasing its computation time will greatly affect the whole systems performance. When an evaluate method is called, instead of controlling the operator type with some if-else or switch-case like statements, we create the nodes accordingly in initialization time and implement the evaluate method of the corresponding operator.

3.6 An Example

In this section one simple example will be demonstrated to clarify used data structures and operations. In real-time usage, many rules can be executed in one analysis. To make the example simpler, event names will be represented as upper-case letters and only condition part of the rules are presented. The example rule is:

(A ~B , C) ; (D && E)

The corresponding AST, AOF and Hash Table figures for this rule example are presented in the following.

AST is created as in Figure 3.13.

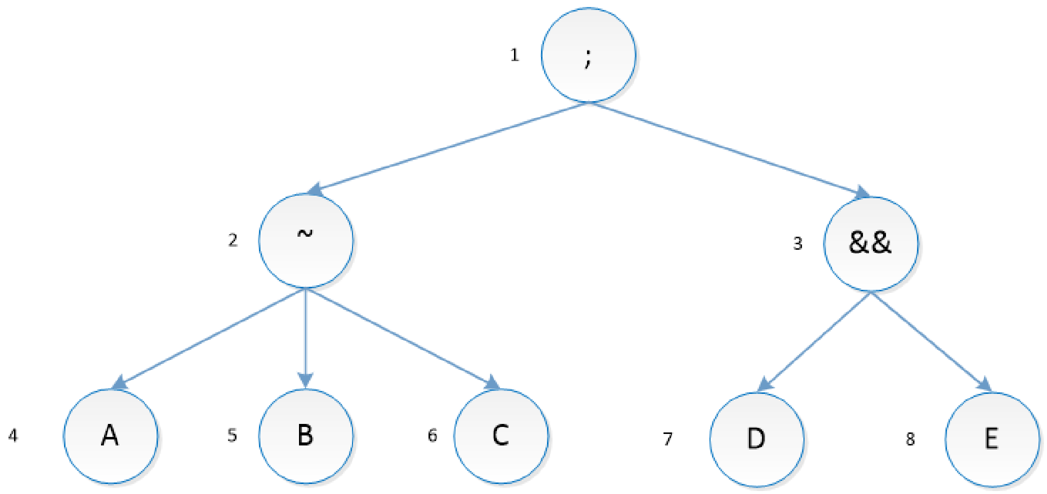


Figure 3.13: AST - At the beginning

After the AST is constructed, its AOF is builded by traversing the AST by the given algorithm. The AOF is only used to hold the current active nodes to define sequence of the nodes. AOF is shown in Figure 3.14

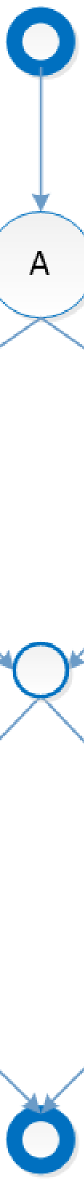
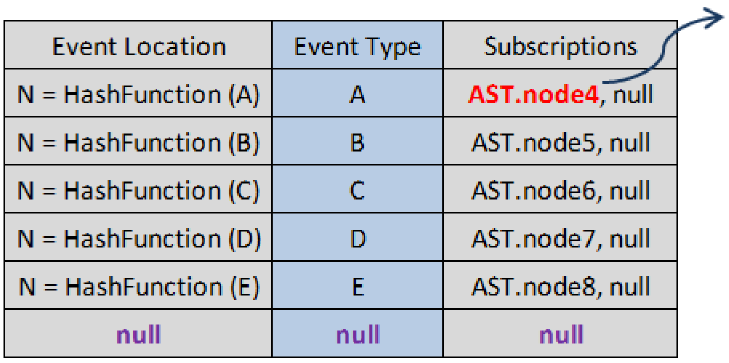


Figure 3.14: AOF - At the beginning

After AOF is constructed. Finally, hash table is built. As explained before, there are "Active" and "Passive" nodes in the hash table. The incoming events will be sent only to the "Active" nodes. Therefore, at the beginning according to the AOF, the only node where the event A is held active and the others are passive. This is shown in Figure 3.15 where the active node is colored with red. So, even if events B, C or the other events are received in the input, they are not processed.



Event Location	Event Type	Subscriptions
N = HashFunction (A)	A	AST.node4, null
N = HashFunction (B)	B	AST.node5, null
N = HashFunction (C)	C	AST.node6, null
N = HashFunction (D)	D	AST.node7, null
N = HashFunction (E)	E	AST.node8, null
null	null	null

Figure 3.15: Hash Table - At the beginning

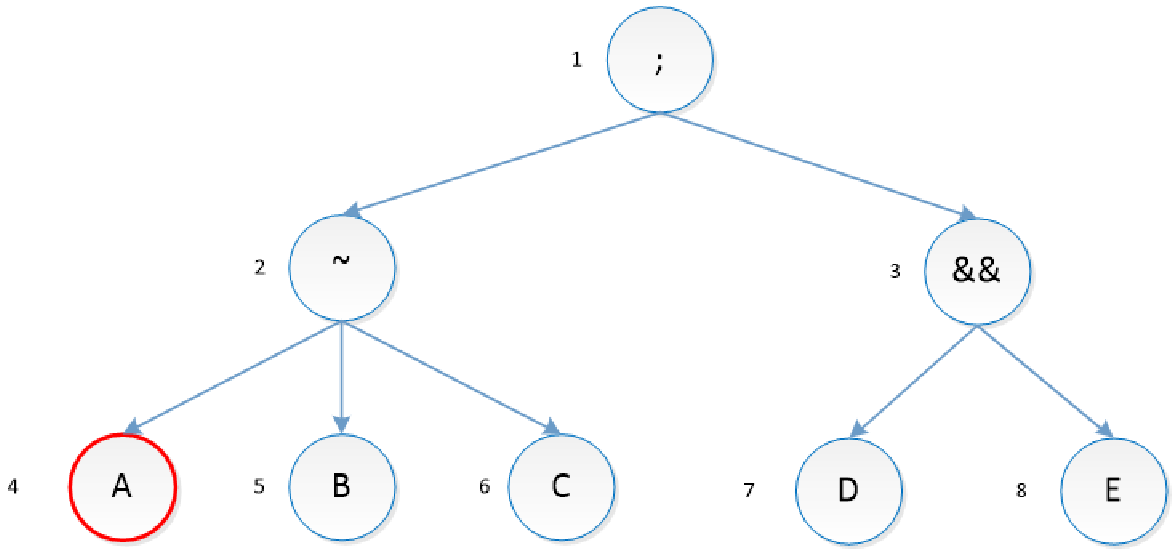
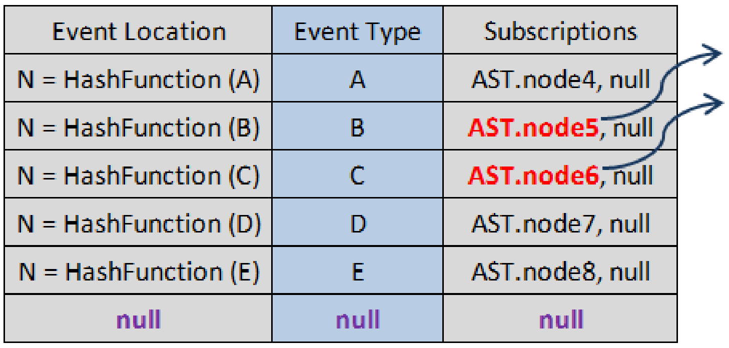


Figure 3.16: AST - After A event was evaluated

After event A arrives at the system as input, B and C nodes will be activated. This is shown in Figure 3.17. Therefore, in the hash table only event B and event C entries are activated. Between B, C events and D, E events there is a sequence operator ";", which causes D and E events to be passive and B and C operators to be active.

After events B and C are received, next and last event D and E are listened to, through the activation of the corresponding nodes in the AST. This is shown in Figure 3.19.

Here is an example for demonstrating the initialization and execution of the system. In run-time there can be many of this execution and initializations that work in parallel. Moreover, this example can demonstrate the usage of temporal



Event Location	Event Type	Subscriptions
N = HashFunction (A)	A	AST.node4, null
N = HashFunction (B)	B	**AST.node5**, null
N = HashFunction (C)	C	**AST.node6**, null
N = HashFunction (D)	D	AST.node7, null
N = HashFunction (E)	E	AST.node8, null
null	**null**	**null**

Figure 3.17: Hash Table - After A event was evaluated

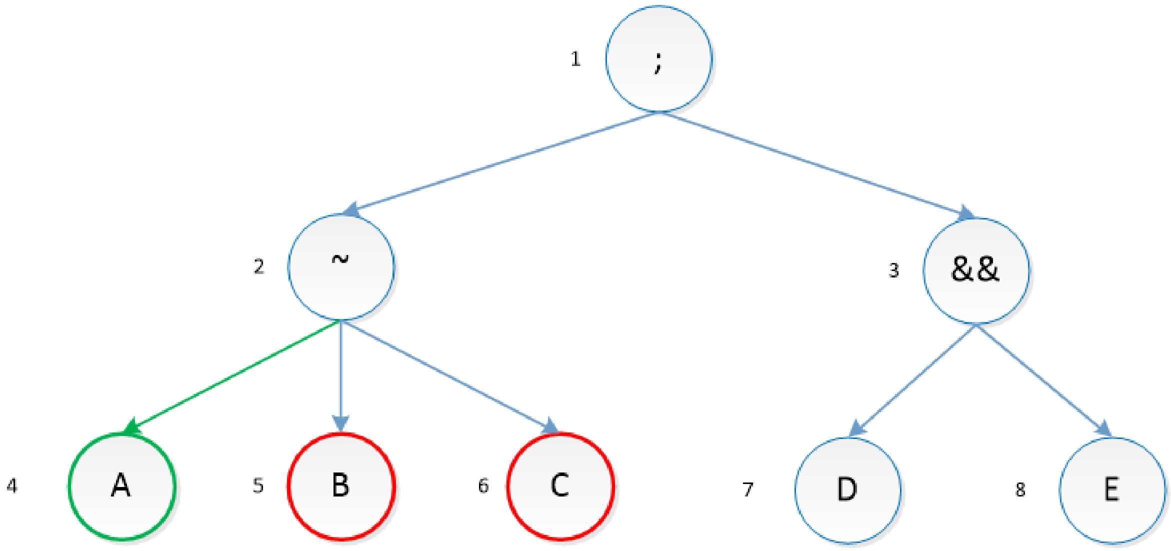
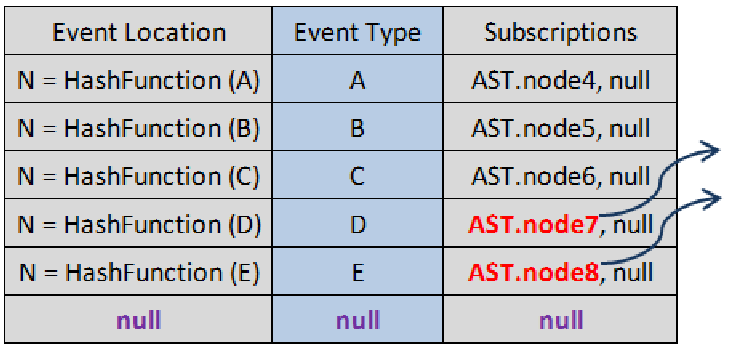


Figure 3.18: AST - After A event was received



Event Location	Event Type	Subscriptions
N = HashFunction (A)	A	AST.node4, null
N = HashFunction (B)	B	AST.node5, null
N = HashFunction (C)	C	AST.node6, null
N = HashFunction (D)	D	**AST.node7**, null
N = HashFunction (E)	E	**AST.node8**, null
null	**null**	**null**

Figure 3.19: Hash Table - After B and C events are evaluated

operators, ";" and "~".

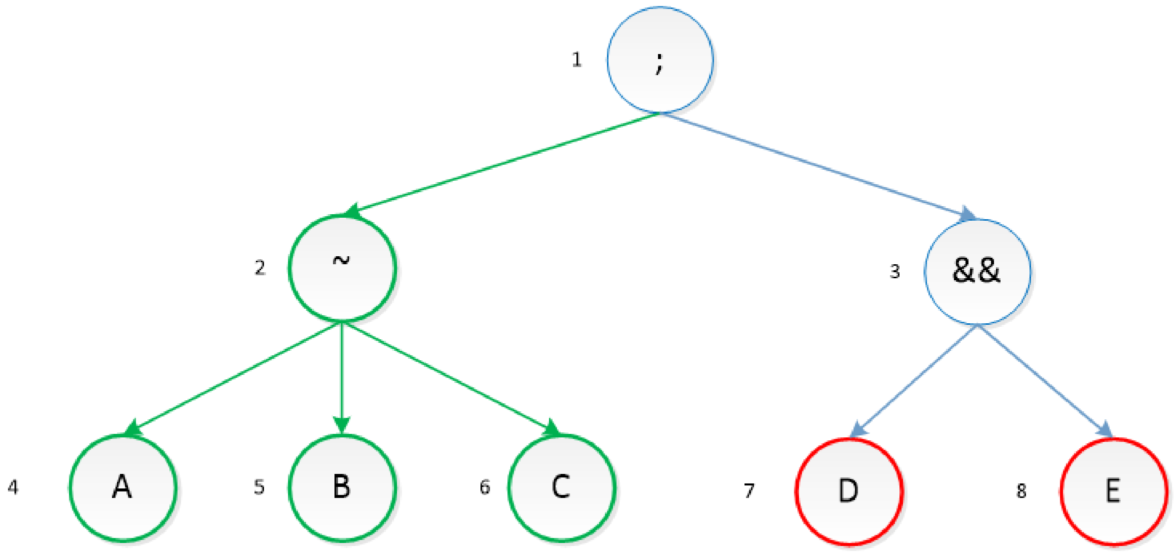


Figure 3.20: AST - After B and C events are evaluated

CHAPTER 4

CREATION OF ANALYSIS FACILITIES AND CONDUCTING RANDOM TESTS

In this thesis, five analysis facilities are created and experimentally evaluated. The first analysis facility allows users to measure time between processing events. Second analysis facility provides a performance evaluation for the generated events, which are events fired by rules, and shows their effects to the system.

The rest of the three analysis facility types are mentioned in section 10.5.2 "Dynamic Analysis of Event Processing Networks" in [8]. The first analysis facility mentioned in that book is the “Termination Problem”, which is infinitely executing cycles that involve some rules. The second analysis facility is “Reachability Issues”. If a rule exists in the rule set but never used during the analysis there is a reachability issue. Third and the last analysis facility is “Output Terminal is Unusable”, which is output events (events fired from action parts of rules) is not consumed by any of the rule in the rule set.

A performance evaluation is presented in the following sections with a corresponding tool and details of our tool and environment information is presented for further studies. After the performance evaluation, a study on the mutual exclusion detection problem and static analysis difficulties in event domain are stated. These topics are supported with examples to facilitate the understanding.

To test the created analysis facilities we created some random tests with randomly created events and rules. A "Random Event Generator" which creates random events with random values at random intervals and a "Random Rule Generator" is implemented for testing purposes. Event and rule numbers to be created can be defined by users.

4.1 Random Event Generator

Two types of events exist in our domain, "belirti" and "ariza" type events. "belirti" events are integer valued and "ariza" events are boolean valued events. The value of events are written after the event name separated with a comma. The event generator firstly selects event type as "ariza" or "belirti". Then location of the event is determined after the "kaynak" keyword with two numbers. Next, type of the event for either "ariza" or "belirti" events is determined with a random number between 0 to 99. Lastly, value of the event is determined according to the event type. Value is 0 or 1 for "ariza" events and value is a number for "belirti" events. After an event is created, it is sent to the hash table, simulating a real world event.

For our program to be more efficient, we implemented our program with multithreading capability. The “thread” class of C++ language is used for this facility. Analyses can be performed with millions of input events. An input event goes through many evaluations and processes. When these processes are happening, there is no need for event generator to wait to create the next event. This fact asserts the need for multithreading.

4.2 Random Rule Generator

Rule generation cannot be easily done completely randomly as in the event generation process. There are some operator types that cannot exist together in the same rule or result of some operations cannot be an operand for the other operators. Therefore, we created some rule templates that are correct in terms of syntax, then we created some patterns from these rules. The rule generator randomly selects a rule template then changes its events with randomly created events and save the new rule to use in analyses. Some example rule templates are given in the following. The upper case characters represent generic names of events in the template, that will be changed by the rule generator to randomly created events.

rule1 = "(A || B) && (C && D)->E;F;G;H"

rule2 = "(A   B , C) ; (D != E)->F;G"

rule3 = "(A == B) || (C > D)->E;F"

rule4 = "(A <= B) || (C && D)->E;F"

rule5 = "(A   B , C) ; (D && E)->F;G;H"

rule6 = "-(-A * B + - C)->D,E"

rule7 = "(A & B) | C)->D;E"

rule8 = "(A ; (B || C)) && (D == - E))->F;G"

rule9 = "(A   (B | C) , (D + E))->F;G;H"

rule10 = "(A - B) + (C * D)->E;F"

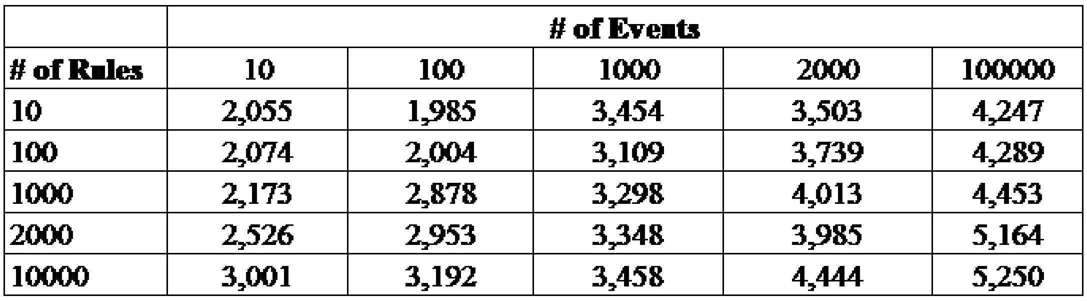
4.3 Performance Evaluation

The previous work [20] on interpreter development using Python has utilized average processing times for arriving events, for performance evaluation. To give a feeling about the performance improvement, we also utilize the similar measurements in our comparisons. In this study, we also perform the same evaluation for the comparison of the two studies. Computation environment information is also presented, because computation time also depends on the environment [20]. The previous study utilized Windows 7 64 bit operating system, Intel I7 CPU Q720 processor running on 1.6 GHz clock, and 8 MB System memory. Our study utilized an environment based on the 64 bit Windows 8 operating system. The processor is Intel® Core i5-4210 @ 1.70 GHz with 8 GB of RAM. We have used the latest version of the C++ programming language which is C++11 and Code::Blocks version 13.12 as an IDE. As a compiler MinGW with gcc version 4.9.2 is used. It is the current latest version and since version 4.9, "thread" and "chrono" headers are supported that requires no other configuration. Whereas in the previous versions users needed to do some configurations.

C++ "chrono" library is used in the implementation of performance evaluation. Timers are set into the places that events enter the system and finish their jobs. There are two different results that an event might cause in the system. First one might cause firing of a rule, and secondly an event entering the system that does not cause a firing of a rule but is still processed probably to advance the AST to a next state. We wanted to determine how much time does each of those cases spend on the average. We have conducted many experiments with huge numbers of randomly generated events.

The results of the performance evaluation are shown in Table 4.1. The number of events that is used in the analysis and results in terms of microseconds are shown. If the number of events and number of rules are small, it is more likely that the case can arrive where there is no combination of inputs that cause an evaluation in any rule. Therefore, for some small numbers, to be able to measure processing time of events, a combination that evaluates is added by hand.

Table4.1: Average Processing Time per Event Arrival in Microseconds



	# of Events				
# of Rules	10	100	1000	2000	100000
10	2,055	1,985	3,454	3,503	4,247
100	2,074	2,004	3,109	3,739	4,289
1000	2,173	2,878	3,298	4,013	4,453
2000	2,526	2,953	3,348	3,985	5,164
10000	3,001	3,192	3,458	4,444	5,250

In Appendix D.1 to D.6 the confidence interval study for these results is given. All the experimentations are with the sample size of 30, and in table 4.1 the average results are provided. For all the experimentations the variability is below 5% for 95% confidence level with sample size of 30.

In this study we also aimed to see the effect of a programming language on the performance of a program. Result of the similar study which was conducted during the PhD thesis of Kaya, is given in Table 4.2. In his study he used the Python programming language in his simulator.

Table4.2: Average Processing Times per Event Arrival in Microseconds in Kaya's study

	# of Events					
# of Rules	**1**	**10**	**100**	**2000**	**10000**	**100000**
10	**21,22**	**33,34**	**39,29**	**37,43**	**42,25**	**42,58**
100	**19,05**	**24,61**	**34,34**	**34,21**	**37,58**	**40,59**
1000	**25,63**	**28,40**	**43,69**	**32,77**	**50,37**	**n/a**
2000	**26,97**	**28,50**	**32,77**	**51,19**	**54,28**	**61,22**
10000	**27,03**	**30,28**	**50,37**	**51,19**	**60,61**	**n/a**

We can see that the choice of programming language affects the system performance along with the execution environment. Since the rules and events are randomized for any experiment, a direct comparison is not meaningful. However, a general indicator about the performance can be achieved observing the results. Also, there are other factors that affect the results like CPU usage, number of processes running etc. The environment information is presented to support more meaningful comparisons. Also Python environment is interpretive whereas our C++ environment is compiled that is a big factor in efficiency. Considering all those factors, with a broad interpretation, a more or less 10 times improvement compared to the language processor written in Python seems to be obtained in this study.

Another comparison for the aforementioned interpreters is provided. A trend analysis work is conducted and corresponding curves are plotted for the two interpreters. Figure 4.1 shows trend analysis for the interpreter created in this study and Figure 4.2 shows trend analysis for the interpreter that was created before. The black lines mentioned as linear in the graphs are the created trendlines for the performance evaluation results. A comparison oriented set of curves are also provided in Appendix E where curves for C++ and Python for one experiment (same numbers of events and rules) are plotted in one graph. These curves are plotted based on the provided data corresponding to the Python case. The performance evaluation table in Kaya's study does not include values in the rows that are at the intersection of "1000 rules - 100000 events" and "10000 rules

100000 events". Therefore in Figure 4.2, the lines that show these two cases are shorter than the others, since they do not have evaluation results for 100000 events. The processing times are shown as functions of event counts and curves are plotted for different numbers of rules. Both of the interpreters yield linear trends, however when the equation of trend lines are created it seems that the coefficients for our interpreter seems smaller then the coefficients of the equation of the previously created interpreter. Therefore, it can be stated that our interpreter displays slower increase in processing times with increasing number of events. Furthermore, in figures E.1 to E.5 a detailed comparison of the interpreters is given for the cases with different numbers of rules. In these graphs the behaviour of the two interpreters can easily be compared. As the number of events increases, processing times of both of the interpreters increase. However, it is clearly seen that the increase in the processing times with increasing number of events is slower in our interpreter.

The improvement we have addressed is basically moving from an interpretive run-time environment for the KOTAY interpreter, to a compiled one. Besides, where possible the connections among the data structures are implemented as pointers, reducing the time to exchange data. Any language that provides pointers and is offering comparable run-time performance for the developed code could be used with similar performance. Python, on the other hand, is offering faster development. Although this new language comes with modern interpreters that are acceptable in performance regarding its being interpretive (a performance slowing factor), even for the real-time requirements of the originating project was satisfied by the previous implementation in Python. A very rough speed-up estimation, looking at similar test runs in both implementations, with an approximate factor of 10 was a very attractive gain to invest. however, C++, being object oriented, also is slightly slower in performance when compared to C, and off course when compared to assembly language. There is a trade-off between the ease of development and speed-up. C++ is easier to develop when compared to C. The development ease in this perspective also means the dependability of the developed code. Having better command over the developed code, understanding it faster and easier are the benefits that come with higher-abstraction

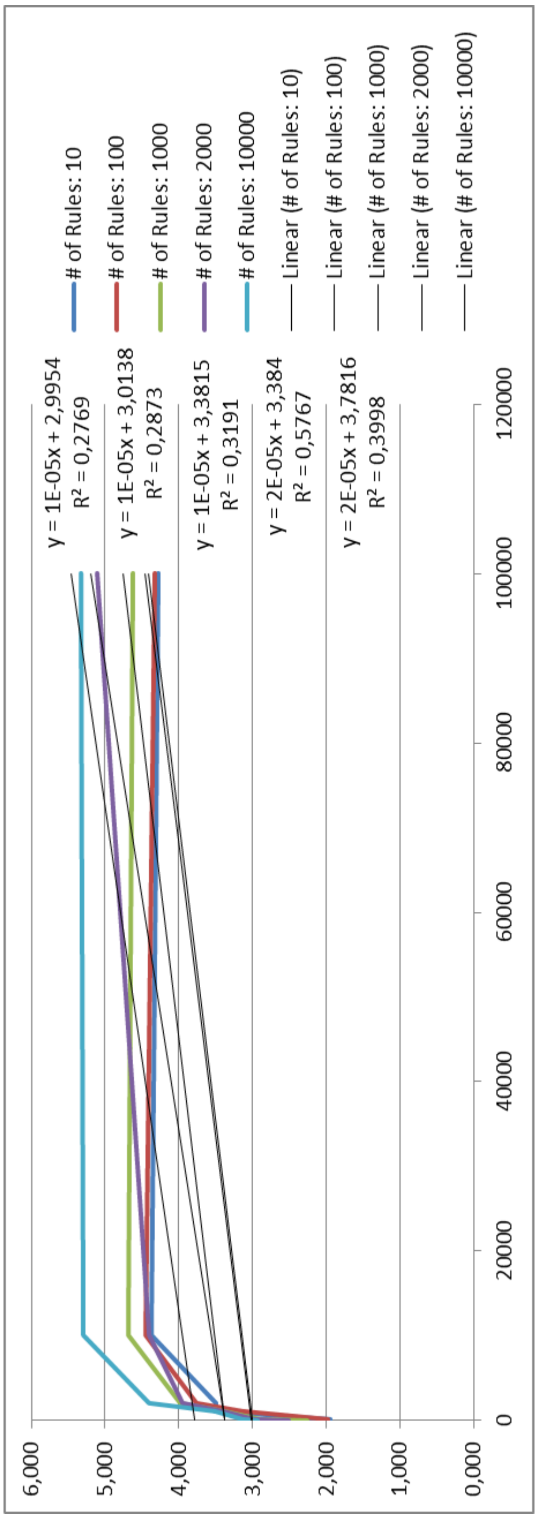


Figure 4.1: Trend Analysis for KOTAY Interpreter - C++ version

level languages. C++ has higher level of abstraction when compared to C. Also, the original project would utilize C++ codes, while a proof of concept developed in Python was acceptable. This is the main reason for having selected the men-

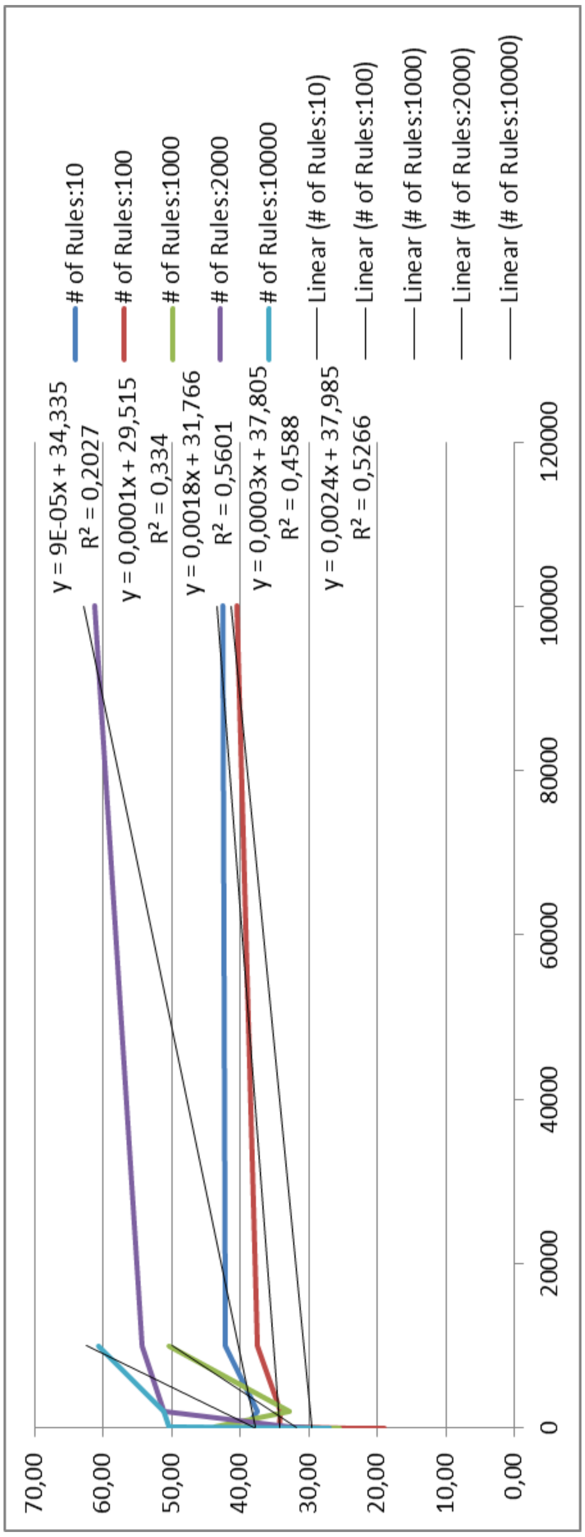


Figure 4.2: Trend Analysis for KOTAY Interpreter - Python version

tioned languages in the development life-cycle of the fault management related project.

Selection of the language does not have a big effect on the treatment of the input. In either case, the random generation of the input events would behave equal. Although the conducted measurements do not provide exact values for comparing the case with different factors, or even the use of different languages, it is not very difficult to guess the factors affecting an overall difference between the C++ and the Python cases. Treating the input events, therefore, would not inflict further conditions other than the general behaviour of the two cases (C++ and Python). There were no specific data structures or handling algorithms changed while implementing with C++. It could be argued however, that the speed-up due to C++ might be less than the overall value, when only input handling is considered: There are less numbers of data structures and less data structures involved in input event handling so the advantages of employing faster techniques may have reduced.

4.4 Static Analysis Difficulties in the Event Processing Domain and Mutual Exclusion Detection Problem in Event Processing Languages

In [26], [27], [28], [29], [30], and [31] authors try to find whether the system has a bounded response time. For this aim four "Special Forms" are defined and systems that comply with one of the forms are said to have a bounded response time. To check for a special form compatibility, primarily one of the three compatibility relations must be fulfilled. Some terms used in these studies are explained in the following.

L_a and L_b are defined as the left sides of the rule a and rule b. For instance, in the below samples, x and y constitutes L_a and m and n constitutes L_b. The parts that are stated as "test" are the condition part of rules. In the below example, test a is $(z < 7)$ and test b is $(l < 6)$

Rule a = " $(z < 7)$ -> $((x = 5), (y = 3))$"

Rule b = " $(l < 6)$ -> $((m = 8), (n = 9))$"

It is stated that two rules are compatible if and only if at least one of the three conditions hold. The three conditions are given as follows:

CR1: Test a and test b are mutually exclusive

CR2: $L_a \cap L_b = \emptyset$

CR3: Suppose $L_a \cap L_b \neq \emptyset$. Then for every variable v in $L_a \cap L_b$, the same expression must be assigned to both in rule a and b.

These studies have been conducted for variable involving rules only; they do not consider the events. However, in terms of analysis, there is not much difference between event and variable concepts. Events can also be represented as variables, with some caution. Events with values can be considered the same as variables, whereas Boolean type events can be considered as variables with only two values. In the above conditions CR2 and CR3 can be adapted to consider events easily. When mutual exclusion detection is considered, there is a problem. Firstly, let us look at formal definition of mutual exclusion given in those studies.

Mutual Exclusion: The main logic behind mutual exclusion is defined as "If two tests are mutually exclusive, then only one of the corresponding rules can be enabled at a time." in [28]. Definition of Mutual Exclusion is given in the following.

Let $\boldsymbol{T} = (v_1, v_2, ..., v_n)$ and let $\overline{v}$ be the vector $< v_1, v_2, ..., v_n >$. With this definition, each test in a program can be viewed as a function f ($\overline{v}$) from the space of $\overline{v}$ to the set true, false . Let f_a be the function corresponding to the test a and let V_a be the subset of the space of $\overline{v}$ for which the function f_a maps to true. Let $V_{a,i}$ be the subset of the values of v_i for which the function f_a may map to true; that is, if the value of the variable v_i is in the subset $V_{a,i}$ then there exists an assignment of values to the variables in the set $\boldsymbol{T}$ - v_i such that the function f_a maps to true. Note that if the variable v_k does not appear in the test a , then $V_{a,k}$ is the entire domain of v_k. We say that two tests a and b are mutually exclusive if and only if the subsets V_a, and V_b of the corresponding functions f_a, f_b are disjoint [28].

From the above formal definition, what it means by mutual exclusion relation can be understood. An example to clarify this subject is given in the following.

The two rules are mutually exclusive:

Rule 1: a == 5 and b == true -> c = 3

Rule 2: a != 5 -> c = 5

The values for which the functions map to true is:

$V_{1,a} = \{5\}$

$V_{1,b} = \{true\}$

$V_{2,a} = \{ -\infty, 5], [5, \infty \} \}$

$V_{2,b} = \{true, false\}$

Since b does not exist in rule 2, according to the definition, the entire domain of b will be in the truth map of b for rule 2. As it can be seen in the above spaces, f_1 and f_2 are disjoint since there is no common value that both rules can fire at the same time. Therefore, rule 1 and rule 2 are mutually exclusive. Important point is that, these rules include variables only (no events).

The following two rules are not mutually exclusive:

Rule 1: (a == 5) and (b == true) -> c = 3

Rule 2: a == 5 -> c = 5

The variable values that functions map to true have a common map, which is a = 5 and b = true. To convert these rules to event processing rules, let variable "a" be "Event A" and "b" be "Event B" and "c" be "Event C".

The rules become:

Rule 1: (A == 5) and (B == true) -> C = 3

Rule 2: A == 5 -> C = 5

In that situation, there is again a combination of event values and sequences where these two rules can fire at the same time. If event B occurs at t_1 with the value "true" and after some time, at time t_2 event A occurs with the value "5", where $t_1 < t_2$ then these two rules can fire at the same time.

Let us consider another scenario:

We have the same rules, but the sequence of events changes. The event A occurs with the value of "5" at time t_1. After some time event B occurs with value "true" at time t_2, where $t_1 < t_2$. If these would be values, they would fire at the same time. However, in this situation at t_1 the second rule is fired, because there is only one condition for that rule to fire and it has occurred. But, rule 1 could not fire at that time, since it is expecting also event B to enter with value "true". So, although the expected value combination for the events exist, since the sequence of occurrence of events was not correct, the two rules could not fire at the same time.

Moreover, in our study, we deal with the two temporal operators: sequence operator ";" and abort operator "~". When analyzing mutual exclusion subject, these operators require extra work. Consider the following example;

Rule 1: (A == 5) ; (B == true) -> C = 3

Rule 2: A == 5 -> C = 5

The above two rules are mutually exclusive in the event processing domain. Because, for the first rule to fire, firstly, A has to occur with value "5" and after that B has to occur with value "true". For the second rule to fire there is only one condition and it is the occurrence of A with value "5". But when it occurs, it will immediately fire and there is no chance for rule 1 to fire at the same time with rule 2. For rule 1 to fire, some time has to pass and after that event B has to occur. When only variables are considered (no events), without temporal operators, their mutual exclusion analysis is considerably easier.

A similar scenario exists for the abort operator;

Rule 1: (A == 5) ~(B == true) , (D < 4) -> C = 3

Rule 2: (A == 5) ~(D < 4) , (B == true) -> C = 5

For rules 1 and 2 to fire, firstly A must occur with value 5. Then for rule 1 to fire, B has to occur with value "true" before D occurs with value smaller than 4. However, this is just the opposite for rule 2. For rule 2 to fire, event D has to occur with value smaller than 4 before B occurs with value "true". Since these two cases cannot happen at the same time, these two rules are mutually exclusive. In the variable case, there are no temporal operators and therefore, their mutual exclusion analysis is considerably easy with respect to the event processing case.

To conclude, in event processing, time is crucial. Occurrence time and sequence of events define the system behaviour. Furthermore, as stated before in the real-time systems section, correctness of the real-time systems actually depends on time and sequence of event occurrences in addition to results of logical computations. However, for variables there is nothing like variable occurrence and sequence; they are time independent. Therefore, mutual exclusion detection problem requires different and extra work in the event processing domain. Moreover, if domain specific operators and behaviours exist, this problem becomes a domain dependent problem and may require specific solutions for every system.

4.5 Analysis Facility 1: Measuring time between the processing of input events

Measuring the time between two specified input events is the first analysis facility we have developed. It is an important facility, because it allows users to see how much time it takes for the program to finish execution.

To test the created analysis facility, we have performed the analysis with randomly created events and rules. A timer is used in the implementation of the analysis. The timer starts before the first input event enters the system and ends when last event finishes its operations. The results of the analysis are given in Table B.1 in Appendix A.

The test is conducted with random events and rules. For five different tests listed in the table, the processing time is zero. The reason is that, none of the input events existed in the rules. These results may change for different event and rule sets. This test only shows the analysis facility is working.

4.6 Analysis Facility 2: Performance and Effects of Generated Events

Second analysis facility we have developed measures the performance and effect of events fired from the action part. These events are referred as "Generated Events". When a rule fires, the events in the action part of the rule become input to the system and they may have a great effect on the system behavior and results. Two types of analysis facilities were developed. First type of facility conducts a performance evaluation for generated events. This facility has some differences from the performance evaluation of input events mentioned in Section 5.3 in the implementation, since distinguishing generated events in the input events requires some work. The curiosity to find whether the performance of generated events is different than input events has been the motivation to include this analysis facility. To test the created analysis facility, randomly created rules and events are used. Result of the tests is listed in Table B.1 in Appendix B.

The second type of analysis facility provides the ratio of computation times for generated events which cause firing of a rule to those which stop in the middle of an evaluation. For testing purposes, random events and rules are used. The result of the tests is listed in Table B.2 in Appendix B.

The tests for these two types of analysis facilities are conducted only to show that facilities are working correctly. For some other tests with different rule and event sets, the results will be different.

4.7 Analysis Facility 3: Static and Dynamic Cycle Warning

This analysis facility is called as the "Termination Problem" in [8]. Termination problem occurs because of a loop involving some rules, resulting with infinitely executing cycles. The third analysis facility in our study warns users for possible cyclic cases statically before run time and dynamically during run time.

Static cycle warnings take place before run time. Events and rules that may create a cycle in run time, are revealed through a syntactic analysis. To clarify this concept, a simple example is presented next. Two very basic rules are typed below. These rules may create a cycle in run time. The first rule can be enabled with the reception of events A and B and as an output, the C event is fired. In rule 2, C or D events are necessary for that rule to fire and as an outcome A event is generated. If the required case occurs, these rules may create a cycle and system crashes. Thus, our tool determines these rules as probable cyclic rules and warns the users.

Rule 1: A && B -> C

Rule 2: C || D -> A

"Static Cycle Warning Rule" logic can be briefly explained as follows:

Static Cycle Warning Rule:

If the action part of a rule m includes one or more events that is found in rule n's condition part and rule n has one or more events in its action part that also exists in rule m's condition part, then these two rules are said to be probable cyclic.

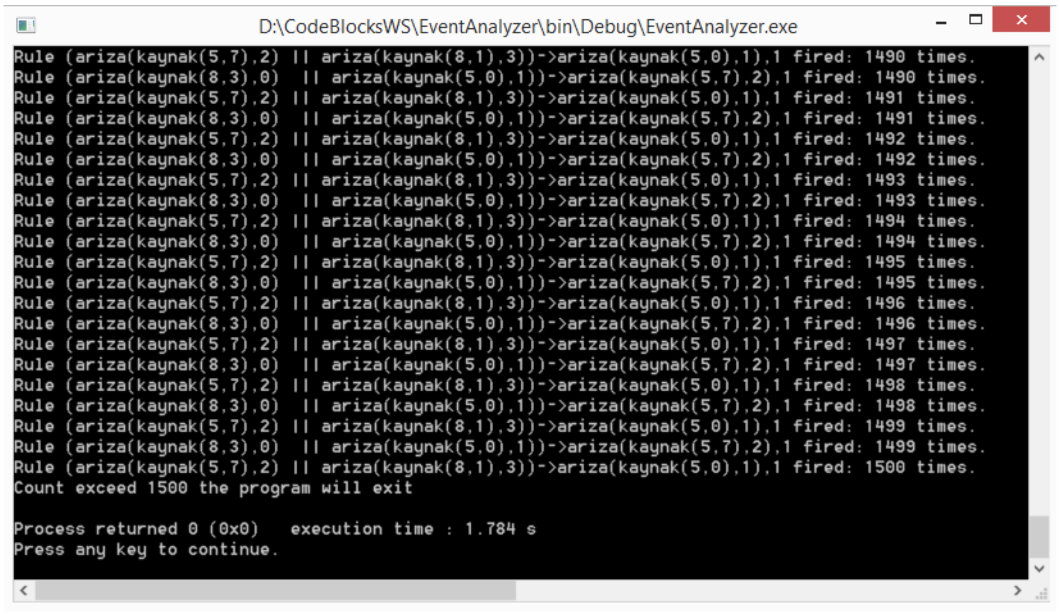
1000 randomly generated rules are used to show the usage of static cycle warning analysis facility. Output of that test placed is in Appendix C. The probable cycles are shown in the following format: "Rule m -> Rule n -> Rule m". Which means, rule m might fire rule n and then rule n might fire rule m. The event that might cause a cycle in these rules is also given afterwards.

Dynamic cycle warning warns users during run time execution. To detect these cycles a number that defines the upper limit for a rule firing is defined, which is called “warning limit”. When this limit is exceeded, the system is further monitored for a short period. If cycle continues the program closes itself, after the number of firings which is called “termination limit”.

A test was conducted with thousand randomly generated rules and hundred events. Then, two rules were added intentionally for the system to enter a cycle. Rule structures are like the following: "A|| B->C" and "C || D->B". The warning limit is given as 1000 and termination limit is given as 1500. Figure 4.3 shows the output; the program first warns the user, then it finishes the program execution.

"(ariza(kaynak(8,3),0) || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1"

"(ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1"



```
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1490 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1490 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1491 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1491 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1492 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1492 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1493 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1493 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1494 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1494 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1495 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1495 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1496 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1496 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1497 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1497 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1498 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1498 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1499 times.
Rule (ariza(kaynak(8,3),0)  || ariza(kaynak(5,0),1))->ariza(kaynak(5,7),2),1 fired: 1499 times.
Rule (ariza(kaynak(5,7),2) || ariza(kaynak(8,1),3))->ariza(kaynak(5,0),1),1 fired: 1500 times.
Count exceed 1500 the program will exit

Process returned 0 (0x0)   execution time : 1.784 s
Press any key to continue.
```

Figure 4.3: Output of Dynamic Cycle Warning Test

4.8 Analysis Facility 4: Coverage Analysis Facility

Fourth analysis facility created in this study provides coverage analysis facility for a given rule and input event set. If a rule in a rule set never fires during an analysis that rule is defined as "unreachable rule". This situation generally occurs because of a mistake, but sometimes can be done on purpose. In both cases, users must be warned. This analysis facility is useful for revealing some missing conditions or mistakes in the input event or rule set. Static analysis for detecting unreachable rules gives possible results and their results are not certain. Rules stated as "can fire" in static analysis may not fire during a dynamic analysis, in our study simulation. These rules may be excluded from the rule set to increase system performance and avoid unnecessary computation.

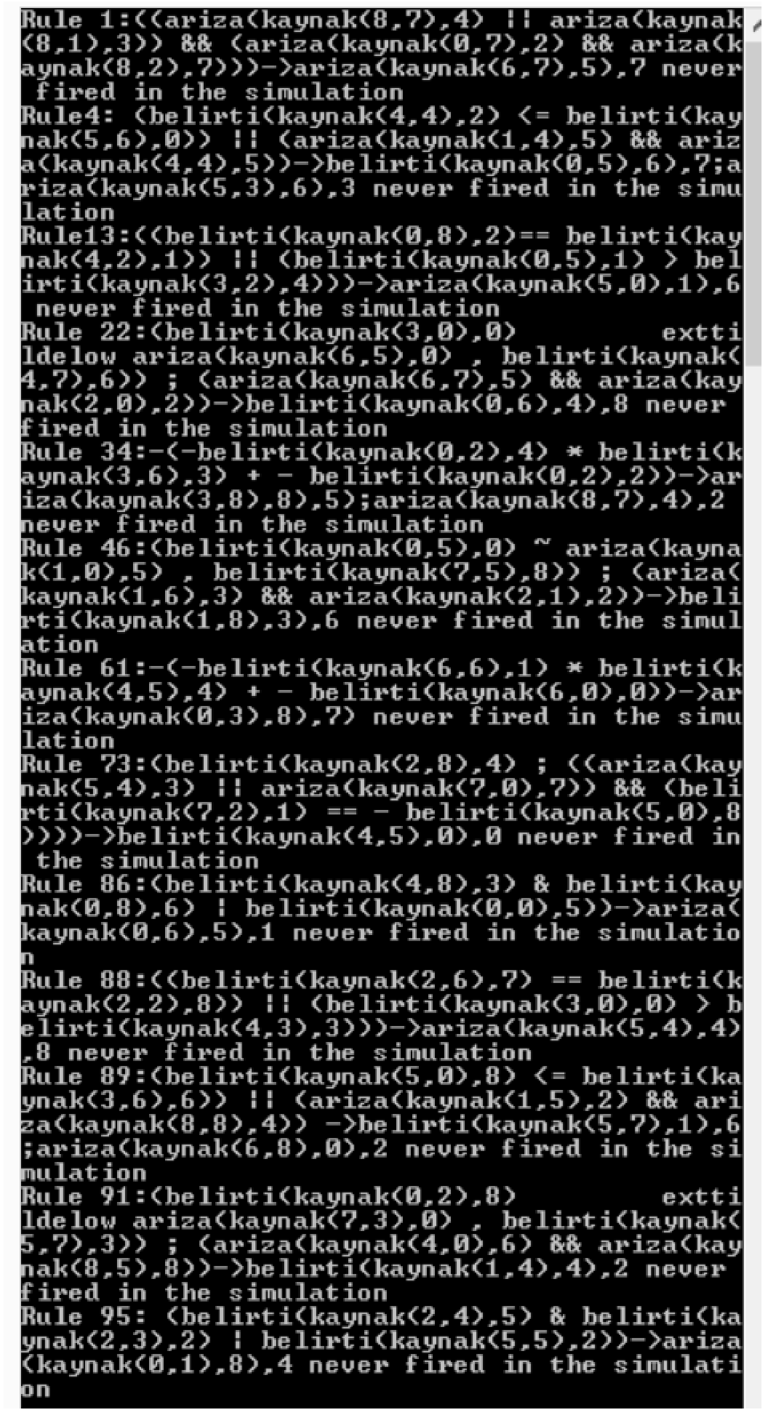
This analysis facility is tested with 100 randomly created rules and 10000 input events. This random experimentation shows that this analysis facility is working. Output of this test is shown in Figure 4.4. This is a random test only to show usage and to produce example output for the developed analysis facility. Results will change for every different rule and event set.

4.9 Analysis Facility 5: Non-Consumed Generated Events Detection

The fifth analysis facility we have developed in this study provides finding generated events that are not received by any of the existing rules. Generated events that are not received by any of the existing rules in the rule set are referred as "Non-consumed Generated Events". These events might point to a problem in the system or its design.

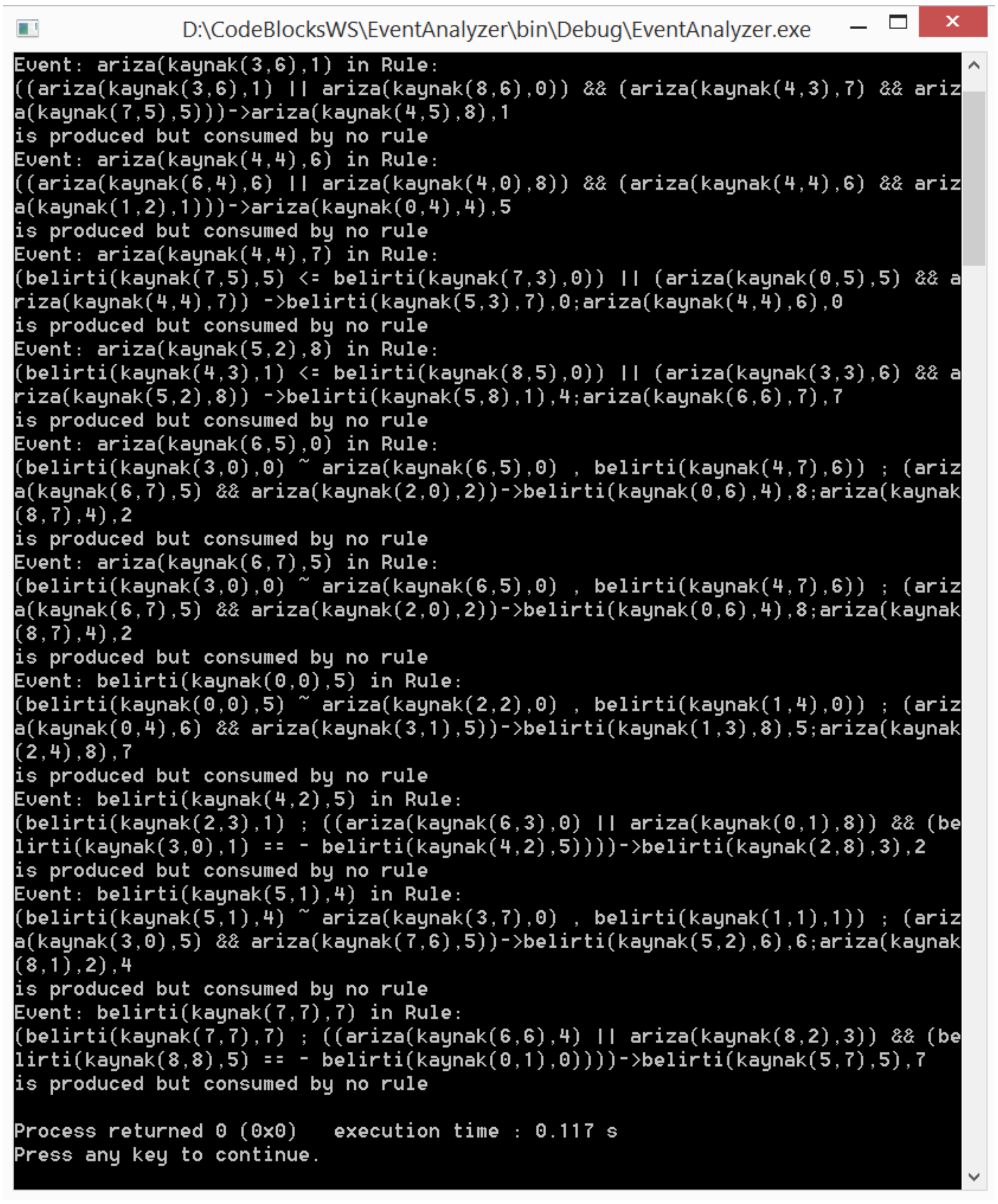
Using static analysis to find non-consumed generated events is not sufficient for critical systems. Static analysis semantically searches condition and action parts of the rules, and state the events as non-consumed generated events, if those generated events do not exist in any of the condition parts. However, during run-time, some generated events may not be received by any of the rules because corresponding part of the rule may not be active, or that part may already have been evaluated. Thus, dynamic analysis is required for an accurate result.

To test this analysis facility 100 rules and 10000 events, both randomly generated, were used. The output is shown in Figure 4.5. The output only shows the sample usage of this analysis facility. Different results occur for each different rule and event set.



```
Rule 1:((ariza(kaynak(8,7),4) || ariza(kaynak
(8,1),3)) && (ariza(kaynak(0,7),2) && ariza(k
aynak(8,2),7)))->ariza(kaynak(6,7),5),7 never
 fired in the simulation
Rule4: (belirti(kaynak(4,4),2) <= belirti(kay
nak(5,6),0)) || (ariza(kaynak(1,4),5) && ariz
a(kaynak(4,4),5))->belirti(kaynak(0,5),6),7;a
riza(kaynak(5,3),6),3 never fired in the simu
lation
Rule13:((belirti(kaynak(0,8),2)== belirti(kay
nak(4,2),1)) || (belirti(kaynak(0,5),1) > bel
irti(kaynak(3,2),4)))->ariza(kaynak(5,0),1),6
 never fired in the simulation
Rule 22:(belirti(kaynak(3,0),0)         exttti
ldelow ariza(kaynak(6,5),0) , belirti(kaynak(
4,7),6)) ; (ariza(kaynak(6,7),5) && ariza(kay
nak(2,0),2))->belirti(kaynak(0,6),4),8 never
fired in the simulation
Rule 34:-(-belirti(kaynak(0,2),4) * belirti(k
aynak(3,6),3) + - belirti(kaynak(0,2),2))->ar
iza(kaynak(3,8),8),5);ariza(kaynak(8,7),4),2
never fired in the simulation
Rule 46:(belirti(kaynak(0,5),0) ~ ariza(kayna
k(1,0),5) , belirti(kaynak(7,5),8)) ; (ariza(
kaynak(1,6),3) && ariza(kaynak(2,1),2))->beli
rti(kaynak(1,8),3),6 never fired in the simul
ation
Rule 61:-(-belirti(kaynak(6,6),1) * belirti(k
aynak(4,5),4) + - belirti(kaynak(6,0),0))->ar
iza(kaynak(0,3),8),7) never fired in the simu
lation
Rule 73:(belirti(kaynak(2,8),4) ; ((ariza(kay
nak(5,4),3) || ariza(kaynak(7,0),7)) && (beli
rti(kaynak(7,2),1) == - belirti(kaynak(5,0),8
))))->belirti(kaynak(4,5),0),0 never fired in
 the simulation
Rule 86:(belirti(kaynak(4,8),3) & belirti(kay
nak(0,8),6) | belirti(kaynak(0,0),5))->ariza(
kaynak(0,6),5),1 never fired in the simulatio
n
Rule 88:((belirti(kaynak(2,6),7) == belirti(k
aynak(2,2),8)) || (belirti(kaynak(3,0),0) > b
elirti(kaynak(4,3),3)))->ariza(kaynak(5,4),4)
,8 never fired in the simulation
Rule 89:(belirti(kaynak(5,0),8) <= belirti(ka
ynak(3,6),6)) || (ariza(kaynak(1,5),2) && ari
za(kaynak(8,8),4)) ->belirti(kaynak(5,7),1),6
;ariza(kaynak(6,8),0),2 never fired in the si
mulation
Rule 91:(belirti(kaynak(0,2),8)         exttti
ldelow ariza(kaynak(7,3),0) , belirti(kaynak(
5,7),3)) ; (ariza(kaynak(4,0),6) && ariza(kay
nak(8,5),8))->belirti(kaynak(1,4),4),2 never
fired in the simulation
Rule 95: (belirti(kaynak(2,4),5) & belirti(ka
ynak(2,3),2) | belirti(kaynak(5,5),2))->ariza
(kaynak(0,1),8),4 never fired in the simulati
on
```

Figure 4.4: Output of Coverage Analysis Facility Test



D:\CodeBlocksWS\EventAnalyzer\bin\Debug\EventAnalyzer.exe

```
Event: ariza(kaynak(3,6),1) in Rule:
((ariza(kaynak(3,6),1) || ariza(kaynak(8,6),0)) && (ariza(kaynak(4,3),7) && ariz
a(kaynak(7,5),5)))->ariza(kaynak(4,5),8),1
is produced but consumed by no rule
Event: ariza(kaynak(4,4),6) in Rule:
((ariza(kaynak(6,4),6) || ariza(kaynak(4,0),8)) && (ariza(kaynak(4,4),6) && ariz
a(kaynak(1,2),1)))->ariza(kaynak(0,4),4),5
is produced but consumed by no rule
Event: ariza(kaynak(4,4),7) in Rule:
(belirti(kaynak(7,5),5) <= belirti(kaynak(7,3),0)) || (ariza(kaynak(0,5),5) && a
riza(kaynak(4,4),7)) ->belirti(kaynak(5,3),7),0;ariza(kaynak(4,4),6),0
is produced but consumed by no rule
Event: ariza(kaynak(5,2),8) in Rule:
(belirti(kaynak(4,3),1) <= belirti(kaynak(8,5),0)) || (ariza(kaynak(3,3),6) && a
riza(kaynak(5,2),8)) ->belirti(kaynak(5,8),1),4;ariza(kaynak(6,6),7),7
is produced but consumed by no rule
Event: ariza(kaynak(6,5),0) in Rule:
(belirti(kaynak(3,0),0) ~ ariza(kaynak(6,5),0) , belirti(kaynak(4,7),6)) ; (ariz
a(kaynak(6,7),5) && ariza(kaynak(2,0),2))->belirti(kaynak(0,6),4),8;ariza(kaynak
(8,7),4),2
is produced but consumed by no rule
Event: ariza(kaynak(6,7),5) in Rule:
(belirti(kaynak(3,0),0) ~ ariza(kaynak(6,5),0) , belirti(kaynak(4,7),6)) ; (ariz
a(kaynak(6,7),5) && ariza(kaynak(2,0),2))->belirti(kaynak(0,6),4),8;ariza(kaynak
(8,7),4),2
is produced but consumed by no rule
Event: belirti(kaynak(0,0),5) in Rule:
(belirti(kaynak(0,0),5) ~ ariza(kaynak(2,2),0) , belirti(kaynak(1,4),0)) ; (ariz
a(kaynak(0,4),6) && ariza(kaynak(3,1),5))->belirti(kaynak(1,3),8),5;ariza(kaynak
(2,4),8),7
is produced but consumed by no rule
Event: belirti(kaynak(4,2),5) in Rule:
(belirti(kaynak(2,3),1) ; ((ariza(kaynak(6,3),0) || ariza(kaynak(0,1),8)) && (be
lirti(kaynak(3,0),1) == - belirti(kaynak(4,2),5))))->belirti(kaynak(2,8),3),2
is produced but consumed by no rule
Event: belirti(kaynak(5,1),4) in Rule:
(belirti(kaynak(5,1),4) ~ ariza(kaynak(3,7),0) , belirti(kaynak(1,1),1)) ; (ariz
a(kaynak(3,0),5) && ariza(kaynak(7,6),5))->belirti(kaynak(5,2),6),6;ariza(kaynak
(8,1),2),4
is produced but consumed by no rule
Event: belirti(kaynak(7,7),7) in Rule:
(belirti(kaynak(7,7),7) ; ((ariza(kaynak(6,6),4) || ariza(kaynak(8,2),3)) && (be
lirti(kaynak(8,8),5) == - belirti(kaynak(0,1),0))))->belirti(kaynak(5,7),5),7
is produced but consumed by no rule

Process returned 0 (0x0)   execution time : 0.117 s
Press any key to continue.
```

Figure 4.5: Output of Non-Consumed Generated Events Detection

CHAPTER 5

CONCLUSION

In this study an simulator and five analysis facilities were developed for a rule-based domain specific language. The language was designed and created before during a PhD study, along with an interpreter using Python for usage in real-time mission critical systems for complex event processing. The performance of the interpreter was also improved with the C++ implementation. The implementation environment has also an effect on the results of the performance evaluation and environment information is provided for future comparisons.

Then, difficulties in static analysis for the event processing domain were investigated. The study shows that if time is in consideration, conducting a static analysis for rule based languages is not easy and requires extra work. Such difficulties were explained carefully and recorded as a future work to be studied further.

Next, five dynamic analysis facilities were created. The first analysis facility allows users of this tool to measure simulation time to predict the real usage time of their system. Second analysis facility shows the effects and performance of events fired from the action parts of the rules. The third analysis facility helps and warns users to find cycles during a simulation. Also a static cycle detection facility was developed to use beforehand. Fourth analysis facility reveals the unused rules during a simulation in a rule set, which decreases performance and creates confusions. The fifth analysis facility in this study discovers the unused output events in the system and presents them to the users. To demonstrate the usage of the analysis facilities, random tests have been conducted with randomly generated rules and events. Test results of the analysis facilities were presented after the definition of facilities and some of them were given in the appendices. With these tests operability of the system can be assessed with different numbers of events and rules and proper results appeared in the outputs.

Finally, improvement of this simulator would be a great future study. Support for different rule based languages can prove to be very useful. Performance improvement is also very critical as in all other studies. Performing new types of analysis and combination of dynamic analysis with static analysis, where possible, would also be a great study for this area. Repeating performance evaluation with the same event and rule set as in the previous PhD study, would be better to make more meaningful comparisons.

REFERENCES

[1] B. Zupan and A.M.K. Cheng. Optimization of rule-based systems using state space graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 10(2):238–254, Mar 1998.

[2] A.K. Mok. Formal analysis of real-time equational rule-based systems. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 308–318, Dec 1989.

[3] James C. Browne, Allen Emerson, Mohamed Gouda, Daniel Miranker, Aloysius Mok, and Rosier Louis. Bounded time fault tolerant rule-based systems. *Telematics and Informatics*, 7:441–454, 1990.

[4] Mok A. K. Wang R. H. Deriving response-time bounds for equational rule-based programs. *Security & Privacy, IEEE 9.2*, pages 50–57, 2011.

[5] Chih-Kan Wang and AloysiusK. Mok. Timing analysis of mrl: A real-time rule-based system. *Real-Time Systems*, 5(1):89–128, 1993.

[6] A.M.K. Cheng and H.-Y. Tsai. A graph-based approach for timing analysis and refinement of ops5 knowledge-based systems. *Knowledge and Data Engineering, IEEE Transactions on*, 16(2):271–288, Feb 2004.

[7] Ozgur Kaya, Seyedsasan Hashemikhabir, Cengiz Togay, and Ali Hikmet Dogru. A rule-based domain specific language for fault management. *J. Integr. Des. Process Sci.*, 14(3):13–23, July 2010.

[8] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[9] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. Ep-sparql: A unified language for event processing and stream reasoning. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 635–644, New York, NY, USA, 2011. ACM.

[11] Kia Teymourian and Adrian Paschke. Enabling knowledge-based complex event processing. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 37:1–37:7, New York, NY, USA, 2010. ACM.

[12] Y. Magid, A. Adi, M. Barnea, D. Botzer, and E. Rabinovich. Application generation framework for real-time complex event processing. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 1162–1167, July 2008.

[13] James L. Schmidt Simon M. Kao Jackson Y. Readk Thomas J. Laffey, Preston A. Cox. Real-time knowledge-based systems. *AI Magazine*, 9(1):27–45, 1988.

[14] P.A. Ramamoorthy and S. Huang. Implementation of rule-based expert systems for time-critical applications using neural networks. In *Systems Engineering, 1989., IEEE International Conference on*, pages 147–150, 1989.

[15] A.D. Lunardhi and K.M. Passino. Verification of dynamic properties of rule-based expert systems. In *Decision and Control, 1991., Proceedings of the 30th IEEE Conference on*, pages 1561–1566 vol.2, Dec 1991.

[16] Charles L. Forgy. Expert systems. chapter Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, pages 324–341. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990.

[17] Roland Stühmer Kay-Uwe Schmidt, Darko Anicic. Event-driven reactivity a survey and requirement analysis. *SBPM Proceedings.*

[18] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification.* John Wiley & Sons, Inc., New York, NY, USA, 1 edition, 2002.

[19] John A Stankovic. Real-time computing. *Byte, pág*, pages 155–162, 1992.

[20] Ozgur Kaya. *A Rule-Based Domain Specific Language For Fault Management.* PhD thesis, METU, Ankara, Turkey, 2014.

[21] Robinson Selvamony. Introduction To The Rete Algorithm. `http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/`, 2010. [Online; accessed 19-May-2015].

[22] Robert B. Doorenbos. *Production Matching for Large Learning Systems.* PhD thesis, Pittsburgh, PA, USA, 1995. UMI Order No. GAX95-22942.

[23] Ing-Ray Chen and Tawei Tsao. A reliability model for real-time rule-based expert systems. *Reliability, IEEE Transactions on*, 44(1):54–62, Mar 1995.

[24] Tin A Nguyen, Walton A Perkins, Thomas J Laffey, and Deanne Pecora. Checking an expert systems knowledge base for consistency and completeness. In *IJCAI*, volume 85, pages 375–378, 1985.

[25] Rick Evertsz. The automated analysis of rule-based systems, based on their procedural semantics. In *IJCAI*, pages 22–29, 1991.

[26] A.M.K. Cheng and C.-H. Chen. Efficient response time bound analysis of real-time rule-based systems. In *Computer Assurance, 1992. COMPASS '92. 'Systems Integrity, Software Safety and Process Security: Building the System Right.', Proceedings of the Seventh Annual Conference on*, pages 63–76, Jun 1992.

[27] A.M.K. Cheng, J.C. Browne, A.K. Mok, and Rwo-Hsi Wang. Analysis of real-time rule-based systems with behavioral constraint assertions specified in estella. *Software Engineering, IEEE Transactions on*, 19(9):863–885, Sep 1993.

[28] A.M.K. Cheng, J.C. Browne, A.K. Mok, and Rwo-Hsi Wang. Estella; a facility for specifying behavioral constraint assertions in real-time rule-based systems. In *Computer Assurance, 1991. COMPASS '91, Systems Integrity, Software Safety and Process Security. Proceedings of the Sixth Annual Conference on*, pages 107–123, Jun 1991.

[29] A.M.K. Cheng and C.-K. Wang. Fast static analysis of real-time rule-based systems to verify their fixed point convergence. In *Computer Assurance, 1990. COMPASS '90, Systems Integrity, Software Safety and Process Security., Proceedings of the Fifth Annual Conference on*, pages 46–56, June 1990.

[30] Jeng-Rung Chen and A.M.K. Cheng. Response time analysis of eql real-time rule-based systems. *Knowledge and Data Engineering, IEEE Transactions on*, 7(1):26–43, Feb 1995.

[31] A.M.K. Cheng and Jeng-Rung Chen. Response time analysis of ops5 production systems. *Knowledge and Data Engineering, IEEE Transactions on*, 12(3):391–409, May 2000.

[32] Charles Lanny Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Pittsburgh, PA, USA, 1979. AAI7919143.

[33] Wikipedia. Reverse polish notation — wikipedia, the free encyclopedia, 2015. [Online; accessed 16-August-2015].

[34] A. C. Scott Suwa, M. and E.H. Shortliffe. An approach to verifying completeness and consistentcy in a rule-based expert system. *The AI Magazine*, pages 16–21, 1982.

[35] INRIA Sophia Antipolis. Verification and validation of knowledge-based program supervision systems. 1995.

[36] RA Stachowitz, CL Chang, TS Stock, and JB Combs. Building validation tools for knowledge-based systems. In *NASA. Lyndon B. Johnson Space Center, Houston, Texas, First Annual Workshop on Space Operations Automation and Robotics(SOAR 87) p 209-216(SEE N 88-17206 09-59)*, 1987.

[37] Petter Fogelqvist. Verification of completeness and consistency in knowledge based systems a design theory. 2011.

[38] Masoud Mansouri-samani, Morris Sloman, and Morris Sloman. Gem - a generalised event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4, 1997.

APPENDIX A

TEST OF ANALYSIS FACILITY 1

TableA.1: Processing Time of Number of Events in Milliseconds

	# of Event				
# of Rules	10	100	1000	10000	100000
10	0	0	0.003	0.048	0.266
100	0	0.0019	0.003	0.034	0.363
1000	0	0.002	0.012	0.097	0.893
2000	0	0.002	0.022	0.203	1.680
10000	0.001	0.008	27.737	82.957	N/A

APPENDIX B

TEST OF ANALYSIS FACILITY 2

TableB.1: Average Processing Time of Events received from Action Part per Event Arrival in Microseconds

	# of Events					
# of Rules	2000		10000		100000	
	Fire	Mid-Eval	Fire	Mid-Eval	Fire	Mid-Eval
10	N/A	N/A	3987	7879	N/A	N/A
100	N/A	4099	N/A	7442	7221	1569
1000	N/A	5788	19923	7371	8665	3822
2000	21765	7553	18524	959	11584	4595
10000	7354	2718	7675	2745	10539	3788

TableB.2: Ratio of Events received from Action Part: Causes Fire and Stays in Mid-Eval

	# of Events					
# of Rules	2000		10000		100000	
	Fire	Mid-Eval	Fire	Mid-Eval	Fire	Mid-Eval
10	N/A	N/A	8%	92%	N/A	N/A
100	0%	100%	0%	100%	7%	93%
1000	6%	94%	15%	85%	10%	90%
2000	10%	90%	13%	87%	11%	89%
10000	21%	79%	21%	79%	21%	79%

APPENDIX C

TEST OF ANALYSIS FACILITY 3

Possible Cycles:

Rule 11 -> Rule 409 -> Rule 11

Event that might cause a cycle: ariza(kaynak(5,0),1)

Rule 31 -> Rule 31 -> Rule 31

Event that might cause a cycle: belirti(kaynak(3,2),2)

Rule 61 -> Rule 427 -> Rule 61

Event that might cause a cycle: belirti(kaynak(4,4),5)

Rule 157 -> Rule 781 -> Rule 157

Event that might cause a cycle: belirti(kaynak(6,0),0)

Rule 336 -> Rule 794 -> Rule 336

Event that might cause a cycle: belirti(kaynak(8,3),3)

Rule 409 -> Rule 11 -> Rule 409

Event that might cause a cycle: belirti(kaynak(3,2),4)

Rule 427 -> Rule 61 -> Rule 427

Event that might cause a cycle: belirti(kaynak(1,3),5)

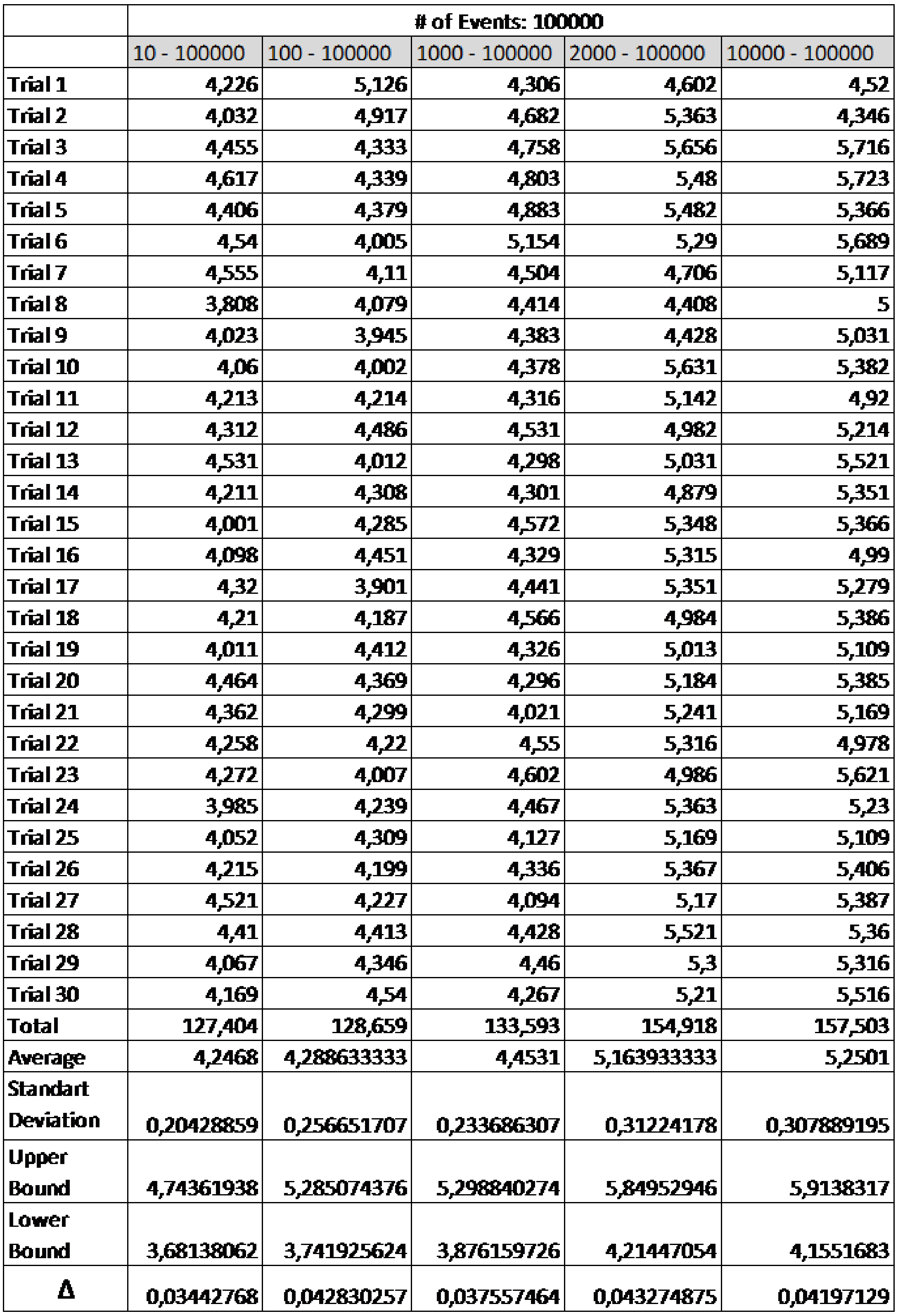
Rule 430 -> Rule 430 -> Rule 430

```
Event that might cause a cycle: belirti(kaynak(7,2),5)

Rule 434 -> Rule 556 -> Rule 434

Event that might cause a cycle: ariza(kaynak(5,6),1)

Rule 479 -> Rule 817 -> Rule 479

Event that might cause a cycle: ariza(kaynak(1,8),2)

Rule 556 -> Rule 434 -> Rule 556

Event that might cause a cycle: belirti(kaynak(4,0),4)

Rule 781 -> Rule 157 -> Rule 781

Event that might cause a cycle: belirti(kaynak(6,7),3)

Rule 794 -> Rule 336 -> Rule 794

Event that might cause a cycle: ariza(kaynak(4,7),0)

Rule 817 -> Rule 479 -> Rule 817

Event that might cause a cycle: belirti(kaynak(3,0),2)

Rule 853 -> Rule 853 -> Rule 853

Event that might cause a cycle: belirti(kaynak(7,3),7)

Rule 903 -> Rule 928 -> Rule 903

Event that might cause a cycle: ariza(kaynak(7,3),0)

Rule 928 -> Rule 903 -> Rule 928

Event that might cause a cycle: belirti(kaynak(0,4),7)
```

APPENDIX D

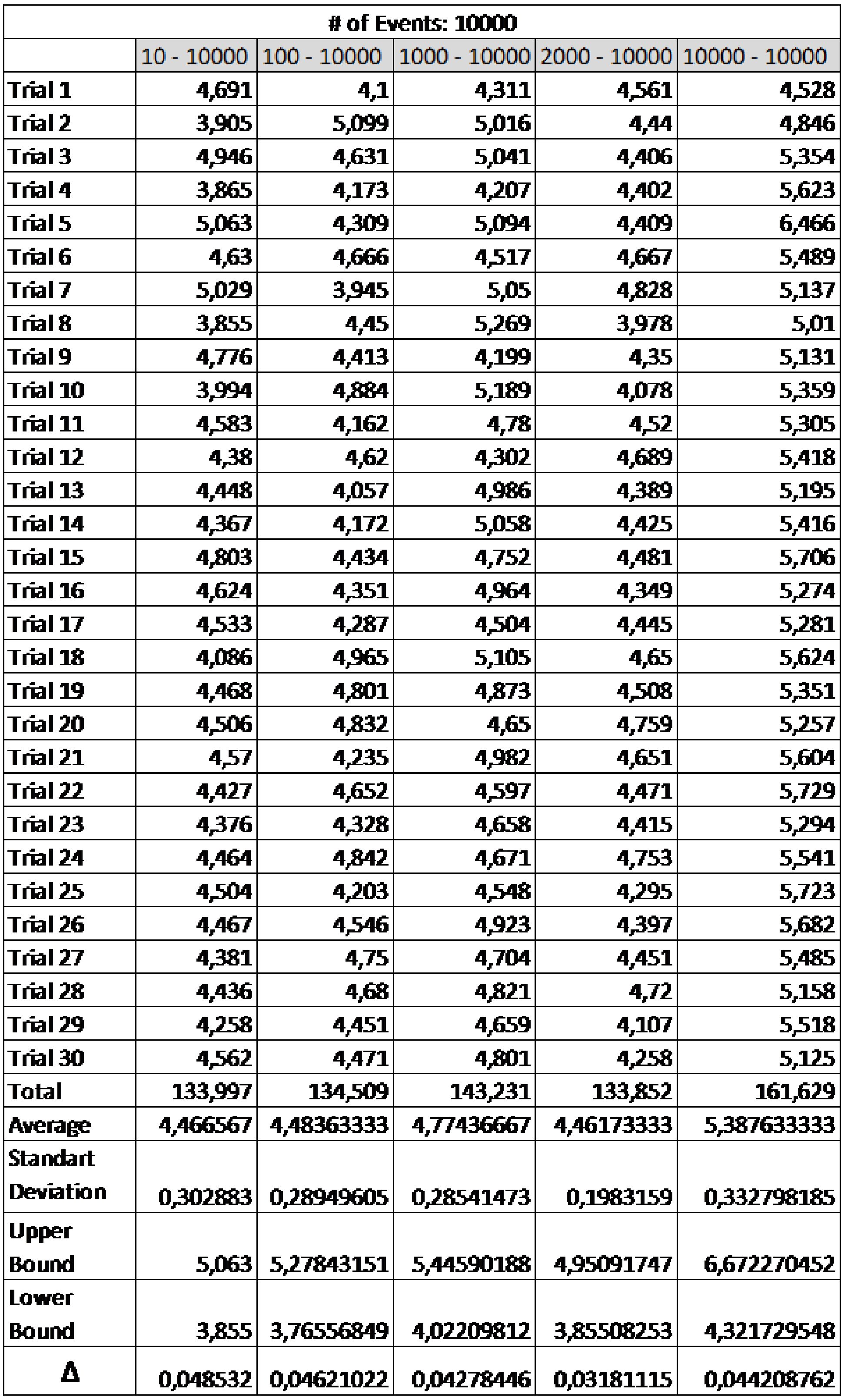
CONFIDENCE INTERVAL STUDY FOR PERFORMANCE EVALUATION

TableD.1: Processing Time of Number of Events in Milliseconds for 100000 events



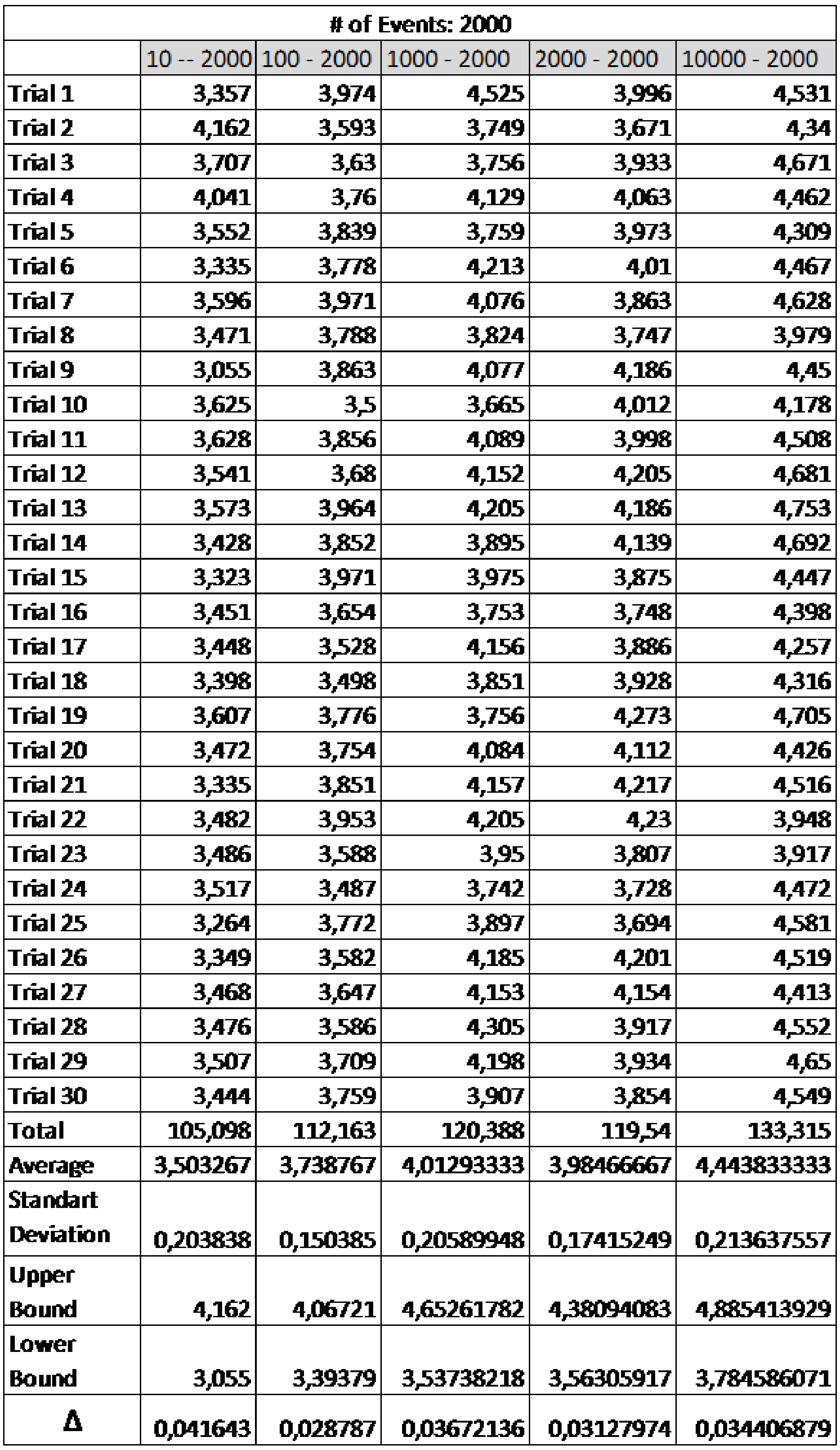
	# of Events: 100000				
	10 - 100000	100 - 100000	1000 - 100000	2000 - 100000	10000 - 100000
Trial 1	4,226	5,126	4,306	4,602	4,52
Trial 2	4,032	4,917	4,682	5,363	4,346
Trial 3	4,455	4,333	4,758	5,656	5,716
Trial 4	4,617	4,339	4,803	5,48	5,723
Trial 5	4,406	4,379	4,883	5,482	5,366
Trial 6	4,54	4,005	5,154	5,29	5,689
Trial 7	4,555	4,11	4,504	4,706	5,117
Trial 8	3,808	4,079	4,414	4,408	5
Trial 9	4,023	3,945	4,383	4,428	5,031
Trial 10	4,06	4,002	4,378	5,631	5,382
Trial 11	4,213	4,214	4,316	5,142	4,92
Trial 12	4,312	4,486	4,531	4,982	5,214
Trial 13	4,531	4,012	4,298	5,031	5,521
Trial 14	4,211	4,308	4,301	4,879	5,351
Trial 15	4,001	4,285	4,572	5,348	5,366
Trial 16	4,098	4,451	4,329	5,315	4,99
Trial 17	4,32	3,901	4,441	5,351	5,279
Trial 18	4,21	4,187	4,566	4,984	5,386
Trial 19	4,011	4,412	4,326	5,013	5,109
Trial 20	4,464	4,369	4,296	5,184	5,385
Trial 21	4,362	4,299	4,021	5,241	5,169
Trial 22	4,258	4,22	4,55	5,316	4,978
Trial 23	4,272	4,007	4,602	4,986	5,621
Trial 24	3,985	4,239	4,467	5,363	5,23
Trial 25	4,052	4,309	4,127	5,169	5,109
Trial 26	4,215	4,199	4,336	5,367	5,406
Trial 27	4,521	4,227	4,094	5,17	5,387
Trial 28	4,41	4,413	4,428	5,521	5,36
Trial 29	4,067	4,346	4,46	5,3	5,316
Trial 30	4,169	4,54	4,267	5,21	5,516
Total	127,404	128,659	133,593	154,918	157,503
Average	4,2468	4,288633333	4,4531	5,163933333	5,2501
Standart Deviation	0,20428859	0,256651707	0,233686307	0,31224178	0,307889195
Upper Bound	4,74361938	5,285074376	5,298840274	5,84952946	5,9138317
Lower Bound	3,68138062	3,741925624	3,876159726	4,21447054	4,1551683
Δ	0,03442768	0,042830257	0,037557464	0,043274875	0,04197129

TableD.2: Processing Time of Number of Events in Milliseconds for 10000 events



# of Events: 10000					
	10 - 10000	100 - 10000	1000 - 10000	2000 - 10000	10000 - 10000
Trial 1	4,691	4,1	4,311	4,561	4,528
Trial 2	3,905	5,099	5,016	4,44	4,846
Trial 3	4,946	4,631	5,041	4,406	5,354
Trial 4	3,865	4,173	4,207	4,402	5,623
Trial 5	5,063	4,309	5,094	4,409	6,466
Trial 6	4,63	4,666	4,517	4,667	5,489
Trial 7	5,029	3,945	5,05	4,828	5,137
Trial 8	3,855	4,45	5,269	3,978	5,01
Trial 9	4,776	4,413	4,199	4,35	5,131
Trial 10	3,994	4,884	5,189	4,078	5,359
Trial 11	4,583	4,162	4,78	4,52	5,305
Trial 12	4,38	4,62	4,302	4,689	5,418
Trial 13	4,448	4,057	4,986	4,389	5,195
Trial 14	4,367	4,172	5,058	4,425	5,416
Trial 15	4,803	4,434	4,752	4,481	5,706
Trial 16	4,624	4,351	4,964	4,349	5,274
Trial 17	4,533	4,287	4,504	4,445	5,281
Trial 18	4,086	4,965	5,105	4,65	5,624
Trial 19	4,468	4,801	4,873	4,508	5,351
Trial 20	4,506	4,832	4,65	4,759	5,257
Trial 21	4,57	4,235	4,982	4,651	5,604
Trial 22	4,427	4,652	4,597	4,471	5,729
Trial 23	4,376	4,328	4,658	4,415	5,294
Trial 24	4,464	4,842	4,671	4,753	5,541
Trial 25	4,504	4,203	4,548	4,295	5,723
Trial 26	4,467	4,546	4,923	4,397	5,682
Trial 27	4,381	4,75	4,704	4,451	5,485
Trial 28	4,436	4,68	4,821	4,72	5,158
Trial 29	4,258	4,451	4,659	4,107	5,518
Trial 30	4,562	4,471	4,801	4,258	5,125
Total	133,997	134,509	143,231	133,852	161,629
Average	4,466567	4,48363333	4,77436667	4,46173333	5,387633333
Standart Deviation	0,302883	0,28949605	0,28541473	0,1983159	0,332798185
Upper Bound	5,063	5,27843151	5,44590188	4,95091747	6,672270452
Lower Bound	3,855	3,76556849	4,02209812	3,85508253	4,321729548
Δ	0,048532	0,04621022	0,04278446	0,03181115	0,044208762

TableD.3: Processing Time of Number of Events in Milliseconds for 2000 events

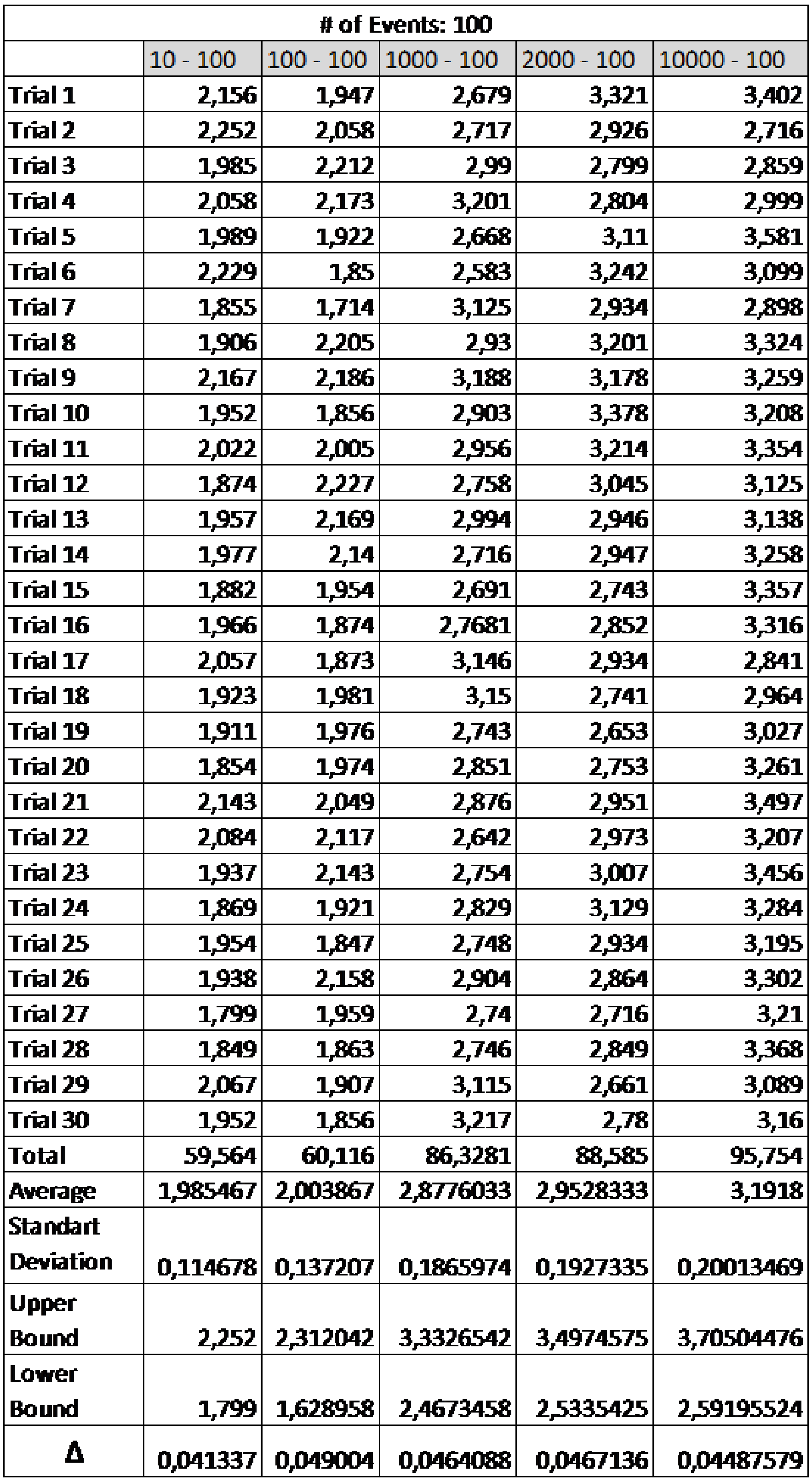


# of Events: 2000					
	10 -- 2000	100 - 2000	1000 - 2000	2000 - 2000	10000 - 2000
Trial 1	3,357	3,974	4,525	3,996	4,531
Trial 2	4,162	3,593	3,749	3,671	4,34
Trial 3	3,707	3,63	3,756	3,933	4,671
Trial 4	4,041	3,76	4,129	4,063	4,462
Trial 5	3,552	3,839	3,759	3,973	4,309
Trial 6	3,335	3,778	4,213	4,01	4,467
Trial 7	3,596	3,971	4,076	3,863	4,628
Trial 8	3,471	3,788	3,824	3,747	3,979
Trial 9	3,055	3,863	4,077	4,186	4,45
Trial 10	3,625	3,5	3,665	4,012	4,178
Trial 11	3,628	3,856	4,089	3,998	4,508
Trial 12	3,541	3,68	4,152	4,205	4,681
Trial 13	3,573	3,964	4,205	4,186	4,753
Trial 14	3,428	3,852	3,895	4,139	4,692
Trial 15	3,323	3,971	3,975	3,875	4,447
Trial 16	3,451	3,654	3,753	3,748	4,398
Trial 17	3,448	3,528	4,156	3,886	4,257
Trial 18	3,398	3,498	3,851	3,928	4,316
Trial 19	3,607	3,776	3,756	4,273	4,705
Trial 20	3,472	3,754	4,084	4,112	4,426
Trial 21	3,335	3,851	4,157	4,217	4,516
Trial 22	3,482	3,953	4,205	4,23	3,948
Trial 23	3,486	3,588	3,95	3,807	3,917
Trial 24	3,517	3,487	3,742	3,728	4,472
Trial 25	3,264	3,772	3,897	3,694	4,581
Trial 26	3,349	3,582	4,185	4,201	4,519
Trial 27	3,468	3,647	4,153	4,154	4,413
Trial 28	3,476	3,586	4,305	3,917	4,552
Trial 29	3,507	3,709	4,198	3,934	4,65
Trial 30	3,444	3,759	3,907	3,854	4,549
Total	105,098	112,163	120,388	119,54	133,315
Average	3,503267	3,738767	4,01293333	3,98466667	4,443833333
Standart Deviation	0,203838	0,150385	0,20589948	0,17415249	0,213637557
Upper Bound	4,162	4,06721	4,65261782	4,38094083	4,885413929
Lower Bound	3,055	3,39379	3,53738218	3,56305917	3,784586071
Δ	0,041643	0,028787	0,03672136	0,03127974	0,034406879

TableD.4: Processing Time of Number of Events in Milliseconds for 1000 events

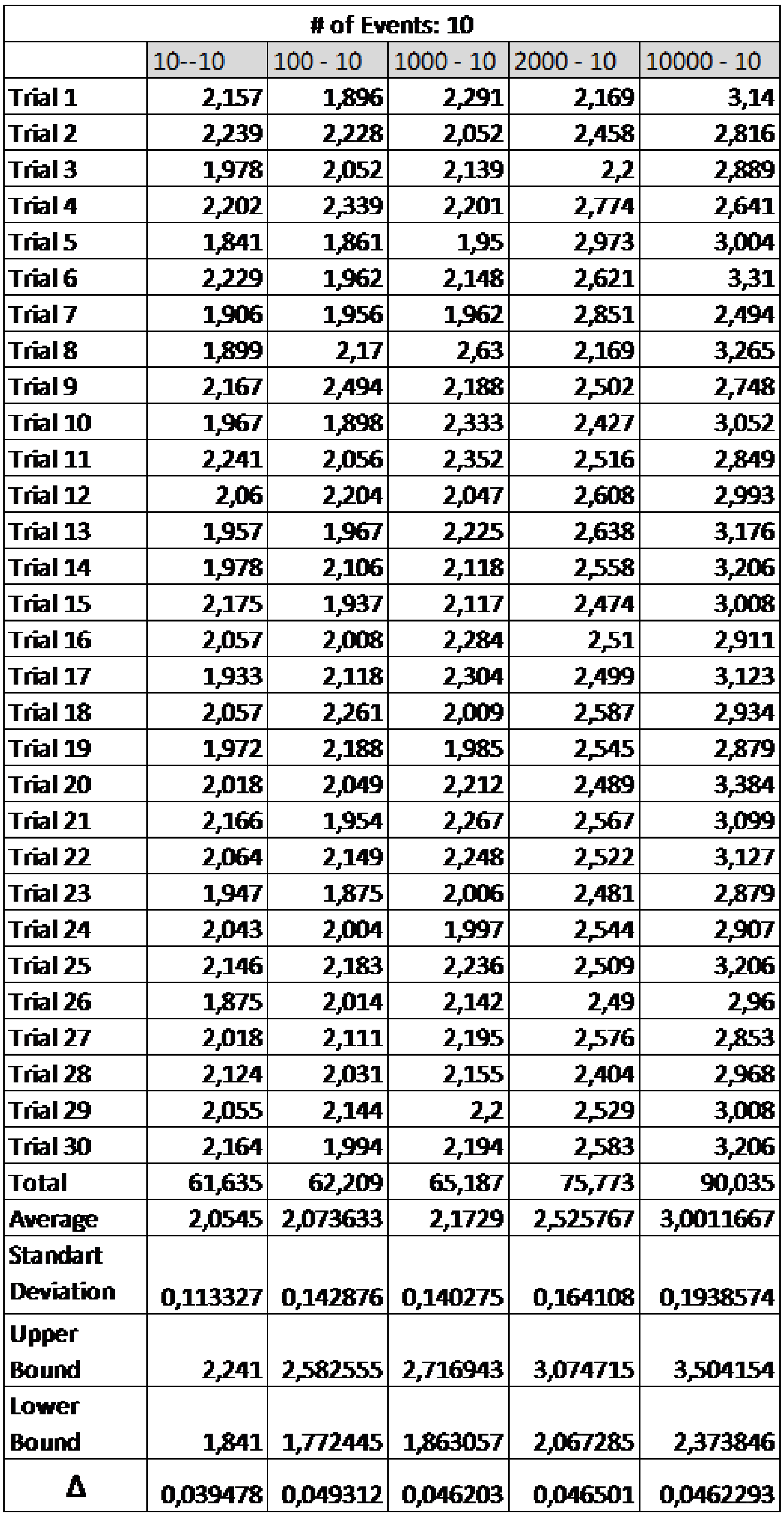
# of Events: 1000					
	10 -- 1000	100 - 1000	1000 - 1000	2000 - 1000	10000 - 1000
Trial 1	3,078	2,936	2,957	3,038	3,146
Trial 2	2,931	3,421	3,25	3,118	3,326
Trial 3	4,028	3,011	3,184	3,482	3,629
Trial 4	3,955	2,56	3,178	3,069	3,258
Trial 5	2,99	3,517	3,163	2,941	3,645
Trial 6	3,532	3,45	3,638	3,546	3,421
Trial 7	3,461	2,864	3,643	3,698	3,587
Trial 8	3,756	3,332	3,34	3,406	3,582
Trial 9	3,104	3,014	3,716	3,458	3,763
Trial 10	3,168	3,261	3,56	3,682	3,693
Trial 11	3,627	3,215	3,408	3,552	3,612
Trial 12	3,574	3,118	3,584	3,447	3,753
Trial 13	3,588	3,367	3,517	3,218	3,358
Trial 14	3,416	2,958	3,204	3,657	3,456
Trial 15	3,338	2,891	3,249	3,404	3,354
Trial 16	3,48	3,15	3,019	3,149	3,486
Trial 17	3,716	3,246	3,174	3,258	3,557
Trial 18	3,488	3,219	3,297	3,167	3,082
Trial 19	3,439	3,145	2,917	3,267	3,346
Trial 20	3,58	3,117	3,276	2,928	3,725
Trial 21	3,417	3,257	3,384	3,158	3,689
Trial 22	3,442	2,904	3,416	3,425	3,094
Trial 23	3,429	2,88	3,534	3,529	3,127
Trial 24	3,419	3,204	2,854	3,394	3,647
Trial 25	3,521	3,216	3,351	3,756	3,68
Trial 26	3,451	3,155	3,446	3,618	3,821
Trial 27	3,387	3,197	3,057	2,941	3,067
Trial 28	3,408	2,912	3,251	3,417	3,543
Trial 29	3,473	2,891	2,91	3,449	2,94
Trial 30	3,429	2,873	3,46	3,275	3,349
Total	103,625	93,281	98,937	100,447	103,736
Average	3,454167	3,109367	3,2979	3,3482333	3,457866667
Standart Deviation	0,237453	0,210852	0,22890047	0,2332192	0,239476614
Upper Bound	4,028	3,647687	3,85787398	3,9005507	3,969429143
Lower Bound	2,931	2,429313	2,71212602	2,7834493	2,791570857
Δ	0,049199	0,048532	0,04967463	0,049851	0,049565593

TableD.5: Processing Time of Number of Events in Milliseconds for 100 events



# of Events: 100					
	10 - 100	100 - 100	1000 - 100	2000 - 100	10000 - 100
Trial 1	2,156	1,947	2,679	3,321	3,402
Trial 2	2,252	2,058	2,717	2,926	2,716
Trial 3	1,985	2,212	2,99	2,799	2,859
Trial 4	2,058	2,173	3,201	2,804	2,999
Trial 5	1,989	1,922	2,668	3,11	3,581
Trial 6	2,229	1,85	2,583	3,242	3,099
Trial 7	1,855	1,714	3,125	2,934	2,898
Trial 8	1,906	2,205	2,93	3,201	3,324
Trial 9	2,167	2,186	3,188	3,178	3,259
Trial 10	1,952	1,856	2,903	3,378	3,208
Trial 11	2,022	2,005	2,956	3,214	3,354
Trial 12	1,874	2,227	2,758	3,045	3,125
Trial 13	1,957	2,169	2,994	2,946	3,138
Trial 14	1,977	2,14	2,716	2,947	3,258
Trial 15	1,882	1,954	2,691	2,743	3,357
Trial 16	1,966	1,874	2,7681	2,852	3,316
Trial 17	2,057	1,873	3,146	2,934	2,841
Trial 18	1,923	1,981	3,15	2,741	2,964
Trial 19	1,911	1,976	2,743	2,653	3,027
Trial 20	1,854	1,974	2,851	2,753	3,261
Trial 21	2,143	2,049	2,876	2,951	3,497
Trial 22	2,084	2,117	2,642	2,973	3,207
Trial 23	1,937	2,143	2,754	3,007	3,456
Trial 24	1,869	1,921	2,829	3,129	3,284
Trial 25	1,954	1,847	2,748	2,934	3,195
Trial 26	1,938	2,158	2,904	2,864	3,302
Trial 27	1,799	1,959	2,74	2,716	3,21
Trial 28	1,849	1,863	2,746	2,849	3,368
Trial 29	2,067	1,907	3,115	2,661	3,089
Trial 30	1,952	1,856	3,217	2,78	3,16
Total	59,564	60,116	86,3281	88,585	95,754
Average	1,985467	2,003867	2,8776033	2,9528333	3,1918
Standart Deviation	0,114678	0,137207	0,1865974	0,1927335	0,20013469
Upper Bound	2,252	2,312042	3,3326542	3,4974575	3,70504476
Lower Bound	1,799	1,628958	2,4673458	2,5335425	2,59195524
Δ	0,041337	0,049004	0,0464088	0,0467136	0,04487579

TableD.6: Processing Time of Number of Events in Milliseconds for 10 events



# of Events: 10					
	10--10	100 - 10	1000 - 10	2000 - 10	10000 - 10
Trial 1	2,157	1,896	2,291	2,169	3,14
Trial 2	2,239	2,228	2,052	2,458	2,816
Trial 3	1,978	2,052	2,139	2,2	2,889
Trial 4	2,202	2,339	2,201	2,774	2,641
Trial 5	1,841	1,861	1,95	2,973	3,004
Trial 6	2,229	1,962	2,148	2,621	3,31
Trial 7	1,906	1,956	1,962	2,851	2,494
Trial 8	1,899	2,17	2,63	2,169	3,265
Trial 9	2,167	2,494	2,188	2,502	2,748
Trial 10	1,967	1,898	2,333	2,427	3,052
Trial 11	2,241	2,056	2,352	2,516	2,849
Trial 12	2,06	2,204	2,047	2,608	2,993
Trial 13	1,957	1,967	2,225	2,638	3,176
Trial 14	1,978	2,106	2,118	2,558	3,206
Trial 15	2,175	1,937	2,117	2,474	3,008
Trial 16	2,057	2,008	2,284	2,51	2,911
Trial 17	1,933	2,118	2,304	2,499	3,123
Trial 18	2,057	2,261	2,009	2,587	2,934
Trial 19	1,972	2,188	1,985	2,545	2,879
Trial 20	2,018	2,049	2,212	2,489	3,384
Trial 21	2,166	1,954	2,267	2,567	3,099
Trial 22	2,064	2,149	2,248	2,522	3,127
Trial 23	1,947	1,875	2,006	2,481	2,879
Trial 24	2,043	2,004	1,997	2,544	2,907
Trial 25	2,146	2,183	2,236	2,509	3,206
Trial 26	1,875	2,014	2,142	2,49	2,96
Trial 27	2,018	2,111	2,195	2,576	2,853
Trial 28	2,124	2,031	2,155	2,404	2,968
Trial 29	2,055	2,144	2,2	2,529	3,008
Trial 30	2,164	1,994	2,194	2,583	3,206
Total	61,635	62,209	65,187	75,773	90,035
Average	2,0545	2,073633	2,1729	2,525767	3,0011667
Standart Deviation	0,113327	0,142876	0,140275	0,164108	0,1938574
Upper Bound	2,241	2,582555	2,716943	3,074715	3,504154
Lower Bound	1,841	1,772445	1,863057	2,067285	2,373846
Δ	0,039478	0,049312	0,046203	0,046501	0,0462293

APPENDIX E

TREND ANALYSIS STUDY FOR THE KOTAY INTERPRETERS

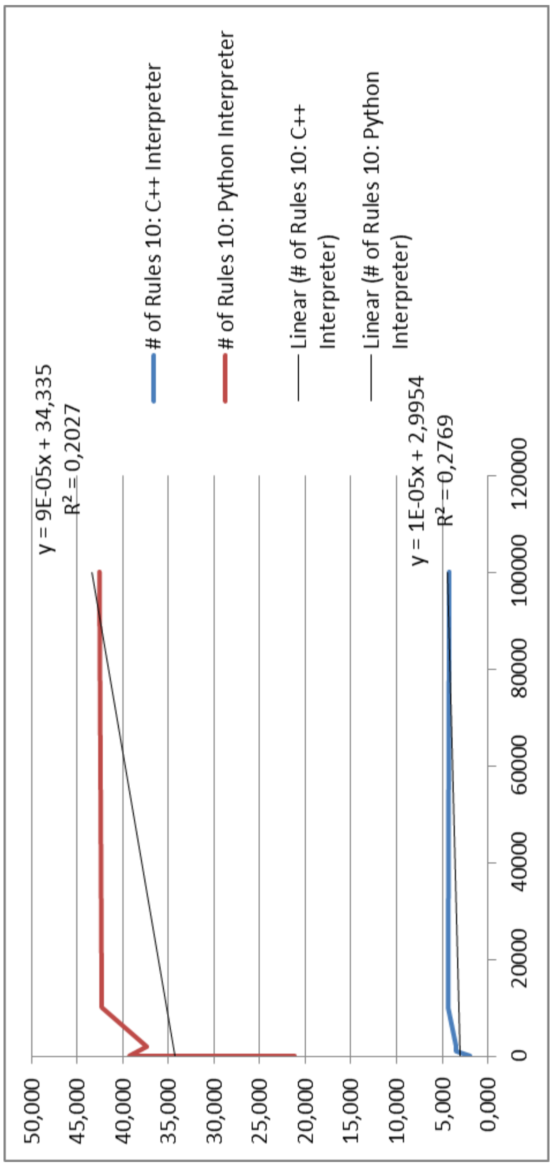


Figure E.1: Trend Analysis comparison of KOTAY Interpreters - Number of Rules - 10

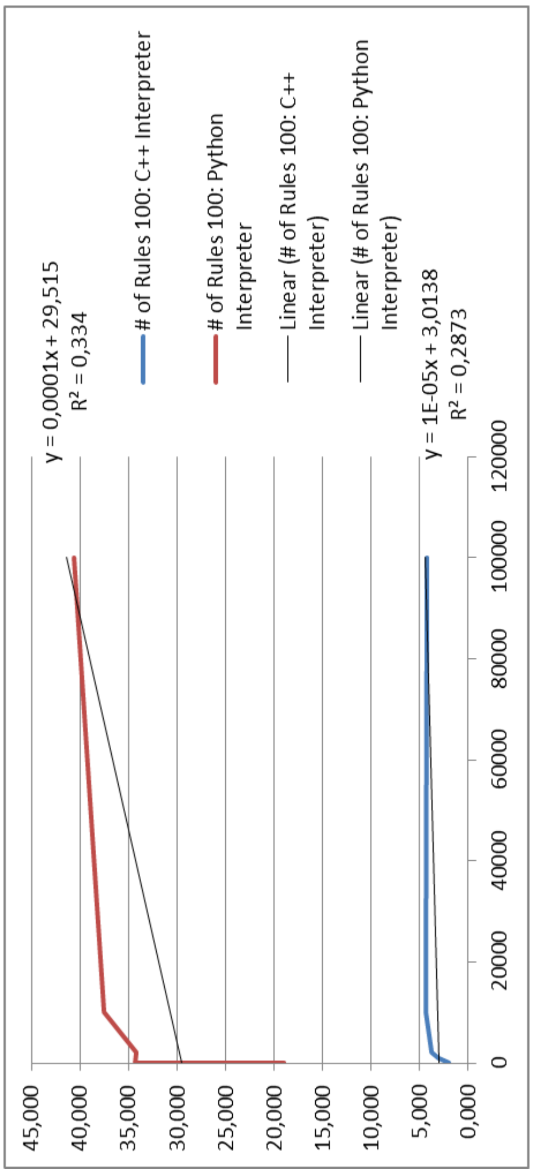


Figure E.2: Trend Analysis comparison of KOTAY Interpreters - Number of Rules - 100

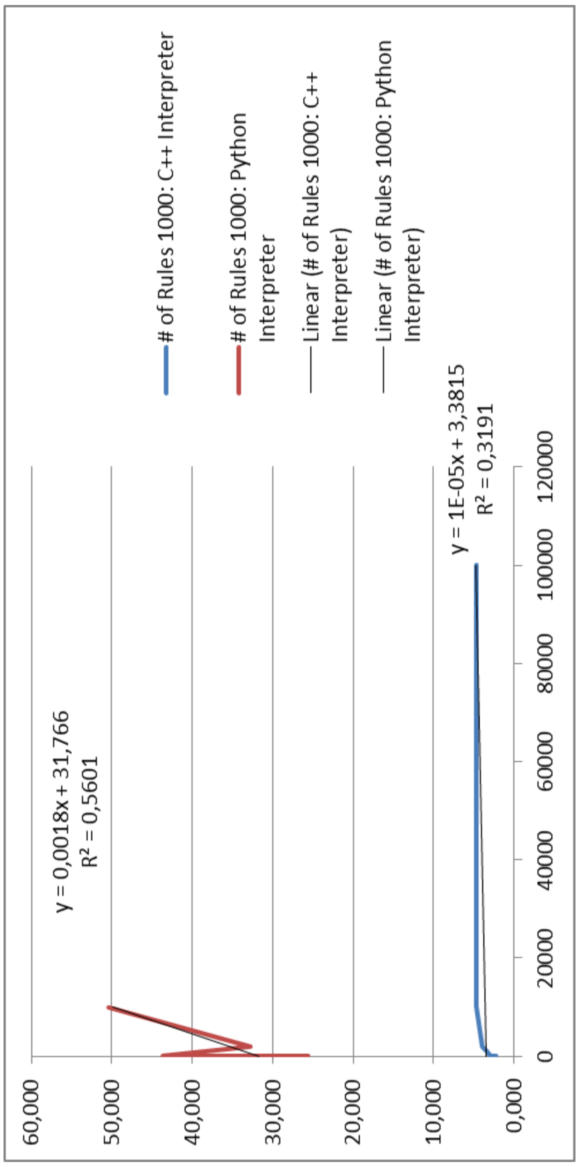


Figure E.3: Trend Analysis comparison of KOTAY Interpreters - Number of Rules: 1000

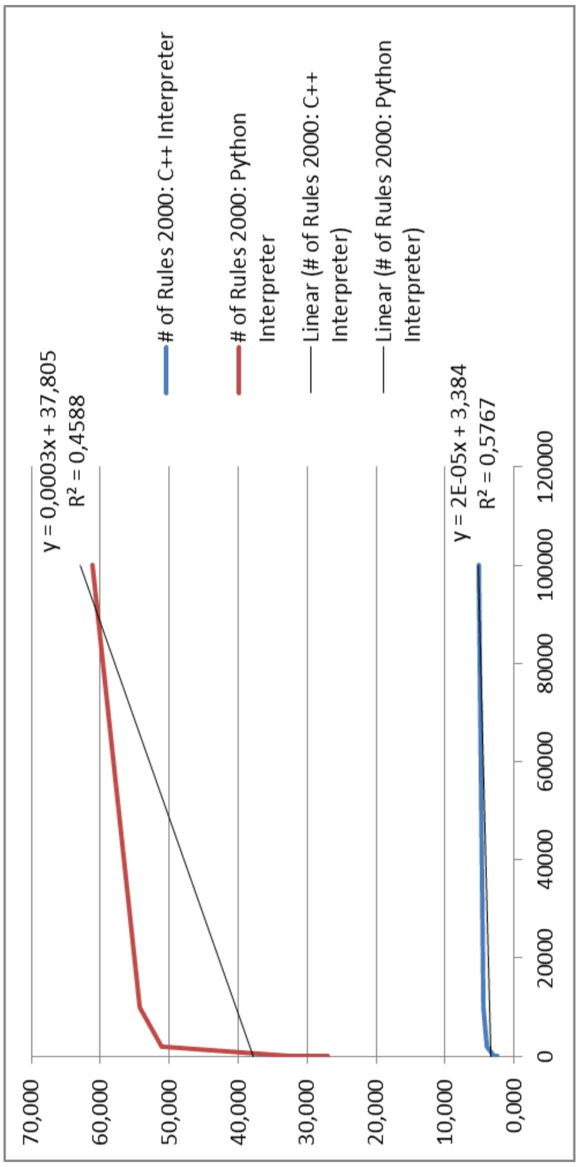


Figure E.4: Trend Analysis comparison of KOTAY Interpreters - Number of Rules: 2000

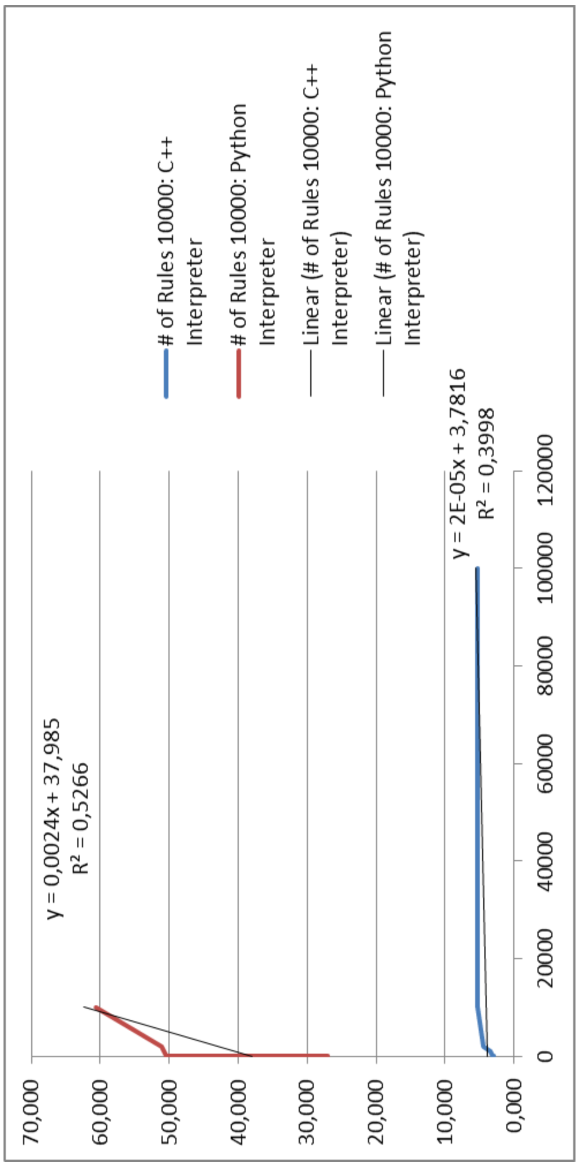


Figure E.5: Trend Analysis comparison of KOTAY Interpreters - Number of Rules: 10000