POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

INCREMENTAL REACHABILITY CHECKING OF KERNELC PROGRAMS USING MATCHING LOGIC

Relatore: Prof. Carlo Ghezzi
Correlatore: Ing. Domenico Bianculli
Ing. Antonio Filieri
Correlatore ASP: Prof. Marco Torchiano

Tesi di Laurea di:
Alessandro Maria Rizzi, matricola 783504

Anno Accademico 2012-2013

Ai miei genitori

Abstract

Verification is an important activity in the software development process.

Although testing can help in finding errors in programs, formal software verification techniques have proved to assure the development of applications that dependably satisfy their requirements. Nevertheless, these formal verification techniques are often characterized by high performance requirements, which negatively impact on the application of these techniques in the context of software that is continuously subjected to changes.

In this scenario, the possibility to reuse—when verifying a new version of an already verified program— the intermediate results related to the unchanged parts in the original program, could save time and resources; incremental verification could contribute to the effectiveness of the application of formal verification techniques.

In this work we explore the application of incremental verification in the context of reachability checking of KernelC programs using matching logic. KernelC is a subset of the C programming language that supports important features like the heap; matching logic consists of a language-independent proof system to reason about programs in any language that has a rewrite-based operational semantics.

We achieve incrementality by using a syntactic-semantic approach: the reachability checking procedure is encoded as semantic attributes of a grammar in operator-precedence form; this specific class of grammars guarantee the support for incremental parsing and hence incremental evaluation of semantic attributes.

Estratto (in italiano)

La verifica è una fase importante dello sviluppo del software.

Sebbene il testing possa trovare diversi errori nei programmi, le tecniche di verifica formale possono ragionevolmente assicurare che le applicazioni sviluppate soddisferanno i requisiti. Tuttavia, queste tecniche sono spesso caratterizzate dalla richiesta di alte prestazioni, che ne impediscono l'applicazione su software continuamente soggetto a cambiamenti.

In questo scenario, la possibilità di riutilizzare, nella verifica di una nuova versione di un programma già verificato, i risultati intermedi della parti immutate del programma originario, possono risparmiare tempo e risorse; la verifica incrementale può contribuire all'efficacia di tali tecniche formali.

In questo lavoro tratteremo l'applicazione di verifica incrementale nel contesto del controllo di raggiungibilità operato su programmi in linguaggio KernelC attraverso matching logic. KernelC è un sottoinsieme del linguaggio C che supporta importanti caratteristiche come la gestione della memoria; matching logic è un sistema di dimostrazione indipendente dal linguaggio che può trattare programmi scritti in qualunque linguaggio che abbia una semantica operazionale basata su riscritture. L'approccio utilizzato per ottenere l'incrementalità è di tipo sintattico-semantico: la verifica di raggiungibilità è espressa attraverso attributi semantici di una grammatica in forma a precedenza di operatori; questa particolare classe di grammatiche può operare in maniera incrementale e quindi consente una valutazione incrementale degli attributi semantici.

Contents

Abstract **I**

Contents **II**

List of Figures **V**

List of Tables **VI**

Listings **VII**

Ringraziamenti **X**

1 Introduction **1**
1.1 Motivations . . . 2
1.2 Contributions . . . 4
1.3 Thesis organization . . . 5

2 Background concepts **6**
2.1 Hoare logic . . . 6
2.1.1 Rules . . . 7
2.2 Matching logic . . . 8
2.2.1 Definition . . . 11
2.2.2 Reachability rules . . . 11
2.2.3 Matching logic proof system . . . 12
2.3 Operator-precedence grammars . . . 14
2.3.1 Context free grammars . . . 14
2.3.2 Operator-precedence grammars . . . 15
2.3.3 Properties of operator-precedence grammars . . . 17
2.4 Attribute grammars . . . 19

3 Related Work **21**
3.1 Incrementality from change encapsulation 21
3.2 Incrementality by change anticipation 22
3.3 Syntax-driven incrementality 23
3.4 Incremental model-checking 23

4 Reachability analysis of KernelC programs **25**
4.1 Syntactical part 25
4.2 Configuration term structure 26
4.3 Semantic part 27
4.4 Attribute evaluation example 43

5 Implementation **45**
5.1 Syntactic part 45
5.1.1 Grammar 46
5.1.2 Parser 47
5.1.3 Semantics 48
5.2 Kernel C semantic 48
5.3 Solver 49
5.3.1 Code 51
5.3.2 Formula 52
5.3.3 System operation 53

6 Validation **56**
6.1 KernelC specifications compliance 56
6.1.1 Error detection 57
6.1.2 Correctness of simple programs 58
6.1.3 Heap management 62
6.1.4 Call stack 69
6.2 Evaluation of the incremental approach 70
6.2.1 Scenario 1 70
6.2.2 Scenario 2 74
6.2.3 Scenario 3 78
6.2.4 Summing up 80

7 Conclusions **81**
7.1 Future work 82

Bibliography **83**

List of Figures

2.1 Example of operator grammar 16
2.2 Example of operator precedence matrix 16
2.3 Example of syntax tree 17
2.4 Example of attribute grammar 20

4.1 Example of KernelC attribute evaluation 43

5.1 UML Class diagram of grammar package 46
5.2 UML Class diagram of parser package 47
5.3 UML Class diagram of KernelC semantics package 51
5.4 UML Class diagram of solver package 52
5.5 UML Class diagram of config package 53
5.6 UML Class diagram of code package 54
5.7 UML Class diagram of rule package 55

6.1 Detail of KernelC syntax tree 71
6.2 Graphical representation of scenario 1 tests results 73
6.3 Graphical comparison of scenario 1 tests results 75
6.4 Graphical representation of scenario 2 tests results 76
6.5 Graphical representation of scenario 3 tests results 79

List of Tables

6.1 Results of scenario 1 tests (average of ten measures) 72
6.2 Comparison between scenario 1 results 74
6.3 Results of scenario 2 tests (average of ten measures) 77
6.4 Results of scenario 3 tests (average of ten measures) 78

Listings

4.1 code of KernelC attribute evaluation 43
5.1 Example of *malloc* definition 50
5.2 Example of *free* definition 50
6.1 Division by zero 57
6.2 Uninitialized variable 57
6.3 Unallocated location 58
6.4 Uninitialized memory 58
6.5 Average of three numbers 59
6.6 Minimum of two numbers 59
6.7 Maximum of two numbers 59
6.8 Multiplication performed as a series of additions 60
6.9 Sum of the first n numbers with recursion 61
6.10 Sum of the first n numbers with iteration 61
6.11 Verification of function properties 61
6.12 Retrieve first element of a list 62
6.13 Retrieve the tail of a list 63
6.14 Addition of a new element at the top of the list 64
6.15 Swapping of two values 65
6.16 Program which retrieves the length of a list 66
6.17 Program which retrieves the sum of values of a list 68
6.18 f can be called only from g 69
6.19 f can be called only if h is in the stack 69

Ringraziamenti

Desidero innanzitutto ringraziare la mia famiglia, che mi è stata vicina durante il percorso universitario. Vorrei ringraziare in particolar modo il mio relatore, Prof. Carlo Ghezzi, per aver seguito il mio lavoro di tesi con grande competenza e disponibilità. Ringrazio inoltre l'Ing. Domenico Bianculli e l'Ing. Antonio Filieri per il supporto offertomi durante questi mesi. Infine, un grazie speciale ai miei compagni di studi, ai colleghi dell'ASP e agli amici del POuL per i bei momenti vissuti durante questi anni.

Chapter 1

Introduction

> *"Incipe; dimidium facti est cœpisse. Supersit*
> *Dimidium: rursum hoc incipe, et efficies."*
>
> Decimus Magnus Ausonius

A fundamental phase in software development is the one which deals with verifying that software behaves correctly. Although accurate testing can discover many wrong behaviours, formal software verification techniques have proved to assure the development of applications that dependably satisfy their requirements.

However, since formal verification techniques are time consuming and software changes continuously, incremental verification methods, i.e., methods capable of reusing the previous results when verifying a new version of a program, are very useful, since they can significantly reduce the time required to perform the verification.

In this work we explore the application of incremental verification in the context of reachability checking of KernelC programs using matching logic. KernelC is a subset of the C programming language that supports important features like the heap; matching logic consists of a language-independent proof system to reason about programs in any language that has a rewrite-based operational semantics.

We achieve incrementality by using a syntactic-semantic approach: the reachability checking procedure is encoded as semantic attributes of a grammar in operator-precedence form; this specific class of grammars guarantee the support for incremental parsing and hence incremental evaluation of semantic attributes.

Based on the results of our evaluation, we can see that:

1. syntactic-semantic techniques are suitable for expressing reachability checking problems expressed through matching logic.
2. the use of incremental can greatly reduce the verification time.

1.1 Motivations

Software products are extremely subjected to defects. These are the result of different errors which can affect any phase of software development. First of all software specification is given in natural language, which is intrinsically ambiguous, and human errors are possible in requirement analysis. Moreover, errors can also be introduced in the development phase. Therefore it is very likely that software will not behave as expected.

There are different ways to deal with this. One of them is testing, which consists in providing a series of different inputs to the program and check the output with respect to an oracle.

However the limits of this practice lie in the impossibility of verifying every possible input. To overcome this it is possible to use more powerful techniques: formal verification methods. In this way the entire domain of the problem is analyzed guaranteeing that no deviations are present.

A peculiar feature of software is its dynamism. A software product is subjected to continuous changes. These changes do not only occur in the development phase, but also in the maintenance one. In fact it is quite common that software will continue to evolve to meet environmental changes.

Since formal verification methods are expensive from a computational time perspective and software is characterized by a series of different versions, an incremental approach can be a way to optimize this process.

Assume we have a program of which we have verified a version P_1 and now we want to verify a new version P_2, which will likely be very similar to P_1: we would like to exploit this fact by trying to verify only the part of P_2 affected by changes, reusing as much as possible the intermediate verification results obtained while verifying P_1.

In this way we can look at incrementality as a way for improving the speed of verification.

Verification & validation

In order to guarantee that a program works correctly different procedures can be used.

The terms "verification and validation" define the process of asserting that a software product meets its specification and that it performs what

it is intended for. These are complementary parts which have different purposes. According to [5] the former answers the question "Are we building the product right?" while the latter answers the question "Are we building the right product?".

The validation part has the task of asserting if the software requirements are correct. It has been defined in the Capability Maturity Model (CMM v1.1) [2] as "The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.".

The verification part instead deals with assuring that the program behaviour follows the requirements. According to CMM v1.1 [2] it is "The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.".

One way in which this process ca be achieved is by means of testing. This activity has been defined in [28] as "the process of executing a program with the intent of finding errors". In practice it consists in trying different inputs for a given program in order to find a fault, which is an incorrect behaviour of the program. However it is impossible for this method to guarantee that the program contains no flaws.

Instead the use of formal verification methods can assure that the software will behave correctly for every possible input. The common thing of this methods is the formalization of the domain of the problem (the program), which is translated into an abstract mathematical model and given the properties of such model it is possible to derive formal proof of the requirements.

These different verification techniques can be divided in two big categories: deductive verification and model checking. The former is based on deriving a proof of correctness from the system specifications, which implies that the desired properties are satisfied.

The latter involves the complete exploration of the mathematical model. In practice, the full state space of the domain of the problem is analyzed in order to find a possible state which violates the properties desired for the software. This last method is the one adopted in this work: we analyze the space of the possible internal states of a program and the system of transitions between them in order to find a path admitted by the program which brings us in a state where a given property is violated; if we do not find such a state, it means the program is correct with respect to the specification.

1.2 Contributions

The main contribution of this work is the adaptation of the reachability checking methods based on matching logic in order to support incrementality. Matching logic is a general framework to perform program verification of a generic language. It is designed to be extremely flexible (it can easily support complex structure like an heap) and easily extensible (it is possible to add new constructs without modifying the existing one) [33]. The key concept for Matching logic is semantic definition of a programming language: the definition for each construct of the selected programming language of a mapping between the program states before that construct and the program states after it. More precisely they are given as a mapping between different set of states (or pattern). An actual state is said to match a pattern if it belongs to the set of states defined by it. The reasons for its choice are its great potential to manage complex programming languages (which requires to deal with side effects and manage complex structure like heap).

This work also demonstrates the power of incremental syntactic-semantic methods in term of performance obtained. In this work we obtain incrementality through the use of formal grammars. The two formalisms are operator-precedence grammars and attribute grammars. The formers are used for their peculiar properties (in particular the locality one) which allow a syntactic incrementality: which is only the changed parts of the program are re-parsed. The latter instead are responsible for the semantic incrementality: the verification problem is solved through an evaluation of attributes. Attribute grammars allow to only re-compute those attributes related to the modified parts of the program. Of course semantic incrementality cannot be achieved without syntactic incrementality, since syntactical structure of the input program drives the attribute evaluation. The advantages of this approach are the fact that it is totally automatic: parsing can be restricted only to the changed parts.

In this work we support the KernelC language, mostly because there is a non-incremental version of the reachability checking algorithm, to which it is possible to compare our implementation.

KernelC is a nontrivial subset of the C language. It has the if-else construct, while loop, memory support, all operators of C. The only type allowed is the integer one (it is not possible to use floating-point numbers, strings, enums), but it is possible to define and allocate structs.

However our implementation is quite different from the non-incremental previous one. This work has been implemented in Java and uses the external Satisfiability Modulo Theories (SMT) solver Z3. Instead the available

implementation is composed by a parser written in Java that translates the input program into an intermediate representation, which is passed to the actual verifier written in Maude [7].

Since the tools we have used are quite different from the ones utilized by the previous implementation, some changes were required. In particular we changed the format of annotations, which are the parts of the input which specify the properties to check.

The syntactic-semantic incrementality is achieved using an S-attributed grammar built upon an operator-precedence grammar.

We have evaluated our approach on a set of KernelC programs, contained in the distribution of the match program (cited in [30]). The results show the feasibility of applying a syntactic-semantic approach, which can reduce the time required by the verification process.

1.3 Thesis organization

The thesis is organized as follows. In chapter 2 we present the background concepts regarding formal methods for software verification and formal grammars. In chapter 3 we recall the state of the art on incremental software verification methods. After that, in chapter 4, it is presented the theoretical idea behind this work, namely the formal grammars developed. In chapter 5 we explain our actual implementation, while in chapter 6 we present the results of the evaluation, which compares our implementation with the non-incremental one. Finally, chapter 7 concludes this document and outline future research directions.

Chapter 2

Background concepts

> *"A semantic definition of a particular set of command types, then, is a rule for constructing, for any command of one of these types, a verification condition on the antecedents and consequents."*
>
> Robert Floyd

In this chapter we present the concepts upon which this work is based. First of all we present the formal systems for reasoning about the correctness of computer programs. We start from the work of C. A. R. Hoare which, although quite old, contains the foundations of formal verification of computer programs, and then we move to the one which is specifically used in this thesis, namely the development of matching logic. Afterwards, we present the other concepts which allow to develop an incremental approach: operator-precedence grammars, which have interesting properties for developing an incremental parser, and attribute grammars, which define the structure the work should have.

2.1 Hoare logic

Hoare logic is the first attempt to formalize the concept of correctness of a computer program. It has been presented in 1969 by C. A. R. Hoare in [21]. The main idea in Hoare's work is the introduction of the Hoare triple, which describes how an instruction changes the machine state.

$$\{P\}C\{Q\} \tag{2.1}$$

The three components of the triple are the precondition (P), the code (C) and the postcondition (Q). The precondition is a first-order-logic (FOL)

predicate which asserts what is guarantee to be true before the instruction; the code is the actual instruction considered and the postcondition is another FOL predicate which holds all the properties which are valid after the execution of the instruction.

The triple can be read in this way: if the precondition is true and the instruction is executed then the postcondition holds.

2.1.1 Rules

In addition to the general framework, Hoare provided also several rules which describe the behaviour of a simple programming language, by specifying how its basic blocks behave. The general form of the rule is an implication (as shown in 2.2). In this case the meaning of the formula is that if A is true than B holds.

$$\frac{A}{B} \tag{2.2}$$

Assignment rule

$$\frac{}{\{P[E/x]\}\ x = E\ \{P\}} \tag{2.3}$$

This rule specifies the behaviour of an assignment statement where E can be a given expression. It states that everything that was true for the right hand side of the assignment is now true also for the left hand side. This is achieved by specifying the precondition as the substitution of the free occurrences of x with the expression E.

Skip rule

$$\frac{}{\{P\}\ \textbf{skip}\ \{P\}} \tag{2.4}$$

This rule simply states that after an empty statement nothing changes.

Composition

$$\frac{\{P\}\ S\ \{Q\}\ ,\ \{Q\}\ T\ \{R\}}{\{P\}\ S;T\ \{R\}} \tag{2.5}$$

This rule simply states that different instructions can be combined together if the postcondition of the first is equal to the precondition of the second.

Conditional

$$\frac{\{B \wedge P\} \quad S \quad \{Q\}\,, \qquad \{\neg B \wedge P\} \quad T \quad \{Q\}}{\{P\} \quad \textbf{if(}\ B\ \textbf{)\{}\ S\ \textbf{\}else\{}\ T\ \textbf{\}} \quad \{Q\}} \tag{2.6}$$

This rule describes the behaviour of a conditional construct. It has the effects of splitting the machine state in two parts: one in which the condition is true and so the `then` branch is taken and another one in which the condition is false and the `else` branch is followed instead.

Loop

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \textbf{while(}\ B\ \textbf{)\{}\ S\ \textbf{\}}\ \{\neg B \wedge I\}} \tag{2.7}$$

The loop is a tricky construct, because it can theoretically be taken an infinite number of times. So to be treated it requires the definition of a special first-order-logic predicate: the *invariant*. This formula embodies what remains true after every iteration of the loop, allowing to summarize the whole loop behaviour in a single iteration. It is important to emphasize that this system is only able to give partial correctness verification, which is it cannot guarantee the termination of the program. A more sophisticated rule can guarantee also the total correctness which is the fact that the code will always terminate.

Weakening

$$\frac{R \Rightarrow P, \{P\}\ S\ \{Q\}}{\{R\}\ S\ \{Q\}} \tag{2.8}$$

$$\frac{\{P\}\ S\ \{Q\}\ , Q \Rightarrow R}{\{P\}\ S\ \{R\}} \tag{2.9}$$

These rules adapt another rule to a stronger precondition or to a weaker postcondition.

2.2 Matching logic

Hoare logic, although providing a general system for correctness verification, has several drawbacks. One of them is being excessively abstract, i.e., it does not consider how the computations are actually performed, only focusing on high-level aspects of the language. Indeed it assumes that the formalization of programming language constructs is provided. This fact can be problematic if the formalization is not explicitly defined, because the meaning of language constructs can be not so clear and ambiguous situations

are possible. For example in C language an expression like this "`i-i++`", where '`i`' is an integer variable, is problematic: according to C language specifications, although it is a legal C expression, its result is undefined. To avoid such problems a formal semantics of the language is needed. Starting from these criticisms G. Roşu and others develop a formal, syntax-oriented compositional proof system: Matching logic ([33],[30],[31],[32]).

In order to introduce Matching logic, we present a general way of modeling computer programs known as Kripke structure.

A *Kripke structure* is a tuple $K = \langle S, I, \delta, AP, L \rangle$ where:

- S is a finite set of states.
- $I \subseteq S$ is set of initial states.
- $\delta : S \to S$ is a transition relation.
- AP is a set of atomic propositions.
- $L : S \to 2^{AP}$ is a labeling function, which maps each state in S to a set of atomic propositions.

We can define a set of terminal states $T = \{s \in S | \nexists s' \in S : (s, s') \in \delta\}$

Moreover we can evaluate a FOL formula f in a state $s \in S$ if $AP_f \subseteq AP$ where AP_f is the set of the atomic propositions in f.

Given that for each $a \in AP$ and $s \in S$ $s \models a$ iff $a \in L(s)$ and $s \models \neg a$ iff $a \notin L(s)$.

Having a truth value for each atomic proposition in f in state s we can apply the FOL logic rules to give a truth value to f in s.

We can think of a computer program as a finite set of states S which represent each possible machine configuration and each instruction performs a transition from one state to another. Every state has some properties associated to it, represented by the elements of AP. We can identify a subset I of these states which are the possible initial configurations, or the possible machine configuration present when the program starts and, on the other hand, a set of states T which represent all the possible final configurations, or all the machine configurations where the program can end.

In this structure Hoare logic can be expressed as follows: if we have a precondition P and a postcondition Q such that all the atomic propositions of P and Q are contained in AP, we consider the set $R = \{s \in S \mid s \models P\}$ and a set $U = \{s \in S \mid s \models Q\}$. Stating $I \subseteq R$ defines a precondition, whereas $T \subseteq U$ is defines to a postcondition.

Matching logic is just a particular case of this general model. The state (called configuration) defines a sort of first order logic with equality terms.

Moreover this sort is formed by a set of FOL with equality terms of different sorts which can have this form again. This general form can be adapted to different programming languages. In facts the subdivisions of the configuration in sub-terms make it compositional: in the sense that they are not atomic but constituted by different sub-configurations. This fact allows to easily extend the composition of the configurations with the addition of other symbols to extend the given language if some additive axioms to describe their behaviour are provided. For example in the language IMP (also shown in [33]) the configuration term is a set of two structures: one named k which contains the list of instructions to be executed and another one named *env* which is a map from a string to an integer which represents the defined variables with the value they carry. An example of configuration is shown in formula 2.10 which means that the instruction to be executed is an assignment of the value 2 to `x` and `x` has actually a value of three.

$$\langle \langle \texttt{x:=2} \rangle_k \, \langle \texttt{x} \mapsto 3 \rangle_{env} \rangle \tag{2.10}$$

In Matching logic the program instructions to be executed are part of the configuration so we define a language-dependent (partial) function which maps a configuration to another; moreover there is not in principle any distinction between the program instruction part of the configuration and the rest.

In order to easily encode all possible transitions, they are defined for a set of configurations called *configuration pattern*. This is achieved by using first order logic with equality formulae that can contain free and bound variables.

An example of a pattern which is matched by the above configuration is the following:

$$\exists a, \rho((\square = \langle \langle \texttt{x:=2} \rangle_k \, \langle \texttt{x} \mapsto a, \rho \rangle_{env} \rangle) \wedge a \geq 0) \tag{2.11}$$

In this configuration pattern we have a which is a generic integer, `x` which expresses the variable name, $\square$ is a special term which defines the considered configuration and ρ which represent the rest of the content of *env*, i.e., other variable maps.

This configuration pattern matches every configuration which have k corresponding to the statement "`x:=2`" and *env* which contains, besides (eventually) other variables, a variable named `x` with associated an integer non negative value.

2.2.1 Definition

We summarize here the formal definition of Matching logic present in [33]. We want to define a matching logic proof system for the generic language $\mathcal{L}$. First of all we define the first order logic with equality specification of our language as $\mathcal{L} = (S_{\mathcal{L}}, \Sigma_{\mathcal{L}}, \mathcal{F}_{\mathcal{L}})$ where $S_{\mathcal{L}}$ is the set of the sorts of the language, $\Sigma_{\mathcal{L}}$ is the set of operations of the language over the sort in $S_{\mathcal{L}}$ (which describe the syntactical features of the language) and $\mathcal{F}_{\mathcal{L}}$ is a set of formulae which describes the semantic features of the language along with the useful properties of our operations. Let us consider a given model $\mathcal{T}_{\mathcal{L}}$ of $\mathcal{L}$. We consider a particolar sort in $S_{\mathcal{L}}$ which is *Cfg* and holds the configurations. We denote with *Var* a sortwise infinite set of variables and $\square$ a variable of sort *Cfg* such that $\square \notin Var$.

Definition 1 Configuration patterns *(or patterns) are FOL with equality formulae over* $\Sigma_{\mathcal{L}}$ *which have the form:* $\exists X.((\square = c) \wedge \varphi$.

- $X \subset Var$ *are the (pattern) bound variables; the remaining ones are the (pattern) free variables;* $\square$ *appears exactly once per pattern.*
- c *is the pattern structure; a term of sort Cfg.*
- φ *is the (pattern) constraint, a generic FOL with equality formula.*

Definition 2 *A* valuation $(\gamma, \tau) : Var \cup \square \rightarrow \mathcal{T}_{\mathcal{L}}$ *is a function where* γ *is a configuration which corresponds to* $\square$ *and* τ *maps Var to* $\mathcal{T}_{\mathcal{L}}$.

Definition 3 *Configuration* γ matches *pattern* $\Phi = \exists X.((\square = c) \wedge \varphi$ *iff exists a* $\tau : Var \rightarrow \mathcal{T}_{\mathcal{L}}$ *such that* $(\gamma, \tau) \models \Phi$ *that is* $\models \Phi(\gamma, \tau)$.

Definition 4 *A pattern* Φ *is* weakly well-defined *iff for any* $\tau : Var \rightarrow \mathcal{T}_{\mathcal{L}}$ *there is some configuration* $\gamma \in Cfg$ *such that* $(\gamma, \tau) \models \Phi$*; iff* γ *is unique it is well-defined.*

2.2.2 Reachability rules

In this section we formally define the reachability concept in Matching logic as stated in [30].

Definition 5 *A* reachability rule *is a pair of patterns* $\varphi \Rightarrow \varphi'$.

Definition 6 *A* reachability system S *is a set of reachability rules.*

Definition 7 *A* transition system $(\mathcal{T}_\mathcal{L}, \Rightarrow_\mathcal{S}^{\mathcal{T}_\mathcal{L}})$ *is inducted by* $\mathcal{S}$ *on* $\mathcal{T}_\mathcal{L}$*.* $\gamma \Rightarrow_\mathcal{S}^{\mathcal{T}_\mathcal{L}} \gamma'$ *for* $\gamma, \gamma' \in Cfg$ *iff exists a reachability rule* $\varphi \Rightarrow \varphi'$ *in* $\mathcal{S}$ *and a function* $\tau : Var \to \mathcal{T}_\mathcal{L}$ *such that* $(\gamma, \tau) \models \varphi$ *and* $(\gamma', \tau) \models \varphi'$ *.*

Definition 8 *A configuration* $\gamma \in Cfg$ *is said to* terminate *in* $(\mathcal{T}_\mathcal{L}, \Rightarrow_\mathcal{S}^{\mathcal{T}_\mathcal{L}})$ *iff in the transition system there is not an infinite sequence starting with* γ*.*

Definition 9 *A reachability rule* $\varphi \Rightarrow \varphi'$ *is* (weakly) well-defined *iff* φ' *are (weakly) well-defined.*

Definition 10 *A reachability system* $\mathcal{S}$ *is* (weakly) well-defined *iff each rule is (weakly) well-defined.*

Definition 11 $\mathcal{S} \models \varphi \Rightarrow \varphi'$ *where* $\mathcal{S}$ *is a reachability system and* $\varphi \Rightarrow \varphi'$ *a reachability rule iff for all* $\gamma \in Cfg$ *such that* γ *terminates in* $(\mathcal{T}_\mathcal{L}, \Rightarrow_\mathcal{S}^{\mathcal{T}_\mathcal{L}})$ *and for all function* $\tau : Var \to \mathcal{T}_\mathcal{L}$ *such that* $(\gamma, \tau) \models \varphi$ *there is some* γ' *such that* $(\gamma', \tau) \models \varphi'$

2.2.3 Matching logic proof system

The matching logic proof system (as presented in [30]) is composed by eight general proof rules which, together with the language-specific ones, form the proof system. The general form of the derivation is shown in 2.12, where $\mathcal{A}$ and $\mathcal{C}$ are two sets of rules.

$$\mathcal{A} \vdash_\mathcal{C} \varphi \Rightarrow \varphi' \tag{2.12}$$

The form 2.13 means that the set $\mathcal{C}$ is empty.

$$\mathcal{A} \vdash \varphi \Rightarrow \varphi' \tag{2.13}$$

At the beginning the set $\mathcal{A}$ contains all the language-specific rules, i.e., the operational semantics of the language, while the set $\mathcal{C}$ is empty. $\mathcal{A}$ has the role of containing all the sound derivations while $\mathcal{C}$ (called circularity set) contains some derivations which are not proven yet. This is useful while verifying situations like a loop through invariant or a recursive function call. For example in order to verify the correctness of a recursive function we must follow the first call and then assume that the function itself would be correct and substitute the subsequent call with what we want to prove: otherwise we undergo into an infinite recursion process.

This proof system is sound and relative complete for any weakly well-defined reachability system $\mathcal{S}$ [30]. The completeness is relative to an "oracle" to check first order logic validity of configuration model.

Axiom

$$\frac{\varphi \Rightarrow \varphi' \in \mathcal{A}}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'} \tag{2.14}$$

This rule means that if a rule is contained in the set $\mathcal{A}$ is derived by the system.

Reflexivity

$$\frac{\cdot}{\mathcal{A} \vdash \varphi \Rightarrow \varphi} \tag{2.15}$$

This rule add the reflexivity property to rules in $\mathcal{A}$. It is important to underline that $\mathcal{C}$ must be empty in this case.

Transitivity

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow^+ \varphi_2 \;, \quad \mathcal{A} \cup \mathcal{C} \vdash \varphi_2 \Rightarrow \varphi_3}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_3} \tag{2.16}$$

This rule applies transitivity to configurations. The symbol $\Rightarrow^+$ means that at least one step has to be provided. This fact, together with the reflexive rule, has the task to make sure that the first step in derivation is sound.

Logic framing

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \;, \quad \psi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \wedge \psi \Rightarrow \varphi' \wedge \psi} \tag{2.17}$$

This rule, where ϕ is a patternless first order logic with equality formula, guarantees that more logical constraints can be added if they are not structural (patternless).

Consequence

$$\frac{\models \varphi_1 \rightarrow \varphi_1' \;, \quad \mathcal{A} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \;, \quad \models \varphi_2' \rightarrow \varphi_2}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2} \tag{2.18}$$

This rule is analogous to the Hoare's weaking rule.

Case analysis

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \;, \quad \mathcal{A} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{A} \vdash_{\mathcal{C}} \varphi_1 \vee \varphi_2 \Rightarrow \varphi} \tag{2.19}$$

Also this rule is equivalent to Hoare's one: the conditional rule.

Abstraction

$$\frac{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' , \quad X \cap FreeVars(\varphi') = \emptyset}{\mathcal{A} \vdash_{\mathcal{C}} \exists X \varphi \Rightarrow \varphi'} \tag{2.20}$$

This rule is also present in Hoare logic but, while being derivable in Hoare logic, in Matching logic has to be added as axioms.

Circularity

$$\frac{\mathcal{A} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{A} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'} \tag{2.21}$$

This rule has the role of adding a claim relative to a circular behaviour. The other rules guarantee that that claim has to be used after one sound step.

2.3 Operator-precedence grammars

2.3.1 Context free grammars

In this section we summarize context free grammars providing definition for productions, immediate derivation, derivation language and empty string which will be used in the presentation of Operator-precedence grammars.

Definition 1 *A* context-free grammar *(CFG) is a tuple* $G = \langle \mathcal{V}, \Sigma, P, S \rangle$ *where:*

- $\mathcal{V}$ *is a finite set of non-terminal symbols.*
- Σ *is a finite set of terminal symbols such that* $\mathcal{V} \cap \Sigma = \emptyset$*.*
- $P : \mathcal{V} \to (\mathcal{V} \cup \Sigma)^*$ *(where* $*$ *is the Kleene star operator) is a finite relation whose member are called productions.*
- $S \in \mathcal{V}$ *is the start symbol (or axiom).*

Definition 2 *A* production *(or rule) is a pair* $(\alpha, \beta) \in P$ *where* $\alpha \in \mathcal{V}$ *and* $\beta \in (\mathcal{V} \cup \Sigma)^*$*.*

Definition 3 $u \Rightarrow v$ *is an* immediate derivation *where* $u, v \in (\mathcal{V} \cup \Sigma)^*$ *iff exists* $(\alpha, \beta) \in P$ *and* $u_1, u_2 \in (\mathcal{V} \cup \Sigma)^*$ *such that* $u = u_1 \alpha u_2$ *and* $v = u_1 \beta u_2$

We can define an immediate derivation relation $R : (\mathcal{V} \cup \Sigma)^* \to (\mathcal{V} \cup \Sigma)^*$ where $(u, v) \in R$ iff $u \Rightarrow v$.

Definition 4 $u \Rightarrow^* v$ *is a* derivation *iff* $(u, v) \in R^*$ *where* R^* *is the reflexive transitive closure of* R*.*

Definition 5 *The* language *of a grammar* $G = \langle \mathcal{V}, \Sigma, P, S \rangle$ *is the set* $L(G) = \{u \in (\mathcal{V} \cup \Sigma)^* | S \Rightarrow^* u\}$.

Definition 6 ε *is the* empty string*, which is the string that does not contain any symbol.*

Definition 7 *A* derivation tree *(or parse tree) of a string* $w \in \Sigma^*$ *generated by a context-free grammar* $G = \langle \mathcal{V}, \Sigma, P, S \rangle$ *is a tree* T *where:*

- *each node of* T *is labeled by a symbol* $a \in \mathcal{V} \cup \Sigma \cup \varepsilon$.
- *the label of the root is* S.
- *if a node labeled by* a *has the children labeled by* $a_1, ..., a_n$ *then* $(a, a_1, ..., a_n) \in P$.
- w *is equal to the labels of the leaves of* T *concatenated from left to right.*

2.3.2 Operator-precedence grammars

In this section we formally define operator-precedence grammars.

Definition 8 *A rule* $(\alpha, \beta) \in P$ *is in* operator form *iff there are not* $u, v \in (\mathcal{V} \cup \Sigma)^*$, $A, B \in \mathcal{V}$ *such that* $\beta = uABv$ *(that is the right hand side of the rule does not contain two adjacent non-terminals).*

Definition 9 *A grammar* G *is an* operator grammar *(OG) if it is composed only by rules in operator form.*

Definition 10 *The* left terminal set *of a non-terminal* A *of an operator grammar* G *is:* $\mathcal{L}_G(A) = \{a \in \Sigma | A \Rightarrow^* Bau\}$ *where* $B \in \mathcal{V} \cup \{\varepsilon\}$ *and* $u \in (\mathcal{V} \cup \Sigma)^*$.

Definition 11 *The* right terminal set *of a non-terminal* A *of an operator grammar* G *is:* $\mathcal{R}_G(A) = \{a \in \Sigma | A \Rightarrow^* uaB\}$ *where* $B \in \mathcal{V} \cup \{\varepsilon\}$ *and* $u \in (\mathcal{V} \cup \Sigma)^*$.

Definition 12 *Given an operator grammar* $G = \langle \mathcal{V}, \Sigma, P, S \rangle$, $a, b \in \Sigma$, $u, v \in (\mathcal{V} \cup \Sigma)^*$ *we say that:*

- a *and* b *have* equal precedence $(a \doteq b)$ *iff exists* A *such that* $(A, uaBbv) \in P$ *where* $B \in \mathcal{V} \cup \{\varepsilon\}$.

Figure 2.1: Example of operator grammar

$\langle S \rangle$ ::= $\langle A \rangle$ | $\langle B \rangle$
$\langle A \rangle$ ::= $\langle A \rangle$ '+' $\langle B \rangle$ | $\langle B \rangle$ '+' $\langle B \rangle$
$\langle B \rangle$ ::= $\langle B \rangle$ '*' 'n' | 'n'

Figure 2.2: Example of operator precedence matrix

	'n'	'*'	'+'
'n'		$\gtrdot$	$\gtrdot$
'*'	$\doteq$		
'+'	$\lessdot$	$\lessdot$	$\gtrdot$

- a takes precedence *over* b $(a \gtrdot b)$ *iff exists* A *such that* $(A, uDbv) \in P$ *and* $a \in \mathcal{R}_G(D)$ *where* $D \in \mathcal{V}$.

- a yields precedence *to* b $(a \lessdot b)$ *iff exists* A *such that* $(A, uaDv) \in P$ *and* $b \in \mathcal{L}_G(D)$ *where* $D \in \mathcal{V}$.

Definition 13 *An* operator precedence matrix *(OPM)* M *of a grammar* G $(M = OPM(G))$ *is a relation* $\Sigma \times \Sigma \to \{\doteq, \gtrdot, \lessdot\}$ *which associates to each pair (a,b) with* $a, b \in \Sigma$ *the set* $m_{a,b}$ *of their operator precedence relations.*

Definition 14 *An* operator grammar G *is an operator-precedence grammar iff its OPM is conflict-free, that is for each pair* (a, b) *with* $a, b \in \Sigma$, $|m_{a,b}| \leq 1$.

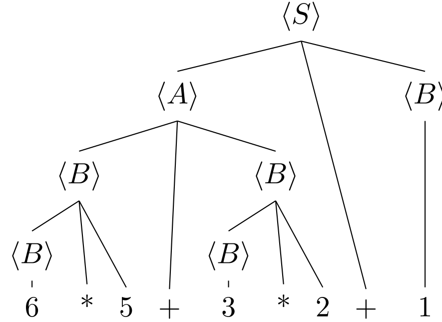
An example of operator precedence grammar is shown in Figure 2.1, which defines expression composed of summations and products of natural numbers. Its non-terminals are $\langle A \rangle$,$\langle B \rangle$ and $\langle S \rangle$; $\langle S \rangle$ is the axiom. Its terminals are '*','+' and 'n' which denotes any natural number. Its operator precedence matrix is shown in Figure 2.2.

The following definition is not specific to operator-precedence grammars. It is presented here because it has been applied to operator-precedence grammars in this work in order to simplify the parser construction.

Definition 15 *A grammar* $G = \langle \mathcal{V}, \Sigma, P, S \rangle$ *is in* Fisher normal form *iff all the following statements hold:*

- *for all* $A, B \in \mathcal{V}$ *if* $(A, u) \in P$ *and* $(B, u) \in P$ *where* $u \in (\mathcal{V} \cup \Sigma)^*$ *then* $A = B$.

Figure 2.3: Example of syntax tree



- *for all* $A \in \mathcal{V}$ *if* $(A, D) \in P$ *and* $D \in \mathcal{V}$ *then* $A = S$.
- *for all* $A \in \mathcal{V}$ *if* $(A, \varepsilon) \in P$ *then* $A = S$.

2.3.3 Properties of operator-precedence grammars

In this section we present the important properties which characterize operator-precedence grammars; many of them are described in [11].

The first feature deals with how the parsing is conducted. Assume the input string being implicit delimited by a new terminal symbol '\$' such that for all $a \in \Sigma$ $m_{\$,a} = \{\lessdot\}$ and $m_{a,\$} = \{\gtrdot\}$ first. Then let us extend the input string by inserting between every pair of terminals their operator precedence. In such string, every sequence of terminals enclosed by a $\lessdot$ and a $\gtrdot$ uniquely determines a right hand side to be substituted. This fact is also valid if some reductions have been performed on the input string: it is sufficient to ignore the non-terminals inside the string and compute the precedence between terminals as usual: again the sub-string enclosed by a $\lessdot$ and a $\gtrdot$ (eventually extended with surrounding non-terminals) determines the right hand side to be applied.

An example related to the grammar presented in 2.1 is shown in 2.3. If we consider the sub-string '`3*2`' we find that it is derived from a non-terminal $\langle B \rangle$, we can note that $m_{+,3} = \{\lessdot\}$ and $m_{2,+} = \{\gtrdot\}$. Again if we consider the sub-string '`6*5+3*2`', derived from non-terminal $\langle A \rangle$, we find $m_{\$,+} = \{\lessdot\}$ and $m_{+,+} = \{\gtrdot\}$, since we have to consider the next derivation and ignoring the non-terminal $\langle B \rangle$.

Another important property in the scope of this work is the *locality property*, which is the base of the syntactic incrementality and the reason for which we have chosen this formalism to build this work upon. The locality property states the following:

Proposition 1 *Given* $A \in \mathcal{V}$*,* $a, b \in \Sigma$ *and* $u, v, w \in \Sigma^*$ *if* $aAb \Rightarrow^* awb$ *then* $S \Rightarrow^* uawbv$ *iff* $S \Rightarrow^* uaAbv \Rightarrow^* uawbv$*.*

The above two properties allows to easily develop an incremental parser. Suppose we have parsed a string $s = uawbv$ and then we have to parse a string $s' = uaw'bv$. We start to parse the minimum context (a sequence of terminals enclosed by a $\lessdot$ and a $\gtrdot$) which contains all the symbols changed. We proceed to parse that string extending it until we find a derivation $aAb \Rightarrow^* aw'b$. For the Proposition 1 we can stop the parsing process; we say that a *matching-condition* with the previous derivation $aAb \Rightarrow^* awb$ is satisfied. Now it is sufficient to substitute the derivation $A \Rightarrow^* w$ with the new one $A \Rightarrow^* w'$. This fact guarantees a syntactic incrementality: only a small part of the input has to be reparsed after an edit.

For example if we replace '`3*2`' by '`4`' in the example in Figure 2.3, we only have to parse '`4`' since $m_{\texttt{+},\texttt{4}} = \{\lessdot\}$, $m_{\texttt{4},\texttt{+}} = \{\gtrdot\}$ and the matching condition is satisfied, because the non-terminal $\langle B \rangle$ generates both '`4`' and '`3*2`'.

2.4 Attribute grammars

Attribute grammars are a formalism first presented by Knuth in [24] to assign a "meaning" to a string recognised by a Context Free Grammar. They extend context-free grammars by adding attributes to symbols and semantic functions to productions.

Definition 1 *An* attribute grammar *(AG) is a tuple* $AG = \langle G, A, R \rangle$ *where:*

- G *is a context-free grammar.*
- A *is a finite set of attributes.*
- R *is a finite set of semantic rules.*

A finite set of *attributes* $A(k)$ is associated to every symbol of the grammar $k \in \Sigma \cup \mathcal{V}$. $A(k)$ is partitioned into two subsets: the *inherited attributes* $I(k)$ and the *synthesized attributes* $S(k)$ such that $I(k) \cup S(k) = \emptyset$.

$$A = \bigcup_{k \in \Sigma \cup \mathcal{V}} A(k) \tag{2.22}$$

A production $(k_0, k_1 ... k_n) \in P$ has an *attribute occurrence* $k_i.a$ if $a \in A(k_i)$, where $k_i \in \Sigma \cup \mathcal{V}, 1 \leq i \leq n$.

Every production $p \in P$ has a set of rules R_p.

For every attribute occurrence $k_0.a$ such that $a \in S(k_0)$ there is one and only rule r of the form $k_0.a = f(h_1, ..., h_j)$ with $k \geq 0$ such that $r \in R_p$.

For every attribute occurrence $k_i.a$ such that $a \in S(k_i)$ and $1 \leq i \leq n$ there is one and only rule r of the form $k_i.a = f(h_1, ..., h_j)$ with $k \geq 0$ such that $r \in R_p$.

Every h_l with $1 \leq l \leq j$ is an attribute occurrence in p.

f is a semantic function which maps the value of an attribute occurrence to the value of its arguments.

$$R = \bigcup_{k \in \Sigma \cup \mathcal{V}} R(k) \tag{2.23}$$

Definition 2 *An* attribute tree *for a string* $w \in \Sigma^*$ *is a derivation tree of* w *where each node* n *labeled by* $b \in \Sigma \cup \mathcal{V}$ *has an attribute instance for every attribute* $a \in A(b)$.

Definition 3 Attribute evaluation *of a string* $w \in \Sigma^*$ *is the process of computing the values of attribute instances of an attribute tree* T *of* w *according to the semantic rules* R.

Figure 2.4: Example of attribute grammar

$\langle S\rangle$ ::= $\langle A\rangle$ { $value(\langle S\rangle) = value(\langle A\rangle)$ }
$\langle S\rangle$ ::= $\langle B\rangle$ { $value(\langle S\rangle) = value(\langle B\rangle)$ }
$\langle A_0\rangle$ ::= $\langle A_1\rangle$ '+' $\langle B\rangle$ { $value(\langle A_0\rangle) = value(\langle A_1\rangle) + value(\langle B\rangle)$ }
$\langle A\rangle$ ::= $\langle B_1\rangle$ '+' $\langle B_2\rangle$ { $value(\langle A\rangle) = value(\langle B_1\rangle) + value(\langle B_2\rangle)$ }
$\langle B_0\rangle$ ::= $\langle B_1\rangle$ '*' 'n' { $value(\langle B_0\rangle) = value(\langle B_1\rangle) * eval(\text{'n'})$ }
$\langle B\rangle$ ::= 'n' { $value(\langle B\rangle) = eval(\text{'n'})$ }

Even if it is possible to characterize different types of attribute grammars for this work we just need the definition of the S-attributed ones.

Definition 4 *An attribute grammar is* S-attributed *if it has only synthesized attributes.*

While being quite primitive, this kind of attribute grammars is enough powerful to express any attribute calculation, as stated in [24].

The first advantage of using an S-attributed grammar is that if used with a bottom-up parser allows to evaluate all the attributes of the tree with just one traversal.

Another advantage can be observed from an incremental prospective. Assume we have evaluated an attribute tree T corresponding to a certain input and then we want to evaluate an attribute tree T' which is derived from a modified input. We should have an efficient way of recomputing the attributes of the new tree, i.e., re-evaluating only the ones which are changed. An S-attributed grammar guarantees that the change on an attribute of a node n will reflect only to the (grand)parents of n.

As seen in the previous section, after a change in the input we obtain a new subtree which substitutes a node in the old tree. From a semantic point of view is sufficient to compute the attributes of the root of the new subtree and to propagate them in the old tree toward the root.

An example of attribute grammar is shown in Figure 2.4. This grammar calculates the algebraic value of an expression belonging to the grammar in 2.1. The only attribute of the grammar is *value* which contains the numeric value of that node. The function *eval* returns the numeric value of the terminal. In each node of the tree the value of the expression is computed from the values of its children.

Chapter 3

Related Work

> *"A problem never exists in isolation; it is surrounded by other problems in space and time. The more of the context of a problem that a scientist can comprehend, the greater are his chances of finding a truly adequate solution."*
>
> Russell L. Ackoff

In the field of incremental verification, many different approaches have been proposed in the past. They can be characterized in three different categories depending the way the incrementality is obtained: from change encapsulation, from change anticipation and with a syntax driven approach ([17]). The latter is the most important for our purpose, since it is the one adopted in this work.

In this section first we review these three approaches. Then we present other work in the field of model-checking.

3.1 Incrementality from change encapsulation

This approach deals with a complex system by considering the whole system as composed by different modules with an approach called assume-guarantee [23]. According to this approach each module has to guarantee that a given property, or its contract toward the other modules, holds.

The idea is that a certain module can guarantee its property assuming that another module would deliver another property. In this way a composition claim, which is the union of the property guaranteed by the modules, can be given by valuating every single module alone.

This approach works in an incremental way if the changes are local (or encapsulated) in a module: which is in spite of the change the module

continues to guarantee its contract. Otherwise the whole system has to be verified again and the composition claim does not hold.

Moreover, in order to use this technique, the system has to be divided properly into modules: a wrong partitioning would cause poor performance in the incremental verification. Nevertheless since the module itself encapsulates high-level concepts, the low level properties are ignored.

An example of work based on this approach is [8], which applies a compositional approach to the general framework for incremental model-checking. In [25] the compositional verification is applied to probabilistic and non-deterministic systems, while in [16] it has been used to perform a model-checking of a multithreaded software system.

3.2 Incrementality by change anticipation

This approach is based on a program optimization technique called partial evaluation.

This technique consists in generating from a program P a new program P' which has the same behaviour of P but runs faster. The reason for this lies in the fact that some input values are known at compile time.

We can think of a program P as a mapping from input data I to output data O ($P : I \to O$). I can be subdivided in two partitions: I_{static}, which contains the input data known at compile-time, and $I_{dynamic}$, which is composed by run-time input data. The new program P' (or residual program) is obtained by precomputing the inputs which belong to I_{static} ($P : I_{dynamic} \to O$).

This procedure can be exploited in verification methods which are based on parameters, like probabilistic verification, in a method called parametric analysis. The result is computed in a parametric formulae. As soon as values for the parameters are available, they are substituted in the formula to obtain the final value. If the values change the precomputed formulae can be reused.

A work which exploits this approach is [15], which uses the parametric approach of model-checking of probabilistic computational tree logic (PCTL) over discrete-time Markov chain (presented in [12]) to develop an incremental probabilistic model-checking.

3.3 Syntax-driven incrementality

This approach relies on the formal grammar which describes the syntactical structure of the verification target. If such grammar is built in a certain way it can be used as a frame for the analysis.

This technique derives from the studies for an incremental compilation from both syntactical and semantic point of view. An example of the former is [19]), whereas the latter is discussed in, e.g., [22].

In fact this approach is very general: any problem that can be expressed through the evaluation of an attribute grammar can exploit it.

Its two main advantages are the total automation of the process, and its general nature. The former is due to the way in which the semantic evaluation is performed: the parser can automatically recognize the part of the input to be reparsed and according the the generated tree only the useful semantic rules are triggered. The latter means that there are no limitations in the kind and the broadness of the changes.

A workflow verification through an attribute grammar is performed in [14]; however, it is not incremental. More important for the sake of this work is instead [4], where the framework SiDECAR, which is the one used in the developing of this work, is presented. SiDECAR is a general framework for the evaluation of S-attributed grammars built upon operator-precedence grammars in an incremental way. In the article it is applied to probabilistic verification and reachability analysis. The latter is the basis for this work which, in a certain way, extends it.

3.4 Incremental model-checking

Related to this aspect, the main idea is to reuse as much as possible the representation explored during verification, and to reanalyze only the new parts of the space state. This technique is followed by [34], which performs model checking in the mu-calculus by analyzing the changes to be applied to the labeled transition system verified.

In [20], instead, it is applied the lazy-abstraction algorithm which builds the abstract reachability tree, a representation of the region of the abstract space tree reachable of the program. The incrementality is given by the computation of the difference in the abstract reachability tree and analyzing only the parts that differ.

Reference [26] provides an incremental explicit space-state model-checking procedure. It performs a graph search from the initial state, trying to reach states that violates the given properties avoiding to visit the same state

many times. The incrementality is obtained by analyzing the difference in the state-space graph to find what explorations have to be performed. A similar work is [35], which can perform the new analysis skipping the parts of the state-space which will not change behaviour after the changes.

Finally in [10] the analysis is recorded in a “derivation graph”. After a change, it is altered by adding the new derivations and removing the ones no more present in the program. All these techniques reported have the disadvantages of being bound to the model used to represent the state-space, and thus are difficult to adapt for a different one.

Chapter 4

Reachability analysis of KernelC programs with matching logic

> *"The final test of a theory is its capacity to solve the problems which originated it."*
>
> George Dantzig

This chapter describes our approach for incremental reachability checking using matching logic. We designed a matching logic-like reachability system for KernelC using an S-attributed grammar over an operator-precedence grammar; the attributes compute the matching logic reachability rules which describe the behaviour of the program.

In addition to achieve incrementality, we also want to merge the reachability rules as much as possible: we combine a sequence of rules which can be applied one after the other, into a single one which summarizes the entire combination. The reason for this is to reducing the computation time by decreasing the length of the paths in the transition system.

4.1 Syntactical part

In this section we present the grammar of KernelC that will be used in the work.

This grammar is based on the one provided with the original Matching logic KernelC verifier [1]. It has been modified in order to be transformed in a operator-precedence grammar.

During this transformation a problem arose related to the tokens '+', '-', '*', '\&': they appears as operators with different arity (unary and binary). Indeed in the different cases the operators have different precedence. In order to manage this a preprocessor phase has been introduced to distinguish between the two versions of each token. However this addition stage does not affect incrementality since it can be done incrementally too.

The resulting grammar is almost the same as the original. There are only minor changes, of which the most import one is the absence of the single statement blocks in **while** loops or **if else** conditionals.

This has been done to keep the grammar compact due to the limit of operators grammar, since it is not possible to use two adjacent non-terminals.

The complete grammar is presented in Appendix A.

4.2 Configuration term structure

Here we describe the matching logic configuration term chosen in order to represent the state of a KernelC program.

The configuration is subdivided into the following terms:

c represent the code of our programs. It is a list of these types:

- A code token, which is a representation of a token in the input string.
- A value (optionally with a type).

In KernelC the only data type is integer. So value is always an integer. However, in order to use the structures, it is important to maintain a type for some values. A type is divided into two parts. The first represents the name of the type and can be any structure defined in the code plus **int** or **void**, which are the built-in types. The second is an integer which represents the level of indirection, which is the number of indirections to be performed in order to reach the value.

env represents the environment. It is a map from variable names to a value possible with its type.

mem represents the heap. It is a map between a memory location and its content.

fstack represents the stack. It is a list containing the previous code and environment prior the the last function call.

fun represents the function. It is a map between a function name and the list of types of its parameters and the actual function code.

size is a map between a **struct** name and its size.

struct is a map between a **struct** field and its relative position inside the **struct**.

We present the rules which map a pattern configuration to another in the next section also providing an S-attributed grammar to compute them.

4.3 Semantic part

First of all we describe the attribute structure of our grammar.

The two main attributes of our grammar are `C` and `R`.

`C` describes the code related the symbol to which it belongs, which is if it belongs to a node n it describes the code of the subtree which has n as root. Its structure is the same as the one of c presented in the previous section. `R` instead is the set of reachability rules generated. In this grammar we focus only on the rules which are directly generated from code and not from the ones obtained by merging different ones together.

Other attributes are used to carry a specific information through the tree. `N` contains a variable identifier.

`L`,`D`, `OC` and `FC` are used in managing the function call part. `L` contains a list of identifiers with their types while `D` is a list of code structures which represent a list of expressions. `OC` and `FC` represent (like `C`) a part of the code and share the same structure.

A type is carried in `TYPE` and `PTR` attributes. `TYPE` carries the name of the type. `PTR` carries the number of indirection.

Finally, in order to generate the struct related rules, the following attributes are used:

- `SIZE` contains a list of elements whose sizes must be known.
- `STRUCT` contains the list of parameters which a structure contains.
- `SIZE'` contain the list of the sizes of the computed structs.

The reachability rules are written in the attribute grammar in the following form $\varphi \Rightarrow \varphi'$ where φ and φ' are configuration patterns. Each pattern contains only the relevant sub-terms. The c term represents only the relevant part, i.e., it does not consider the remaining part of c which does not change.

One important thing on how the reachability rules work is related to the different components of c: code token and value. Code tokens represent the actual input program, and contain information which are available at compile time. Values are instead a way or representing information only available at run-time. This fact is the key of our reachability system, because many rules expect to "receive" the result of other rule computations. In this way is possible to decouple the effect of the different rules on the different parts of the program.

We now present the rules and provide a brief explanation.

$\langle program \rangle_0$ ::= $\langle programChoice \rangle_1$ FinalAnnotation$_2$? $\langle moreAnnotation \rangle_3$?
{
$C_0 = C_1$
$R_0 = R_1$
}

This rule has only the purpose of reporting the computed attribute to the root node ($\langle program \rangle$).

$\langle global_declaration \rangle_0$ ::= '`struct`'$_1$ IDENTIFIER$_2$ '`{`'$_3$ $\langle declaration \rangle_4$ '`}`'$_5$ $\langle global_declaration1 \rangle_6$? {
$C_0 = C_1 \circ C_2 \circ C_3 \circ C_4 \circ C_5 \circ C_6$
$R_0 = R_6 \cup \langle c, struct, size \rangle \Rightarrow \langle c', struct', size' \rangle$
where:
$c = C_1 \circ C_2 \circ C_3 \circ C_4 \circ C_5$
$c' = \emptyset$
$size = value_2.SIZE_4 \cup \langle D_4(0) \rightarrow i_0, ..., D_4(n-1) \rightarrow i_{n-1} \rangle$
$struct' = value_2.STRUCT_4 \cup struct \cup \langle value_2.D_4(0) \rightarrow k_0, ..., value_2.D_4(n-1) \rightarrow k_{n-1} \rangle$
$k_0 = 0$
$k_j = k_{j-1} + i_{j-1}$
$size' = value_2.SIZE'_4 \cup size \cup \langle value_2 \rightarrow k_n \rangle$
}
| ($\langle function_declaration \rangle | \langle parameter \rangle$)$_1$ '`;`'$_2$ $\langle programChoice \rangle_3$ {
$C_0 = C_1 \circ C_2 \circ C_3$
$R_0 = R_1 \cup R_3$
}

$\langle global_declaration1 \rangle_0$ ::= '`;`'$_1$ $\langle programChoice \rangle_2$ {
$C_0 = C_1 \circ C_2$
$R_0 = R_2$
}

These rules compute the attributes of the different declarations of the language. The first is related to the computation of a **struct** reachability rule. This rule is characterized by having in the *size* term of the first configuration pattern the sizes of all types used in the structure, while adding in the *size* term of the second configuration pattern, the ones related to the new defined types (if any), while in the *struct* term of the second pattern the relative position of the **struct** elements are added. To do this the attribute rule uses the data produced in the $\langle declaration \rangle$ node.

$\langle parameter \rangle_0$ ::= $\langle type \rangle_1$ IDENTIFIER$_2$ {

$C_0 = c$

$R_0 = \langle c, e \rangle \Rightarrow \langle c', e' \rangle$

$L_0 = \langle TYPE_1, PTR_1, N_2 \rangle$

where:

$c = C_1 \circ C_2$

$c' = \emptyset$

$e' = e \cup \langle N_2 \mapsto (TYPE_1, PTR_1) undef \rangle$

}

| $(\langle type \rangle | \langle ptr_type \rangle)_1$ '*'$_2$ $\langle id \rangle_3$ {

$C_0 = c$

$R_0 = \langle c, e \rangle \Rightarrow \langle c', e' \rangle$

$L_0 = \langle TYPE_1, PTR_1 + 1, N_3 \rangle$

where:

$c = C_1 \circ C_2 \circ C_3$

$c' = \emptyset$

$e' = e \cup \langle N_3 \mapsto (TYPE_1, PTR_1 + 1) undef \rangle$

}

These rules handle the variable declaration. The reachability rule added states that there is a new variable in the environment which has the *undefined* value. In addition it also writes the type information in the proper attribute. The decision whether to use the reachability rule or the parameter information is taken by another node which will have more information regarding the position of the statement.

$\langle parameter_list \rangle_0$ ::= $\langle parameter \rangle_1$ ','$_2$ $(\langle parameter_list \rangle | \langle parameter \rangle)_3$ {

$c = C_1 \circ C_2 \circ C_3$

$L_0 = L_1 \circ L_3$

}

These rules just define a list of parameters.

$\langle function_definition\rangle_0$ ::= $\langle type\rangle_1$ IDENTIFIER$_2$ $\langle function_definition2\rangle_3$ {

$C_0 = c$

$R_0 = R_3 \cup \langle c, ..., fun\rangle \Rightarrow \langle c', ..., fun'\rangle$

where:

$c = C_1 \circ C_2 \circ C_3$

$c' = OC_3$

$fun' = fun \cup \langle N_2 \mapsto \langle\langle(TYPE_1, PTR_1), L_3\rangle, FC_3\rangle\rangle$

}

| $(\langle type\rangle|\langle ptr_type\rangle)_1$ '*'$_2$ $\langle function_definition1\rangle_3$ {

$C_0 = c$

$R_0 = R_3 \cup \langle c, ..., fun\rangle \Rightarrow \langle c', ..., fun'\rangle$

where:

$c = C_1 \circ C_2 \circ C_3$

$c' = OC_3$

$fun' = fun \cup \langle N_3 \mapsto \langle\langle(TYPE_1, PTR_1 + 1), L_3\rangle, FC_3\rangle\rangle$

}

$\langle function_definition1\rangle_0$ ::= IDENTIFIER$_1$ $\langle function_definition2\rangle_2$ {

$C_0 = C_1 \circ C_2$

$R_0 = R_2$

$FC_0 = FC_2$

$L_0 = L_2$

$OC_0 = OC_2$

$N_0 = value_1$

}

$\langle function_definition2\rangle_0$ ::= '('$_1$ $\langle parameterChoice\rangle_2$? ')'$_3$ Annotation$_4$?

'{'$_5$ $\langle functionChoice\rangle_6$? '}'$_7$ $\langle programChoice\rangle_8$? {

$C_0 = C_1 \circ C_2 \circ C_3 \circ C_4 \circ C_5 \circ C_6 \circ C_7 \circ C_8$

$R_0 = R_6 \cup R_8$

$FC_0 = C_6$

$L_0 = L_2$

$OC_0 = C_8$

}

These rules define a new function. The important ones are related to $\langle$*function_definition*$\rangle$ non-terminal, since $\langle$*function_definition1*$\rangle$ and $\langle$*function_definition2*$\rangle$ just pass the data to it. The reachability rule added states that the piece of code of the function define a new function in the *fun* term, which associates the function name to all data related to the function (return type, input

parameter and function code).

$\langle compound_declaration \rangle_0$ ::= $(\langle function_declaration \rangle | \langle parameter \rangle)_1$ '`;`'$_2$ $\langle functionChoice \rangle_3$

{
$C_0 = C_1 \cup C_3$
$R_0 = R_1 \cup R_3$
}

$\langle compound_declaration1 \rangle_0$ ::= '`;`'$_1$ $\langle compound_declaration \rangle_2$ {
$C_0 = C_2$
$R_0 = R_2$
}

$\langle separator \rangle_0$ ::= '`;`'$_1$ {
$C_0 = C_1$
$R_0 = \emptyset$
}

$\langle separator1 \rangle_0$ ::= '`;`'$_1$ $\langle compoundStatementChoice \rangle_2$ {
$C_0 = C_2$
$R_0 = R_2$
}

These rules have only the purpose of passing attributes inside the tree.

$\langle compound_statement \rangle_0$ ::= '`return`'$_1$ $\langle expressionChoice \rangle_2$? '`;`'$_3$ $\langle compoundStatementChoice \rangle_4$

{
$C_0 = C_2 \circ C_3 \circ C_4$
$R_0 = R_2 \cup R_4 \cup \langle c, e, fstack \rangle \Rightarrow \langle c'', e', fstack' \rangle$
where:
$c = i \circ C_3$
$c'' = i \circ c'$
$fstack' = \langle e', c' \rangle \leftarrow fstack.pop$
}

| '`if`'$_1$ '`(`'$_2$ $\langle expressionChoice \rangle_3$ '`)`'$_4$ '`{`'$_5$ $\langle statementChoice \rangle_6$ '`}`'$_7$
('`else`'$_8$ '`{`'$_9$ $\langle statementChoice \rangle_{10}$ '`}`'$_{11}$)? $\langle compoundStatementChoice \rangle_{12}$

{
$C_0 = C_3 \circ C_5 \circ C_6 \circ C_7 \circ C_9 \circ C_{10} \circ C_{11} \circ C_{12}$
$R_0 = R_3 \cup R_6 \cup R_{10} \cup R_{12} \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$ if $i \neq 0 \cup \langle c, env \rangle \Rightarrow \langle c'', env \rangle$ if $i = 0$
where:
$c = i \circ C_5 \circ C_6 \circ C_7 \circ C_9 \circ C_{10} \circ C_{11}$
$c' = C_6$

$c'' = C_{10}$
}
| $\langle expressionChoice \rangle_1$ ‘`;`’$_2$ $\langle compoundStatementChoice \rangle_3$ {
$C_0 = C_1 \circ C_2 \circ C_3$
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$
where:
$c = i \circ C_2$
$c' = \emptyset$
}
| Annotation$_1$? ‘`while`’$_2$ ‘`(`’$_3$ $\langle expressionChoice \rangle_4$ ‘`)`’$_5$
‘`{`’$_6$ $\langle statementChoice \rangle_7$ ‘`}`’$_8$ $\langle compoundStatementChoice \rangle_9$ {
$C_0 = C_4 \circ C_6 \circ C_7 \circ C_8 \circ C_9$
$R_0 = R_4 \cup R_7 \cup R_9 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$ if $i \neq 0 \cup \langle c, env \rangle \Rightarrow \langle c'', env \rangle$
if $i = 0$
where:
$c = i \circ C_6 \circ C_7 \circ C_8$
$c' = C_3 \circ C_4 \circ C_6 \circ C_7 \circ C_8$
$c'' = \emptyset$
}
| ‘`{`’$_1$ $\langle compound_declaration \rangle_2$ ‘`}`’$_3$ $\langle compoundStatementChoice \rangle_4$ {
$C_0 = C_2 \circ C_4$
$R_0 = R_2 \cup R_4$
}

$\langle statement \rangle_0$ ::= ‘`return`’$_1$ $\langle expressionChoice \rangle_2$? ‘`;`’$_3$ {
$C_0 = C_2 \circ C_3$
$R_0 = R_2 \cup \langle c, e, fstack \rangle \Rightarrow \langle c'', e', fstack' \rangle$
where:
$c = i \circ C_3$
$c'' = i \circ c'$
$fstack' = \langle e', c' \rangle \leftarrow fstack.pop$
}
| $\langle expressionChoice \rangle_1$ ‘`;`’$_2$ {
$C_0 = C_1 \circ C_2$
$R_0 = R_1 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$
where:
$c = i \circ C_2$
$c' = \emptyset$
}
| ‘`if`’$_1$ ‘`(`’$_2$ $\langle expressionChoice \rangle_3$ ‘`)`’$_4$ ‘`{`’$_5$ $\langle statementChoice \rangle_6$ ‘`}`’$_7$

(‘else’$_8$ ‘{’$_9$ $\langle statementChoice\rangle_{10}$ ‘}’$_{11}$)? {
$C_0 = C_3 \circ C_5 \circ C_6 \circ C_7 \circ C_9 \circ C_{10}$
$R_0 = R_3 \cup R_6 \cup R_{10} \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle$ if $i \neq 0 \cup \langle c, env\rangle \Rightarrow \langle c'', env\rangle$ if $i = 0$
where:
$c = i \circ C_5 \circ C_6 \circ C_7 \circ C_9 \circ C_{10}$
$c' = C_6$
$c'' = C_{10}$
}
| Annotation$_1$? ‘while’$_2$ ‘(’$_3$ $\langle expressionChoice\rangle_4$ ‘)’$_5$
‘{’$_6$ $\langle statementChoice\rangle_7$ ‘}’$_8$ {
$C_0 = C_4 \circ C_6 \circ C_7 \circ C_8$
$R_0 = R_4 \cup R_7 \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle$ if $i \neq 0 \cup \langle c, env\rangle \Rightarrow \langle c'', env\rangle$ if $i = 0$
where:
$c = i \circ C_6 \circ C_7 \circ C_8$
$c' = C_3 \circ C_4 \circ C_6 \circ C_7 \circ C_8$
$c'' = \emptyset$
}

These rules manage the behaviour of the various constructs of the language. The **return** construct is managed by a reachability rule which restores the environment and the code prior the function call. The **if−else** construct is managed by providing two reachability rules: one for each possible branch. The **while** construct is handled by one reachability rule which will unroll it if the condition is satisfied and by another rule which will skip it if the condition is not satisfied.

$\langle declaration\rangle_0$ ::= ($\langle function_declaration\rangle$|$\langle parameter\rangle$)$_1$ ‘;’$_2$ $\langle declaration\rangle_3$?
{
$C_0 = C_1 \circ C_2 \circ C_3$
$L_0 = L_1 \circ L_3$
$SIZE_0 = SIZE_3 \cup L_1$
$STRUCT_0 = L_1 \cup STRUCT_3$
$SIZE'_0 = SIZE'_3$
}
| ‘struct’$_1$ IDENTIFIER$_2$ ‘{’$_3$ $\langle declaration\rangle_4$ ‘}’$_5$ ($\langle declaration1\rangle$|$\langle separator\rangle$)$_6$
{
$C_0 = C_1 \circ C_2 \circ C_3 \circ C_4 \circ C_5 \circ C_6$
$L_0 = \langle value_2, 0, value_2\rangle \cup L_6$
$SIZE_0 = SIZE_6 \setminus (value_2.SIZE'_4 \cup value_2 \rightarrow k_n) \cup value_2.SIZE_4 \cup$

$L_4(0) \to i_0, ..., L_4(n-1) \to i_{n-1}$
$STRUCT_0 = STRUCT_6 \cup value_2.STRUCT_4 \cup value_2.L_4(0) \to k_0, ..., value_2.L_4(n-1) \to k_{n-1}$
$SIZE'_0 = SIZE'_6 \cup value_2.SIZE'_4 \cup value_2 \to k_n$
where:
$k_0 = 0$
$k_j = k_{j-1} + i_{j-1}$
}

$\langle declaration1 \rangle_0$ ::= ‘`;`’$_1$ $\langle declaration \rangle_2$ {
$C_0 = C_1 \circ C_2$
$L_0 = L_2$
$SIZE_0 = SIZE_2$
$STRUCT_0 = STRUCT_2$
$SIZE'_0 = SIZE'_2$
}

These rules synthesize **struct** related attributes. These attributes will be used in the ⟨*global_declaration*⟩ node to compute the reachability rule of the **struct**.

$\langle type \rangle_0$ ::= ‘`struct`’$_1$ IDENTIFIER$_2${
$C_0 = C_1 \circ C_2$
$TYPE_0 = value_1$
$PTR_0 = 0$
}
| ‘`void`’$_1$ {
$C_0 = C_1$
$TYPE_0 = void$
$PTR_0 = 0$
}
| ‘`int`’$_1$ {
$C_0 = C_1$
$TYPE_0 = int$
$PTR_0 = 0$
}

$\langle type2 \rangle_0$ ::= ‘`(`’$_1$ ($\langle type \rangle | \langle ptr_type \rangle$)$_2$ ‘`)`’$_3$ {
$C_0 = C_2$
$TYPE_0 = TYPE_2$
$PTR_0 = PTR_2$
}

$\langle ptr_type \rangle_0$ ::= $(\langle type \rangle | \langle ptr_type \rangle)_1$ '*'$_2$ {

$C_0 = C_1$

$TYPE_0 = TYPE_1$

$PTR_0 = PTR_1 + 1$

}

These rules just calculate the attributes related to data type.

$\langle empty_fcall \rangle_0$ ::= IDENTIFIER$_1$ $\langle empty_expression \rangle_2$ {

$R_0 = \langle c, env, ..., fun, fstack \rangle \Rightarrow \langle c', env', fun, fstack' \rangle$

$C_0 = c$

where:

$c = C_1$

$\langle value_1 \mapsto \langle l, c' \rangle \rangle \in fun$

$fstack' = fstack \circ \langle env, code \setminus c \rangle$

$env' = env$

}

This rule handles a function call with no argument. The reachability rules manage the call by saving environment and code in *fstack* and setting the function code.

$\langle id \rangle_0$ ::= IDENTIFIER$_1$ {

$R_0 = \langle c, env \rangle \Rightarrow \langle h, env \rangle$

$C_0 = c$

$N_0 = value_1$

where:

$c = C_1$

$\langle value_1 \mapsto h \rangle \in env$

}

This rule just handles the use of variables. It produces a reachability rule which substitutes a code token representing a variable identifier with the according value taken from the environment.

$\langle argument_expression_list \rangle_0$ ::= $\langle expressionChoice \rangle_1$ ','$_2$ $\langle expressionChoice \rangle_3$

{

$R_0 = R_1 \cup R_3$

$D_0 = C_1 \circ C_3$

}

| $\langle expressionChoice \rangle_1$ ','$_2$ $\langle argument_expression_list \rangle_3$ {

$R_0 = R_1 \cup R_3$

$D_0 = C_1 \circ D_3$

}

These rules just calculate the attributes related to an expression list, by creating a list.

$\langle \mathit{unary_expression} \rangle_0$ ::= ‘`sizeof`’$_1$ $(\langle \mathit{unaryChoice} \rangle | \langle \mathit{type2} \rangle)_2$ {

$R_0 = R_2 \cup \langle c, size \rangle \Rightarrow \langle j, size \rangle$

$C_0 = C_2 \circ C_1$

$N_0 = N_2$

where:

$c = i \circ C_1$

$j = size(i.type)$

}

| (‘`~`’|‘`!`’|‘`+`[1]’|‘`-`[1]’)$_1$ $\langle \mathit{postfixChoice} \rangle_2$ {

$R_0 = R_2 \cup \langle c \rangle \Rightarrow \langle j \rangle$

$C_0 = C_2 \circ C_1$

$N_0 = N_2$

where:

$c = i \circ C_1$

$j = op_1 i$

}

| ‘`&`[1]’$_1$ $\langle \mathit{postfixChoice} \rangle_2${

$R_0 = R_2 \cup \langle c, mem \rangle \Rightarrow \langle j, mem \rangle$

$C_0 = C_2 \circ C_1$

$N_0 = N_2$

where:

$c = i \circ C_1$

$\langle j \mapsto i \rangle \in mem$

}

| ‘`*`[1]’$_1$ $\langle \mathit{postfixChoice} \rangle_2${

$R_0 = R_2 \cup \langle c, mem \rangle \Rightarrow \langle j, mem \rangle$

$C_0 = C_2 \circ C_1$

$N_0 = N_2$

where:

$c = i \circ C_1$

$\langle i \mapsto j \rangle \in mem$

}

| (‘`++`’|‘`--`’)$_1$ $\langle \mathit{unaryChoice} \rangle_2$ {

$R_0 = R_2 \cup \langle c \rangle \Rightarrow \langle j \rangle$

$C_0 = C_2 \circ C_1$

$N_0 = N_2$

[1] This are the unary versions of the tokens.

where:
$c = i \circ C_1$
$j = op_1 i$
}

$\langle assignment_expression \rangle_0$ ::= $\langle unaryChoice \rangle_1$ ‘=’$_2$ $\langle expressionChoice \rangle_3$ {
$R_0 = R_3 \cup \langle c, env \rangle \Rightarrow \langle i, env' \rangle$
$C_0 = C_3 \circ C_2 \circ C_1$
where:
$c = i \circ C_2 \circ C_1$
$env' = env \circ \langle N_1 \mapsto i \rangle$
}
| $\langle unaryChoice \rangle_1$ (‘+=’|‘-=’|‘*=’|‘/=’|‘%=’)$_2$ $\langle expressionChoice \rangle_3$ {
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env' \rangle \Rightarrow \langle k, env'' \rangle$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3 \circ C_2 \circ i$
$c'' = j \circ C_2 \circ i$
$k = i \, op_2 \, j$
$env'' = env' \circ \langle N_1 \mapsto i \, op_2 \, j \rangle$
}
| $\langle unaryChoice \rangle_1$ (‘^=’|‘|=’|‘&=’|‘<<=’|‘>>=’)$_2$ $\langle expressionChoice \rangle_3$ {
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env' \rangle \Rightarrow \langle k, env'' \rangle$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3 \circ C_2 \circ i$
$c'' = j \circ C_2 \circ i$
$k = i \, op_2 \, j$
$env'' = env' \circ \langle N_1 \mapsto i \, op_2 \, j \rangle$
}

$\langle conditional_expression \rangle_0$ ::= $\langle logicalOrChoice \rangle_1$ ‘?’$_2$ $\langle expressionChoice \rangle_3$ ‘:’$_4$
$\langle conditionalChoice \rangle_5$ {
$R_0 = R_1 \cup R_3 \cup R_5 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$ if $i \neq 0 \cup \langle c, env \rangle \Rightarrow \langle c'', env \rangle$
if $i = 0$
$C_0 = C_1 \circ C_2 \circ C_3 \circ C_4 \circ C_5$
where:
$c = i \circ C_2 \circ C_3 \circ C_4 \circ C_5$

$c' = C_3$
$c'' = C_5$
}

$\langle logical_or_expression \rangle_0$ ::= $\langle logicalOrChoice \rangle_1$ '||'$_2$ $\langle logicalAndChoice \rangle_3$ {
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle 1, env \rangle$ if $i \neq 0 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$ if $i = 0$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3$
}

$\langle logical_and_expression \rangle_0$ ::= $\langle logicalAndChoice \rangle_1$ '&&'$_2$ $\langle inclusiveOrChoice \rangle_3$
{
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$ if $i \neq 0 \cup \langle c, env \rangle \Rightarrow \langle 0, env \rangle$ if $i = 0$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3$
}

$\langle inclusive_or_expression \rangle_0$ ::= $\langle inclusiveOrChoice \rangle_1$ '|'$_2$ $\langle exclusiveOrChoice \rangle_3$
{
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env \rangle \Rightarrow \langle k, env \rangle$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3 \circ C_2 \circ i$
$c'' = j \circ C_2 \circ i$
$k = iop_2 j$
}

$\langle exclusive_or_expression \rangle_0$ ::= $\langle exclusiveOrChoice \rangle_1$ '^'$_2$ $\langle andChoice \rangle_3$ {
$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env \rangle \Rightarrow \langle k, env \rangle$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_2 \circ C_3$
$c' = C_3 \circ C_2 \circ i$
$c'' = j \circ C_2 \circ i$
$k = iop_2 j$
}

$\langle and_expression\rangle_0$ ::= $\langle andChoice\rangle_1$ '&'$_2$ $\langle equalityChoice\rangle_3$ {

$R_0 = R_1 \cup R_3 \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle \cup \langle c'', env\rangle \Rightarrow \langle k, env\rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle equality_expression\rangle_0$ ::= $\langle equalityChoice\rangle_1$ ('=='|'!=')$_2$ $\langle relationalChoice\rangle_3$

{

$R_0 = R_1 \cup R_3 \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle \cup \langle c'', env\rangle \Rightarrow \langle k, env\rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle relational_expression\rangle_0$ ::= $\langle relationalChoice\rangle_1$ ('>'|'<'|'>='|'<=')$_2$ $\langle shiftChoice\rangle_3$

{

$R_0 = R_1 \cup R_3 \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle \cup \langle c'', env\rangle \Rightarrow \langle k, env\rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle shift_expression\rangle_0$ ::= $\langle shiftChoice\rangle_1$ ('>>'|'<<')$_2$ $\langle additiveChoice\rangle_3$ {

$R_0 = R_1 \cup R_3 \cup \langle c, env\rangle \Rightarrow \langle c', env\rangle \cup \langle c'', env\rangle \Rightarrow \langle k, env\rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle additive_expression \rangle_0$ ::= $\langle additiveChoice \rangle_1$ (‘+’|‘-’)$_2$ $\langle multiplicativeChoice \rangle_3$

{

$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env \rangle \Rightarrow \langle k, env \rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle multiplicative_expression \rangle_0$ = $\langle multiplicativeChoice \rangle_1$ (‘*’|‘/’|‘%’)$_2$ $\langle postfixChoice \rangle_3$

{

$R_0 = R_1 \cup R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env \rangle \Rightarrow \langle k, env \rangle$

$C_0 = C_1 \circ C_2 \circ C_3$

where:

$c = i \circ C_2 \circ C_3$

$c' = C_3 \circ C_2 \circ i$

$c'' = j \circ C_2 \circ i$

$k = iop_2 j$

}

$\langle cast_expression \rangle_0$ ::= ‘(’$_1$ ($\langle type \rangle$|$\langle ptr_type \rangle$)$_2$ ‘)’$_3$ $\langle postfixChoice \rangle_4$ {

$R_0 = R_3 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle$

$C_0 = C_4 \circ C_2$

$N_0 = N_3$

where:

$c = i \circ C_2$

$c' = (TYPE_2, PTR_2)i$

}

$\langle nested_expression \rangle_0$ ::= ‘(’$_1$ $\langle expressionChoice \rangle_2$ ‘)’$_3$ {

$R_0 = R_2$

$C_0 = C_2$

}

These rules define the behaviour of the various operators of the language. They work by adding reachability rules which receive a value and produce a new one. Since the value will be known only at runtime, it is handled by a value item in the code part of the rule. In order to work, the reachability rules produced expect that other rules have done their job by converting a

part of the code expressed as tokens into its value. If the operator is unary the only thing to do is to add the code in postfix form and add a reachability rule which will modify the value received according to the operator nature. If the operator is binary this part is trickier. The reason is the fact that only the first operand can be computed by the other rules, while the second one will remain in the form of the program token. So the first reachability rule added just invert the code so that the second operand in program tokens form is now at the beginning of the code (and so can match other reachability rules) while in the second part there is the first operand in the form of value. Then the other rule will actually encode the computation of the operation, by substituting the two values separated by the operator token with the operation value.

$\langle postfix_expression \rangle_0$ ::= Constant$_1$ $\langle postfix_expression1 \rangle_2$? {
$R_0 = R_2 \cup \langle c, env \rangle \Rightarrow \langle i, env \rangle$
$C_0 = C_1 \circ C_2$
$N_0 = undefined$
where:
$c = C_1$
$i = value_1$
}
| IDENTIFIER$_1$ ($\langle nested_expression \rangle$|$\langle postfix_expression1 \rangle$|$\langle postfix_expression2 \rangle$)$_2$
{
$R_0 = R_2 \cup \langle c, env_0, ..., fun, fstack \rangle \Rightarrow \langle c', env', fun, fstack' \rangle$
if $\langle D_2(0), env_0 \rangle \Rightarrow \langle i_0, env_1 \rangle, ..., \langle D_2(n-1), env_{n-1} \rangle \Rightarrow \langle i_{n-1}, env_n \rangle$
$C_0 = c$
where:
$c = C_1 \circ C_2$
$\langle value_1 \mapsto \langle l, c' \rangle \rangle \in fun$
$fstack' = fstack \circ \langle env_n, code \setminus c \rangle$
$env' = env_n \circ subst(l, i_0, ..., i_{n-1})$
}

$\langle postfix_expression1 \rangle_0$ ::= '`[`'$_1$ $\langle expressionChoice \rangle_2$ '`]`'$_3$ $\langle postfix_expression1 \rangle_4$?
{
$R_0 = R_2 \cup R_4 \cup \langle c, env \rangle \Rightarrow \langle c', env \rangle \cup \langle c'', env', mem, size \rangle \Rightarrow \langle k, env', mem, size \rangle$
$C_0 = C_1 \circ C_2 \circ C_4$
where:
$c = i \circ C_1 \circ C_2$
$c' = C_2 \circ C_1 \circ i$
$c'' = j \circ C_1 \circ i$

$\langle i \rightarrow s \rangle \in size$
$\langle i + s * j \rightarrow k \rangle \in mem$
}
| ('.'|'->')$_1$ IDENTIFIER$_2$ $\langle$*postfix_expression1*$\rangle_3$? {
$R_0 = R_3 \cup \langle c, env, mem, size \rangle \Rightarrow \langle k, env, mem, size \rangle$
$C_0 = C_1 \circ C_2 \circ C_3$
where:
$c = i \circ C_1 \circ C_2$
$\langle i.value_2 \rightarrow j \rangle \in struct$
$\langle i +_{Int} j \rightarrow k \rangle \in mem$
}
| ('++'|'--')$_1$ $\langle$*postfix_expression1*$\rangle_2$?{
$C_0 = C_1 \circ C_2$
$R_0 = R_2 \cup \langle c, env \rangle \Rightarrow \langle i, env' \rangle$
where:
$c = i \circ C_1$
$env' = env \circ \langle N_1 \mapsto op_1 i \rangle$
}

$\langle$*postfix_expression2*$\rangle_0$ ::= '('$_1$ $\langle$*expressionChoice*$\rangle_2$? ')'$_3$ $\langle$*postfix_expression1*$\rangle_4${
$C_0 = c$
$R_0 = R_2 \cup R_4$
$D_0 = D_2$
where:
$c = C_1 \circ C_2 \circ C_3 \circ C_4$
}
| '('$_1$ $\langle$*argument_expression_list*$\rangle_2$ ')'$_3$ $\langle$*postfix_expression1*$\rangle_4$?{
$C_0 = c$
$R_0 = R_2 \cup R_4$
$D_0 = D_2$
where:
$c = C_1 \circ C_2 \circ C_3 \circ C_4$
}

These rules define the behaviour of a postfix expression. In the case of a constant they simply substitute the code token with the value of the constant. In the case of a function call, the call is performed by setting as code the code of the function and saving the environment and the code in *fstack*. In the case of a postfix operator the same considerations of unary operators have to be done.

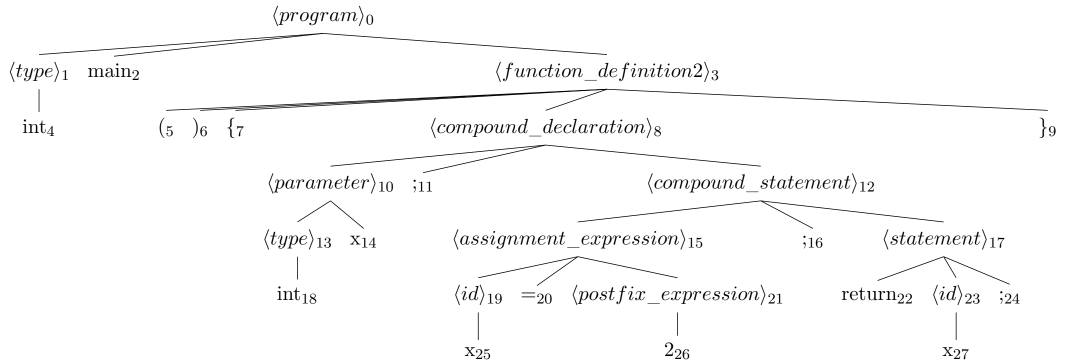
4.4 Attribute evaluation example

In this section we show an example of attribute evaluation on the abstract syntax tree depicted in Figure 4.1, corresponding to the input program shown in Listing 4.1; we assume each terminal is identified by a token t_i.

Listing 4.1: code of KernelC attribute evaluation

```
1 int main()
2 {
3   int x;
4   x = 2;
5   return x;
6 }
```

Figure 4.1: Example of KernelC attribute evaluation



$C_1 = \{t_4\}$
$TYPE_1 = int$
$PTR_1 = 0$

$C_{13} = \{t_{18}\}$
$TYPE_{13} = int$
$PTR_{13} = 0$

$C_{10} = \{t_{18}, t_{14}\}$
$R_{10} = \langle\{t_{18}, t_{14}\}, e\rangle \Rightarrow \langle\emptyset, e \cup \langle \mathtt{x} \mapsto (int, 0)undef\rangle\rangle$
$L_{10} = \langle int, 0, \mathtt{x}\rangle$

$R_{19} = \langle\{t_{25}\}, env\rangle \Rightarrow \langle h, env\rangle$ with $\langle \mathtt{x} \mapsto h\rangle \in env$
$C_{19} = \{t_{25}\}$
$N_{19} = \mathtt{x}$

$R_{21} = \langle\{t_{26}\}, env\rangle \Rightarrow \langle 2, env\rangle$
$C_{21} = \{t_{26}\}$
$N_{21} = undefined$

$R_{15} = R_{21} \cup \langle\{i, t_{26}, t_{20}\}, env\rangle \Rightarrow \langle i, env \circ \langle \mathtt{x} \mapsto i\rangle\rangle$
$C_{15} = C_{21} \circ \{t_{26}, t_{20}\}$

$R_{23} = \langle\{t_{27}\}, env\rangle \Rightarrow \langle h, env\rangle$ with $\langle \mathtt{x} \mapsto h\rangle \in env$
$C_{23} = \{t_{27}\}$
$N_{23} = \mathtt{x}$

$C_{17} = C_{23} \circ \{t_{24}\}$
$R_{17} = R_{23} \cup \langle\{i, t_{24}\}, e, fstack\rangle \Rightarrow \langle i \circ c', e', fstack'\rangle$
where:
$fstack' = \langle e', c'\rangle \leftarrow fstack.pop$

$C_{12} = C_{15} \circ \{t_{16}\} \circ C_{17}$
$R_{12} = R_{15} \cup R_{17} \cup \langle\{i, t_{16}\}, env\rangle \Rightarrow \langle\emptyset, env\rangle$

$C_8 = C_{10} \cup C_{12}$
$R_8 = R_{10} \cup R_{12}$

$C_3 = \{t_5, t_6, t_7\} \circ C_8 \circ \{t_9\}$
$R_3 = R_8$
$FC_3 = C_8$
$L_3 = \emptyset$
$OC_3 = \emptyset$
$C_0 = \{t_4, t_2\} \circ C_3$
$R_0 = R_3 \cup \langle\{t_4, t_2\} \circ C_3, ..., fun\rangle \Rightarrow \langle\emptyset, ..., fun'\rangle$
where:
$fun' = fun \cup \langle \mathtt{main} \mapsto \langle\langle(int, 0), \emptyset\rangle, C_8\rangle\rangle$

Chapter 5

Implementation

"In theory, theory and practice are the same. In practice, they are not."

Jan L. A. van de Snepscheut

In this chapter we present the implementation of the tool accompanying this work.

The implementation is totally different from the original work on KernelC [1]. The original KernelC verifier has been mainly implemented in Maude [7], which is a logic programming language based on term rewriting [27]. It is constituted of two parts. One part is a simple parser written in Java and based on ANTLR. It reads the KernelC program and translates it into a Maude input. The second one is the actual verifier, it is written in Maude and it is based on the language definition of KernelC.

Our implementation follows an Object-oriented paradigm and is written in Java. It is composed of different packages and is divided in two groups. The first one, the syntactical one, implements the parsing of the input program and drives the attribute evaluation. It is independent from the programming language used. The second group of packages is language-specific; the central part is the implementation of attribute grammar evaluation.

The two groups of packages are "linked" by the concept of attribute grammar: in the first group it is represented by the abstract class *SemanticVisitor*; in the second one, there is the actual implementation.

5.1 Syntactic part

This part is composed by the following packages:

- *grammar*: it defines an input operator-precedence grammar and generates its operator precedence matrix (see Figure 5.1).

- *parser*: it contains the operators grammar parsers (see Figure 5.2).
- *semantics*: it contains the template of an attribute grammar.

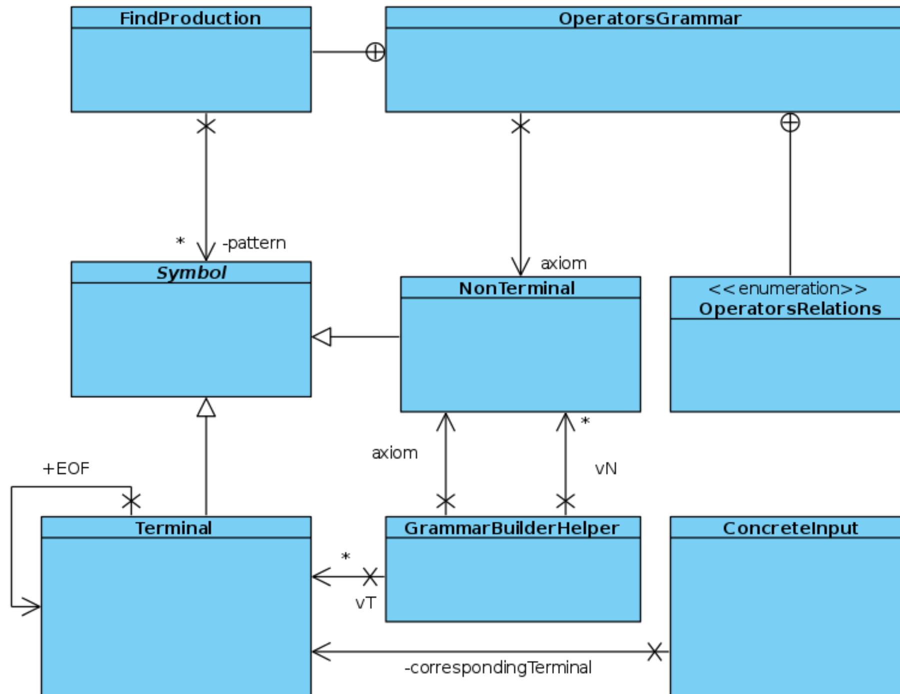


Figure 5.1: UML Class diagram of grammar package

5.1.1 Grammar

This package can be divided in two parts: one composed by the primitive elements which forms a grammar, and one that contains the actual grammar elements.

The first part is composed by the class *Symbol*, *Terminal*, *NonTerminal* and *ConcreteInput*. *Symbol* is the abstract element that represents a generic symbol of the grammar. It is implemented by the two classes *Terminal* and *NonTerminal* which represents the elements of the grammars. The last class *ConcreteInput* represents the actual terminals of a grammar. It is different from the *Terminal* one because, while *Terminal* represents a generic terminal of the grammar, *ConcreteInput* represents an instance of it. This does not happen with non-terminals since they can be considered equals while different terminals can have different representations.

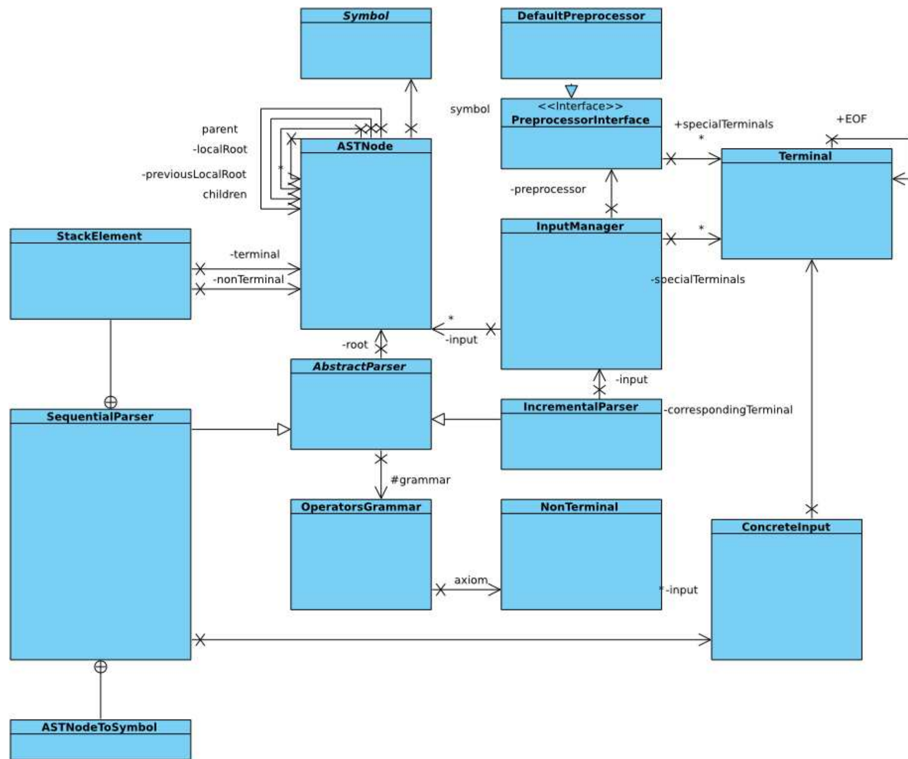


Figure 5.2: UML Class diagram of parser package

The second part is composed by *OperatorGrammar* and *GrammarBuilderHelper*. The important one is the former, since the latter is just a factory class for it. The most important method of *OperatorsGrammar* is *generatePrecedenceTable* which constructs the operator precedence matrix of the grammar.

5.1.2 Parser

This package can be divided in three parts. The first part contains the basic elements of the parsing process which are the classes *ASTNode* and *InputManager*.

The former, as the name suggests, contains a node of the abstract syntax tree. The latter instead maintains the input to be processed for a reason which will be explained later. The second one is formed by the actual parsers: it is composed by *AbstractParser*, *SequentialParser* and *IncrementalParser*. While the first is just a generic class which defines a basic parser structure, the others are implementation operator-precedence grammars parsers.

The difference between the two is the fact that the former reads always the whole input, while the latter tries to analyze the smallest part of the input, if it derives from the previous parsed input with some changes.

The main method that realizes this is *edit*, which receives a change as the starting position from the beginning of previous input, the number of tokens to be deleted from that position, and a list of tokens to be inserted after the deletion. It returns an *ASTNode* which contains the root of the syntax tree.

Also the *SequentialParser* class implementation can provide a (partial) incremental parsing: although it has to analyze the whole input, it is able to recognize if a certain reduction has been computed before and thus reuse it. So, while lacking syntactical incrementality, it is suitable for semantic incrementality.

The last part is composed of *PreprocessorInterface* and *DefaultPreprocessor*. This works with *InputManager* to form a preprocessor, which performs small changes to the input tokens. This component has been introduced to deal with the problem of two versions of some tokens with different arity and precedence. In this way a single token is threaded in the grammar as different ones while this part dealt with fictional tokens used in the productions.

PreprocessorInterface is an interface that specifies which methods must be defined by a preprocessor; *DefaultPreprocessor* a class which implements a preprocessor which does nothing, i.e., maps each token to itself.

5.1.3 Semantics

This package defines the structure of attribute grammars using the abstract class *SemanticVisitor*, based on the visitor design pattern.

An attribute grammar is defined as a class which extends *SemanticVisitor*. For each attribute we define an attribute class and a visitor class which extends *SemanticVisitor*.

The visitor class must define the method *visit* which computes the attribute. It is possible to retrieve an attribute value associated to a node with the method *getValue* and add an attribute value for a node with the method *insertValue*.

5.2 Kernel C semantic

The Kernel C semantic part is divided as follows:

- *syntax*: a package which contains the Kernel C grammar definition and its preprocessor.
- *semantics*: a package which performs the attribute evaluation.
- *solver*: this part actually performs program verification.

The first two parts contains a grammar definition and concrete class implementing (extending) *PreprocessorInterface* and *SemanticVisitor*. The last one contains the detailed program analysis and is analyzed in the next section. An overview of the class structure of these part is shown in Figure 5.3, 5.4, 5.5, 5.6, 5.7.

5.3 Solver

This is the main component of our tool. Before illustrating it, we explain how the approach presented in the previous chapter has been casted into an object-oriented implementation that relies on an SMT solver (namely, Z3 [13]). First of all the configuration structure has been slightly modified. Each configuration is now composed by three parts: the first represents the code, the second is a first order logic formula which describes the properties of the configuration and the third one is composed by the real configuration and represents the configuration structure. The code part, which corresponds to the configuration sub-term c, has been separated from the configuration since the code structure drives the attribute evaluation. The logic formula is the way in which most of the properties of the configuration are encoded. It is used to translate the verification problem in a format understandable by the SMT solver. The "structural" part is instead the one which resembles the matching logic configuration. Its main task is to perform configuration mapping but it can also maintain some properties.

The number of different configurations has also been reduced. There are only the *env* and the *heap*.

The *env* has been enriched with *fstack* since they are used together and this determines a simpler structure of the configuration rules.

The *heap* configuration has the only purpose of mapping between different configurations, while all its properties are managed in the FOL-formula. In particular, it is managed with one integer and three maps from integers to integers: the integer contains the position reached in memory allocation. The arrays are maps from a memory cell to a certain information. One contains if a certain cell is allocated (and if so the pointer of the first cell in the

allocation). The second if the cell is initialized or not. The third contains the memory content of the cell.

Memory management is used by the built-in functions: *sbrk* and *brk*. There are two versions of *sbrk*: one has no input parameters and returns the position of the next cell. The second accepts two parameters: an index and a cell. If the owner of the cell at the position of index is the owner passed to the function it frees that cell and returns 1; otherwise returns 0. *brk* allocate the next cell receiving a parameter which contains the cell owner. With these functions is possible to implement *malloc* (shown in Listing 5.1) and *free* (Listing 5.2), which are the memory management functions provided by the standard C library.

Listing 5.1: Example of malloc definition

```
1  void* malloc(int size){
2    int c;
3    void* ptr;
4    if(size<1){
5      return 0;
6    }
7    ptr = sbrk();
8    c=0;
9    while(c<size){
10     brk(ptr);
11     c=c-1;
12   }
13   return ptr;
14 }
```

Listing 5.2: Example of free definition

```
1 void* free(void* base){
2   void* ptr;
3   ptr=base;
4   while(sbrk(ptr,base)){
5     ptr=ptr+1;
6   }
7 }
```

The *code* instead has been enriched to substitute the structures *size*, *struct*, *fun*. A configuration is represented by the class *Configuration* and is composed by the three parts presented before. A rule, represented by *Rule*, identifies a transition between a set configuration to another set of configuration. This element can be labeled by an instance of *Metadata* which carries additional information of the rule, which are used in the verification process.

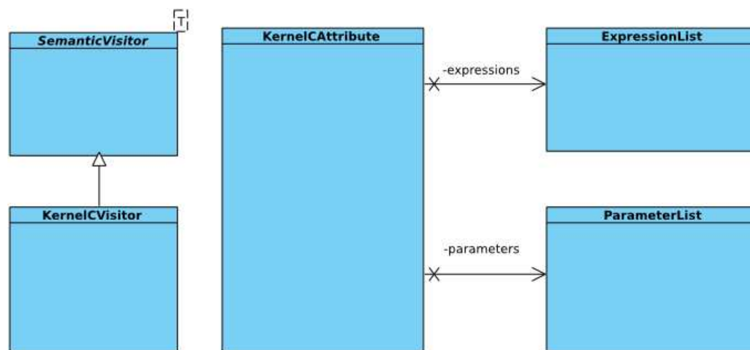


Figure 5.3: UML Class diagram of KernelC semantics package

5.3.1 Code

The main class *Code* contains a list of basic code elements, represented by the abstract class *CodeElement*.

The class *Token* represents the actual code from the program.

Instead the class *CodeValue* represents a variable with a certain type. Value and type of the variable can be determined only at run-time, so each variable is represented with a unique string which identifies an integer variable in the formula.

The class *VariableCode* deals with functions and structs. It is composed by a string which is the actual value of an input token. The difference w.r.t. *Token* is that in the latter the value of the token is not important since it is used only locally to refer to that particular token.

Finally there is the *SpecialToken* which are new token which have special meaning. Some of them are used in the management of function calls. During function call the code is simply appended at the top of the code. The problem of this is dealing with the *return* statement: if there is one in the middle of a function all the subsequent code must be dropped away. This is done by the *SpecialTokens* *returnToken*, *endOfFunction* and *globalMatch*.

When a *return* is found a *returnToken* is placed. There are then two general rules, one that deletes the following *CodeElement* after *returnToken* and one that performs the exit from the call if the sequence *returnTokenendOfFunction* is found. To achieve this the token *endOfFunction* is put at the end of each function and *globalMatch* has the property of matching almost all *CodeElements* but *endOfFunction*.

The matching between code part is done by a comparison one by one of the first *CodeElements* in the code.

The comparison depends on the type, for example two *Tokens* are equal

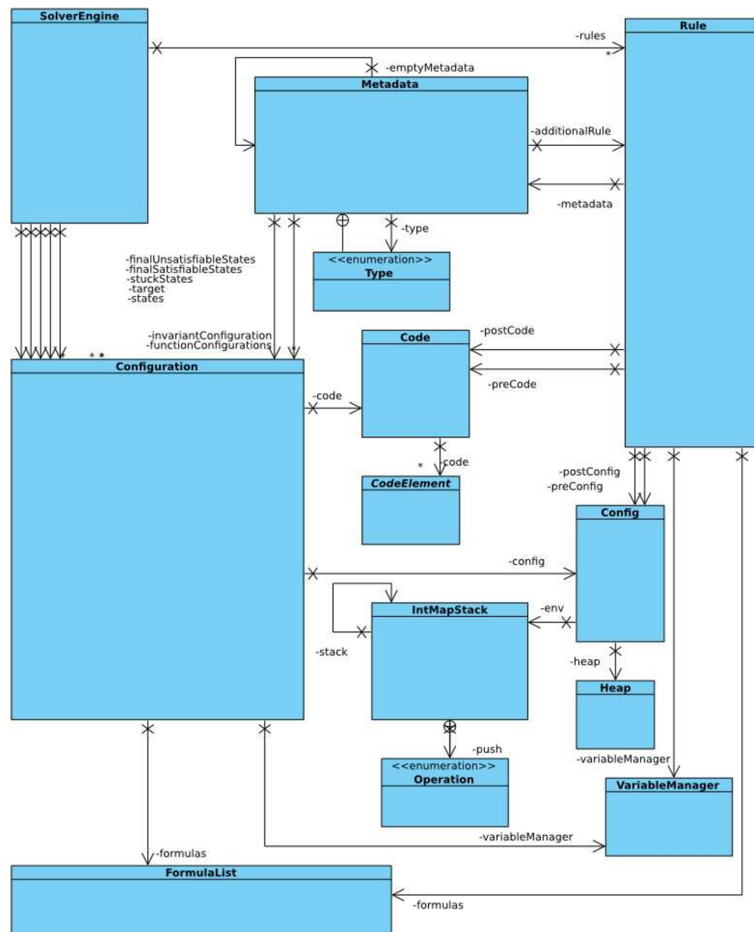


Figure 5.4: UML Class diagram of solver package

if they refers to the same terminal, two *CodeValue* are always equal and two *VariableCode* are equal if their values are.

5.3.2 Formula

The management of the FOL-formula that contains all the constraints of the configuration is performed by the class *FormulaList*.

It allows the introduction of new constraints and interfacing with the external SMT solver.

New constraints are added either by providing methods for adding a specific constraint of by unifying constraints from different configurations.

The interface with the SMT solver is provided through the method *check* which returns if the current formula is satisfiable or not. Since all verification issues can be reduced to a satisfiability problem this is enough for this work.

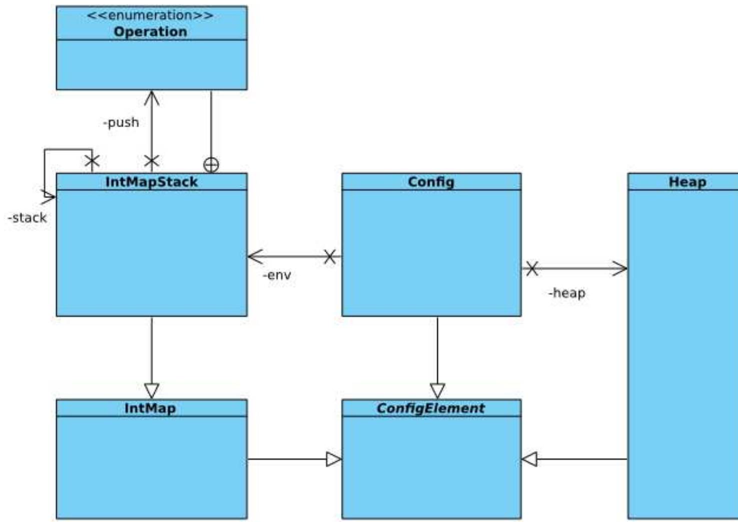


Figure 5.5: UML Class diagram of config package

In addition *FormulaList* encapsulates the SMT solver used (Z3 in our case). So in the event of substituting the external solver, only this class has to be changed.

One issue is due to the fact that in a rule a single formula is used to contain both preconditions and postconditions. The problem that would arise is that if we have a rule R which has a precondition p and maps the set of configurations S to S', the effect of applying that rule would be to map $T \subseteq S$ to $T' \subseteq S'$ where $T \models p$. In that way it is possible to "lose" a subset of configurations which are not allowed by the program.

To overcome this we introduce for each precondition a complementary rule which maps the configuration which does not hold the precondition to an "error configuration". From this configuration, represented by the *SpecialToken stuck*, it is impossible to move and of course it is neither a final one. So the verification would fail as expected.

We made this implementation decision to reduce the number of satisfiability checks, which are computationally intensive. In fact these checks are not performed for every new configuration reached, except for final ones and before rules which can trigger an infinite path, namely loops and function calls.

5.3.3 System operation

The system can perform two types of operations: it can start from a set of configuration and check if all the possible path satisfies a certain property

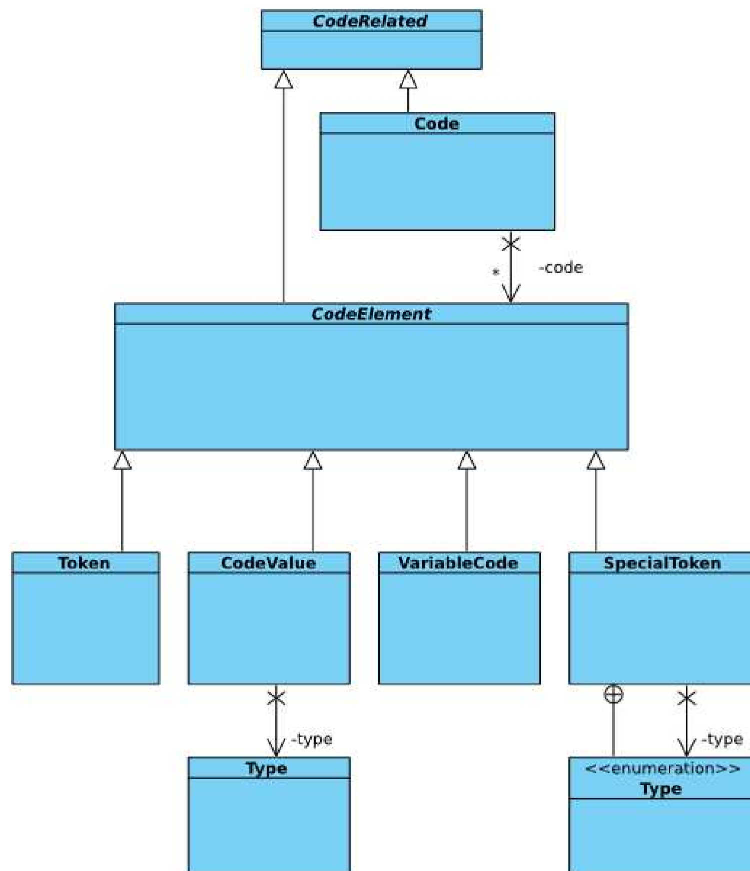


Figure 5.6: UML Class diagram of code package

or produce new rules from the existing ones.

The first task is performed by the class *SolverEngine* and is the general operating mode and is mainly conducted in the method *solve*.

This method starts from the current set of states and continues to apply rule until it is possible. A set of state cannot be processed anymore either if no rule can match it or it is a final one, i.e., matches the target configuration. The final sets of configurations are divided in two groups if the constraints in the formula are satisfiable or not.

In order to verify if a given postcondition hold is sufficient to use as final configuration set one as follows: the code part should be empty, since all the code must have been executed; the structure part should contain all the variable needed and the formula should be the negation of the postcondition

In this way it is sufficient to verify that every path reaches the final configuration without satisfying it. In this way we only need to know if a given formula is satisfiable or not.

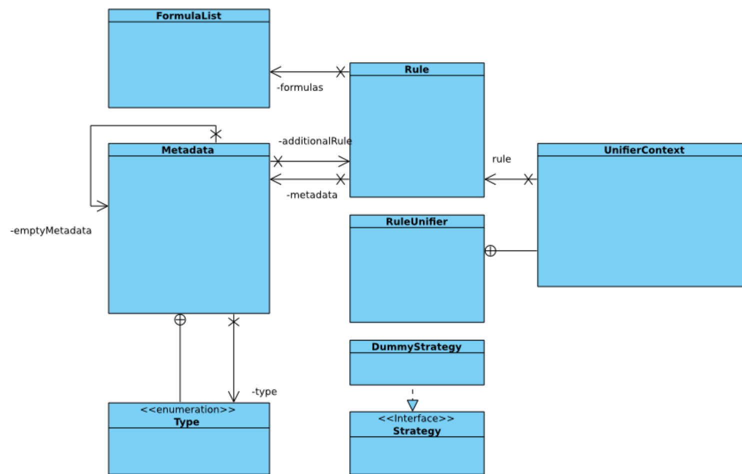


Figure 5.7: UML Class diagram of rule package

The other operating mode tries to deduce new rules in order to speed up the process. This is carried by the class *RuleUnifier* which tries to unify different rules according a code pattern. The decision whether perform a certain unification or not is done by a subclass of *Strategy* according to the additional information in *Metadata*. The rationale for this is that a certain unification can continue endlessly, are expensive to perform. This behaviour is typical of loops and functional call, whose unification is problematic.

The basic behaviour is to avoid unification. First of all we recall that the purpose of this task is to save verification time. Rule unification is performed in a local context with limited knowledge of the whole program. So it is possible that a loop which would perform fixed number of iterations in the global program would loop forever in the fragment considered.

Another issue of unification is the possibility of a performance loss. This can happen if too many rules are generated (so it takes longer to scan them) and if there are many rules which match the same configuration set not managed in a proper way, since an exponential number of new configuration sets can be generated.

Chapter 6

Validation

> *"It is a capital mistake to theorize before you have all the evidence. It biases the judgment."*
>
> Sir Arthur Ignatius Conan Doyle

In this chapter we report on the experimental evaluation of the tool. In Section 6.1, the tool is validated against a broad set of test cases from the official test suite of KernelC. The purpose of the tests is twofold: on one hand we test the compliance of our tool to the KernelC language, on the other we prove the correctness of its results by comparing with the original, non incremental, implementation. In Section 6.2, we instead focus on the performance of our tool and the impact of incrementality. To this purpose, we created several versions of the KernelC testsuite and used them as a benchmark to assess the impact of incrementality.

6.1 KernelC specifications compliance

The programs presented here have the purpose of verifying the compliance of our implementation of a KernelC reachability checking is with respect to the language specifications. We selected some examples from the ones provided with the official matching logic tool, MatchC, [1]; we present them in the next four subsections. The first group includes short code snippets using only the basic features of the language, whose purpose is to show how error detection works in KernelC. We then focus on more complex test cases aiming at stressing the capabilities of KernelC in proving program correctness. The second section contains simple programs with loops or recursive function calls. In the third section, programs contains also heap

management statements. Finally in the last section we show how checking properties involving the call stack state can be performed by our tool.

6.1.1 Error detection

We present this first four programs with the purpose of showing that our system is capable of recognizing various kinds of error.

The first one (shown in Listing 6.1) is contains a division by zero, which would lead to an undefined behavior. This kind of error is managed in our tool by adding special configurations which represent an error state. These configurations behave like a sink state: once they are reached it is not possible to move from them and reaching a final configuration. In this way the system can detect this sort of errors.

Listing 6.1: Division by zero

```
1 int main()
2 {
3   int x;
4   x = 0;
5   return 3/x;
6 }
```

The second one (shown in Listing 6.2) shows how the system checks that every variable used is initialized. According to the C language specification this is a legal program which would lead to an undefined behavior. However this behaviour is not allowed by KernelC.

This error is detected by the *env* part of configuration. When a variable is declared, it is marked as uninitialized. In this state it is not possible to retrieve the value of such variable. The verification fails since it is impossible to reach a final configuration, because the program flows is halted before the value of `x` is obtained.

Listing 6.2: Uninitialized variable

```
1 int main()
2 {
3   int x;
4   return x;
5 }
```

The third one (shown in Listing 6.3) contains an access to a memory address not allocated. A similar C program would crash after the dereferentiation of `x`. This error is recognized by the heap manager. The program flows is stuck in an error state configuration caused by the attempt of accessing an unallocated location.

Listing 6.3: Unallocated location

```
1 int main()
2 {
3   int *x;
4   x = (int *) 1000;
5   return *x;
6 }
```

The last one (shown in 6.4) shows what happens if an undefined memory location is read. The way in which this error is detected is similar to the previous one. The difference is that the error state is caused by the fact that the requested memory location is not initialized.

Listing 6.4: Uninitialized memory

```
1 struct listNode {
2   int val;
3   struct listNode *next;
4 };
5
6 int main()
7 {
8   struct listNode *x;
9   x = (struct listNode*) malloc(sizeof(struct listNode));
10  return x->next;
11 }
12
13 void* malloc(int size){
14   int c;
15   void* ptr;
16   if(size<1){
17     return 0;
18   }
19   ptr = sbrk();
20   c=0;
21   while(c<size){
22     brk(ptr);
23     c=c+1;
24   }
25   return ptr;
26 }
```

6.1.2 Correctness of simple programs

In this section we consider some simple programs, which also contain recursive functions and loops. When compared to the examples in the original MatchC distribution, these programs are different in the format of the an-

notations used to express the properties to check. We changed the format to adapt the annotations so that they could be used in conjunction with an SMT solver; the original one were meant to be used with Maude.

Our tool supports two types of annotation: one related to functions and one for defining a loop invariant; since the next examples deal with simple function verification, we explain first function annotations.

The purpose of a function annotation is to specify what that function is expected to do, i.e., what the verifier has to check about that function. This kind of annotation is divided in different parts. The first one declares which variables are used in subsequent parts. The first variable reported has a special meaning, since it identifies the return value of the function. The others have to match the parameters of the function. Then it is possible to have another part regarding heap, which we analyze later. Next there is the core of the annotation. There are two parts which represent respectively the precondition and postcondition. These are formed by two SMTLIB2 ([9]) strings enclosed between square brackets. The first states what is required to hold when the function is called. The second states what must hold when the function returns.

The first examples are simple programs that compute, respectively, the average of three numbers (Listing 6.5), the minimum (Listing 6.6) and the maximum (Listing 6.7) of two numbers.

Listing 6.5: Average of three numbers

```
1 int average(int x, int y, int z)
2 //@ <ret;x,y,z> [] [ ( assert (= ret (div (+ x y z ) 3) ) ) ]
3 {
4   int sum;
5   sum = x + y + z;
6   return sum / 3;
7 }
```

Listing 6.6: Minimum of two numbers

```
1 int minimum(int x, int y)
2 //@ <ret;x,y> [] [ ( assert (and (<= ret x ) (<= ret y ) ) ) ]
3 {
4   if (x < y) return x;
5   return y;
6 }
```

Listing 6.7: Maximum of two numbers

```
1 int maximum(int x, int y)
2 //@ <ret;x,y> [] [ ( assert (and (>= ret x ) (>= ret y ) ) ) ]
```

```
3 {
4   if (x < y) return y;
5   return x;
6 }
```

We now move to case of recursive functions. Recursive functions are harder to handle since it is not possible to unroll the unlimited sequence of recursive calls. So the trick used is the one presented in Section 2.2.3, regarding the matching logic proof system: we perform only the first call of a recursive function. Then we substitute to the subsequent calls the definition of the function which is provided by the annotation.

This technique is very powerful but has one drawback: it provides only partial correctness proofs. So each proof of this kind only guarantees that if the function returns the postcondition is satisfied. It does not guarantee function call termination.

Listing 6.8 and 6.9 show program with recursive function. The former is a simple function which computes a multiplication between two numbers which is performed as a sequence of summations. The latter instead computes the summation of the first n numbers.

Listing 6.8: Multiplication performed as a series of additions

```
1 int multiplication_by_addition(int x, int y)
2 //@ <ret;x,y> [] [ ( assert (= ret (* x y ) ) ) ]
3 {
4   if (x == 0) return 0;
5   if (x < 0) return -multiplication_by_addition(-x,y);
6   return y + multiplication_by_addition(x - 1, y);
7 }
```

Listing 6.9: Sum of the first n numbers with recursion

```
1 int sum_recursive(int n)
2 //@ <ret;n> [( assert (>= n 0) )] [ ( assert (= ret (div (* n (+ n 1) ) 2) ) ) ]
3 {
4   if (n <= 0) return 0;
5   return n + sum_recursive(n-1);
6 }
```

Starting from the last example provided, we can rewrite it in an iterative way (Listing 6.10). In order to present this new program, we introduce a loop invariant. A loop invariant is the same concept introduced while presenting Hoare logic in Section 2.1.1: we have to define what always holds after each iteration of the loop. We do this using a specific kind of annotations. The format of these annotations is similar to those related to functions. The first part is devoted to variable declarations. The first one has no special meaning since there is no return value. Then we have two parts enclosed within square brackets. The first is an SMT string and declares the invariant. The second is a list of variables. These one are the variables which do not change during loop iterations.

Listing 6.10: Sum of the first n numbers with iteration

```
1  int sum_iterative(int n)
2  //@ <ret;n> [( assert (>= n 0) )] [ ( assert (= ret (div (* n (+ n 1) ) 2) ) ) ]
3  {
4    int s;
5    int i;
6
7    s = 0;
8    i = n;
9
10   /*@ <i,n,s>
11       [ (assert (and (= s (div (* (- i n ) (+ i n 1 )) 2)) (>= n 0 ) (>= i n )) )]
12       [i] */
13   while (n > 0) {
14     s += n;
15     n -= 1;
16   }
17
18   return s;
19 }
```

We end this section by showing an unannotated function (Listing 6.11). Its properties (in this case the fact of being associative and commutative) have to be extracted by the SMT solver from how the return value is formed.

Listing 6.11: Verification of function properties

```
1  int f(int x, int y)
2  {
3    return x + y + x*y;
4  }
5
6  int comm_assoc(int x, int y, int z)
7  //@ <ret> [] [ ( assert (= ret 1) ) ]
8  {
9    return f(x,y) == f(y,x) && f(x,f(y,z)) == f(f(x,y),z);
10 }
```

6.1.3 Heap management

Now we want use our system to verify programs which use the heap. As we can immediately notice, the annotations used are much more complex. However the only change in their structure is the presence of the variables which specify the heap. This new part of the annotation is composed by eight variables. The first four variables identify the heap at the beginning of the function while the other four identify the heap at the end. We recall that heap is characterized by four variables of which the first three are array of integers and the fourth is an integer variable. The first is the *heap* and contains the actual heap data mapped to a certain position. The second is the *memory map* which contains a value different from zero for the cells allocated. The third is the *declaration map* and contains a value different from zero for the cells whose data are initialized. The fourth contains the position which will be the next allocated in memory.

To verify programs based on lists we declare the FOL function `isList` which represents a function starting from the position given as argument. Then we give some properties of lists regarding the structure of the memory. In particular we state that a list pointer must be non-negative and if it is strictly positive, then the node of the list is correctly allocated in memory and defined. Moreover all these properties are valid for the next node, if the list pointer is strictly positive. However the function given is a particular case of general list properties, since the heap variables are not parameters of the FOL function, but are hard-coded in its definition. The reason for this is that Z3 has some limitations in managing quantifiers.

In the first program (Listing 6.12), we analyze a function which returns the first element of a list. We declare that the passed parameter is a list and the value returned must satisfy the definition of the function. In addition the heap must not change.

Listing 6.12: Retrieve first element of a list

```
1  struct listNode {
2    int val;
3    struct listNode *next;
4  };
5
6
7  int head(struct listNode *x)
8  /*@ <ret;x> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
9    (declare-fun isList( Int ) Bool )
10   (assert (and
11   (forall ((i Int)) (=> (and (isList i) (> i 0))
12     (isList (select heap (+ i 1))) ))
13   (forall ((i Int)) (=> (and (isList i) (> i 0))
14     (> pos (+ i 1 ) )))
15   (forall ((i Int)) (=> (and (isList i) (> i 0))
16     (and ( = 1 (select dec i ) ) ( = 1 (select dec (+ i 1 ) ) ))) )
17   (forall ((i Int)) (=> (and (isList i) (> i 0))
18     (and ( > 0 (select mm i ) ) ( > 0 (select mm (+ i 1 ) ) ))) )
19   (forall ((i Int)) (=> (isList i) (>= i 0)))
20     (and (isList x ) (> x 0 ) )
21   )) ]
22 [
23   (declare-fun isList( Int ) Bool )
24   (assert (and
25   (= ret (select heap x ) )
26   (= heap heap1 )
27   (= mm mm1 )
28   (= dec dec1 )
29   (= pos pos1 )
30 ))] */
31 {
32   return x->val;
33 }
```

The next program (shown in Listing 6.13) is very similar to the previous one. The only difference is that this one returns the rest of the list instead of the first element. The annotation is very similar the the previous one, the only change is due to the different definition of the function.

Listing 6.13: Retrieve the tail of a list

```
1  struct listNode {
2    int val;
3    struct listNode *next;
4  };
5
6
7  struct listNode* tail(struct listNode *x)
```

```
8  /*@ <ret;x> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
9    (declare-fun isList( Int ) Bool )
10   (assert (and
11   (forall ((i Int)) (=> (and (isList i) (> i 0))
12     (isList (select heap (+ i 1))) ))
13   (forall ((i Int)) (=> (and (isList i) (> i 0))
14     (> pos (+ i 1 ) )))
15   (forall ((i Int)) (=> (and (isList i) (> i 0))
16     (and ( = 1 (select dec i ) ) ( = 1 (select dec (+ i 1 ) ) ))) )
17   (forall ((i Int)) (=> (and (isList i) (> i 0))
18     (and ( > 0 (select mm i ) ) ( > 0 (select mm (+ i 1 ) ) ))) )
19   (forall ((i Int)) (=> (isList i) (>= i 0)))
20     (and (isList x ) (> x 0 ) )
21   )) ]
22 [
23   (declare-fun isList( Int ) Bool )
24   (assert (and
25   (= ret (select heap (+ x 1) ) )
26   (= heap heap1 )
27   (= mm mm1 )
28   (= dec dec1 )
29   (= pos pos1 )
30 ))] */
31 {
32   return x->next;
33 }
```

The next example (shown in Listing 6.14) is more interesting, because the heap is also modified. It performs the addition of a new element which is added at top of the list. Indeed the new element is allocated and initialized. In order to capture this the annotation has been slightly modified by allowing only the allocation of the new element. In addition the `isList` FOL function has been removed since not used in this scenario. A peculiar condition is the fact that the position of the heap must be strictly positive, since zero is regarded as an always unallocated cell. Although this condition is automatically added at the entry point of the program, this must be specified as precondition in function proofs.

Listing 6.14: Addition of a new element at the top of the list

```
1 struct listNode {
2   int val;
3   struct listNode *next;
4 };
5
6
7 struct listNode* add(int v, struct listNode* x)
8 /*@ <ret;x,v> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
```

```
 9   (assert (and
10   (> pos 0 ) )) ]
11 [
12   (assert (and
13   (= pos1 (+ pos 2) )
14   (= ret pos )
15   (forall ((i Int)) (=> (and (> i 0) (< i pos )) (= (select mm i ) (select mm1 i ) )))
16   (forall ((i Int)) (=> (and (> i 0) (< i pos )) (= (select dec i ) (select dec1 i ) )))
17   (forall ((i Int)) (=> (and (> i 0) (< i pos )) (= (select heap i ) (select heap1 i ) )))
18   (= x (select heap1 (+ ret 1) ) )
19   (= v (select heap1 ret ) )
20   ))] */
21 {
22   struct listNode* y;
23
24   y = (struct listNode*) malloc (sizeof(struct listNode));
25   y->val = v;
26   y->next = x;
27
28   return y;
29 }
30
31 void* malloc(int size){
32   int c;
33   void* ptr;
34   if(size<1){
35     return 0;
36   }
37   ptr = sbrk();
38   c=0;
39   while(c<size){
40     brk(ptr);
41     c=c+1;
42   }
43   return ptr;
44 }
```

In the next example we verify the swapping between the first two values in a list.

Listing 6.15: Swapping of two values

```
1 struct nodeList {
2   int val;
3   struct nodeList *next;
4 };
5
6
7 struct nodeList* swap(struct nodeList* x)
8 /*@ <ret;x> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
```

```
9  (declare-fun isList( Int ) Bool )
10 (assert (and
11 (> (select heap (+ 1 x ) ) 0)
12 (forall ((i Int)) (=> (and (isList i) (> i 0)) (isList (select heap (+ i 1))) ))
13 (forall ((i Int)) (=> (and (isList i) (> i 0)) (> pos (+ i 1 ) )))
14 (forall ((i Int)) (=> (and (isList i) (> i 0))
15   (and ( = 1 (select dec i ) ) ( = 1 (select dec (+ i 1 ) ) ))) )
16 (forall ((i Int)) (=> (and (isList i) (> i 0))
17   (and ( > 0 (select mm i ) ) ( > 0 (select mm (+ i 1 ) ) ))) )
18 (forall ((i Int)) (=> (isList i) (>= i 0)))
19 (isList x )
20 (> x 0 ) )) ]
21 [
22 (assert (and
23 (= ret (select heap (+ x 1) ))
24 (= x (select heap1 (+ ret 1 )))
25 (= pos1 pos )
26 (forall ((i Int)) (=> (and (> i 0) (< i pos )) (= (select mm i ) (select mm1 i ) )))
27 (forall ((i Int)) (=> (and (> i 0) (< i pos )) (= (select dec i ) (select dec1 i ) )))
28 (forall ((i Int)) (=> (and (> i 0) (< i pos ) (not (= i (+ x 1) )) (not (= i (+ ret 1)
       ))) (= (select heap i ) (select heap1 i ) )))
29 ))] */
30 {
31   struct nodeList* p;
32
33   p = x;
34   x = x->next;
35   p->next = x->next;
36   x->next = p;
37
38   return x;
39 }
```

The last two programs are more complex. We want to verify a program (shown in Listing 6.16) which returns the length of a list and another one (shown in Listing 6.17) which returns the sum of all the values in a list.

The additional complexity lies in the recursive nature of the functions. To catch these behaviours the FOL functions `length` and `sum` are defined such that the former is equal to the length of the list and the latter is equal to the sum of all the values of the list.

Listing 6.16: Program which retrieves the length of a list

```
1 struct listNode {
2   int val;
3   struct listNode *next;
4 };
5
6
```

```
 7 int length_recursive(struct listNode* x)
 8 /*@ <ret;x> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
 9   (declare-fun isList( Int ) Bool )
10   (assert (and
11   (forall ((i Int)) (=> (and (isList i) (> i 0)) (isList (select heap (+ i 1))) ))
12   (forall ((i Int)) (=> (and (isList i) (> i 0)) (> pos (+ i 1 ) )))
13   (forall ((i Int)) (=> (and (isList i) (> i 0))
14     (and ( = 1 (select dec i ) ) ( = 1 (select dec (+ i 1 ) ) ))) )
15   (forall ((i Int)) (=> (and (isList i) (> i 0))
16     (and ( > 0 (select mm i ) ) ( > 0 (select mm (+ i 1 ) ) ))) )
17   (forall ((i Int)) (=> (isList i) (>= i 0)))
18   (isList x )
19   )) ]
20 [
21   (declare-fun length( Int ) Int )
22   (assert
23   (=> (and
24   (forall ((i Int)) (=> (= i 0) (= (length i ) 0) ))
25   (forall ((i Int)) (=> (> i 0)
26     (= (length i ) (+ (length (select heap (+ i 1) ) ) 1) ) ))
27   ) (and
28   (= heap1 heap )
29   (= mm1 mm )
30   (= dec1 dec )
31   (= pos1 pos )
32   (= ret (length x ) )
33   ))
34 )] */
35 {
36   if (x == 0)
37     return 0;
38 
39   return 1 + length_recursive(x->next);
40 }
```

Listing 6.17: Program which retrieves the sum of values of a list

```
1  struct listNode {
2    int val;
3    struct listNode *next;
4  };
5
6
7  int sum_recursive(struct listNode* x)
8  /*@ <ret;x> <heap,mm,dec,pos,heap1,mm1,dec1,pos1> [
9    (declare-fun isList( Int ) Bool )
10   (assert (and
11   (forall ((i Int)) (=> (and (isList i) (> i 0))
12     (isList (select heap (+ i 1))) ))
13   (forall ((i Int)) (=> (and (isList i) (> i 0))
14     (> pos (+ i 1 ) )))
15   (forall ((i Int)) (=> (and (isList i) (> i 0))
16     (and ( = 1 (select dec i ) ) ( = 1 (select dec (+ i 1 ) ) ))) )
17   (forall ((i Int)) (=> (and (isList i) (> i 0))
18     (and ( > 0 (select mm i ) ) ( > 0 (select mm (+ i 1 ) ) ))) )
19   (forall ((i Int)) (=> (isList i) (>= i 0)))
20   (isList x )
21   )) ]
22 [
23   (declare-fun sum( Int ) Int )
24   (assert
25   (=> (and
26   (forall ((i Int)) (=> (= i 0) (= (sum i ) 0) ))
27   (forall ((i Int)) (=> (> i 0)
28     (= (sum i ) (+ (sum (select heap (+ i 1) ) ) (select heap i )) ) ))
29   ) (and
30   (= heap1 heap )
31   (= mm1 mm )
32   (= dec1 dec )
33   (= pos1 pos )
34   (= ret (sum x ) )
35   ))
36 )] */
37 {
38   if (x == NULL)
39     return 0;
40
41   return x->val + sum_recursive(x->next);
42 }
```

6.1.4 Call stack

These two programs show the feature of checking the call stack of a function. In the first one (shown in Listing 6.18) the function `f` can only be called by `g`, while in the second one (shown in Listing 6.19) at the moment of the call to `f`, `h` must be in the stack.

To specify this kind of properties the function annotation has been extended with an additional field. This new field specifies a list of admissible stack records. The exclamation mark means that one of these records must be found considering only the last one, thus it means that such function must be called only from the functions provided. In all previous cases, our tool was able to process the input program and successfully perform the verification.

Listing 6.18: f can be called only from g

```
1  void f()
2  //@<ret> [(assert true)] [(assert true)] !<g>
3  {
4  }
5  void g(){
6    f();
7  }
8  void h(){
9    g();
10   g();
11 }
12 int main(){
13   h();
14 }
```

Listing 6.19: f can be called only if h is in the stack

```
1  void f()
2  //@<ret> [(assert true)] [(assert true)] <h>
3  {
4  }
5  void g(){
6    f();
7  }
8  void h(){
9    g();
10 }
11 int main(){
12   h();
13   h();
14 }
```

6.2 Evaluation of the incremental approach

Until now we have shown that we can verify the correct behavior of a program according to the language specification. We now evaluate the benefits provided by our incremental approach with respect to the non-incremental case.

We selected different programs from the ones we have analyzed so far. Starting from a very simple program composed only by the list definition and the `malloc` function, we add these programs one by one and perform the verification with our tool, simulating the possible evolution of a program to which new function definitions are added over time. For each program, we performed verification both incrementally and non-incrementally.

The programs we selected are:

- The functions `maximum`, `minimum`, `average`, `sumIterative`, `sumRecursive`, `multiplicationByAddition` from the basic examples.
- The functions `head`, `tail`, `lengthRecursive`, `sumRecursive`, `add` from the heap ones.
- The program `hInStackWhenF` from the call stack ones.

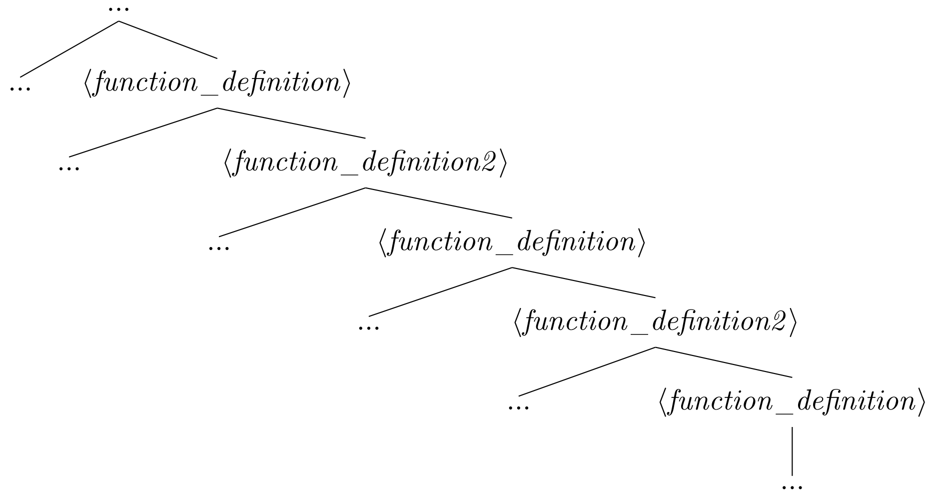
Since performances in a syntax driven incrementality approach vary depending on how the input is arranged we perform tests under different conditions:

1. each new function is added at the end of the previous program.
2. each new function is added before the previous ones (but after the list definition).
3. each new function is added at the beginning of the previous program.

6.2.1 Scenario 1

This is the worst-case from an incremental point of view, due to the bottom-up parser used: for every new function added all the previous function definition nodes has to be revisited in the attribute grammar tree. To better explain this fact we show the KernelC syntax tree in Figure 6.1. Each $\langle \mathit{function_definition} \rangle$ non-terminal represents a certain function where the majority of the verification is performed. The problem is that $\langle \mathit{function_definition2} \rangle$ which contains the body of the function has as a child a $\langle \mathit{function_definition} \rangle$

Figure 6.1: Detail of KernelC syntax tree. The chain of function definitions begins at the top-left of the figure and end at the bottom-right



non-terminal. This forms a long chain which connects all the function definitions of KernelC language. If we add a new node at the end of the chain, all the previous one have to be recomputed. Then, since the computations performed in ⟨*function_definition*⟩ nodes are quite expensive, this impact negatively on the performance of the incremental method.

However in order to reduce this penalty, an optimization has been inserted. The computation of the attributes at ⟨*function_definition*⟩ nodes is split in two parts. In the first one function is checked by only using the information available from the body of the function. In the second stage the information coming from the other children of the node is used. In the case that the changes are only outside the function, the first step is re-used.

For these reasons we expect that the duration of the non-incremental verification will constantly increase, since we are adding more entities to compute every time. On the other hand the incremental ones will show different times which will depend on the complexity of the verification. So we expect that the performance of the checks related to `maximum`, `minimum` and `average` will be low, while `sumIterative`, `sumRecursive` and `multiplicationByAddition` a little higher and the ones regarding the heap will be more expensive in terms of time.

In addition we have to consider the positions of the list and `malloc` function definitions. This has a big impact on the heap examples. If we refer again to Figure 6.1, we must imagine these definitions are placed above all other nodes. To complete the verifications which involve the use of that

Table 6.1: Results of scenario 1 tests (average of ten measures)

	Non-incremental	**Incremental**
average	81ms	38ms
+sumIterative	155ms	87ms
+sumRecursive	188ms	97ms
+multiplicationByAddition	283ms	117ms
+hInStackWhenF	302ms	92ms
+maximum	316ms	64ms
+minimum	341ms	73ms
+head	384ms	108ms
+tail	513ms	137ms
+add	767ms	442ms
+listLengthRecursive	1084ms	557ms
+listSumRecursive	1242ms	684ms

structure or `malloc` function, it will be necessary to reach the root node. Therefore we expect that from the addition of `head` on, also the time of incremental executions will grow continuously.

Table 6.1 shows the results of this test. We can notice that both incremental and non-incremental tests behave as predicted. In particular we can observe for the incremental ones that the timings reflect more or less the complexity of the property to check.

From the `head` function on, the predicted increase in incremental duration is observable. This result shows that the incremental method used is always more efficient than the non-incremental one. Figure 6.2 gives a better comparison between the incremental and non-incremental performances.

We notice that the speed gain obtained through incrementality is quite consistent. The speedup obtained by incrementality ranges from 174% obtained when adding `add` to 494% obtained after adding `maximum`, with an average of 227%.

For the sake of comparison we provide in Table 6.2 the result of the same scenario performed without the optimization of the two steps function evaluation, which has been introduced to overcome the problem related to the peculiar structure of KernelC operator-precedence grammar.

In Figure 6.3, we compare the three different executions (non-incremental, non-optimized incremental, optimized incremental). The results are pretty interesting, showing an averaged speedup of 150% obtained through optimization.

Figure 6.2: Graphical representation of scenario 1 tests results

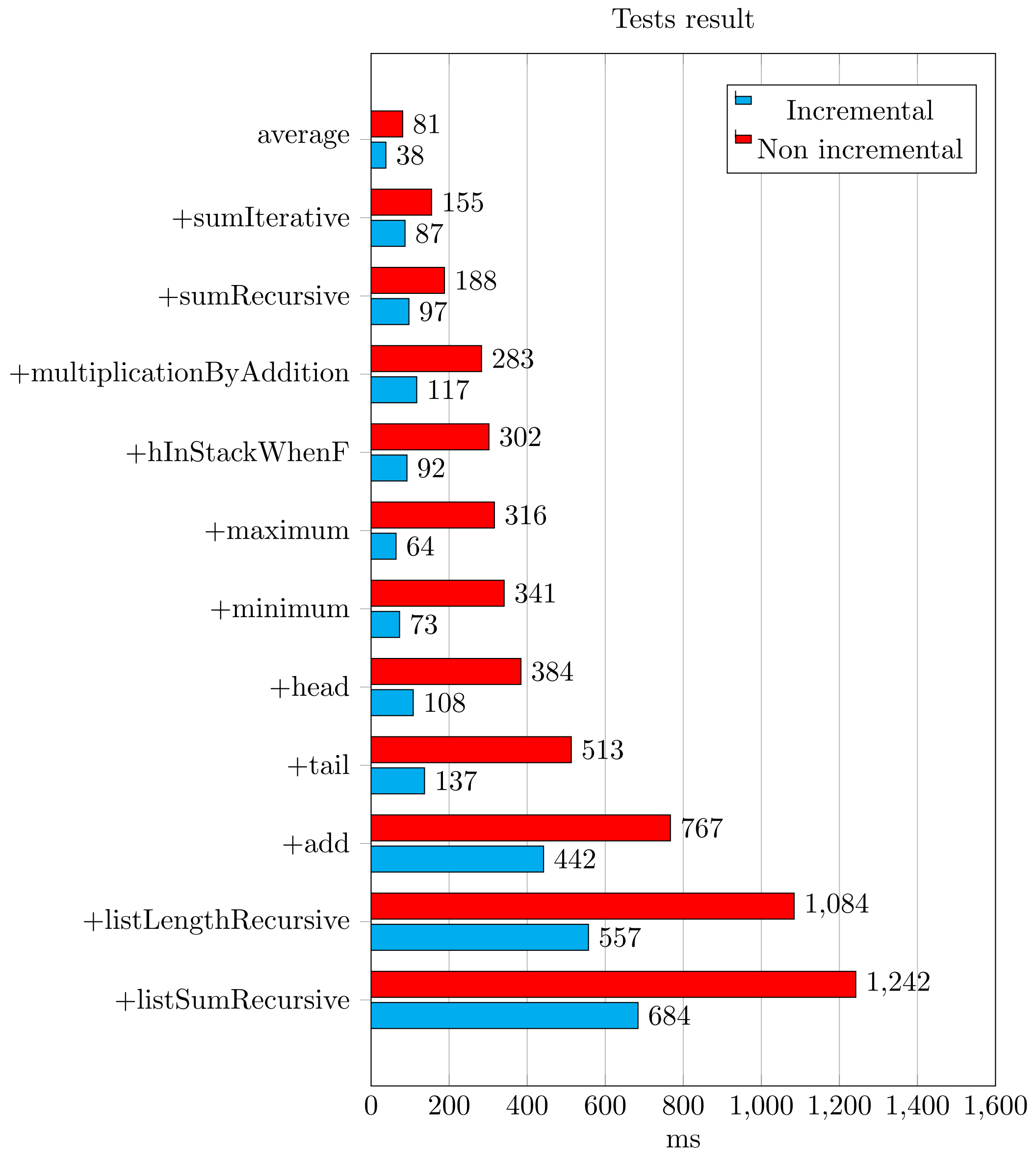


Table 6.2: Comparison between scenario 1 results (average of ten measures)

	From scratch	**Non-optimized**	**Optimized**
average	81ms	41ms	38ms
+sumIterative	155ms	98ms	87ms
+sumRecursive	188ms	96ms	97ms
+multiplicationByAddition	283ms	157ms	117ms
+hInStackWhenF	302ms	153ms	92ms
+maximum	316ms	158ms	64ms
+minimum	341ms	167ms	73ms
+head	384ms	213ms	108ms
+tail	513ms	261ms	137ms
+add	767ms	595ms	442ms
+listLengthRecursive	1084ms	858ms	557ms
+listSumRecursive	1242ms	938ms	684ms

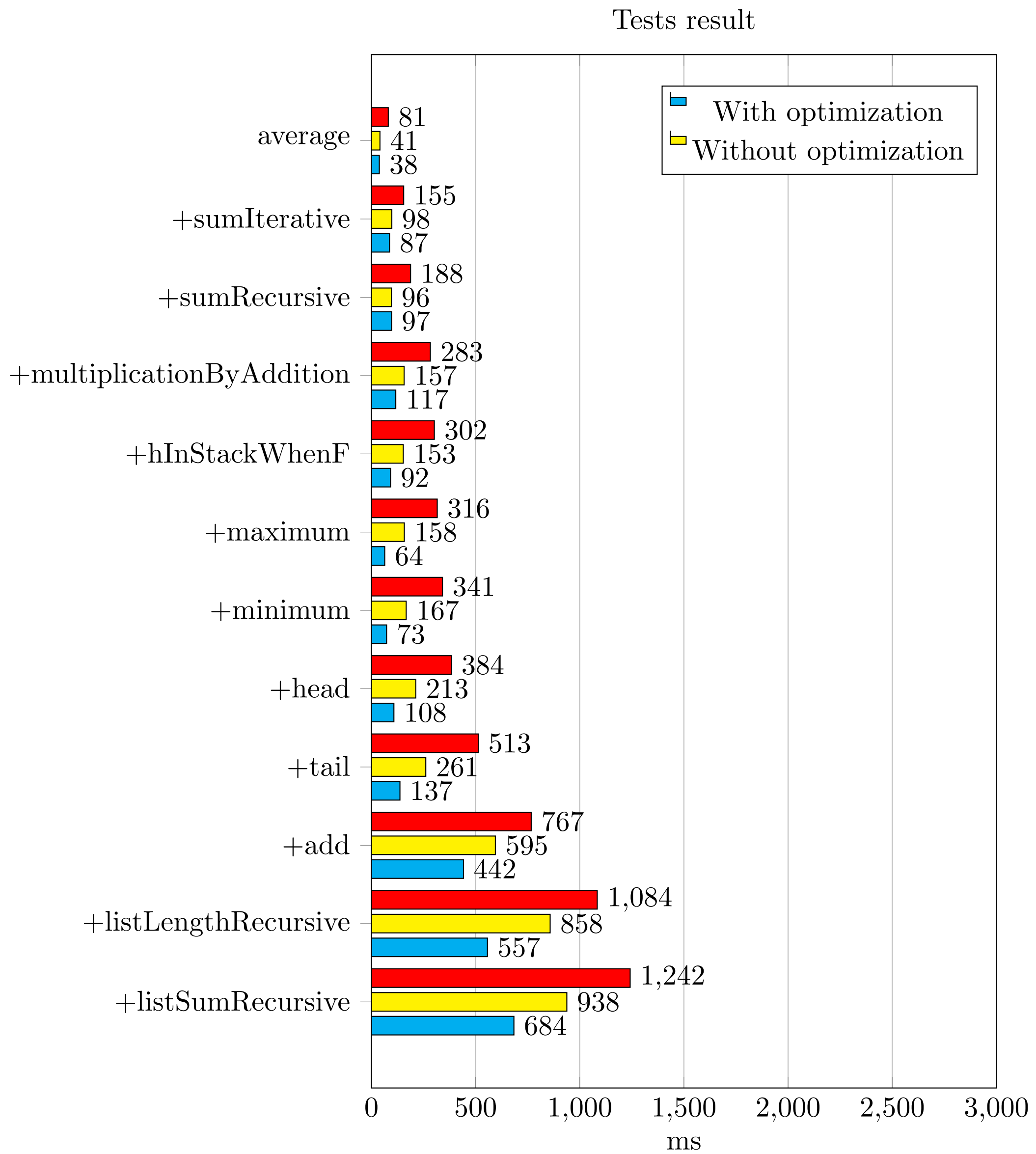
We expect that the results in next section runs should be better than these ones. However we can say from now that the syntactic-semantic incrementality can greatly improve the performance of reachability checking.

6.2.2 Scenario 2

Now we analyze the case in which every new function will be added at the top of the programs but after the list definition. We can make the same considerations we have done before: the non-incremental runs will always increase while the incremental ones will depend on the complexity of the newly added function. Again we expect that starting from the programs which make use of the list definition, the incremental times will begin to continuously increase, since a part of the check for each function has to be deferred in a higher position in the tree and so will take less advantage of incrementality. In addition we expect that these results will be much better, since incremental methods are working in a more favourable condition. We present the test results in Table 6.3, with another graphical representation in Figure 6.4.

Again we can observe that our predictions are met: the overall behaviour is similar to the previous one, with continuous increasing in non-incremental runs and from `head` example of incremental ones. If we compute the speedup, we obtain values between 132% of `listSumRecursive` and 1296% of `minimum`, with an average of 253%. These values are clearly better than the previous ones, meaning that program structure has surely

Figure 6.3: Graphical comparison of scenario 1 tests results



an effect in the performance of our method.

Figure 6.4: Graphical representation of scenario 2 tests results

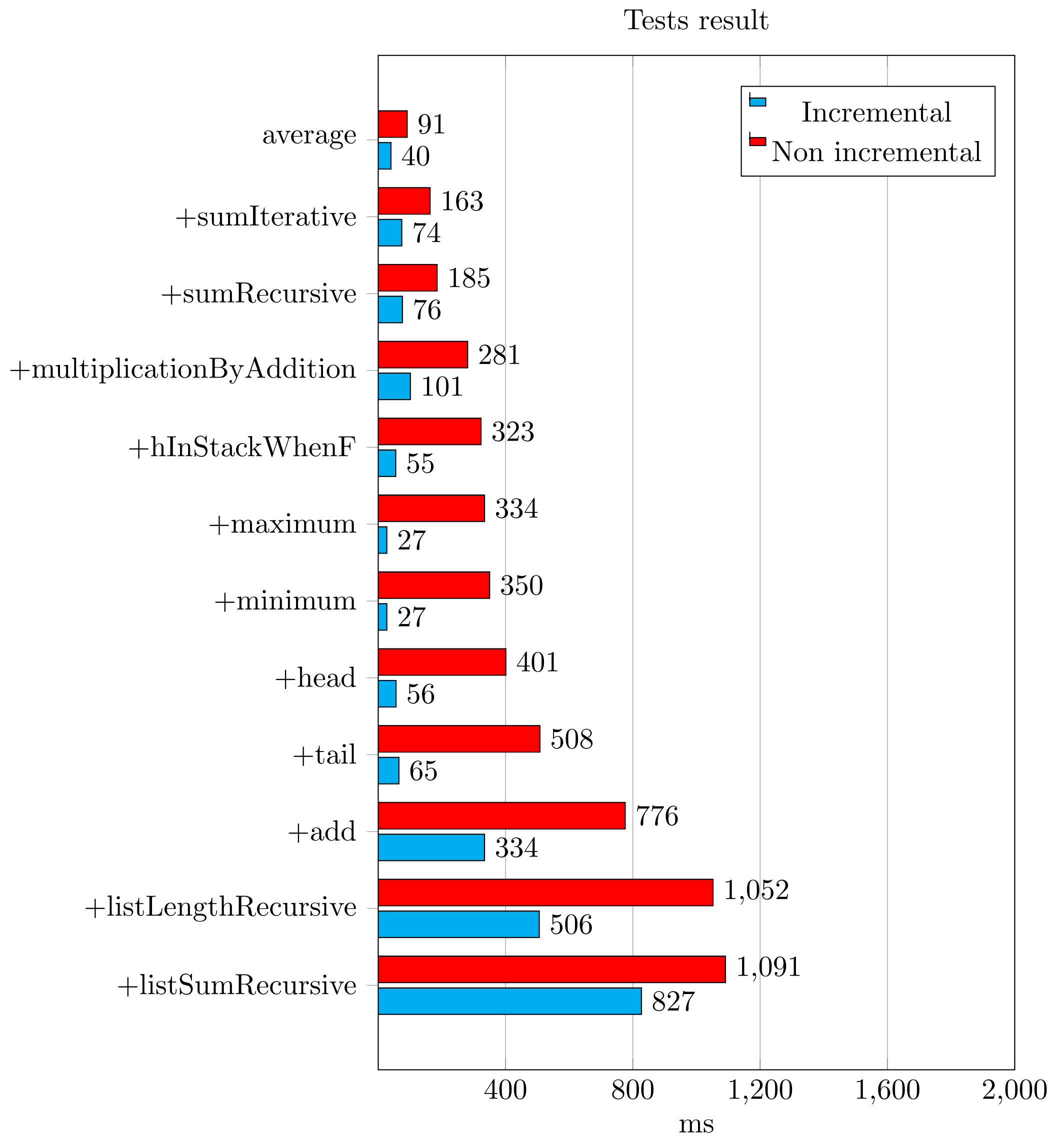


Table 6.3: Results of scenario 2 tests (average of ten measures)

	Non-incremental	**Incremental**
average	91ms	40ms
+sumIterative	163ms	74ms
+sumRecursive	185ms	76ms
+multiplicationByAddition	281ms	101ms
+hInStackWhenF	323ms	55ms
+maximum	334ms	27ms
+minimum	350ms	27ms
+head	401ms	56ms
+tail	508ms	65ms
+add	776ms	334ms
+listLengthRecursive	1052ms	506ms
+listSumRecursive	1091ms	827ms

Table 6.4: Results of scenario 3 tests (average of ten measures)

	Non-incremental	**Incremental**
average	100ms	28ms
+sumIterative	151ms	63ms
+sumRecursive	193ms	72ms
+multiplicationByAddition	301ms	95ms
+hInStackWhenF	346ms	55ms
+maximum	349ms	27ms
+minimum	367ms	25ms
+head	387ms	50ms
+tail	477ms	53ms
+add	761ms	344ms
+listLengthRecursive	1086ms	149ms
+listSumRecursive	1454ms	386ms

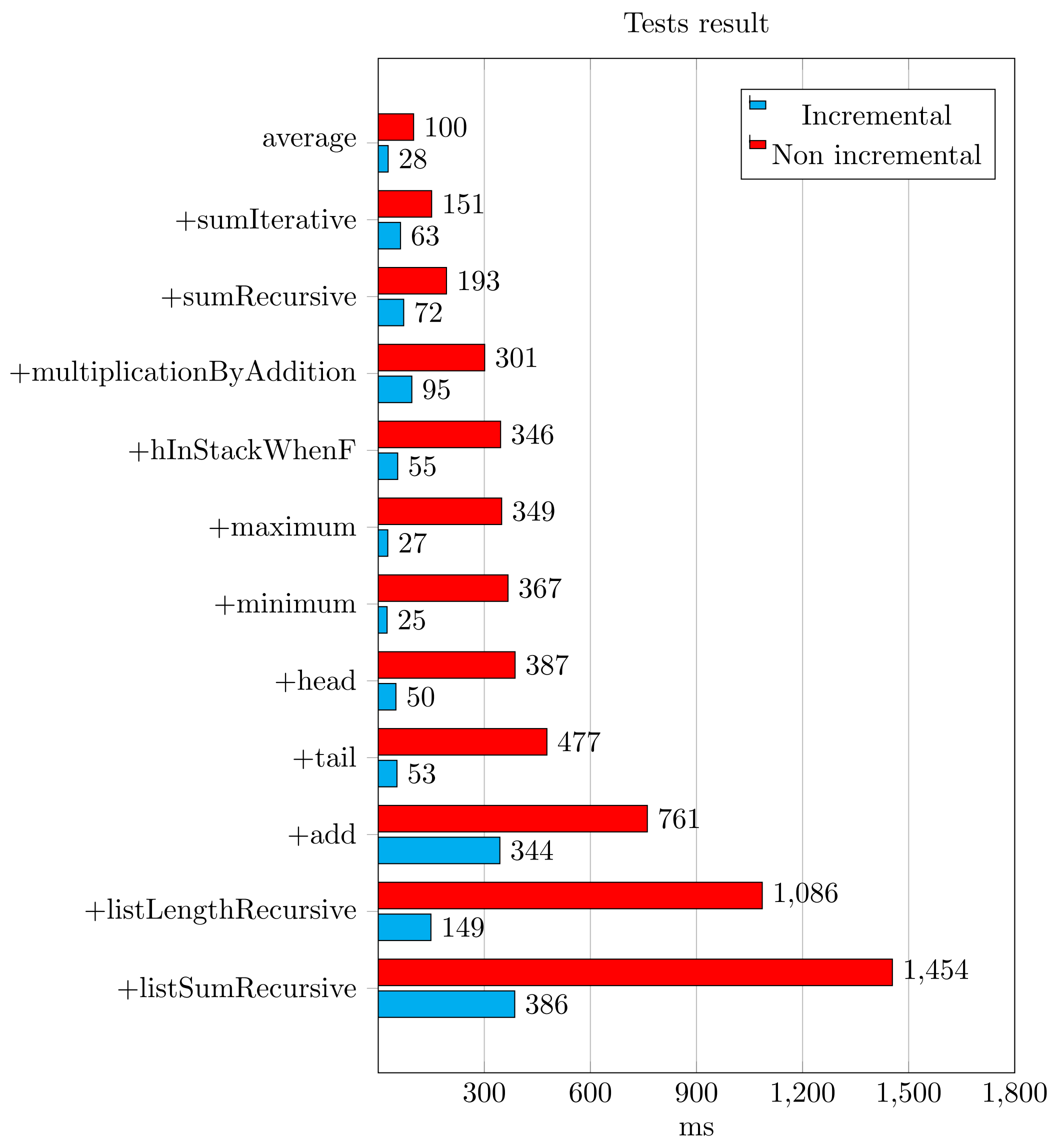
6.2.3 Scenario 3

In this scenario we try to evaluate the gain in case of adding a new function above the whole previous program. In this way, referring to Figure 6.1, we can imagine to add a new function at the top-left part of the tree. So the attributes to be recomputed are very few. For these reason while expecting the same behaviour for non-incremental executions, which should monotonically increase, we expect something different for the incremental ones.

In fact, since all the required information is known when the new function is visited, we no longer expect to see the accumulation from the heap examples on. So we imagine that incremental executions will only be characterized by the complexity of each check, from the simplest, like `maximum` or `minimum`, passing through more complex ones, like `sumIterative` and `multiplicationByAddition` to reach the most complex ones, which involve the heap. Of course we expect to have better results than the previous case since the number of operations to be performed is the lowest possible.

In Table 6.4 we show the results, also plotted in Figure 6.5.

Figure 6.5: Graphical representation of scenario 3 tests results



The resulting speedup is clearly better than before: we obtain a speedup which ranges from 221% of `add` to 1468% of `minimum` with an average of 443%. However this represents the best possible case, where incrementality can perform at its best.

6.2.4 Summing up

As we have seen, the use of a syntactic-semantic incremental methodology in reachability checking can have a significant impact on performance. This gain can be reduced by the way in which the changes to the program are conducted; the program structure determines how the amount of the reusable part of previous verification results. Finally, we have seen that the complexity of the verification along with the dependency inside the code (in the form of function calls or structure definitions) can determine the amount of time required for the incremental verification task.

Chapter 7

Conclusions

> *"One's destination is never a place, but rather a new way of looking at things."*
>
> Henry Valentine Miller

In this work we have applied a general syntactic-semantic incremental technique for performing reachability checking of KernelC programs annotated with matching logic properties. This technique can efficiently check a new version of a program by reusing the results of the previous verification step related to unchanged parts. This approach is based on the syntactic structure of the target program by exploiting the properties of operator-precedence grammars, which we have presented in chapter 2. We have developed our work targeting the language KernelC, a subset of C language, which supports advanced features like heap.

he core of this work consisted in the adaptation of a general method for program verification provided by matching logic according to a syntactic-semantic incremental technique based on attribute grammars. In chapter 4 we have presented the general idea which we followed in the development of this problem. We have formulated the matching logic reachability system as an S-attributed grammar.

In chapter 6 we have evaluated our implementation. First of all we have verified that it is compliant with KernelC specification, by applying some examples provided by the official implementation of matching logic. Then we have measured the gain in performance obtained through incrementality. We found that in every case analyzed with our implementation the incremental verification is significantly faster than the non-incremental version.

7.1 Future work

As part of future work, we plan to perform some improvements of the supported language features, for example by adding I/O management in verification. Other possible improvements are related to: user interaction, by improving the annotations format in order to be more understandable and possibly compatible with the original implementation, and providing more information about the checking which has been performed and the problems encountered. Furthermore, a more thorough assessment of the benefits derived from incrementality is needed. This task is quite hard because it is difficult to find situations which represent the typical scenario of software development. In this work we have analyzed only the addition of different functions, whereas different situations are possible and incremental performance can greatly vary. Moreover the example provided has been quite simple and cannot represent the behaviour of a complex situation. The major issue is that KernelC cannot be use in a real world environment and thus analyze its performance.

Broader developments regard the generalization of this implementation in order to create a general framework for incremental verification. The idea is to provide a way which can automatically generate an incremental verifier for a certain language from its specifications.

Other progress in this area can affect the context in which incremental verification techniques can be used. Besides agile software development, another field which can greatly benefit of such verification technique is "open world software" [3],[6]. In this paradigm a program is composed by different independent pieces outside of it and out of its control. These components can change over time. In order to permit this, continuous software verification is required. In addition, since this software can run on limited-power device, the verification method to be used must have low execution times. Therefore incremental verification could have an important role in the development of this kind of software.

Bibliography

[1] MatchC Verifier.
`http://fsl.cs.illinois.edu/index.php/Special:MatchCOnline`.

[2] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[3] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, 2006.

[4] Domenico Bianculli, Antonio Filieri, Carlo Ghezzi, and Dino Mandrioli. A syntactic-semantic approach to incremental verification. *CoRR*, abs/1304.8034, 2013.

[5] Barry Boehm. *Software risk management.* Springer, 1989.

[6] Radu Calinescu, Carlo Ghezzi, Marta Kwiatkowska, and Raffaela Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, September 2012.

[7] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.6), 2011.

[8] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'03, pages 331–346, Berlin, Heidelberg, 2003. Springer-Verlag.

[9] David R. Cok. The smt-libv2 language and tools: A tutorial. Technical report, Technical report, GrammaTech, Inc, 2011.

[10] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for inter-procedural anal-

ysis of safety properties. In *Proceedings of the 17th international conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 449–461. Springer, 2005.

[11] Stefano Crespi Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. In *Proceedings of the 4th international conference on Language and Automata Theory and Applications*, LATA'10, pages 214–226, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] Conrado Daws. Symbolic and parametric model checking of discrete-time markov chains. In *Proceedings of the 1st international conference on Theoretical Aspects of Computing*, ICTAC'04, pages 280–294, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems, Theory and practice of software*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] Salvatore Distefano, Antonio Filieri, Carlo Ghezzi, and Raffaela Mirandola. A compositional method for reliability analysis of workflows affected by multiple failure modes. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*, CBSE '11, pages 149–158, New York, NY, USA, 2011. ACM.

[15] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd international conference on Software Engineering*, ICSE '11, pages 341–350, New York, NY, USA, 2011. ACM.

[16] Cormac Flanagan and Shaz Qadeer. Assume-guarantee model checking. Technical report, 2003.

[17] Carlo Ghezzi. Evolution, adaptation, and the quest for incrementality. In Radu Calinescu and David Garlan, editors, *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *Lecture Notes in Computer Science*, pages 369–379. Springer Berlin Heidelberg, 2012.

[18] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandioli. Fundamentals of software engineering, 2003.

[19] Carlo Ghezzi and Dino Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, January 1979.

[20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. *Verification: Theory and Practice*, pages 180–181, 2004.

[21] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[22] Fahimeh Jalili. A general incremental evaluator for attribute grammars. *Sci. Comput. Program.*, 5(1):83–96, February 1985.

[23] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[24] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[25] Marta Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Assume-guarantee verification for probabilistic systems. In J. Esparza and R. Majumdar, editors, *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6105 of *LNCS*, pages 23–37. Springer, 2010.

[26] Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 291–300, New York, NY, USA, 2008. ACM.

[27] Narciso Martı-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[28] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing*. Wiley, second edition, June 2004.

[29] Jukka Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.

[30] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. In *Proceedings of the 27th conference on Object-Oriented Programming, Systems, Languages, and Applications (OOP-SLA'12)*, pages 555–574. ACM, 2012.

[31] Grigore Roşu and Andrei Ştefănescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th international symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.

[32] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th symposium on Logic in Computer Science (LICS'13)*. IEEE, June 2013.

[33] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Michael Johnson and Dusko Pavlovic, editors, *Proceedings of the 13th international conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, 2010.

[34] Oleg V. Sokolsky and Scott A. Smolka. Incremental model checking in the modal mu-calculus. In *Proceedings of the 6th international conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 351–363. Springer, 1994.

[35] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. Regression model checking. In *Proceedings of the 25th IEEE international conference on Software Maintenance (ICSM'09)*, pages 115–124. IEEE, 2009.

Appendix A

KernelC operator-precedence grammar

In this appendix we present the complete KernelC grammar, modified in order to become an operator-precedence grammar.

The non-terminals ending with ‘Choice’ has been added just to provide a more clear representation of the grammar [1].

⟨programChoice⟩ ::= ⟨global_declaration⟩
| ⟨function_declaration⟩
| ⟨parameter⟩
| ⟨function_definition⟩

⟨compoundStatementChoice⟩ ::= ⟨annot⟩
| ⟨compound_statement⟩
| ⟨statement⟩
| ⟨separator⟩
| ⟨separator1⟩

⟨statementChoice⟩ ::= ⟨compoundStatementChoice⟩
| ⟨compound_declaration⟩

⟨functionChoice⟩ ::= ⟨compound_declaration⟩
| ⟨annot⟩
| ⟨compound_statement⟩
| ⟨statement⟩
| ⟨separator1⟩
| ⟨separator⟩

[1] They are forbidden in Fisher normal form grammars.

⟨*expressionChoice*⟩ ::= ⟨*conditionalChoice*⟩
| ⟨*assignment_expression*⟩

⟨*conditionalChoice*⟩ ::= ⟨*logicalOrChoice*⟩
| ⟨*conditional_expression*⟩

⟨*logicalOrChoice*⟩ ::= ⟨*logicalAndChoice*⟩
| ⟨*logical_or_expression*⟩

⟨*logicalAndChoice*⟩ ::= ⟨*inclusiveOrChoice*⟩
| ⟨*logical_and_expression*⟩

⟨*inclusiveOrChoice*⟩ ::= ⟨*exclusiveOrChoice*⟩
| ⟨*inclusive_or_expression*⟩

⟨*exclusiveOrChoice*⟩ ::= ⟨*andChoice*⟩
| ⟨*exclusive_or_expression*⟩

⟨*andChoice*⟩ ::= ⟨*equalityChoice*⟩
| ⟨*and_expression*⟩

⟨*equalityChoice*⟩ ::= ⟨*relationalChoice*⟩
| ⟨*equality_expression*⟩

⟨*relationalChoice*⟩ ::= ⟨*shiftChoice*⟩
| ⟨*relational_expression*⟩

⟨*shiftChoice*⟩ ::= ⟨*additiveChoice*⟩
| ⟨*shift_expression*⟩

⟨*additiveChoice*⟩ ::= ⟨*multiplicativeChoice*⟩
| ⟨*additive_expression*⟩

⟨*multiplicativeChoice*⟩ ::= ⟨*postfixChoice*⟩
| ⟨*multiplicative_expression*⟩

⟨*postfixChoice*⟩ ::= ⟨*unaryChoice*⟩
| ⟨*cast_expression*⟩

⟨*unaryChoice*⟩ ::= ⟨*empty_fcall*⟩
| ⟨*id*⟩
| ⟨*nested_expression*⟩
| ⟨*postfix_expression*⟩
| ⟨*unary_expression*⟩

⟨*parameterChoice*⟩ ::= ⟨*parameter_list*⟩
  | ⟨*parameter*⟩

⟨*program*⟩ ::= ⟨*programChoice*⟩ FinalAnnotation? ⟨*moreAnnotation*⟩?

⟨*global_declaration*⟩ ::= '`struct`' IDENTIFIER '`{`' ⟨*declaration*⟩ '`}`' ⟨*global_declaration1*⟩?
  | (⟨*function_declaration*⟩|⟨*parameter*⟩) '`;`' ⟨*programChoice*⟩

⟨*global_declaration1*⟩ ::= '`;`' ⟨*programChoice*⟩

⟨*function_declaration*⟩ ::= (⟨*type*⟩|⟨*ptr_type*⟩) '`*`' (⟨*function_declaration1*⟩|⟨*empty_fcall*⟩)
  | ⟨*type*⟩ IDENTIFIER ⟨*function_declaration2*⟩

⟨*function_declaration1*⟩ = IDENTIFIER ⟨*function_declaration2*⟩

⟨*function_declaration2*⟩ = '`(`' ⟨*parameterChoice*⟩ '`)`' Annotation?
  | '`(`' '`)`' Annotation

⟨*parameter*⟩ ::= ⟨*type*⟩ IDENTIFIER
  | (⟨*type*⟩|⟨*ptr_type*⟩) '`*`' ⟨*id*⟩

⟨*parameter_list*⟩ ::= ⟨*parameter*⟩ '`,`' (⟨*parameter_list*⟩|⟨*parameter*⟩)

⟨*function_definition*⟩ ::= ⟨*type*⟩ IDENTIFIER ⟨*function_definition2*⟩
  | (⟨*type*⟩|⟨*ptr_type*⟩) '`*`' ⟨*function_definition1*⟩

⟨*function_definition1*⟩ ::= IDENTIFIER ⟨*function_definition2*⟩

⟨*function_definition2*⟩ ::= '`(`' ⟨*parameterChoice*⟩? '`)`' Annotation?
    '`{`' ⟨*functionChoice*⟩? '`}`' ⟨*programChoice*⟩?

⟨*compound_declaration*⟩ ::= (⟨*function_declaration*⟩|⟨*parameter*⟩) '`;`' ⟨*functionChoice*⟩

⟨*compound_declaration1*⟩ ::= '`;`' ⟨*compound_declaration*⟩

⟨*compound_statement*⟩ ::= '`return`' ⟨*expressionChoice*⟩? '`;`' ⟨*compoundStatementChoice*⟩
  | '`if`' '`(`' ⟨*expressionChoice*⟩ '`)`' '`{`' ⟨*statementChoice*⟩ '`}`'
    ('`else`' '`{`' ⟨*statementChoice*⟩ '`}`')? ⟨*compoundStatementChoice*⟩
  | ⟨*expressionChoice*⟩ '`;`' ⟨*compoundStatementChoice*⟩
  | Annotation? '`while`' '`(`' ⟨*expressionChoice*⟩ '`)`'
    '`{`' ⟨*statementChoice*⟩ '`}`' ⟨*compoundStatementChoice*⟩
  | '`{`' ⟨*compound_declaration*⟩ '`}`' ⟨*compoundStatementChoice*⟩

⟨*statement*⟩ ::= ‘`return`’ ⟨*expressionChoice*⟩? ‘`;`’
| ⟨*expressionChoice*⟩ ‘`;`’
| ‘`if`’ ‘`(`’ ⟨*expressionChoice*⟩ ‘`)`’ ‘`{`’ ⟨*statementChoice*⟩ ‘`}`’
(‘`else`’ ‘`{`’ ⟨*statementChoice*⟩ ‘`}`’)?
| Annotation? ‘`while`’ ‘`(`’ ⟨*expressionChoice*⟩ ‘`)`’
‘`{`’ ⟨*statementChoice*⟩ ‘`}`’

⟨*separator*⟩ ::= ‘`;`’

⟨*separator1*⟩ ::= ‘`;`’ ⟨*compoundStatementChoice*⟩

⟨*declaration*⟩ ::= (⟨*function_declaration*⟩|⟨*parameter*⟩) ‘`;`’ ⟨*declaration*⟩?
| ‘`struct`’ IDENTIFIER ‘`{`’ ⟨*declaration*⟩ ‘`}`’ (⟨*declaration1*⟩|⟨*separator*⟩)

⟨*declaration1*⟩ ::= ‘`;`’ ⟨*declaration*⟩

⟨*type*⟩ ::= ‘`struct`’ IDENTIFIER
| ‘`void`’
| ‘`int`’

⟨*type2*⟩ ::= ‘`(`’ (⟨*type*⟩|⟨*ptr_type*⟩) ‘`)`’

⟨*ptr_type*⟩ ::= (⟨*type*⟩|⟨*ptr_type*⟩) ‘`*`’

⟨*empty_fcall*⟩ ::= IDENTIFIER ⟨*empty_expression*⟩

⟨*empty_expression*⟩ ::= ‘`(`’ ‘`)`’

⟨*id*⟩ ::= IDENTIFIER

⟨*annot*⟩ ::= Annotation

⟨*moreAnnotation*⟩ ::= FinalAnnotation ⟨*moreAnnotation*⟩?

⟨*argument_expression_list*⟩ ::= ⟨*expressionChoice*⟩ ‘`,`’ ⟨*expressionChoice*⟩
| ⟨*expressionChoice*⟩ ‘`,`’ ⟨*argument_expression_list*⟩

⟨*unary_expression*⟩ ::= ‘`sizeof`’ (⟨*unaryChoice*⟩|⟨*type2*⟩)
| (‘`~`’|‘`!`’|‘`*`[1]’|‘`&`[1]’|‘`+`[1]’|‘`-`[1]’) ⟨*postfixChoice*⟩
| (‘`++`’|‘`--`’) ⟨*unaryChoice*⟩

⟨*assignment_expression*⟩ ::= ⟨*unaryChoice*⟩ ‘`=`’ ⟨*expressionChoice*⟩
| ⟨*unaryChoice*⟩ (‘`+=`’|‘`-=`’|‘`*=`’|‘`/=`’|‘`%=`’) ⟨*expressionChoice*⟩
| ⟨*unaryChoice*⟩ (‘`^=`’|‘`|=`’|‘`&=`’|‘`<<=`’|‘`>>=`’) ⟨*expressionChoice*⟩

[1] This are the unary versions of the tokens.

⟨*conditional_expression*⟩ ::= ⟨*logicalOrChoice*⟩ ‘?’ ⟨*expressionChoice*⟩ ‘:’ ⟨*conditionalChoice*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logicalOrChoice*⟩ ‘||’ ⟨*logicalAndChoice*⟩

⟨*logical_and_expression*⟩ ::= ⟨*logicalAndChoice*⟩ ‘&&’ ⟨*inclusiveOrChoice*⟩

⟨*inclusive_or_expression*⟩ ::= ⟨*inclusiveOrChoice*⟩ ‘|’ ⟨*exclusiveOrChoice*⟩

⟨*exclusive_or_expression*⟩ ::= ⟨*exclusiveOrChoice*⟩ ‘^’ ⟨*andChoice*⟩

⟨*and_expression*⟩ ::= ⟨*andChoice*⟩ ‘&’ ⟨*equalityChoice*⟩

⟨*equality_expression*⟩ ::= ⟨*equalityChoice*⟩ (‘==’|‘!=’) ⟨*relationalChoice*⟩

⟨*relational_expression*⟩ ::= ⟨*relationalChoice*⟩ (‘>’|‘<’|‘>=’|‘<=’) ⟨*shiftChoice*⟩

⟨*shift_expression*⟩ ::= ⟨*shiftChoice*⟩ (‘>>’|‘<<’) ⟨*additiveChoice*⟩

⟨*additive_expression*⟩ ::= ⟨*additiveChoice*⟩ (‘+’|‘-’) ⟨*multiplicativeChoice*⟩

⟨*multiplicative_expression*⟩ = ⟨*multiplicativeChoice*⟩ (‘*’|‘/’|‘%’) ⟨*postfixChoice*⟩

⟨*cast_expression*⟩ ::= ‘(’ (⟨*type*⟩|⟨*ptr_type*⟩) ‘)’ ⟨*postfixChoice*⟩

⟨*nested_expression*⟩ ::= ‘(’ ⟨*expressionChoice*⟩ ‘)’

⟨*postfix_expression*⟩ ::= Constant ⟨*postfix_expression1*⟩?
  | IDENTIFIER (⟨*nested_expression*⟩|⟨*postfix_expression1*⟩|⟨*postfix_expression2*⟩)

⟨*postfix_expression1*⟩ ::= ‘[’ ⟨*expressionChoice*⟩ ‘]’ ⟨*postfix_expression1*⟩?
  | (‘.’|‘->’) IDENTIFIER ⟨*postfix_expression1*⟩?
  | (‘++’|‘--’) ⟨*postfix_expression1*⟩?

⟨*postfix_expression2*⟩ ::= ‘(’ ⟨*expressionChoice*⟩? ‘)’ ⟨*postfix_expression1*⟩
  | ‘(’ ⟨*argument_expression_list*⟩ ‘)’ ⟨*postfix_expression1*⟩?