Source Code Verification for Embedded Systems using Prolog

Frank Flederer
University of Wuerzburg
Aerospace Information Technology
`frank.flederer@uni-wuerzburg.de`

Ludwig Ostermayer
University of Wuerzburg
Knowledge-based Systems
`ludwig.ostermayer@uni-wuerzburg.de`

Dietmar Seipel
University of Wuerzburg
Knowledge-based Systems
`dietmar.seipel@uni-wuerzburg.de`

Sergio Montenegro
University of Wuerzburg
Aerospace Information Technology
`sergio.montenegro@uni-wuerzburg.de`

System relevant embedded software needs to be reliable and therefore well tested, especially for aerospace systems. A common technique to verify programs is the analysis of their abstract syntax tree (AST). Tree-like structures can be elegantly analyzed with the logic programming language Prolog. Moreover, Prolog offers further advantages for a thorough analysis: On the one hand, it natively provides versatile options to efficiently process tree- or graph-like data structures. On the other hand, Prolog's non-determinism and backtracking allows to test different variations of the program flow without big effort. A rule-based approach with Prolog allows to characterize the verification goals in a concise and declarative way.

In this paper, we describe our approach to verify the source code of a flash file system with the help of Prolog. The flash file system is written in C++ and has been developed particularly for use within satellites. We transform a given abstract syntax tree of C++ source code into Prolog facts and derive a call graph and the execution sequence, which then are further tested against the verification goals. The different program flow branching due to control structures is derived by backtracking as subtrees of the original AST. Finally, these subtrees are verified in Prolog.

We illustrate our approach with a case study, where we search for incorrect applications of semaphores in a flash file system for satellites. We rely on computation tree logic (CTL) and have designed an embedded domain specific language (DSL) in Prolog to express the verification goals.

1 Introduction

There exist several approaches which introduce logic programming into the static source code analysis of software. A brief comparison between logic-based infrastructures concerning detection and extraction is given by Kniesel, Hannemann and Rho [11], who compare different frameworks that enable software analysis and manipulation by an object-oriented program representation. The comparison addresses efficiency and scalability, but also further criteria such as expressiveness, turnaround and availability. The comparison also includes the formulation of a design pattern detection. They compare two frameworks for the analysis of Java source code: JQuery and their own approach JTransformer/CTC. CodeQuest was selected as a reference for performance and scalability. However, JTransformer is limited to process Java source code. It is designed as a plug-in solely for the use with Eclipse[1], a popular integrated development environment (IDE) for Java. But embedded software development often depends on different IDEs or command line tools.

Consens and Mendelzon describe in various publications [6, 7] their query language named GraphLog. It operates on data represented as graph structures, even the queries are described as graph patterns. To

[1] https://www.eclipse.org

Submitted to:
WLP 2016

evaluate a query the graph pattern is searched within the database graph. Data, queries and results can be visualized. The authors argue that many databases can be viewed as graphs. As a case study, they use GraphLog to analyze software structures in [8]. Consens et al. examine the package structure of a complex Windows application in order to remove cycles within the package dependency. However, the data structure for their software analysis is not the full AST. Therefore, this approach is highly specialized and difficult to generalize.

Ciraci describes in his dissertation [4] the graph-based verification of static program constraints [5]. In the first step, the original source code is automatically converted into a intermediate representation which is called SCML (Source Code Modeling Language). SCML is an attributed graph representation of the source code. The SCML expressions can be imported directly from different languages – there are already converters for Java and C/C++. The constraints for the source code are described as graph transformation rules which also written in SCML. To notify the user for violated constraints, information nodes are inserted locally into the original SCML. The user can write Prolog rules to further investigate the information nodes. However, the program flow is not covered in Ciraci's analysis.

Crew [9] created a ASTLOG, a Prolog-like programming language, to specifically process ASTs. ASTLOG does not save a given AST into an internal database, but references the elements of the AST directly. A difference to a fact base in Prolog is the Current Object which only is implicitly available to rules. The rules are evaluated in the context of the Current Object; similar to a visitor in the visitor pattern. Even ASTLOG is Prolog-like, compatibility is not given.

Centaur [3] is a generic interactive debugging system. The input is a formal specification of the syntax and semantics of the used programming language. The specifications are described in Prolog which are used to convert the given source code into the internal data format (Virtual Tree Processor). Albeit Centaur provides an interactive investigation process, the automatic pattern recognition is hard to modify.

Ballance, Graham and Van De Vanter present Pan [1], an integrated development environment, which allows to analyze source code loaded by its editor. They use logic programming in conjunction with Logical Constraint Grammars differently from Prolog. For instance, the logic database is partitioned into several collections and code in Pan is interpreted separated from data which makes it less flexible compared to homoiconic Prolog.

Wahler, Seipel, Von Gudenberg and Fischer propose an XML structure for the representation of abstract syntax trees [14]. On that structure, they apply an algorithm inspired from data mining for searching clones within the source code by finding frequent itemsets in the XML. In case studies they run the analysis on the Java Development Kit and the Dislog Development Kit. Additionally to the program analysis, the tool Squash of Boehm, Seipel, Sickmann and Wetzka can be used for designing, analyzing and refactoring relational database applications [2]. In their approach, schema definitions and queries from SQL are mapped to an XML representation called SquashML. Like an AST the SQL representation in SquashML is a tree structure. By using the query and transformation language FnQuery for tree structures, they can use Prolog rules to describe modifications of the structure of the relational database. Additionally, Squash provides a visualization of the relations and join paths within SQL queries.

Source Code Verification for Embedded Systems Embedded systems are heavily used in mission critical parts, e. g. in the field of aerospace. Because of the preferred usage of established and time-proven software, modern cutting-edge technology is seldom applied. If new software technology and concepts are used in critical embedded systems, then the software has to be well tested and approved before it is applied in real critical missions. The majority of software in embedded systems is written in the two programming languages C (51 %) and C++ (30 %) [12]; both are sometimes processed by the same tools, e.g. Clang/LLVM.

In a satellite project, we have developed a tailored file system for the embedded use on a satellite, which makes use of several flash memory chips and one MRAM – a non-volatile random access memory. The file system will be used in the real-time operating system named RODOS[2] which is developed at the German Aerospace Center (DLR) and the chair for aerospace information system at the university of Wuerzburg. Like the operating system RODOS, the file system itself is written in C++. To make sure that everything is implemented correctly, we want to verify the file system formally. But in extend to the usual static analysis of software, we want to be able to test patterns within the source code that are specific for the domain of our own flash file system. Similar to JTransformer[11], which operates with Prolog on Java source code, we want to perform static source code analyses by using Prolog rules – but for C/C++ software. There exists an extension to JTransformer, named StarTransformer[3], which enables the analysis of other languages as well, but no application for C/C++ could be found yet. This paper shows our first attempts that make use of non-deterministic Prolog rules for the static source code analysis for embedded software that is written in C/C++. For future work we are planning to extend our approach with refined techniques of static software analysis like symbolic execution.

In this paper, we use Prolog rules to define the verification patterns over the original source code for the following reasons. Firstly, we are able to extend the analysis by writing comprehensible Prolog rules. Secondly, Prolog enables an easy definition of domain specific languages to match the requirements of specific software parts, e.g. our file system. Thirdly, as we heavily operate on graph structures, we can make use of existing graph investigation techniques and libraries which are already available in certain Prolog systems. Additionally, we make use of another advantage of Prolog: the non-determinism. Because of control structures within the original source code, there might be different program flow branches which might be used during the software runtime – depending on the actual results of the evaluation of the conditions. To test different program flows, Prolog's backtracking mechanism is used to test several variants of program flows. For our implementation, we use the established Prolog system SWI-Prolog [15].

Organization of the Paper The remainder of the paper is organized as follows. In Section 2 a short overview of the process is given. Section 3 shows the representation of the original source code in abstract syntax trees. Out of an abstract syntax tree, we extract in Section 4 a call graph for methods and functions. For the investigation of sequent calls over different method and function calls, we create in Section 5 an execution sequence tree. According to control structures within the source code and the generated execution sequence tree, we generate different subtrees of the abstract syntax tree in Section 5 which will be the foundation for non-deterministic execution sequence trees. In Section 6 we use our approach to analyze possibly issues with semaphores and conclude with future work in Section 7.

2 Overview of the Verification Process

The verification process transforms the source code into different graph representations in subsequent stages. Figure 1 outlines the different stages and graphs.

[2]Real-time On-board Dependable Operating System: http://www.dlr.de/irs/desktopdefault.aspx/tabid-5976/9736_read-19576/

[3]https://sewiki.iai.uni-bonn.de/research/jtransformer/api/meta/startransformer

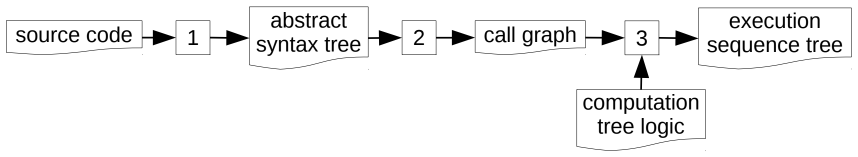


Figure 1: the verification process is partitioned into three subsequent stages

At first, a program using Clang transforms the corresponding *abstract syntax tree* (AST) into Prolog facts which then are stored into the Prolog database. Secondly, declarative Prolog rules match graph patterns of method calls within the AST and extract information about method calls into a *call graph* (CG). In the last stage, traversing Prolog rules construct *execution sequence trees* (EST) that contain information about method calls that are subsequently executed after a given method call. Due to user-defined *computation tree logic* (CTL) rules and different program flows in the software, the process generates different non-deterministic variations of the EST, called NESTs. The traversing process directly evaluates the CTL rules on the nodes during building the NEST; therefore, it detects NESTs with failing or successful CTL checks early. This reduces the number of NESTs that are investigated.

3 Representation of the Abstract Syntax Tree in Prolog

The first stage of the overall process (stage 1 in Figure 1) is an automatic transformation of the structure of source code into Prolog facts. We use Clang, a C/C++ frontend for LLVM[4], to extract the full AST into Prolog facts. The resulting facts (AST nodes) have the following structure:

```
node(File, Ast_Order, Id, Par_Id, Type, Src_Loc, Params).
```

For every C/C++ source file, Clang creates a separate AST. The node IDs are unique for the scope of a single AST. However, our process operates on the complete AST of the full software, therefore, each node is extended by the name of the source file (`File`, first argument of `node/7`). The second argument (`Ast_Order`) defines the relative location of a node among its siblings. The argument `Id` represents an identifier of the node within the source file's AST, and `Parent_Id` is the identifier of its parent node, in the same AST. The fifth argument (`Type`) names the node Type (e. g. `IfStmt`, `CXXMethodDecl`). The argument `Src_Loc` contains the location of the node within the source code (`File:Line:Column`). Depending on the `Type`, a node comes with specific parameters which differ in number and in data types. Therefore, the last element (`Parameters`) is a Prolog list which contains the specific extra information.

Example Representation of a Source Code as an AST Apart from the syntactical structure of the source code, Clang also includes references which describe semantic links between nodes of the AST. Figure 2 shows an AST of the following source code:

```
// File: test.h
class Test {
public:
  bool var;
  void methodA();
  void methodB();
```

[4]Low Level Virtual Machine: http://llvm.org

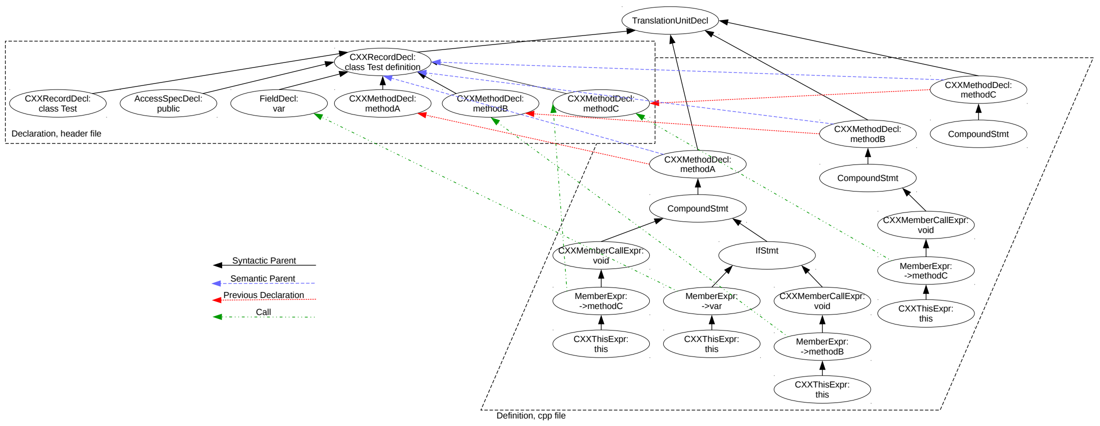


Figure 2: Example of an AST from Clang with Semantic References

```
  void methodC(); };

// File: test.cpp
#include "test.h"
void Test::methodA() {
  methodC();
  if(var) methodB(); }
void Test::methodB() {
  methodC(); }
void Test::methodC() { }
```

The type of the root node is always `TranslationUnitDecl`. Due to `#include` statements in the source file, Clang adds nodes from additional files, i. e. header files, to the AST. In the example the AST is built from a source and a header file; Figure 2 separates the nodes by their origin (dashed boxes). The header file declares the class `Test` (node `CXXRecordDecl`[5]), the member variable `var` (node `FieldDecl`) and the three methods `methodA`, `methodB`, and `methodC` (node `CXXMethodDecl`). The implementation (definition) of the methods is written in the source file instead of the header file. Therefore, the method declaration nodes `CXXMethodDecl` from the header file lack syntactic children (black edges). Instead, they have semantic children (red edges) which describe their implementations (definitions). The implementations (definitions), though, are syntactically direct children of the root node (`TranslationUnitDecl`).

The body of a method is always enclosed in curly braces; the `CompoundStmt` stands for this enclosed compound of statements. Consequently, the children of `CompoundStmt` describe the method's body. A method call is defined by three subsequent nodes; `CXXMemberCallExpr` specifies the return type, `MemberExpr` describes the member name (for the method or variable), and `CXXThisExpr` defines the variable (object) on which the call or access is executed. Additionally, Clang adds a reference to the declaration of the called method to the `MemberExpr` nodes (green edges). If a method is already declared before, Clang adds a reference from the `CxxMethodDecl` to the previous declaration node (also `CxxMethodDecl`). The `methodA` calls one or two other methods; `methodC` is always called whereas the call of `methodB` depends on the evaluation result of the if statement. The left child of the node `IfStmt` defines the condition part, and the right child defines the then part. Therefore, the call of `methodB` is

[5]The prefix CXX in the types of the nodes stands for C++.

only executed if the value of `var` is `true`.

Regarding the extra (colored) edges, the resulting structure is actually not a tree; instead, it is a directed acyclic graph (DAG). As mentioned above, the process in the first stage transforms the AST from Clang into Prolog facts. Here are some example nodes:

```
% node(+File, +AST_Order, +Id, +Par_Id, +Type, +Src_Loc, +Params) <-
node(
   'src/test.cpp', 4, '0x2b8e2a0', '0x2b8d8f0', 'CXXRecordDecl',
   '<data/src/test.cpp:1:1, line:13:1>',
   ['class', 'Test', 'definition']).
node(
   'src/test.cpp', 7, '0x2b8e4c0', '0x2b8e2a0', 'CXXMethodDecl',
   '<line:3:3, line:6:3>',
   ['methodA', 'void (void)']).
```

Prolog Rules for Accessing AST Structures For an easy access of the AST we define several rules. Two predicates are used for accessing syntactic relations; `edge/3` represents the syntactic edges of the AST, and `transitive/3` determines ancestors or descendants of nodes. Additionally, we define particular predicates for accessing different node types. As mentioned above, the different types of nodes include different extra information in the list `Parameters`. The rules extract the extra information and provide them as arguments of the predicate. The following example shows the rules for `edge/3`, `transitive/3`, `if_stmt/3`, and `binary_operator/5`.

```
% edge(-File, -Parent, -Child) <-
edge(File, Parent, Child) :-
   node(File, _, Child, Parent, _, _).

% transitive(-File, -Ancestor, -Child) <-
transitive(File, Ancestor, Child) :-
   edge(File, Ancestor, Child).
transitive(File, Ancestor, Child) :-
   edge(File, Parent, Child),
   transitive(File, Ancestor, Parent).

% if_stmt(-File, -Id, -AST_Order) <-
if_stmt(File, Id, AST_Order) :-
   node(File, AST_Order, Id, _, 'IfStmt', _).

% binary_operator(-File, -Id, -AST_Order, -Type, -Name) <-
binary_operator(File, Id, AST_Order, Type, Name) :-
   node(File, AST_Order, Id, _, 'BinaryOperator', [_, Type, Name]).
```

The predicate `if_stmt/3` represents a node with the type `IfStmt`. The predicate `binary_operator/5` represents a node with the type `BinaryOperator` and additionally extracts the return type of the binary operator and its name.

4 Call Graph in Prolog

The AST created by Clang provides references from calling methods to the called methods, see Section 3. Using this information we create a graph that only contains information about method calls. We name

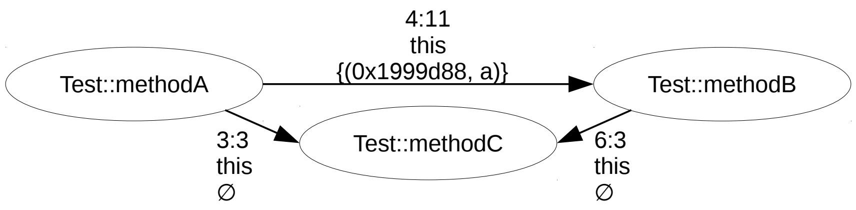


Figure 3: Our example displayed as call a graph

the graph *call graph* (*CG*), which can be expressed as a tuple

$$CG = (V_C, E_C, N_C, L_C, W_C, C_C, D_C, \mathscr{N}_C, \mathscr{L}_C, \mathscr{W}_C, \mathscr{C}_C),$$

where V_C is a set of nodes, E_C is a set of directed edges (v_0, v_1) with $v_0, v_1 \in V_C$, N_C is a set of method names, L_C is a set of locations within the source code, W_C is a set of variable names, C_C is a set of conditions IDs, $D_C = \{a, b\}$ is a set of condition categories, $\mathscr{N}_C$ is a labeling function $\mathscr{N}_C : V_C \to N_C$, $\mathscr{L}_C$ is a labeling function $\mathscr{L}_C : E_C \to L_C$, $\mathscr{W}_C$ is a labeling function $\mathscr{W}_C : E_C \to W_C$, and $\mathscr{C}_C$ is a labeling function $\mathscr{C}_C : E_C \to (C_C, D_C)^n$ with $n \in \mathbb{N}_0$.

For the investigation of different program flows, we use information about control structures. Either one or several nested control structures within method bodies decide whether a method call is skipped. The transformation process labels the edges with a set of control structures IDs that affect the method call. Some control structures like `if` ensure an exclusive execution of their children. E. g. for the control structure `if` either the then part or the else part is executed – never both parts at the same time. To cover the exclusive execution, the transformation process classifies the control structures IDs into different categories; an identifier is either categorized as `a` (i.e. then part) or `b` (i.e. else part).

Figure 3 shows the resulting CG for the AST from Figure 2. The nodes are labeled with the methods' names. The edges are labeled with the source code location, the variable (object) name on which the call is invoked, and the set of the control structures IDs they depend on.

Detection Rules for Code Patterns of Methods Information about structures like method declarations, their definitions, method calls etc. is stored within the AST. The information is not available in single nodes, but rather in subgraphs of the AST. In Prolog we define graph patterns specifically for structures like method definition, method calls etc. The following example shows such a rule that detects a method declaration.

```
% method_decl(-File, -Method_Id, -Full_Name) <-
method_decl(File, Method_Id, Full_Name) :-
   cxx_method_decl(File, Meth_Id, Func_Name, _),
   transitive(File, Class_Id, Meth_Id),
   cxx_record_decl(File, Class_Id, 'class', Class_Name),
   namespace(File, Method_Id, Namespace_Name),
   atomic_list_concat([Namespace_Name, '::', Func_Name], Full_Name).
```

The rule searches for a pattern that consists of a node `CXXMethodDecl` (predicate `cxx_method_decl/4`) and a ancestor node `CXXRecordDecl` (predicate `cxx_record_decl/4`) with the record type `'class'`. Generally, the rule searches for a class definition which contains a method definition. The node of the type `CXXMethodDecl` contains only the local name of the method, not the full one. To obtain the full name, the rules for `namespace/3` recursively traverse the AST up and joins the appropriate scope names (e. g. from classes or namespaces). After that, `atomic_list_concat/2` merges that prefix and the local name.

Detection Rules for Control Structures There are different types of structures, which lead to conditional execution of operations, e g. method calls. On the one hand there are obvious keywords for control structures, e. g. `do`, `for`, `if`, `while` and the ternary operator `?`. On the other hand there are other structures which implicitly skip operations. For example, an *or* (`||`) executes the second operand only if the first one is false. Otherwise the second operand is not executed at all (short-circuit evaluation).

We define rules that identify control structures and determine the root nodes of the conditionally executed subtrees. The following example shows a rule for an `if` statement with a then part and an else part:

```
% if(-File, -Cond_Id, -When_Id, -Else_Id) <-
if(File, Cond_Id, When_Id, Else_Id) :-
   if_stmt(File, Id, _),
   edge(File, Id, Cond_Id),
   edge(File, Id, When_Id),
   edge(File, Id, Else_Id),
   node(File, Cond_Order, Cond_Id, _, _, _),
   node(File, When_Order, When_Id, _, _, _),
   node(File, Else_Order, Else_Id, _, _, _),
   Cond_Order < When_Order,
   When_Order < Else_Order.
```

The rule searches a node `IfStmt` (predicate `if_stmt/3`) and its three children. The three children are the root nodes of different subtrees that stand for the different parts of the `if`; the first subtree represents the condition part, the second subtree represents the then part, and the third subtree represents the else part. The order of the children defines their assignment to the parts of the `if`. Therefore, the rule determines the relative order using `node/5`, and after that, the orders are compared. We also define a similar rule for `if`s without an else part, which is not shown here.

Virtual Methods An important feature of the object-oriented programming paradigm is the inheritance of classes. Subclasses can overwrite methods if the method is declared as `virtual` in the superclass. Due to inheritance, a pointer to a superclass can also point to a corresponding subclass. Then, if an overwritten method is called, the implementation in the subclass is called instead of the one in the superclass.

As mentioned in Section 3, the node of a method call references the declaration of the called method. However, it always references the class for which the pointer is declared, not the derived classes. Therefore, we generate multiple CGs that call overwritten methods instead.

5 Building Execution Sequence Trees in Prolog

The verification process investigates subsequent method calls that are executed after a given method call. A tree structure, which we call Execution Sequence Tree (EST), represents the subsequent method calls. Following the program flow, the subsequent method calls from the successor nodes are added subsequently to a list via depth-first search on the CG. After traversing the children nodes, parents are visited to add additional methods that are called after a return. Several parent methods can call the current method, therefore, a branching in the EST is introduced to represent different program flows.

The evaluations of control structures result in different program flows which we want to investigate. Therefore, non-deterministic rules create different variants of ESTs that represent various program flows; we call them Non-deterministic Execution Sequence Trees (NEST). To search programming errors, the creation process of NESTs applies Computation Tree Logic (CTL) during its work; the CTL e. g. describes programming errors. This enables early CTL evaluation results, even before NESTs are created completely. Consequently, some control structures are excluded from the NESTs and, therefore, can be ignored; therefore, the number of investigated NESTs is reduced.

Definition of Execution Sequence Trees The tree structure EST can be expressed as a tuple

$$EST = (V_E, E_E, N_E, L_E, W_E, T_E, C_E, D_E, \mathscr{N}_E, \mathscr{L}_E, \mathscr{W}_E, \mathscr{T}_E, \mathscr{C}_E),$$

where V_E is a set of nodes, E_E is a set of directed edges (v_0, v_1) with $v_0, v_1 \in V_E$, N_E is a set of method names, L_E is a set of locations within the source code, W_E is a set of variable (object) names on which the calls are executed, $T_E = \{c, p, r\}$ is a set of call types, for the call of child methods (c), the return to parents (p), or the root node (r), C_E is a set of condition IDs, D_E is a set of sets of categories of conditions, $\mathscr{N}_E$ is a labeling function $\mathscr{N}_E : V_E \to N_E$, $\mathscr{L}_E$ is a labeling function $\mathscr{L}_E : V_E \to L_E$, $\mathscr{W}_E$ is a labeling function $\mathscr{W}_E : V_E \to W_E$, $\mathscr{T}_E$ is a labeling function $\mathscr{T}_E : V_E \to T_E$, and $\mathscr{C}_E$ is a labeling function $\mathscr{C}_E : V_E \to ((C_E, D_E)^n)^m$ with $n, m \in \mathbb{N}_0$.

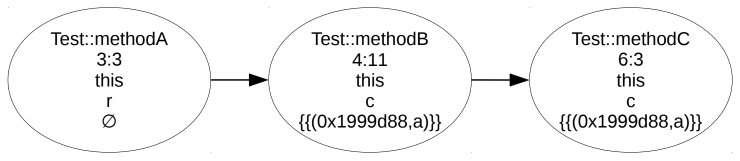
The basis for an EST is a single call (edge within the CG). At first, the called method is used as root node including the information about the call, i. e. the location in the source code and the variable (object) name. According to the program flow, traversing rules add subsequent method calls to the tree with regard to the following guidelines:

Children Coming from an entry edge, the current node is added to the end of the active branch of the EST including the information of the entry edge. After that, all successors are traversed in a depth-first search, whereby siblings are ordered by their location within the source code.

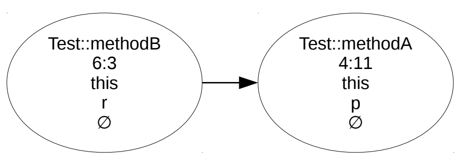
Parents After processing all children or detecting a return node, the traversing rules proceed to the parent nodes. They add each parent node to the last node of the active branch of the EST. As a result, the traversing rules create new branchings which then are separately processed, beginning with the children.

Recursion To avoid endless loops in the traversing process, recursions are handled specially. If there is a recursion (direct or mutual method calls) within the CG, then only one complete run is added to the active branch of the EST. After that, the depth-first search is aborted and backtracked to the next node after the entrance to the recursion. However, the information about which methods are called in the recursion remains in the EST due to the one complete run that was added.

In the CG, information about conditions are assigned to the edges. To make the information available in the EST, the transformation process adds it to the target node. Additionally, the information is propagated for all successors from the CG and added to the nodes of the EST appropriately.



(a) Full EST for methodA → methodC call



(b) Full EST for methodB → methodC call

Figure 4: Execution Sequence Trees of the Semaphore Example

Keeping our example from Figure 3, Figure 4 shows two ESTs which base on different calls of `methodC`. The first one (Subfigure (a)) bases on the method call `methodA`→`methodC`. Consequently, `methodA` becomes the root node. The depth-first search traverses all children that are called after the base call `methodA`$\rightarrow$`methodC`. The location of `methodA`$\rightarrow$`methodB` (4:11) is greater than the base call `methodA`$\rightarrow$`methodC`. Therefore, `methodB` becomes the successor node in the EST. After that, the process adds `methodC` to the EST because `methodB`$\rightarrow$`methodC` is the next subsequent method call within `methodB`. Then, no other subsequent method calls are left to add.

The second EST (Subfigure (b)) bases on the method call `methodB`$\rightarrow$`methodC`. Consequently, the method `methodB` becomes the root node. There are no children in the CG for `methodB`, therefore, it proceeds to the parent nodes and adds its parent `methodA` to the EST following the parent edge. The process does not add any child nodes because the entry edge (`4:11`) is underneath the other method call `methodA`$\rightarrow$`methodC` (`7:3`).

Bringing Non-Determinism to the Execution Sequence Tree Control structures affect the execution of the software; some method calls are only executed in certain program flows. To investigate different program flows for errors, non-deterministic Prolog rules create different variants (subtrees) of ESTs, each representing a program flow. Due to the non-determinism in the creation process, we name the variants of the EST *Non-deterministic Execution Sequence Trees* (NEST). A NEST is identified by the set I of valid control structures and their categories (tupels). The set of nodes of a NEST $V'_E \subseteq V_E$ is composed two subsets: $V'_E = V_D \cup V'_N$

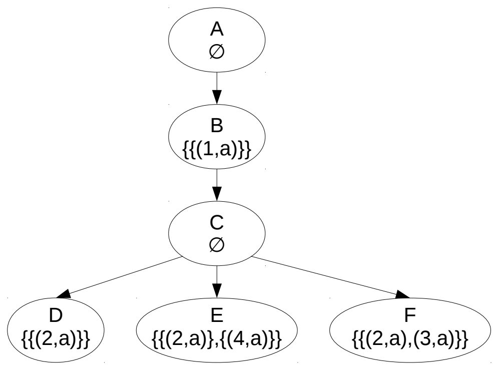
- $V_D = \{v \in V_E \mid \mathscr{C}_E(v) = \emptyset\}$ is the set of nodes which are present in every variation of NESTs (deterministic)
- $V'_N = \{v \in V_E \mid \mathscr{C}_E(v) \subseteq I\}$ is the set of nodes which are dependent of control structures. Therefore, the set differs for every NEST.

The edges of a NEST $E'_E \subseteq E_E$ are reduced according to the nodes. Only edges whose both nodes are part of the NEST become an edge of the NEST: $e = (v_1, v_2) \in E_E \wedge v_1, v_2 \in V'_E \Rightarrow e \in E'_E$.

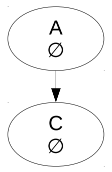
Figure 5 shows a complete EST and sample variations of NESTs. For reasons of comprehensibility only the name of methods and the set of condition IDs are printed; unimportant labelings are omitted. The full EST in Subfigure (a) contains nodes with varying control structure dependencies. The nodes `B` and `D` are valid for single conditions (`(1,a)` resp. `(2,a)`). The node `E` depends on the two single conditions `(2,a)` and `(4,a)`. If one control structures is valid, node `E` becomes part of the NEST. The Node `F` is only included in a NEST if both conditions, `(2,a)` and `(3,a)`, are valid.

Subfigures (b) to (d) show different variants of NESTs. The NEST with no valid control structures is the least complex one. It only contains nodes with empty sets of control structures, as Subfigure (b) shows. Every node with a set of control structures that are valid for a NEST, becomes part of the NEST. Subfigure (c) shows the NEST with one valid control structure: `(2,a)`. The Node `F` contains a combined set `{(2,a),(3,a)}`; however, the condition `(3,a)` is invalid, therefore, the node is left out. For NEST of Subfigure (d) both, `(2,a)` and `(3,a)` are valid. Consequently, nodes with only `(2,a)` (nodes `D` and `E`) and nodes with combined conditions (node `F`) are included in the NEST.

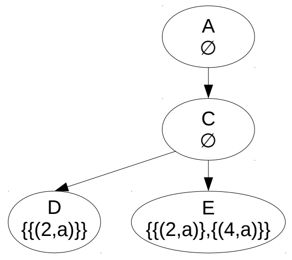
Limitation of the Range of CTL Rules Rules for CTL check the properties of nodes in a tree. The number of checked nodes depends on the rules, the properties of the nodes and on the quantifiers. Table 1 gives an overviews about the range of CTL quantifiers. As soon as a success or a fail is determined, the EST creation stops and ignores the rest of the tree. Consequently, the creation process for NESTs ignores control structures that are only used in ignored nodes; therefore, the number of created NESTs decreases.



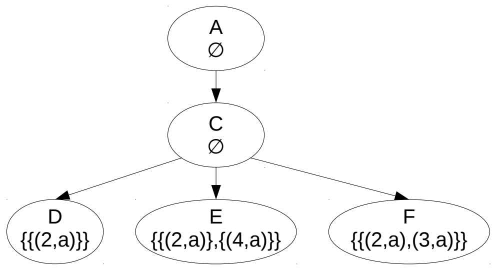
(a) Example for a full EST



(b) NEST without non-deterministic nodes



(c) NEST with nodes for [(2,a)]



(d) NEST with nodes for [(2,a),(3,a)]

Figure 5: Some examples for NESTs for a given full EST

Quantifier	Success		Fail	
	Condition	Ignored Nodes	Condition	Ignored Nodes
AF	a node per path successes	remaining nodes per path	all nodes in a path fail	remaining paths
AG	all nodes success	none	one node fails	remaining nodes
AX	all next nodes success	remaining nodes per path	a next node fails	remaining nodes
AU	first rule successes until second rule successes	remaining nodes per path	first and second rule fail	remaining nodes
EF	one node successes	remaining nodes	all nodes in all paths fail	none
EG	all nodes for a path success	remaining paths	a node of the last path fails	remaining nodes of the last path
EX	next node of a path successes	remaining nodes	next nodes in all paths fail	remaining nodes of all paths

Table 1: Maximum range of CTL rules

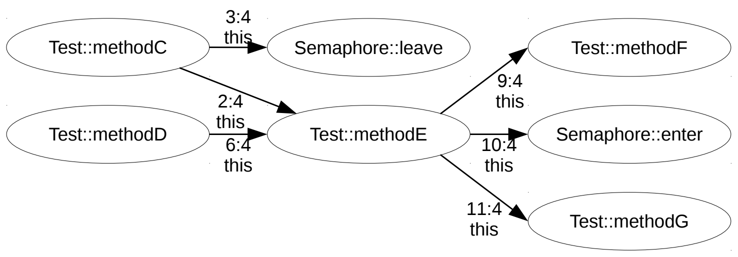


Figure 6: Call Graph of the Incomplete Semaphore Usage

Prolog rules traverse the CG and construct the NESTs, simultaneously applying the user-defined CTL rules. If the CTL rules successes prematurely, the creation process proceeds with the next NEST. However, if the CTL rule fails, the process aborts and the user is informed about the problematic NEST.

6 Case Study using Computation Tree Logic

A case study investigates the usage of semaphores in a flash file system developed for the real-time operating system RODOS. Semaphores are used for synchronizing concurrent threads in RODOS. They are provided as classes with the methods `enter` and `leave`. Consequently, we can investigate their usage on the CG and NESTs. We investigate the static source code, therefore, there are limitations regarding variable values. For example, we cannot detect whether two pointers refer to the same object, because we do not have access to the actual values.

Before introducing problematic semaphore usages, we show a correct usage:

```
void SemaTest::methodA() {
   sema.enter();
   methodB();
   sema.leave(); }
```

The semaphore object `sema` is entered (`enter()`) once and left (`leave()`), eventually.

Semaphore entered but never left Firstly, we investigate missing Semaphore leavings. Figure 6 illustrates the following example:

```
void Test::methodC() {
   methodE();
   sema.leave(); }
void Test::methodD() {
   methodE(); }
void Test::methodE() {
   methodF();
   sema.enter();
   methodG(); }
void Test::methodF() { }
void Test::methodG() { }
```

The `methodC` first calls `methodE`, which enters the semaphore. Then, `methodE` is left while keeping the semaphore entered. After returning to `methodC` the semaphore is left. Therefore, for `methodC`, the semaphore is used correctly.

If `methodE` is called, it only calls `methodE` which enters the semaphore. After that, the execution returns from `methodE` is left, but there are no other calls within `methodD`. Consequently, if the semaphore is not left in a call after returning from `methodD`; we found an incorrect use of semaphores.

Based on the method call `Test::methodE`→`Semaphore::enter` we generate an EST and apply a CTL rule:

$$AF\,\mathscr{N} = Semaphore :: leave \wedge \mathscr{W} = w_r$$

On every branch of the EST (A), finally (F) the rule is applied: there should be a node with the method name $\mathscr{N}$ = `Semaphore::leave`. Additionally, the variable (object) name $\mathscr{W}$ must be the same as the one from the root node (w_r), to assure that the same semaphore is investigated. In Prolog, we define the CTL rules as binary predicates. The first argument is a predicate `est_node/5` of the root node in the EST, the second one is a predicate `est_node/5` of the currently investigated node:

```
% sema_leave(+Root, +Node) <-
sema_leave(Root, Node) :-
   Root = est_node(_, _, _, Var_1, _),
   Node = est_node(_, [Name_2| _], _, Var_2, _),
   Name_2 = 'RODOS::Semaphore::leave',
   Var_1 = Var_2.
```

Semaphore entered twice before left Secondly, we check whether a semaphore is entered multiple times before it is left. An example clarifies the issue:

```
void SemaTest::methodH() {
   sema.enter();
   methodB();
   sema.enter();
   methodB();
   sema.leave(); }
```

The semaphore is entered, before and after the first call of `methodB`, but only left after the second call. This programming error leads to an infinite blocking of the program at the second call of `sema.enter()`. To find this issue in the source code, we describe the correct usage in CTL:

$$A(\neg(\mathscr{N} = Semaphore :: enter \wedge \mathscr{W} = w_r)\ U\ (\mathscr{N} = Semaphore :: leave \wedge \mathscr{W} = w_r))$$

This CTL formula consists of two separate rules; for all branches (A) the first one must be true until (U) the second one becomes true. The first rules expresses that the investigated node must *not* have the method name $\mathscr{N}$ = `Semaphore::enter` on the same variable (object) name $\mathscr{W}$ as the one of the root node (w_r). The second rule expresses that the investigated node *must* have the method name $\mathscr{N}$ = `Semaphore::leave` on the same variable (object) name $\mathscr{W}$ as the one of the root node (w_r). That means, that a semaphore should not be entered before it is left.

Above, we had already written the second rule in Prolog; the first rule in Prolog follows:

```
% sema_not_enter(+Root, +Node) <-
sema_not_enter(Root, Node) :-
   Root = est_node(_, _, _, Var_1, _),
   Node = est_node(_, [Name_2| _], _, Var_2, _),
   ( Var_1 \= Var_2
   ; Name_2 \= 'RODOS::Semaphore::enter' ).
```

This rule successes if the variable (object) names or the method names of the root and the current node differ.

7 Conclusions and Future Work

Using Prolog provides a flexible and concise definition of validation goals. Furthermore, it enables the definition of domain-specific rules for validation. For future work, we intend to demonstrate the versatility by further evaluations for the flash file system for BIROS. We aim at providing a verification that precisely addresses issues of flash file system, for example the retention of data in unpredicted failures. By using Prolog's non-determinism enables the effortless investigation of different program flows. Additionally, we are interested in further applications of Prolog's features like non-determinism for embedded software, beyond file systems.

The EST, which we introduced in Section 5, allows the application of CTL for method calls in the program flow to investigate programming errors. By an early application of CTL during the creation process of NESTs, we already reduce the number of generated NESTs. To further decrease the number of NESTs, we are going to introduce Symbolic Execution to our process. Symbolic execution analyzes the domain of the values of variables, which limits the non-determinism for control structures.

References

[1] R. Ballance, S. Graham, and M. V. D. Vanter. The pan language-based editing system for integrated development. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, SDE 4, pages 77–93, New York, NY, USA, 1990. ACM.

[2] A. M. Boehm, D. Seipel, A. Sickmann, and M. Wetzka. Squash: A tool for analyzing, tuning and refactoring relational database applications. In *Applications of Declarative Programming and Knowledge Management, 17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Würzburg, Germany, October 4-6, 2007, Revised Selected Papers*, pages 82–98, 2007.

[3] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 3, pages 14–24, New York, NY, USA, 1988. ACM.

[4] S. Ciraci. *Graph Based Verification of Software Evolution Requirements*. PhD thesis, Univ. of Twente, Enschede, November 2009. CTIT Ph.D. thesis series no. 09-162.

[5] S. Ciraci, P. van den Broek, and M. Aksit. Graph-based verification of static program constraints. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2265–2272, New York, NY, USA, 2010. ACM.

[6] M. Consens and A. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

[7] M. Consens and A. Mendelzon. Low complexity aggregation in graphlog and datalog. In *in GraphLog and Datalog, Theoretical Computer Science 116*, pages 95–116, 1993.

[8] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and querying software structures. In *International Conference on Software Engineering*, pages 138–156, Melbourne, Australia, May 1992. IEEE.

[9] R. Crew. Astlog: A language for examining abstract syntax trees. In C. Ramming, editor, *DSL*. USENIX, 1997.

[10] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[11] G. Kniesel, J. Hannemann, and T. Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*, LATE '07, New York, NY, USA, 2007. ACM.

[12] M. Nahas. *Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems*. PhD thesis, University of Leicester, 5 2009.

[13] D. Seipel, R. von der Weth, S. Abreu, F. Nogatz, and A. Werner. Declarative rules for annotated expert knowledge in change management. In *Proceedings of the 5th Symposium on Languages, Applications, Technologies*, SLATE 2016, pages 81–96. OASICS, Dagstuhl Publishing, 2016.

[14] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, SCAM '04, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.

[15] J. Wielemaker. An overview of the SWI-Prolog programming environment. In F. Mesnard and A. Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.