Graph-Based Verification of Static Program Constraints [*]

Selim Ciraci
University of Twente
PO Box 217, Enschede
7500 AE, The Netherlands
ciracis@ewi.utwente.nl

Pim van den Broek
University of Twente
PO Box 217, Enschede
7500 AE, The Netherlands
pimvdb@ewi.utwente.nl

Mehmet Aksit
University of Twente
PO Box 217, Enschede
7500 AE, The Netherlands
aksit@ewi.utwente.nl

ABSTRACT

Software artifacts usually have static program constraints and these constraints should be satisfied in each reuse. In addition to this, the developers are also required to satisfy the coding conventions used by their organization. Because in a complex software system there are too many coding conventions and program constraints to be satisfied, it becomes a cumbersome task to check them all manually. This paper presents a process and tools that allow computer-aided program constraint checking that work on the source code.

We developed a modeling language called Source Code Modeling Language (SCML) in which program elements from the source code can be represented. In the process, the source code is converted into SCML models. The constraint detection is realized by graph transformation rules which are also modeled in SCML; the rules detect the violation and extract information from the SCML model of the source code to provide feedback on the location of the problem. This information can be queried from a querying mechanism that automatically searches the graph for the extracted information. The process has been applied to an industrial software system.

1. INTRODUCTION

To avoid program faults and to enhance reuse, one has to properly specify the constraints of a program. While reusing by extending or adapting an implementation of the program, it is important that the software engineers satisfy these constraints. Violations of these constraints may introduce errors to the program, hampering the benefits of reuse. In addition to program constraints, the developers also have to follow the coding conventions enforced by their organization. Usually, a complex program has many constraints and the coding conventions are used very frequently. So, it is a cumbersome task to manually verify all these constraints and coding conventions.

In the literature, there is vast amount of research on formalizing requirements and verification of requirements with respect to the software system. From these, we particularly focus on program constraints verification. In this domain, analyzing the abstract syntax tree (AST) has pulled great attention [13, 5, 11, 7, 8]. The elements of the AST are converted to predicates and constraints are programmed using logic based languages in these systems. The verification can automatically be derived from the software. Although these approaches are practical, they have the following drawbacks:

- The program elements in the AST are different from the source code. Constraints on comments and macros, for example, cannot be expressed or checked at the AST.
- In general, the programs that verify the constraints tend to contain many predicates. Therefore, it is hard to comprehend the meaning of these programs and how they are applied on the ASTs.
- When a violation is located by these approaches, this violation is reported to the user with elements from the AST. Due to this, it may be too hard to locate the error in the source code.

The contributions of this paper are a process and a tool that use meta-modeling and graph pattern searching for providing automated detection of static program constraints and coding conventions violations. We utilize a modeling language, called Source Code Modeling Language (SCML), for representing the source code as seen by the developers. The models in SCML are attributed graphs, in which the source code elements are represented by nodes and the relationships between these elements are represented by edges.

We use a two tier approach for constraint checking. In the first tier, the parts of the source code that violate the constraints are detected. This detection is done by graph transformation rules. Graph transformations are a good means of expressing visually complex patterns and checking these patterns. In our approach, the transformation rules are also modeled in SCML; thus, the constraints over all program elements visible at the source code can be checked by the rules.

In our approach, in addition to pattern checking, the rewriting of the transformation rule is used for error reporting. Here, the transformation rules add graph elements, whose

[*]This work has been carried out as a part of the DARWIN project at Philips Healthcare under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

attributes/labels are used for error reporting; that is, by locating these elements the developers can get information about the constraint violation. SCML preserves the physical line numbers of statements; so, the error reports generated by the rewriting include these numbers in addition to a description about the constraint and the names of the program elements.

The graph representation of the source code may contain many nodes/edges and, so, it may be hard for the stakeholders to spot the information about the program constraint and/or the coding convention violations collected by the graph transformation rules. As a consequence, there is need for a way to query the transformed graph and display the information collected by the transformation rules. Furthermore, some constraints can be a combination of other constraints. Predicate logic seems to be a good way from expressing the combination of the constraints. Because of this, in the second tier of our approach, we built the querying system with a the well-known predicate logic system Prolog [6]. Compared to other approaches, in our approach the predicate logic is only used to present the information extracted by the graph transformations in a convenient way.

This paper is organized as follows: the next section provides the example from an industrial software system that is the driving force behind the development of this approach. Section 3 describes the process in detail. Section 4 presents the application of the approach to the industrial software system of Section 2. The work that is related to our approach is described in section 5 and, lastly, section 6 provides the conclusions.

2. MOTIVATING EXAMPLE

In this section, we show an example program constraint from a Magnetic Resonance Imaging (MRI) software system. This program constraint is frequently used in the control software of the amplifiers. The MRI machine consists of amplifiers that are turned on/off at strict time intervals during a scan. The control software is responsible for reading/writing the values to the registers of the amplifiers at the right intervals. If the control software does not read a value at the right interval, the value may become outdated causing errors in the scan process.

The amplifier keeps track of its status in registers called *general status registers.* The amplifier is real-time hardware; when a request to read a register is made, the amplifier returns the value within a strict time limit. If the software reads the value after this limit, then the value may become outdated.

The communication with the amplifier is divided into parts called *sector*s. A sector consists of a request, a duration and a register set. Here, the request is a byte designating the operation of the amplifier; each operation has a unique number. The duration is the time interval after which the value of a register is outdated. The operations of the amplifiers take parameters by reading the values written the status registers by the software; an operation reads a specific set of registers. Similarly, the operations write their return values to these registers. The register set in a sector is a map for defining the registers to which values are written (or from which values are read).

Every different amplifier model is operated by a different control software. The control software for a new amplifier model is implemented by reusing one of the control softwares of the previous models. The control software is a bridge between the MRI software and the amplifier hardware (on average an implementation of the control software has 4485 lines of code). It has an interface of 32 functions and from these functions the MRI software controls the amplifier. These functions are fulfilled by at least one of the amplifier's operations. An interface function is implemented by initializing sectors that request the desired operations from the amplifier.

```
 1: general_stati(DEVICE_STRUCT *device_ptr){
 2: ...
 3: GEN_SEOS(GENERAL_STATI,   device_ptr,   cs_ptr,
    GENERIC_DUR)
 4: ...
 5: }
 6: analog_samples(DEVICE_STRUCT *device_ptr){
 7: ...
 8: cs_ptr->enableADC = true;
 9: GEN_SEOS(SAMPLE_FIRST, device_ptr, cs_ptr, SAM-
    PLE_FIRST_DUR)
10: ...
11: }
```

Figure 1: Two example uses of the sector generation macro; for each macro the constraint is to set the right duration for the right request.

Because the code for sector initialization is repeated a lot, the designers decided to implement it as a macro called *GEN_SEOS.* Figure 1 shows two uses of the macro *GEN_SEOS* (note that this code segment is a simplified version of the actual code segment). The first use, line 3, is to generate sectors for reading the general status registers. The request here is *GENERAL_STATI* and the duration for this request is stored in the macro *GENERIC_DUR.* The second use is at line 9, where the request is called *SAMPLE_FIRST* and the delay it takes to fulfil this request is stored in the macro *SAMPLE_FIRST_DUR.* Besides the duration constraint, line 8 shows another constraint of the request *SAMPLE_FIRST*: the amplifier's analog-to-digital (ADC) converter should be enabled in order to complete this request.

As can be seen from the two examples described above, setting up the registers and the delays for the requests are crucial constraints for the correct operation of the control software. The macro *GEN_SEOS* is used 16 times in the latest implementation of the control software; so, it is time consuming to check each usage for the constraints of the requests manually. The MRI software is a huge system and is tested rigourously with organizational policies like nightly build/tests. An overnight testing for the parameters of the macro is too much time consuming for a nightly test. The developers need a way to check these constraints quickly without going through runtime tests.

3. A PROCESS FOR COMPUTER AIDED CONSTRAINT VERIFICATION

The process for computer-aided constraint verification (CACV) is a two-tier approach: in the first tier the program constraint and coding convention violations are detected and the error report is prepared. The second tier provides means for searching for a combination of the program constraints and/or coding conventions violations. Figure 2 provides an overview of how the approach is used; in the rest of this pa-

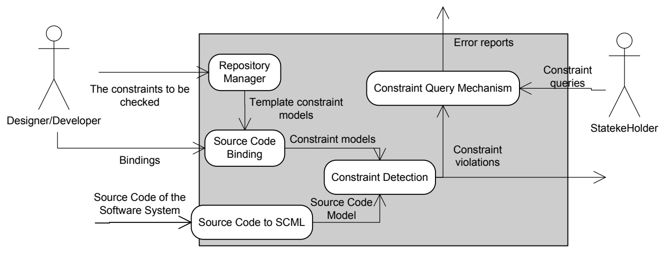

Figure 2: An overview of the CACV approach

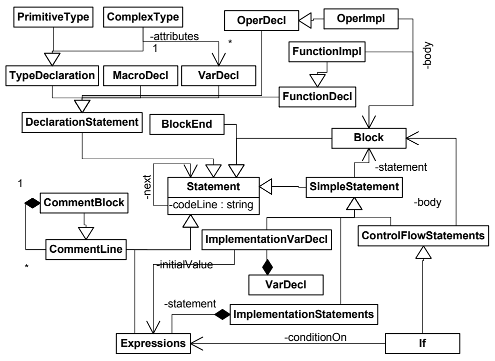

Figure 3: The part of the SCML meta-model showing the relations between different kinds of statements

per we refer to program constraints and coding conventions only as constraints.

In CACV, the constraints are modeled as graph transformation rules with a modeling language called source code modeling language (SCML) that includes the program elements from the source code. The process makes use of a repository to manage/store the modeled constraints and conventions; we use the Computer-Aided Design Evolver tool (CDE) [4] to manage the repository. In the first tier, the developer checks-out the transformation rules modeling the constraints from the repository. The transformation rules in the repository are stored as templates; that is, the names of the program elements are parameterized. If the transformation rule the developer wants to check-out contains parameters, then CDE asks the real values of these parameters from the developer. The developer supplies the actual values for these parameters and with the supplied names CDE binds the constraint models.

The source code is converted to an SCML model, which is an attributed graph. We programmed dedicated converters for Java and C (with preprocessor directives) for this purpose. We also implemented a proof-of-concept (i.e. it cannot parse all statements) converter for Java using the generic text-to-model textual concrete syntax (TCS) [12]. Once the source code is converted and the transformation rules are bound, the constraint violations can be detected with a graph transformation tool; we use GROOVE [16].

If a constraint is violated, GROOVE automatically applies the transformation rule which models that constraint. The transformation rules in CACV add nodes to the graph that contain a description of the constraint. In addition to the description, the SCML preserves the physical line numbers of the statements; so, the rules copy the line numbers of the statements violating a constraint to the node they add. Thus, the developer can learn about the constraint and the line numbers of the statements violating the constraint by locating the node added by the transformation rules.

We added a second tier to CACV that provides querying for constraints, due to the following reasons:

- A constraint can be a combination of other constraints. For example, both $pattern_1$ and $pattern_2$ should be in the design.

- The developers can also query for parts that obey the constraint and when they do, they should get an output stating the constraint is not violated.

- The patterns observed from the source code may be too low level and we need a way to express the constraint at a higher level and convert them to the patterns of the design.

We use the well-known logical programming language Prolog to provide the querying and expressing a combination of the constraints. In our Prolog programs, there are predicates representing the nodes and edges of the graphs. When the evaluation of a Prolog rule reaches these predicates, they execute code in GROOVE that traverses the graph to find the description nodes/edges added by the transformation rules.

In the rest of this section, we describe SCML, how constraints can be modeled with graph transformations to be used with CACV, and how Prolog is tied to the transformation rules.

3.1 Source Code Modeling Language

This section describes the meta-model of SCML; the models in this language are attributed graphs. SCML can be used to represent source code in C/C++ with preprocessor declaratives and Java as graphs. Note that we do not provide the full SCML meta-model in this section, we only describe the elements used in the figures in this paper.

In SCML, the nodes labeled as *Component* refer to source files. Every software entity that belongs to a component is a *Statement*. The ordering between the statements is modeled with the edges labeled *next*. There are four kinds of statements: *declaration statements*, *comment lines*, *simple statements* and *expressions*. Figures 3 presents the relation between these statements.

The declaration statements are statements used for declaring software entities; these statements are not used within a function or method. The meta-model supports variable declarations (nodes with label *VarDecl*), type declarations, macro declarations (nodes with label *MacroDecl*) and function/operation declaration statements (nodes with label *FunctionDecl* and *OperDecl*). All types have designated names modeled as an attribute of the type node. The nodes labeled as *PrimitiveType* refer to data types like *int*. The model supports three complex data types: Enumerations (nodes labeled *EnumType*), structures (nodes labeled *StructureType*) and object type nodes (nodes labeled *ObjectType*). All these types have attributes that are variable declaration nodes. Only object-types and structures (depending on the language) can be connected to method declarations or method implementations with edges labeled *operations*.

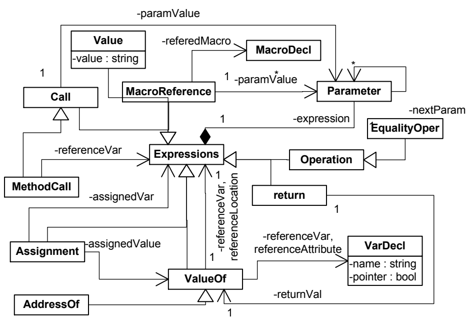

Figure 4: The meta-model of the expression statements in detail.

Nodes labeled as *Signature* represent method/function signatures; the attribute *name* designates the name of the signature. The parameters are modeled with edges connecting the signature node to variable declaration nodes labeled *parameter* and the return type is modeled with an edge connecting the signature node to a type declaration node labeled *returnType*. The method/function declaration or implementation nodes are connected to signature nodes with an edge labeled *signature*.

The body of function/method implementation is represented with block statements that are connected to respective function/method implementations with an edge labeled *body*. The statements of a function/method are represented by nodes labeled *simple statement*. There are three kinds of simple statements: the first kind represents terminated expressions (i.e. terminated with *;*) and this kind is called implementation statements (nodes labeled *ImplementationStatements*), the second kind represents the control flow statements (nodes labeled *ControlFlowStatements* such as an *if* statement) and the third kind represents the variable declarations within functions/methods (nodes labeled *ImplementationVarDecl*). The control flow statements are followed by block statements (i.e. they have bodies). An implementation statement is composed of an expression like a function call. Because of this composition, an implementation statement node is also labeled with the expression it is terminating.

Figure 4 presents the meta-model of expression statements, which includes the following expressions: calls (modeled by nodes labeled *Call*), macro references (modeled by nodes labeled *MacroReference*), returns (modeled by nodes labeled *return*) assignments (modeled by nodes labeled *Assignment*) and values (modeled by nodes labeled *Value*). Calls have three specializations, namely: method calls (nodes labeled *MethodCall*), static method calls (nodes labeled *StaticCall* and constructor calls (nodes labeled *CreateOper*).

The parameter element is a special expression that is composed of other expressions and can be connected to another parameter element with the edge labeled *nextParam*; this edge is used for ordering the parameters. The edge *paramValue* from a call node to a parameter node is used for modeling the parameters passed by a call expression.

The nodes labeled *ValueOf* and *AddressOf* are used to represent a value or an address of a variable and function returns. These nodes can either be connected to other expression nodes or variable declaration nodes with an edge labeled *referenceVar*. A *ValueOf* node connected to a variable declaration node represents that the value of that variable is referred to. Similarly, a *ValueOf* node connected to an expression node means that the resulting value of the expression is referred to (i.e. the return value of a function/method call). These nodes can be connected to attributes of complex types (like a structure) with an edge labeled *referenceAttribute* to model that the value (or the address) of the connected attribute is referenced. The reference to the value (or the address) of an array cell is represented in SCML by edges labeled *referenceLocation* connecting a *ValueOf* (or an *AddressOf*) node to an expression or a variable declaration node.

When a constraint violation is detected, it is important to provide guidelines about the possible location of the problem to the user. We use the physical line numbers for providing such a guideline. The integer attribute called *lineNumber* is added to the simple and declaration statement nodes for storing the line number in SCML models.

3.2 First Tier: Constraint violation detection

The template for modeling a transformation rule that detects a constraint violation is as follows: the expressions/statements that are needed to identify the location (or a usage) of the expressions/statements on which a constraint is defined are modeled in SCML and placed in the left hand-side of the transformation rule. The expressions/statements that follow the constraint are modeled in SCML and placed in the rule as negative application conditions [10] (when the whole pattern depicted by these nodes/edges occurs in the host graph the rule does not match). The right hand-side of the rule adds a node labeled *constraint* (we refer to these edges as constraint nodes), this node is connected to the simple or the declaration statement(s) where the constraint is violated by an edge. The label of this edge is the name of the constraint and it is used in querying for the constraint in the second tier. The text describing the constraint is added as the attribute *description* to the constraint node. Here, the rule can be programmed to concatenate this text with the names of the program elements (an example of this is shown in section 4). Lastly, the rule is modeled to copy the attribute line-number from the implementation and declaration statement(s) where the constraint is violated to the attribute *problemLine* of the constraint node.

In Section 2, the constraints of the sector initialization for the request *GENERAL_STATI* is presented, which is setting the duration of the request to the value of the macro *GENERIC_DUR*. Figure 5 presents a graph transformation rule which is modeled according to the template described above, used for detecting the violation of this constraint.

Both left hand-side and right hand-side of the graph transformation rule are shown in the same graph, using some notational constructions in GROOVE. The dashed edges and the dashed bordered nodes present the negative application conditions. Thick edges and the thick bordered nodes are graph-elements that are added by the transformation rules. All nodes and edges except the circular and the thick ones belong to the left hand-side of the rule. The right hand-side contains all the edges and nodes that are not dashed.

In Figure 5, the node $n4$ is the reference to the macro *GEN_SEOS*. The first parameter of this macro is the value of the variable *GENERAL_STATI* (node $n11$). These nodes are in the left hand-side of the rule, because they are required to identify/locate the usage of the macro *GEN_SEOS*

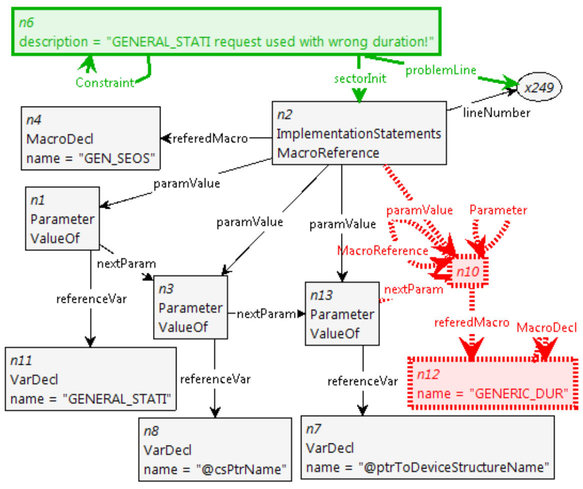

Figure 5: The graph transformation rule modeling the constraint for the request *GENERAL_STATI*

with the parameter *GENERAL_STATI*. The last parameter of the macro *GEN_SEOS* is a macro reference expression (node $n10$) referring to the macro *GENERIC_DUR* (node $n12$). These nodes are negative application conditions of the transformation rule and with these nodes and edges the rule detects the violation of the duration constraint of the request. If the last parameter is a macro reference expression to the macro *GENERIC_DUR* then the rule does not match. If, on the other hand, the last parameter is another expression (e.g. a reference to a different macro) then the constraint is violated and the rule matches.

The constraint node added by this rule is the node $n6$. The edge from the macro reference node $n2$ labeled *lineNumber* represents the attribute with the same name. This edge is connected to the node $x249$, which represents the value of the attribute *lineNumber* and it matches to any value of the attribute. The rule adds an edge labeled *problemLine* from the constraint node to the value node of the attribute *lineNumber*. This is how the copying of the attribute *lineNumber* to the *problemLine* attribute of the constraint node is modeled.

3.2.1 *Parameterizations of Names*

For some constraints, the names of the program elements they work on (e.g. the names of the variables or functions) may change. Rather then modeling a different constraint for each possible name, we parameterize these.

A name or a value that is parameterized starts with the @ character. During the fetch of the transformation rules from the repository, the CDE tool forms a binding file containing all the parameters. The binding file contains a listing of the form $@parameter = givenName$. The developer fills this file. Then, the CDE tool binds the transformation rules by replacing the parameters with the values supplied by the developer.

In the example presented in Figure 1 the values stored in the variables *device_ptr* and *cs_ptr* are "passed" to the macros; however, the developers are free to change the names of these variables in the upcoming versions of the control software. If the names of these variables were fixed in the transformation rule, then the rule could not be used when the names of the variables are changed even though the same constraint still applies. In Figure 5, the name of the variable that holds the register mapping (node $n11$) is set to *@csPtrName* meaning that the name of this variable is parameterized. When the developer wants to check for this constraint, the binding file formed by the CDE tool contains an entry for this parameter and the developer has to supply the name used for this structure (e.g. *cs_ptr*).

3.3 Second Tier: Constraint Querying

In order to find which constraints are violated, the graph production system is *simulated*. The simulation automatically applies the matching transformation rules. It also generates a state-space, in which, the loaded SCML model of the source code is the start state and the transitions are the applied transformation rules. At certain states, no more transformation rules can be applied; these states are called *final states*. The final states are the graphs that contain all the constraint nodes added by the transformation rules.

Manually searching for the constraint nodes in these graphs can be a cumbersome task, depending on the size of the source code. Obviously, there is need for a querying mechanism that automatically searches for the constraint nodes for the constraints the developer is interested in. We use Prolog to provide such a querying mechanism, because it is possible to search for a combination of constraints using Prolog rules. We use Gnu Prolog [1], a Prolog engine written in Java whose predicates can refer to Java methods. We tied this Prolog engine to GROOVE and extended GROOVE with the panel *Constraint Query* from which the developer can enter the queries (or write Prolog programs).

The developer can search for a constraint violation with the predicate *constraint(Name, Line, Description)*. Here, the developer only needs to provide the name of the constraint as *constraint(Name)*. Below the implementation of the predicate *constraint* is given:

```
constraint(Name, Description,Line):- graph(G),
label_edge(G,Name,E),edge_source(E,N),
node_self_edges(G,N,['Constraint']),
node_with_attribute(G,N,'Description',Description),
node_with_attribute(G,N,'problemLine',Line).
```

These predicates all refer to Java methods that search for the final states and the graphs for the provided methods. When a user searches for *constraint('checkSetNodeId')*, the evaluation of the above rule first searches for an edge whose label is *'checkSetNodeId'*. If such a node exists, it finds the *constraint* node this edge is connecting and returns the line number and the description attributes of this node.

Some constraints can be a combination of other constraints. Rather than modeling a new graph transformation rule for these constraints, they can be expressed in terms of other constraints also by using Prolog rules. A Prolog rule for a constraint that depends on three other constraints is presented below:

```
converterClass(ErrorDescription,LineNumber) :-
constraint('checkSetNodeId',ErrorDescription,LineNumber);
constraint('checkGetNodeId',ErrorDescription,LineNumber);
constraint('converterEClass',ErrorDescription,LineNumber).
```

4. APPLICATION OF THE CACV

In this section, we provide examples constraints from the control software showing how our approach addresses the

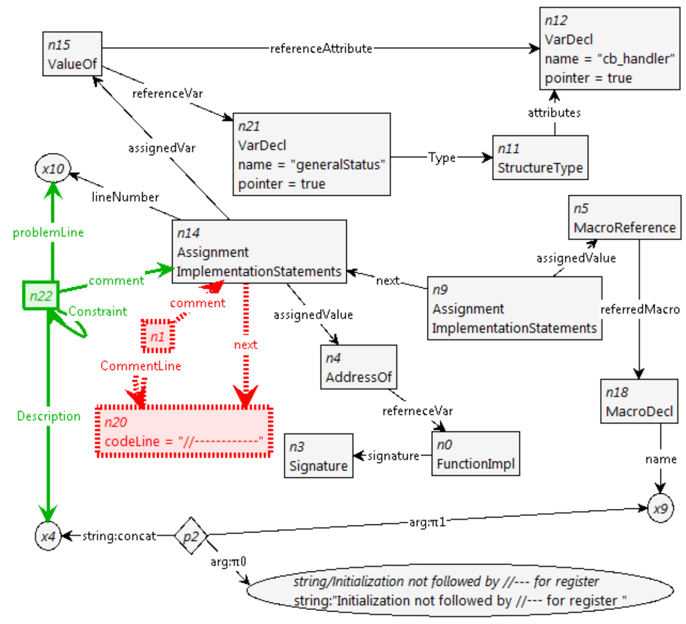

Figure 6: The code convention that checks for the comment block after the initialization of the structures for a register.

drawbacks listed in the introduction section and, then, show how constraints stay valid through evolution.

4.1 Expressivity

Usually, industrial software systems have constraints for increasing the maintainability of the source code. As a result, it is important to verify that the developers follow these constraints. One example of such a constraint from the control software is using comment blocks to separate different consecutive initializations of a structure with the comment line "//–". For example, the general status registers of the amplifier are represented by a structure called *GeneralStatus* in the software. An amplifier has 14 general status; so, the control software has a global array of 14 elements whose type is *GeneralStatus*. In an initialization function, this array is filled; each element's initialization consists of 5 assignment statements. Obviously, when these initializations are placed consecutively it is hard to locate a register's initialization; so, the developers separate them with comment lines.

The AST of the source code is at a lower level than the actual source code the developers sees. Due to this, constraints over some program elements, like comments, cannot be expressed. SCML is designed to represent the source code elements; so, it is possible to express constraints with respect to comments.

Figure 6 presents the transformation rule that checks whether initialization of the structure *GeneralStatus* is followed by the comment block. Here, the node $n14$ is the last assignment statement used for filling the structure. This node is followed by a comment line, node $n20$, whose attribute *codeLine* is set to the comment "//—". The comment line node is a negative application condition of the rule; thus, this rule only matches when the last assignment statement is not followed by the comment block.

The description text for this constraint provides better

```
variable(device_ptr,DPID),variable(cs_ptr,CPID),value(3000,VID),
variable(GENERAL_STATI,GSID),variable(device_request,DREQID),
valueOf(DPIP,DREQID,LID),valueOf(GSID,RID),              as-
sign(LID,RID),
...
variable(dur,DURID),not(valueOf(DPID,DURID,DURASSIGNID),
assign(DURASSIGNID,VID))
```

Figure 7: The constraint for verifying line 3 of Figure 1 expressed in Prolog at the AST level.

guidelines when it also includes the name of the macro used to represent the registers. Because this name changes according to the statement to which the rule matches, the description text in this rule is not fixed. The rule includes the name of the macro into the description text, by concatenating the description text with the name of the macro. In the figure, the node $n18$ is the macro declaration representing the register. The node $x9$ is the value of the attribute *name* of this macro; that is, it is the name of the macro. The description text and the name are used in a production (an attribute operation) represented by node $p2$. The outgoing edge labeled *concat* states that the operation of the production is string concatenation. This edge is connected to the node $x4$ stating that the concatenated string is stored at this node. The node $x4$ is the value of the attribute *Description* of the constraint node $n22$; thus, when the rule is applied the description text contains the name of the macro that represents the register.

4.2 Comprehensibility

The approaches to constraint checking in the literature convert all nodes and edges of the abstract syntax tree to predicates of Prolog-like languages. In Figure 7 the Prolog rule at the AST level used for verifying the constraint over the request *GENERAL_STATI* and it's duration value *GENERIC_DUR* is shown (the transformation rule for detecting the violation of this constraint is shown in Figure 5). First of all, the developer could not understand what this constraint is verifying because there is no reference to the macro *GEN_SEOS* and the macro *GENERIC_DUR*. Secondly, it is very hard to understand the portion of the syntax tree the rule is verifying. Obviously, the developers would better understand what the rule is verifying when the constraint visualizes the syntax tree with program elements the developers is familiar with.

The SCML can be thought of as a syntax tree representing the source code seen by the developers. The SCML models are attributed graphs and they can be visualized with graph editors.

4.3 Error Reporting

It is important to provide guidelines to aid the developers in locating the violation at the source code when a constraint violation is detected. Because of the differences between the abstract syntax and the source code, the guidelines provided by the approaches that work at the AST level may be hard to locate at the source code.

SCML preserves the location of the statements with the attribute named *lineNumber*. Thus, besides the names of the program elements and a description text, the CACV error reports also include the line number of the statement(s) where the violation has occurred. For example, assume the reference to macro *GEN_SEOS* with request *GENERAL_STATI* is used with a duration value other than *GENERIC_DUR* at line 1045 of a source file from the control software. Then, the transformation rule shown in Figure 5 matches and adds the constraint node where the attribute *Description* is *GENERAL_STATI request used with wrong duration* and the attribute *problemLine* is *1045*. The developer can locate this constraint node and learn that she/he used the wrong duration at line *1045*.

For the example given in the above paragraph, the developer and other stakeholders can also use the querying mechanism and query whether she/he has violated the constraint *sectorInit* by typing *constraint('sectorInit')* into the *Constraint Query* panel. Then, the query returns:

Line = 1045

Description = GENERAL_STATI request used with wrong duration

Similarly, it is possible to program a Prolog rule that checks the constraints for all queries as:

```
sectorConstraint(Line,Description) :-
constraint('sectorInit',Line,Description);
constraint('sectorGeneral',Line,Description); ...
```

4.4 Constraints and Software Evolution

We applied CACV to the three different implementations of the gradient amplifier control software to see how many of the constraints remained valid through evolution. From the third implementation of the control software for the latest amplifier model, we identified 10 frequently used constraints, 7 program constraints and 3 coding conventions (coding conventions are used 87 times in total and the 7 program constraints are used 137 times in total in the latest implementation). We modeled these constraints and used the tool to verify whether these constraints are violated or not. It may also be possible that some constraints are only in the previous implementations of the control software or only at the third implementation of the control software. To identify such constraints, we manually searched for constraints at the previous two implementations.

The tool showed no constraint violations in the second implementation (the verification took less than a second) and only one constraint violation in the first implementation (the verification took 5 seconds). Manual search for constraints revealed that the code segment for one of the identified 10 constraints is not used in the first implementation but used in the second and the third implementations of the control software. The code conventions are the same over all versions of the control software. This analysis shows that constraints indeed need to be respected during evolution.

5. RELATED WORK

The CACV approach converts the source code to an SCML model on which the constraints can be checked. There are approaches in the literature where the program constraints can be checked within the domain of the programming language. These approaches can be split into three groups. The approaches in the first group are used for detecting the structural constraints. JQuery [13] is an approach based on predicate logic where the structure of the Java programs is transferred to Prolog facts and the structural constraints are expressed as Prolog facts. The *Java Tools Language* [5] (JTL) is a declarative language designed for selecting Java structural program elements and its semantics are based on predicate logic. Compared to our approach these approaches are limited because they only check the structural constraints and they ignore the implementation of the methods.

The second group of approaches are the approaches based on AST querying. Examples of these approaches include CodeQuest [11] for Java, which is based on a subset of Prolog and ASTLOG [7] for C/C++. Both approaches work on the AST level and thus constraints on program elements like comments and macros cannot be expressed or checked. Our observations with the industry have shown that macros are used very frequently and usually the developers do not know the implementation of the macro. As a result, they cannot write queries to search for constraints that use macros. Moreover, complex industrial software is a combination of different languages; using different tools to check for constraints of different languages complicates the procedure as the developers have to understand both tools. This led to the development of SCML, which can be used to represent both object-oriented and structural languages (i.e. it abstracts from the syntax) and which contains elements from the source code. We presented an example where SCML is used to check whether the developer has used the comment blocks correctly.

Eichberg et al. [8] propose an approach for checking dependency constraints, where the program elements are grouped into *ensembles* and the dependency constraints are defined between the ensembles. This approach converts a subset of AST to predicates; this subset contains declarations and implementation statements that can cause dependencies between program elements. This approach also has two tiers: at the bottom level the core program elements and at the top tier the dependency constraints between the groups of the elements are expressed. The main difference is that our approach is not limited to checking dependency constraints.

In the third group of approaches extensions to type systems are used. The current type checking of programming languages is too limited; it does not check the constraints. Bracha et al. [3] proposed pluggable type systems, extensions to type checkers that include constraints. Andreae et al. [2] implement a plugabble type system for Java AST. This system uses Java annotations and declarative rules for defining semantics of these annotations. Because the type system ignores macros and comments, with this approach the constraints over these program elements cannot be checked.

Once the source code is converted to an SCML model, the graph transformations are used for detecting the constraint violations. The SCML models are attributed graphs, so the transition from model to graph is straight-forward. It may be argued that in the model domain (i.e. without transition to graphs) the constraints can be checked using model transformations. When a constraint violation is detected in our approach, the model is updated with information about the violated constraint. For model updates, graph transformations are more suitable than model transformations [9]. Also, in the literature there are many applications of graph transformations to pattern detection. For example, Niere et al. [15] use graph transformation rules to detect and recover the design patterns from source code semi-automatically. Mens et al. [14] use transformation rules to detect inconsistencies between UML models. In these approaches, the right hand-side of the transformation rule is used for adding nodes/edges for marking which nodes/edges match to the pattern the rule has detected. Due to this wide usage and the suitability of graph transformations to model updates, we used graph transformations in our approach. In addition to marking the nodes, we take advantage of attribute operations to extract information useable for guiding the developers to the locations of the constraint violations.

6. CONCLUSIONS

This paper presents a process for computer-aided verification of static program constraints and coding conventions. The main contribution of this process is using meta-modeling for representing the source files; we called this meta-model Source Code Modeling Language (SCML). In this way, constraints over program elements such as macros and comments can be expressed.

Graph transformations are used for detecting the violations of the constraints. The right hand-side of the rule is used for extracting information from the model of the source code that can provide guidelines to the developers about the problem. An important aspect of the transformation rules in our approach is that the names of the program elements are parameterized when the transformation rules are modeled. The names of variables may change, for example, from implementation to implementation. The constraints over this variable stay the same; so, rather then modeling different constraints for each name, the value to be checked is parameterized.

It may be hard to manage the constraints of a large software system. To address this problem, we utilize a repository where the constraints are stored in a central storage managed by a repository manager tool created by us.

The model of a software component may contain many nodes and edges; thus, it may be hard for the users to locate the nodes labeled *constraint*. We combined Prolog and GROOVE, so that the developer can enter Prolog queries to find constraint violations. In this way, it is also possible to express *high-level* constraints (i.e. constraints that hide information) or express constraints that are a combination of other constraints, using Prolog rules.

Using the approach, we showed how frequently used constraints on source elements, such as macros and comments, can be checked. Moreover, we tested the approach using the history of an industrial software system; we indeed found that it is important to check for constraint violations during evolution.

Acknowledgements

We thank Ivan Kurtev (University of Twente), Pierre van de Laar (Embedded Systems Institute) and Lennart Hofland (Philips Healthcare) for their suggestions and help on this work.

7. REFERENCES

[1] Gnu prolog. http://www.gprolog.org/.

[2] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10):57–74, 2006.

[3] G. Bracha. Pluggable type systems. *OOPSLA workshop on revival of dynamic languages*, 2004.

[4] S. Ciraci, P. van den Broek, and M. Aksit. Framework for computer-aided evolution of object-oriented designs. *COMPSAC*, pages 757–764, 2008.

[5] T. Cohen, J. Y. Gil, and I. Maman. Jtl: the java tools language. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 89–108, New York, NY, USA, 2006. ACM.

[6] M. A. Covington, D. Nute, and A. Vellino. *Prolog programming in depth.* Scott, Foresman & Co., Glenview, IL, USA, 1987.

[7] R. F. Crew. Astlog: a language for examining abstract syntax trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 18–18, Berkeley, CA, USA, 1997. USENIX Association.

[8] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[9] R. Gronmo, B. Moller-Pedersen, and G. K. Olsen. Comparison of three model transformation languages. In *ECMDA-FA '09*, pages 2–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundam. Inf.*, 26(3-4):287–313, 1996.

[11] E. Hajiyev, M. Verbaere, O. de Moor, and K. de Volder. Codequest: querying source code with datalog. In *OOPSLA '05*, pages 102–103, New York, NY, USA, 2005. ACM.

[12] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:: a dsl for the specification of textual concrete syntaxes in model engineering. In *GPCE '06*, pages 249–254, New York, NY, USA, 2006. ACM.

[13] E. McCormick and K. De Volder. Jquery: finding your way through tangled code. In *OOPSLA '04*, pages 9–10, New York, NY, USA, 2004. ACM.

[14] T. Mens, R. van der Straeten, and M. DŠHondt. Detecting and resolving model inconsistencies using transformation dependency analysis. In *Model Driven Eng. Lang. and Sys.*, volume 4199/2006, pages 200–214, 2006.

[15] J. Niere and et al. Towards pattern-based design recovery. In *ICSE '02*, pages 338–348, 2002.

[16] A. Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Trans. with Industrial Relevance*, pages 479–485. Springer-Verlag, 2004.