An abstract specification language for static program analysis

Michael Vistein, Frank Ortmeier, Wolfgang Reif, Ralf Huuck, Ansgar Fehnker

Angaben zur Veröffentlichung / Publication details:

Vistein, Michael, Frank Ortmeier, Wolfgang Reif, Ralf Huuck, and Ansgar Fehnker. 2009. "An abstract specification language for static program analysis." *Electronic Notes in Theoretical Computer Science* 254: 181–97. https://doi.org/10.1016/j.entcs.2009.09.066.

Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 254 (2009) 181–197
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

An Abstract Specification Language for Static Program Analysis

Michael Vistein Frank Ortmeier Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg
Universtitätsstraße 14, 86135 Augsburg, Germany
{vistein, ortmeier, reif}@informatik.uni-augsburg.de

Ralf Huuck Ansgar Fehnker

National ICT Australia Ltd. (NICTA)
University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
{ralf.huuck, ansgar.fehnker}@nicta.com.au

Abstract

Static program analysers typically come with a set of hard-coded checks, leaving little room for the user to add additional properties. In this work, we present a uniform approach to enable the specification of new static analysis checks in a concise manner. In particular, we present our GPSL/GXSL language to define new control flow checks. The language is closely related to CTL specifications that are combined with XPath queries. Moreover, we provide a number of specifications as implemented in our tool Goanna, and report on our experiences of adding new checks.

Keywords: Abstract specification, static analysis, CTL specification, GPSL/GXSL, XPATH query

1 Introduction

Static program analysis has proven to be a useful tool in detecting bugs and vulnerabilities in commercial source code [4,13,2,11]. Much research has been dedicated at improving the speed, depth, and precision of analysis techniques. However, there has been little work on writing static analysis checks in general or extending existing databases of pre-defined checks to meet domain, company or project specific requirements.

doi:10.1016/j.entcs.2009.09.066

In this work we present our static analysis tool GOANNA [10] and the checker language we developed for adding new checks or modifying existing ones according to needs. GOANNA is a state-of-the-art static analysis tool for C/C++ programs that is based on model checking techniques. GOANNA works primarily on a syntactic program abstraction, i.e., control flow graphs (CFG) that are labelled with objects of interest, e.g., where memory is allocated, used or de-allocated, and does CTL model checking on this abstraction. The CTL property expresses syntactic checks as for instance, whether dynamic memory is still referred to after de-allocation.

Our control-flow centred checker language GPSL/GXSL has been motivated by describing typical checking patterns in a straightforward declarative way. It is split in two parts aiming at two different audiences: The first is GPSL, a high-level abstract language that is a collection of the most common checking patterns plus a large library of pre-defined objects of interest. These pre-defined objects themselves are queries such as for locations where memory is allocated, used or de-allocated. This language is targeted to end users, who like to define new checks based on patterns that are instantiated with objects from that library.

The second language is GXSL. This is a lower-level language that is used to build up the library of pre-defined objects and is more targeted towards developers or experts of static analysis tools. GXSL is a language that allows to query information from a program's abstract syntax tree (AST) and also to express direct CTL checks on them. The advantage of GXSL is that new syntactic objects, e.g., finding all the program variables that are declared as `static`, can be defined quickly and, more important, uniformly.

There are several approaches to define static checks and temporal logic patterns. Engler et al. define the meta-compilation language *Metal* [8] to specify new checks. Metal is in essence a state machine written in a C-like language, where certain accepting states are indicating a property violation. Using C as the underlying specification language has advances as well as disadvantages: On the one hand most developers are already familiar with this language, on the other hand it is known to be error prone and it lacks simple abstraction mechanisms. Moreover, Metal is evaluated on individual program paths making it similar to LTL rather than CTL checking.

Mygcc is a research tool for static code analysis using the Condate language[18]. Condate is a minimalistic declarative language for expressing user-defined checks. It is less powerful than Metal, but supports identifying patterns in a program and defining regular path expressions over them. In nature, the declarative approach is similar to GPSL/GXSL, but GOANNA uses more expressive CTL formulas and supports familiar XPath queries in GXSL.

The most prominent approach to encode high-level patterns as CTL formulas (or vice versa) has been presented by Dwyer et al. [6]. These patterns are used to describe common behavioural requirements of reactive systems. The idea is similar to GPSL, however, targeted at a different domain and there is no direct connection of these patterns and matching points of interest in a program as done in Metal, Mygcc or GPSL/GXSL. A more generic approach to build on temporal logics is the *Property Specification Language* (PSL)[1] for writing hardware specific checks. PSL uses CTL as well as LTL to specify temporal relations of hardware signals. PSL is rather hardware centric and not easily applied to other domains such as static code analysis.

A rather different approach has been chosen by Lam et al. [14]. The authors combine a static Program Query Language (PQL) with a dynamic observer. PQL enables to identify syntactic constructs of interest in a program and instruments the application to make the occurrence of these constructs visible to the outside. In a second step those occurrences are observed at run-time with respect to a pre-defined pattern that has been specified as a state machine. While having some similarities with our work PQL focuses primarily on dynamic behaviour and run-time verification.

In summary, there are different approaches for CTL-like specification languages as well as for syntactic matching, but GPSL/GSXL is the first approach to bring this together for static program analysis.

The remainder of this paper is organized as follows: In Section 2 we introduce the Goanna tool and its underlying framework to motivate some of the choices made in designing the GPSL/GXSL language. The language itself is defined in detail in Section 3. We report on some of our experiences on using GPSL/GXSL for real software in Section 4 and present our conclusions in Section 5.

2 The Goanna Tool

In this work, we use an automata-based static analysis framework that is implemented in our tool Goanna. In contrast to typical equation solving approaches to static analysis, the automata based approach [5,12,17,16] defines properties in terms of temporal logic expressions over annotated graphs. The validity of a property can then be checked automatically by graph exploring techniques such as model checking. Goanna [1] itself is a closed source project, but the technical details of the approach can be found in [9,10].

[1] http://nicta.com.au/research/projects/goanna

```
int foo(int n) {
  int n1, n2, res;
  n1 = 1;
  n2 = 1;
  for(int i = 2; i < n; i++) {
    res = n1 + n2;
    n1 = n2;
    n2 = res;
  }
  return res;
}
```

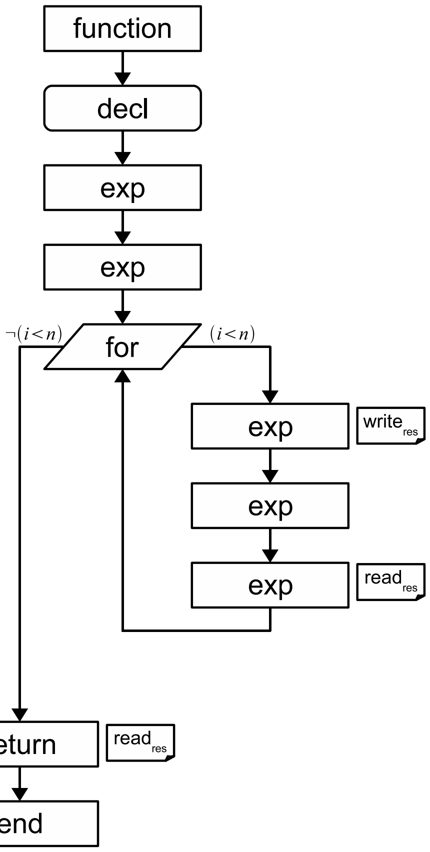

Figure 1. (a) Example C program, and (b) annotated control flow graph (CFG). Each node corresponds to one line-of-code for simplicity.

2.1 Goanna Approach to Source Code Analysis

The basic idea of the Goanna approach is to map a C/C++ program to its corresponding control flow graph (CFG) and to label the CFG with occurrences of syntactic constructs of interest. The CFG together with the labels can easily be mapped to the input language of a model checker or directly translated into a Kripke structure for model checking. Consider the simple example program `foo` in Fig. 1. To check whether variables are written strictly before they are read, we syntactically identify program locations that read or write variables. For variable *res* in Fig. 1 (a) we automatically label these nodes with labels $read_{res}$ and $write_{res}$, respectively, as shown in Fig. 1 (b). Given this annotated CFG, checking whether *res* is always initialized then amounts to checking the following CTL formula.

$$\mathbf{A}\neg read_{res}\mathbf{W}(write_{res} \wedge \neg read_{res}) \tag{1}$$

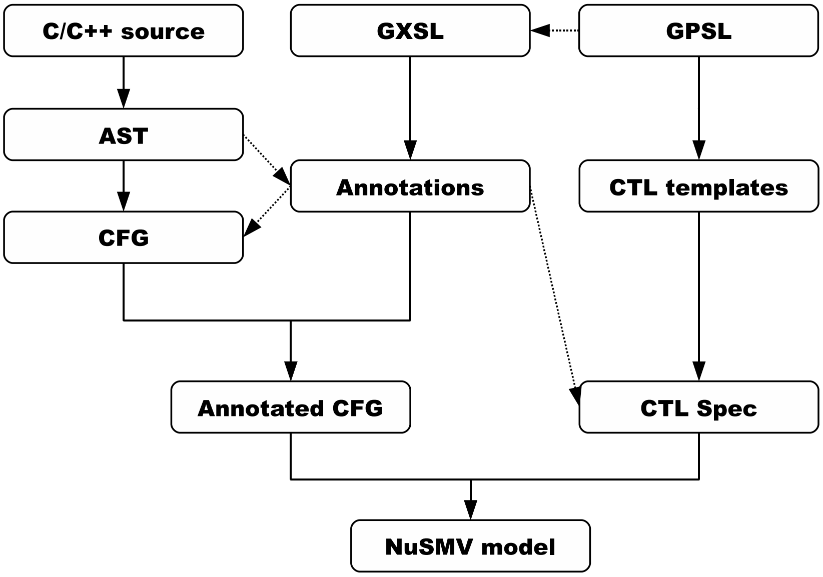

Figure 2. Model checking approach for statically analysing C/C++ code in GOANNA.

CTL uses the path quantifiers $\mathbf{A}$ and $\mathbf{E}$, and the temporal operators $\mathbf{G}, \mathbf{F}, \mathbf{X}$, and $\mathbf{U}$. The (state) formula $\mathbf{A}\phi$ means that ϕ has to hold on all paths, while $\mathbf{E}\phi$ means that ϕ has to hold on some path. The (path) formulae $\mathbf{G}\phi, \mathbf{F}\phi$ and $\mathbf{X}\phi$ mean that ϕ holds globally in all states, in some state, or in the next state of a path, respectively. The *until* $\phi\mathbf{U}\psi$ means that until a state occurs along the path that satisfies ψ, property ϕ has to hold. We also use the *weak until* $\phi\mathbf{W}\psi$. It differs from the *until* in that either ϕ holds until ψ holds, or ϕ holds globally along the path. The *weak until* operator does not require that ψ holds for any state along the paths, as long as ϕ holds everywhere. It can also be expressed in terms of the other operators, and is also called the *unless* operator. CTL formula (1) means that variable *res* can not be read unless it has been written. The clause $(write_{res} \wedge \neg read_{res})$ is used to exclude statements that read and write a variable, like the post-increment `res++` statement. Note, that the annotated CFG in Fig. 1 (b) does not satisfy CTL formula (1).

One advantage of the automata-based approach is that properties can be modified easily to express stronger or weaker requirements by changing the path quantifier, i.e., changing path quantifier $\mathbf{A}$ to $\mathbf{E}$ and vice versa. In the example above replacing $\mathbf{A}$ with $\mathbf{E}$ will check if the variable is initialized on at least one path, rather than on all paths. This also motivated the use of CTL rather than LTL.

2.2 Architecture of the Goanna Tool

Goanna uses NuSMV[15] for model checking the annotated model. In order to analyse a C/C++ program with NuSMV, several steps are required to transform the source code into a suitable Kripke structure, and to generate the associated CTL formulae. We use two specification languages to accomplish this. The *Goanna Properties Specification Language (GPSL)* is used to define CTL templates and the associated atomic propositions. The other language *Goanna XPath Specification Language (GXSL)* is used to map atomic propositions to locations in the source (more precisely, in the control flow graph). Both languages will be described in the next section. The architecture of the automatic NuSMV model creation is depicted in Fig. 2.

The left branch in Fig. 2 reads in the source code, and parses it into an abstract syntax tree (AST). This AST is stored as an XML document. In the next step, the control flow graph is generated out of the AST. The CFG can be seen as a finite state machine, and thus is already quite close to the final NuSMV model. Apart from this, the labels (atomic propositions) and the CTL formulas themselves need to be generated.

The GPSL properties contain templates for the CTL formulas. The templates will be instantiated with query results on the AST at run-time. E.g., queries for all variables, for all reads and all writes. These queries are expressed in GXSL. Also, the CFG nodes for matching query results are subsequently used as labels in the control flow graph, resulting in the annotated control flow graph. In the last step, the annotated control flow graph and the CTL specifications are combined to form the input model for NuSMV. NuSMV performs model checking, and returns a list of all failed CTL specifications. At the moment, this approach supports intra-procedural checks. An extension of the tool as well as of the specification languages to support inter-procedural analysis is currently under development.

3 The GPSL/GXSL Specification Language

Generally, a static analysis check in Goanna requires that certain statements in the source code are executed in a specific order. One standard example for such a property is that, whenever a variable is read, it must have been assigned a value earlier. The specification of a property consists of two parts: Firstly, the temporal relations of certain points-of-interest in the control flow. This is defined with the *Goanna Properties Specification Language (GPSL)*, a library of predefined patterns for commonly used relations. Secondly, to define the points-of-interest, e.g. function calls, variable assignments etc., the *Goanna XPath Specification Language (GXSL)* will be used. It allows the selection of program statements using the XPath query language on the abstract syntax tree of the program under inspection.

3.1 Goanna Property Specification Language

Temporal relations are commonly specified using temporal logics. There are several different flavours of temporal logics; common are *Computational Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). As described in Section 2, Goanna uses CTL. For static analysis, it is helpful to be able to distinguish among different possible execution paths. Statements over implicitly all possible paths, as done in LTL are often not sufficient.

For static analysis purposes, it is desirable to be able to quantify properties over sets of, e.g., program variables. For instance, for all variables of a program a certain property should hold, or for all variables that are used in certain operations only. CTL alone is not enough as it does not allow quantification over sets of atomic propositions. E.g., the CTL formula (1) expresses a check for the variable `res` of program 1 (a) only.

The Goanna Property Specification Language extends CTL in such a way that it is possible to quantify CTL expressions over sets of syntactical constructs of the source code, like all variables, all for-loops, all function calls etc. Moreover, GPSL provides a mechanism to issue tailored warning messages, as well as referring to specific parts in a counter example, i.e., the likely source of an error.
Therefore, any GPSL property consist of the following parts:

- a unique identifier for a check,
- a quantification over a set of objects,
- a CTL template that will be instantiated with these objects, and
- a warning entry, containing the warning message and the position in the counter example to warn about.

We first provide a brief example and then explain the different aspects.

3.1.1 Example

The following example shows the "uninitialised variable" property. This property checks whether a variable in the program has been initialised before being used. The complete grammar of GPSL is included in Appendix A.1.

```
PROP uninitialised_var
  FORALL var IN var_decl()
  ALWAYS write(var) STRICTLY BEFORE read(var)
  WARN LINE last(read(var))
    WITH "Variable␣`" ^ varname(var) ^
    "'␣might␣be␣uninitialised"
```

Table 1: GPSL property that checks for uninitialised variables

The `PROP` keyword marks the beginning of a new property, followed by the name of the property. The following line quantifies the subsequently specified relation over all variables of the program source (see also Section 3.1.2). The relation is specified using a temporal pattern, in this case the `ALWAYS ... STRICTLY BEFORE ...` pattern is applied. More details about the different patterns are explained in Section 3.1.3. The program points that must fulfil the temporal relation are specified using GXSL functions (see also Section 3.2). The last three lines specify the warning message. The line is specified to be the last line where a read of the variable has occurred, and the message itself is specified as string. Warning messages are explained more in detail in Section 3.1.4.

3.1.2 *Quantification*

One important extension of GPSL over CTL is the ability to quantify over expressions. This allows to specify one temporal relation, which is then used several times at runtime to check the relation for all specified items (like all variables of a program). Therefore, GPSL evaluates a GXSL function to extract relevant constructs, like the list of all variables, and provides this information in a newly declared variable that can be used subsequently within the temporal relation.

3.1.3 *Patterns*

GPSL provides a range of pre-defined patterns to specify properties. These patterns define some of the most commonly used temporal relations in static program analysis, and are partly based on ideas by Dwyer et al.[7,6]. Each pattern is mapped to a CTL formula expressing the desired relation. The patterns contain variables a, b, etc. that must refer to GXSL functions like `read(var)` or `write(var)` (see Section 3.2). The control flow graph will be labelled accordingly to the results of these function calls.

The following list contains some of the patterns that are defined in Goanna with their CTL equivalent and an informal explanation of the pattern.

- `NEVER` a, (CTL: `AG !`a)
  This pattern requires that there is no node labelled with a that can be reached.
  This pattern can be used, amongst other things, for enforcing coding standards, e.g. to generally disallow certain functions, statements, etc.
- `EVENTUALLY` a, (CTL: `EF` a)
  This pattern requires that at least one reachable node is labelled with a.
  One usage example for this pattern is also the enforcement of coding standards. It allows to design a check whether some statement (e.g. logging) occurs at least once in each function.
- `ALWAYS` a `BEFORE` b, (CTL: `A [!`b `W` a`]`)
  On all possible execution paths, if there is a node labelled with b, there must have been a node labelled with a before. If there is no node labelled with b, then there is no need for a node labelled with a either. This pattern can be extended with the `STRICTLY` keyword, which checks that the first node labelled with a is not the same one as the node labelled with b. It is realised by the CTL formula `A [!`b `W (`a `& !`b`)]`.
  An example for the application of this pattern is the check for uninitialised variables (c.f. Section 3.1.1).
- `AFTER` a `ALWAYS` b, (CTL: `AG (`$a \rightarrow$ `AF` b`)`)
  If there is a node labelled with a, all possible execution paths from this point on must have at least one node labelled with b.
  This pattern can be used for resource management, e.g. guaranteeing that resources that are allocated are eventually released. An example of this pattern can be seen in Section 3.3.
- `AFTER` a `SOME` b, (CTL: `AG (`$a \rightarrow$ `EF` b`)`)
  If there is a node labelled with a, at least one possible execution path from this point on must have at least one node labelled with b.
  This pattern is a weaker form of the `AFTER` a `ALWAYS` b pattern, which also allows programs where the condition b only happens on some of the possible execution paths.

The previously explained patterns are only a small excerpt of all available patterns. Some of these patterns are used in the examples of this paper, and thus have been chosen for a more detailed explanation. Goanna currently supports over 20 different patterns, and the library is evolving continuously.

3.1.4 Warning Message

If a property is violated a warning message must be displayed to the developer. GPSL provides the means to specify a warning message that can be instantiated with a variable name and a line number, that will result from a certain location in the counter-example. This location will be determined by the occurrence of a certain node in that counter-example. Typically, it is sufficient to use the `first` or `last` such occurrence.

3.2 Goanna XPath Specification Language

To instantiate the GPSL patterns with relevant nodes, we use the GXSL language. The example in Section 3.1.1 contained already GXSL function calls like `read(var)` and `write(var)` that are returning node locations.

The identification of statements can be performed on the *abstract syntax tree* of the program. The XPath[3] language is well suited for expressing queries on tree shaped data, like an abstract syntax tree. However, XPath queries themselves are typically only static strings. For each different program these strings must be dynamically applied depending on variable names, function calls or function parameters. These names are typically not known at the time the query is designed.

The GXSL language extends XPath with functions and variables. A GXSL function is a function that returns a set of nodes of the control flow graph to be labelled[2]. Because the labels generated by Goanna must be unique within the control flow graph, each GXSL function must also have a unique identifier that can be incorporated in the label name. The GXSL functions may have a parameter that can be used to instantiate the generic XPath query (e.g. the `var` in the `read(var)` example). The parameter is specified as a node of the CFG, which allows the GPSL language to use a parameter-less GXSL function to quantify a temporal relation, and to use the single items of the result set as parameters for further GXSL functions. Additionally to the parametrized XPath query and the unique identifier, GXSL also adds a type system.
A GXSL property consist of the following parts:

- a function name, possibly with a parameter,
- an expected input type as well as an output type,
- a unique identifier used for labelling of nodes, and
- the XPath query itself.

[2] Although the GXSL functions work on the abstract syntax tree, the relevant information for the use in GPSL properties are the nodes of the control flow graph. Every item in the abstract syntax tree can be mapped to a node in the control flow graph

3.2.1 Example

The following example defines the `write` function. This function returns the set of all nodes where a variable has been written. The complete grammar of the GXSL language can be found in Appendix A.2.

```
FUN write(node)
  EXP decl RET writevar
  IDENTIFIER @id(node)
  NODESET
    "//Op2[@op='Modif']/*[@role='var'␣and␣@idref='"^
       @id(node) ^"']"
```

Table 2: GXSL function `write`, which selects all nodes where a certain variable is written.

The "`FUN`" keyword starts a new function declaration, followed by the name of the function "write", and the function's parameter "`node`". The "`write`" function expects a parameter of the type "`decl`", which refers to nodes where a variable declaration occurred. It returns nodes of the type "`writevar`" where a variable has been written. The unique identifier is defined to be the `id` attribute of the declaration node (which is always unique). The last part is the XPath query, which is parametrised using the variable `node`. The XPath query searches for every binary operation which is a modification operation. For all these operations, the left hand child (the variable that is assigned a new value) is looked up compared with the variable for which the GXSL function should find write nodes. The query is specified with two strings, which are concatenated with the "`id`" attribute of the parameter "`node`". This part of the query is evaluated to a string at runtime.

3.3 Domain-specific example of GPSL and GXSL

The following listings show a more complicated, domain-specific GPSL property and the necessary GXSL functions. The goal of the property is to check if all database connections have been closed correctly. Two GXSL functions are necessary for this specification. The first function ("`connections()`") returns a list of all variables that are assigned a database connection handle (the database connection function is on the right hand side of an assignment). The second GXSL function ("`close(conn)`") returns all nodes, where a database connection with a certain handle is closed.

```
FUN connections()
  RET dbconn
  NODESET
    "//Op2[@op='Modif']/Var[@role='op1'␣and␣../FnEval[
      @name␣=␣'db_connect']]"

FUN close(conn)
  EXP dbconn RET dbclose
  NODESET
    "//FnEval[@name␣=␣'db_close'␣and␣/Param[@idref␣=␣"
        ^ @id(conn) ^ "]]"
```

The GPSL property makes use of the "after ... always ..." pattern. This pattern ensures that, if some event happens, another event eventually happens on all possible paths of exection.

```
PROP closedb
    FORALL conn IN connections()
    AFTER conn ALWAYS close(conn)
    WARN LINE conn WITH "Database␣connection␣not␣closed"
```

The "closedb" property checks for every connection handle whether its connection is eventually closed (the close method has been called with the appropriate handle). If this property is violated, the line number of the last access to the connection handle will be displayed along with the message "Database connection not closed".

4 Experience

In this section we briefly report on our experiences of using the GPSL/GXSL language to define new checks and its effects on Goanna's scalability.

Writing static analysis checks in a declarative manner rather than as an operational state machine has several advantages and disadvantages: First of all, state machines are easier to be drawn on paper and, therefore, provide a better reference for discussion. However, they require more algorithmic thinking in terms of "if-then-else" rather than expressing a property directly. To this extent, declarative checks are much more succinct. Also, declarative requirements are easy to adapt, e.g., whether a property should hold on all paths or at least on one. This helped greatly to create a variety of checks of different strength quickly.

Generally, learning and developing checks using the GPSL language has been proven straightforward for programmers who used the language. On the other side, the learning curve for GXSL proved to be steeper. We see two reasons for this: On the one hand GXSL requires the developer to be

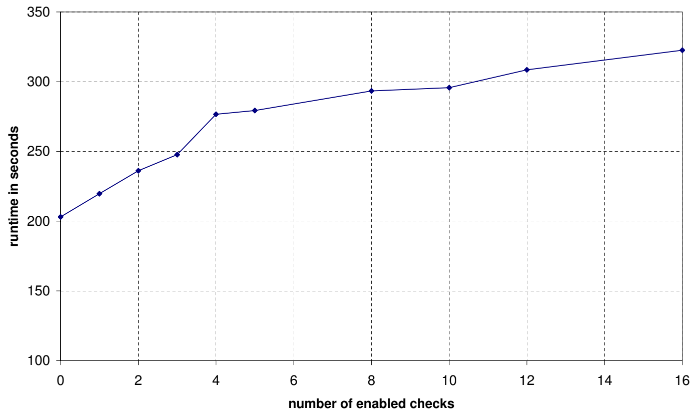

Figure 3. Scalability for increasing numbers of checks.

fairly familiar with the AST Goanna produces. To write XPath queries that pick exactly the right nodes out of the AST and taking the many subtleties of C/C++ into account, involves sufficient experience that needs to build up over time. On the other hand, we currently have no good debugging mechanism for GXSL. This means, once nodes are queried and combined, there is no direct debugging feature/output that shows immediate query results and selected node. This certainly needs to be address to make GPSL more user-friendly.

One positive result of using GPSL/GXSL has been that it can significantly reduce the development time for new checks. Often new properties are written in a matter of minutes and this makes a great difference for the Goanna tool development in general.

While some general performance indicators for Goanna can be found in [10], here we focus on the scalability of checks, i.e., the run-time performance for an increasing numbers of checks. Figure 3 depicts a typical scalability result. We observe the run-time with respect to different numbers of checks for a given software package. Running Goanna without any checks, i.e., just standard compilation and parsing by Goanna took roughly 200s for a software package containing roughly 3, 500 functions. Adding more checks successively increased the run-time only moderately. It is worth to mention that different checks have different complexities and, as seen earlier, a single check should rather been seen as a class of checks as it sometimes ranges over dozens or even hundreds of different program variables. It can be seen, however, that adding 16 classes of checks does not even double the total run-time and it grows sub-linearly overall. The reason for the latter is that certain objects, such as writes or reads to variables are shared by many checks and some optimization effects occur.

In general, GPSL/GXSL has proven to be no worse than hand-optimized hard-coded checks. In fact, in many cases the GPSL/GXSL proved superior due to more efficient caching and reuse of queried objects. The results of GXSL functions are automatically cached without the need for the developer to care about saving results and looking up previous data. With this cache, other properties that use the same set of nodes (e.g. properties which also need nodes where variables are written) immediately benefit from not having to perform potentially time consuming node lookups again.

5 Conclusion

This paper presents an approach for specifying user-defined checks. It allows for convenient, automatic, user-defined labelling of programs and specifying complex temporal relationships easily among those labels using an expressive set of predefined patterns. This specification method has been successfully integrated into a modern static analysis tool (Goanna) and is now a standard component of this tool. Nevertheless the specification languages themselves are independent of the tool and the programming language that should be analysed.

Future work is to reduce some of the limitations of the language as well as extending it include non-control specific requirements. There are a number of features in the Goanna tool, that are not linked with the current GPSL/GXSL language such as: inter-procedural checks, aliasing, or data dependent array value analysis. It would be desirable to extend the current language in a such a fashion that it can make use of the results and features of non-control specific checkers.

Acknowledgement

We like to thank Wilhelm von Rosenberg for his contribution to this work. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] Accellera, "Property Specification Language Reference Manual," (2004), available online at http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

[2] Astrée, *The Astrée Static Code Analyzer*, Website, http://www.astree.ens.fr/; visited on 25 February 2009.

[3] Clark, J. and S. DeRose, "XML Path Language 1.0 (XPath)," W3C (1999).
URL http://www.w3.org/TR/xpath

[4] Coverity, *Prevent*, Website, http://www.coverity.com/html/prevent-for-c-c++.html; visited on 25 February 2009.

[5] Dams, D. and K. Namjoshi, *Orion: High-precision methods for static error analysis of C and C++ programs*, Bell Labs Tech. Mem. ITD-04-45263Z, Lucent Technologies (2004).

[6] Dwyer, M., G. Avrunin and J. Corbett, *Patterns in property specifications for finite-state verification*, in: *Proc. International Conference on Software Engineering*, 1999, pp. 411–420.

[7] Dwyer, M. B., G. S. Avrunin and J. C. Corbett, *Property specification patterns for finite-state verification*, in: *FMSP '98: Proceedings of the second workshop on Formal methods in software practice* (1998), pp. 7–15.

[8] Engler, D., B. Chelf, A. Chou and S. Hallem, *Checking system rules using system-specific, programmer-written compiler extensions*, in: *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, San Diego, CA*, 2000.

[9] Fehnker, A., R. Huuck, P. Jayet, M. Lussenburg and F. Rauch, *Goanna - a static model checker*, in: L. Brim, B. R. Haverkort, M. Leucker and J. van de Pol, editors, *FMICS/PDMC*, Lecture Notes in Computer Science **4346** (2006), pp. 297–300.

[10] Fehnker, A., R. Huuck, P. Jayet, M. Lussenburg and F. Rauch, *Model checking software at compile time*, in: *Proc. First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering TASE '07*, 2007, pp. 45–56.

[11] Grammatech, *Codesurfer*, Website,
http://www.grammatech.com/products/codesurfer/overview.html; visited on 25 February 2009.

[12] Holzmann, G., *Static source code checking for user-defined properties*, in: *Proc. IDPT 2002*, Pasadena, CA, USA, 2002.

[13] Klocwork, *Insight*, Website,
http://www.klocwork.com/products/insight.asp; visited on 25 February 2009.

[14] Martin, M., B. Livshits and M. S. Lam, *Finding application errors and security flaws using pql: a program query language*, in: *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (2005), pp. 365–383.

[15] NuSMV, *A new symbolic model checker*, Website, http://nusmv.irst.itc.it/; visited on 25 February 2009.

[16] Schmidt, D. A., *Data flow analysis is model checking of abstract interpretations*, in: *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), pp. 38–48.

[17] Schmidt, D. A. and B. Steffen, *Program analysis as model checking of abstract interpretations*, in: *Static Analysis Symposium*, 1998, pp. 351–380.

[18] Volanschi, N., *A portable compiler-integrated approach to permanent checking*, in: *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering* (2006), pp. 103–112.

A Language Grammar

This section provides the complete grammar of the GPSL and GXSL languages as implemented in the current version of the Goanna static analysis tool.

A.1 GPSL Grammar

This appendix provides the complete grammar of the GPSL language. The grammar is displayed in the extended Backus-Naur-Form.

```
gpsl         = PROP name expression .
name         = ident [comment] .
comment      = string .
expression   = FORALL ident IN l_function expression
             | pattern warning .
pattern      = NEVER l_function
             | EVENTUALLY l_function
             | FOR EVERY l_function HOLDS l_function
             | FOR SOME l_function HOLDS l_function
             | ALWAYS l_function BEFORE l_function
             | ALWAYS l_function STRICTLY BEFORE l_function
             | SOME l_function BEFORE l_function
             | SOME l_function STRICTLY BEFORE l_function
             | AFTER l_function ALWAYS l_function
             | AFTER l_function ALWAYS l_function BEFORE l_function
             | AFTER l_function ALWAYS l_function STRICTLY BEFORE l_function
             | AFTER l_function ALWAYS HAVE l_function BEFORE l_function
             | AFTER l_function ALWAYS HAVE l_function STRICTLY BEFORE l_function
             | AFTER l_function SOME l_function
             | AFTER l_function SOME l_function BEFORE l_function
             | AFTER l_function SOME l_function STRICTLY BEFORE l_function
             | AFTER l_function HAVE SOME l_function BEFORE l_function
             | AFTER l_function HAVE SOME l_function STRICTLY BEFORE l_function
             | BEFORE l_function SOME l_function AFTER l_function
             | BEFORE l_function SOME l_function STRICTLY AFTER l_function
             | CTL str_function
             | CTLR str_function .
l_function   = function
             | l_function AND l_function
             | l_function OR l_function
             | '(' l_function ')' .
function     = ident [ '(' [ ident ] ')' ] .
warning      = WARN LINE line WITH str_function .
str_function = string
             | function
             | str_function '^' str_function .
line         = ident
             | FIRSTST | LASTST
             | ( FIRST | LAST )  '(' function ')' .
ident        = idents identn* .
idents       = "a" | "b" | ... | "Z" | "-" | "_" .
identn       = "a" | "b" | ... | "Z" | "-" | "_" | "0" | ... | "9" .
string       = '"' (identn | " ")* '"' .
```

A.2 GXSL Grammar

This appendix provides the complete grammar of the GXSL language. The grammar is displayed in the extended Backus-Naur-Form.

```
gxsl          = FUN func types id expr .
func          = ident "(" [ ident ] ")" .
types         = [ EXP strlist ] RET strlist .
id            = IDENTIFIER ( attribute_ref | string ) .
expr          = FORALL ident IN func expr
              | NODESET xpath
              | SELECT select .
xpath         = string
              | attribute_ref
              | xpath "^" xpath .
select        = func WHERE ident "=" attribute_ref .
attribute_ref = "@" ident "(" ident ")" .
strlist       = ident | strlist "," strlist .
ident         = idents identn* .
idents        = "a" | "b" | ... | "Z" | "-" | "_" .
identn        = "a" | "b" | ... | "Z" | "-" | "_" | "0" | ... | "9" .
string        = '"' (identn | " ")* '"' .
```