Program Analysis on Evolving Software

Daniel Kästner, Jan Pohland

Program Analysis on Evolving Software

Daniel Kästner, Jan Pohland
AbsInt GmbH,
Science Park 1, 66123 Saarbrücken, Germany

***Abstract*— Static analysis is well-suited for continuous verification during the software development stage since it only works on the source code and does not require a running system for testing. However, applying the program analysis during software development means that the analysis has to cope with evolving software and evolving analyzer configurations, especially in a model-based development process. In this article we present a unique history-aware concept for program analysis that has been developed for the static analyzer Astrée. It not only provides the ability to backtrack and access previous versions of the analysis configuration, it can also automatically determine the differences between two analysis configurations and relate them to the correct source code versions. Users can explicitly create a revision, i.e. a snapshot of the analysis project; changes of the source code, analysis options, analysis directives and results in different revisions are automatically detected and highlighted. The analyzer provides automatic correctness checks for all specified analysis directives, e.g., to tune the precision of the analyzer or provide information about the environment. This makes software verification applicable during the implementation stage, significantly reduces the effort to adapt the analyzer configuration to new source code versions, and makes analysis results on previous software versions easily reproducible.**

I. Introduction

Traditionally software verification is done in a separate stage at the end of the development process. As a rule of thumb the later a bug is discovered, the greater are the costs to fix them. Therefore software verification should already be started during the implementation stage. Especially static analysis is well-suited for continuous application since it only works on the source code and does not require a running system for testing. As an example, run-time error analysis can be done at the level of software components while developing the software. However, applying the program analysis during software development means that the analysis has to cope with evolving software, especially in a model-based development process.

The static analyzer Astrée detects all potential run-time errors in C programs. If no alarms are reported the absence of run-time errors is proven. However, getting down to zero alarms typically requires some interaction with the analyzer: environment information and side conditions have to be specified and the precision of the analyzer has to be tuned to the software under analysis. In Astrée such interactions are expressed via formal directives which are inserted at specific program points. To support model-based development the directives can also be specified externally, without modifying the source code: they are localized by defining a path to the program point where the directive is to be inserted. So not only the source code, but also the program analysis evolves: starting from an initial analysis, the analysis configuration and options are fine-tuned, and missing information and precision adaptations are added step by step.

Therefore similar to software development the ability to backtrack and access previous versions of the analysis configuration are important development features. Being able to investigate the differences between two analysis configurations and relate them to the correct source code versions is very desirable. Furthermore users should be warned about analysis directives which have become invalid because of source code changes.

All these issues have been addressed by developing a unique history-aware concept for program analysis in Astrée. Users can explicitly create a revision, i.e. a snapshot of the analysis project; changes of the source code, analysis options, analysis directives and results in different revisions are automatically detected and highlighted. The analyzer provides automatic correctness checks for all externally specified analysis directives. This makes software verification applicable during the implementation stage, significantly reduces the effort to adapt the analyzer configuration to new source code versions, and makes analysis results on previous software versions easily reproducible.

II. The Static Analyzer Astrée

The static analyzer Astrée aims at finding all potential run-time in C programs [7]. Basically run-time errors are errors that occur during run-time of the software. Astrée focuses on run-time errors which correspond to undefined or unspecified behavior with respect to the semantics of the programming language C99 [6]. Examples are arithmetic exceptions (e.g. divide by zero), overflows, invalid pointer accesses and manipulations, or array bound errors [10]. The C standard provides a list of unspecified and undefined behaviors in Section J of [6].

Astrée is *sound* which means that the analysis never omits to signal an error that can appear in some execution environment. If no potential error is signaled, definitely no run-time error can occur: the absence of run-time errors has been proven. This is possible since Astrée is based on the theory of abstract interpretation [3], a mathematically rigorous formalism providing a semantics-based methodology for static program analysis. It provides rules for defining an abstract semantics that approximates the concrete semantics of the program and is efficiently computable. The correctness of the abstraction, i.e., the soundness of the analyzer can be formally proven; all necessary correctness proofs for Astrée have been done and published, e.g., [9], [8]. Abstract Interpretation, like model checking and theorem proving, is recognized as a formal verification method and recommended by the DO-178C and other safety standards (cf. Formal Methods Supplement [12] to DO-178C [13]).

While the primary goal of Astrée is to prove the absence of run-time errors, the analysis can also be leveraged to compute further program properties relevant for functional safety. Astrée detects read accesses to uninitialized variables, detects shared

variables accessed by asynchronous threads, computes detailed control and data flow reports, and enables users to prove user-defined static assertions. The static assertions can be applied to arbitrary C expressions so that functional program properties can be addressed. When Astrée does not report an assertion failure alarm, the correctness of the asserted expression has been formally proven. Astrée is sound for floating-point computations and handles them precisely and safely [2]. It takes all possible rounding errors into account which makes it possible to prove the stability of numeric algorithms (e.g., digital filters) with floating-point arithmetic [5].

Astrée finds all potential run-time errors, but it may err on the other side and produce false alarms. The design of the analyzer aims at reaching the zero false alarm objective [2], which was accomplished for the first time on large industrial applications at the end of November 2003[4]. Only with zero alarms the absence of run-time errors is automatically proven. For keeping the initial number of false alarms low, a high analysis precision is mandatory, which is achieved by computing a variety of predefined abstract domains. Any remaining alarm has to be manually checked by the developers – and this manual effort should be as low as possible. Astrée provides intuitive and powerful mechanisms to to investigate the reasons for alarms. As an example, alarm contexts can be interactively explored: all parents in the call stack, relevant loop iterations or conditional statements can be visited per mouse click, and the computed value ranges of variables can be displayed for all abstract domains in all potential context.

If there is a true error it has to be fixed. A false alarm can often be eliminated by a suitable parameterization of Astrée. If the error cannot occur due to certain preconditions which are not known to Astrée, they can be made available to Astrée via dedicated directives. These directives make the side conditions explicit which have to be satisfied for a correct program execution. If the false alarm is caused by insufficient analysis precision, steering directives are available that allow to locally tune the analysis precision to eliminate the false alarm [8]. That means that in one analysis run important program parts can be analyzed very precisely while less relevant parts can be analyzed very quickly – without compromising system safety.

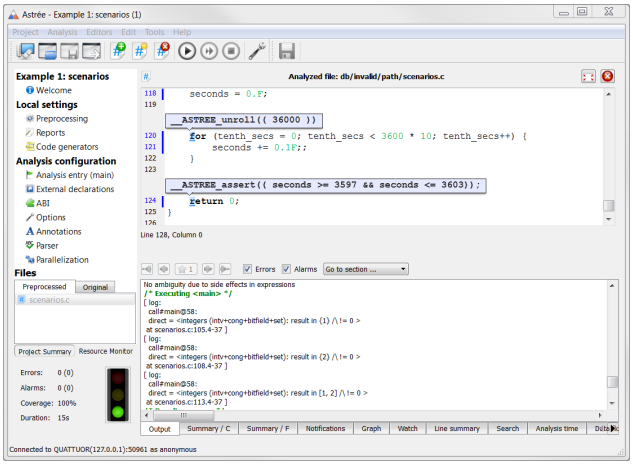

Figure 1: AAL annotations displayed in code view

All directives can either be written directly in the source code, or they can be specified in a formal language AAL [1] and stored in a dedicated file without any source code changes. An AAL annotation consists of an Astrée directive and a path specifying the program point to insert the directive at. The path is specified in a robust way by exploiting the program's syntactical structure without relying on line number information. An example of the syntax of an AAL annotation is shown here:

```
main { + 1 if {then: + 1 statement }}
insert before: __ASTREE_assert(( x >= 0));
```

All AAL annotations are displayed in Astrée's source code editors as overlays at the specified code locations, as shown in the Astrée screen shot of Fig 1.

The AAL language is a prerequisite for supporting model-based code generators. It makes it possible to separate the annotations from the source code, so that when the code is regenerated, all previously generated annotations from structurally unchanged code parts are still valid, even if the line numbers change.

III. Exploiting the Analysis History

In today's development infrastructures it is standard practice to manage source code with a revision control system. While it is theoretically possible to store report files of an analyzer run with each revision this is seldom done, and in order to reproduce analysis results for an older revision, the analysis typically has to be set up from scratch.

We have extended Astrée by an integrated revision system which works orthogonally to the standard revision control systems already in place. Its main purpose is to automatically compute and display differences between the setup and results of multiple analyzer runs on the same or different revisions of the same project. This approach makes it possible to reproduce analysis results on old revisions by mouse-click, to easily compare the analysis results of different source code versions, to compare the analyzer configuration of different revisions, etc.

This revision system is complemented by a mechanism to automatically check the correctness of Astrée directives specified in the AAL language. With AAL the locations can be specified in a robust way without relying on line numbers but if the code structure changes too much, they still may become invalid. The key idea to address this issue is to manage a reference file which keeps track of relevant source code properties for each AAL annotation. When a pre-configured analysis is re-run on an updated version of the source code, users will be notified about potentially invalid analysis directives. In the following sections both mechanism will be explained in more detail.

A. Analysis Revisions

Astrée has a client-server architecture. The client is used to set up the analysis and investigate the analysis results. The analysis itself is performed by the Astrée server which can run either on the same computer, or on a dedicated server machine. The Astrée server manages analysis projects in its data directory. Each analysis project has a unique analysis ID

and all required information is stored under a folder named by this analysis ID. This information includes e.g., the original and preprocessed source files, the option setting, the log file of the analyzer, and a file containing all specified AAL annotations.

The creation of a new revision can be seen as a snapshot of the current state: the current state is saved, and copies are created for new revisions. The same base revision can be cloned to several child revisions, e.g., to apply different analyzer configurations and later investigate the differences in the corresponding analysis results. To create a new analysis revision for a new source code revision first a new analysis revision can be cloned from the current analysis state, and in the new analysis revision the source code files are updated from the normal source code repository. After running the analysis on the new files in the new revision the analysis results can be compared.

The `Differences viewer` enables users to efficiently compare several analysis revisions. It shows the total number of alarms and errors, the number of alarms per category, the option setting of the different analysis revisions, differences in the Astrée annotations specified by the user, the number of total statements and unreachable statements per file and the percentage of unreachable code per file. The `Differences Viewer` can also be applied to the XML report file of an analysis, so that also reports of unrevisioned analysis runs can be compared.

B. Automatically Validating User Annotations

AAL [1] is a formal language that makes it possible to place an Astrée directive in the code without actually modifying the source code. This is a prerequisite for model-based development where the code is automatically regenerated after each model change. It is also important when the code has been frozen and can only be modified after a formal change process, or when the person doing the analysis is not entitled to modify the code. For robustness reasons, AAL is not based on line number information but on the structure of the program's abstract syntax tree. AAL annotations are stored in a dedicated file and the specified directives are inserted into the internal syntax tree of a function body generated in the parsing stage of Astrée.

Still, in the life cycle of a software project the source code will change and such changes may cause previously written Astrée annotations to become invalid. The specified position may not exist any more, the directive might end up in an unwanted location, or the directive may become semantically invalid. Our goal is to warn the user about such invalid annotations.

AAL annotations consist of three parts [11]: a path description, an ordering, and the Astrée directive to be located. For inserting directives into a function body, the syntactic elements that occur before the insertion place – the *significant* statements – must be specified which can simply be done by counting the corresponding statement types. For robustness and usability, the most specific statement types should be preferred, e.g. labeled statements, loops and if statements. This is the purpose of the path description which is composed of a set of distance expressions. In the following AAL annotation:

```
main { +3 loops +1 if {then: +2 statements}}
     insert after: __ASTREE_assert((x > 0));
```

`+1 if` is a distance expression, `main { +3 loops +1 if {then: +2 statements}}` is the path description, `after` the ordering. It inserts an assertion after the second statement of the *then* branch of the first *if* statements after the third loop in the function.

When inserting a directive into the syntax tree the significant statements of the path in the syntax tree are stored in a dedicated history file, a reference file. In our example, the significant statements are three loops, one *if* statement and 2 unspecific statements. This information can be used to detect changes in the source code, that might lead to the directive being placed at a wrong position when restarting the analysis on the new source code version.

1) Invalid Path Description: In the easiest case of an invalid annotation the target point of the annotation's path description does not exist in the Astrée internal syntax tree. Then, the path description of the annotation does not match to the internal syntax tree and it is invalid. In this case, the annotation inserter reports an error during the attempt to insert the annotation at the specified program point.

2) Detecting Structurally Invalid Annotations: If the annotation inserter is able to insert an annotation inside a modified source code version, it cannot be guaranteed that the annotation's path description addresses the correct target point it was written for in the old version. If the annotation is inserted to a wrong target point, the annotation is *structurally invalid*.

It is possible to detect this kind of errors with the help of the reference file created in the last analyzer run before the source code modification. The history information from the reference file and a new temporary reference file are compared. Users can control for which analyzer runs the reference file should be created or updated.

We have developed two algorithms to exploit the history information to detect structurally invalid annotations, which will shortly be summarized in the following. They can be used to classify the annotations as *possibly* correct or *possibly* incorrect. In general, the correctness of annotations in a modified source code version cannot be guaranteed, since this would be equivalent to deciding whether two different programs have the same semantics, which is an undecidable problem [14].

- *Annotation Path Comparison*: In this approach the significant statements of the annotation path of each annotation are compared in the old and the new reference file. To be independent from formatting and line or column numbers, the comparison is done on the elements of the internal syntax tree of Astrée. Differences in statements not contained in the annotation path are irrelevant for the structural validity of an annotation.
- *Statement Counting*: Here the numbers of the available statements of the type counted by the distance-expressions is counted. This way even differences caused by duplicated identical statements are detected.

IV. Experimental Results

In this section we summarize our experiments to assess the usability of the history evaluation approaches. The experiments were performed on an Intel Core2Duo with 8GB RAM under openSUSE Linux 64-bit.

To evaluate the work flow using analysis revisions we investigated three real-life industry projects:

- **Project A** Avionics application consisting of 530 preprocessed source files using 24 MB disk space.
- **Project B** Automotive application consisting of 168 preprocessed source files using 37 MB disk space.
- **Project C** Automotive application consisting of more than 1200 preprocessed source files with a size of 622 MB and 16.8 million lines of code with up to 99,000 lines per file (generated from more than 3,000 original source and header files).

Tab. 2 shows the revision size, the time to create a new revision, and the time to delete a revision. To account for variations of the file system I/O load the reported times are are the maximum times observed in five analyzer runs.

Project	Revision Size	Creation	Deletion
A	400 MB	7.1s	54ms
B	210 MB	3.6s	52ms
C	2.3 GB	67.7s	49ms

Figure 2: Evaluation of Analysis Revisioning Mechanism.

As we can see, creating new revisions takes a couple of seconds while project deletion is in the milliseconds range, even for large industry applications.

The execution time of the Difference Viewer is dominated by the time needed to compare the XML report files of the analysis revisions to be compared. The maximal comparison time observed in our experiments was about two minutes.

Regarding the performance of the algorithms for checking the validity of AAL annotations in all experiments we conducted the time needed to write a reference file, run the Annotation Path Comparison algorithm, and run the Statement Counting algorithm are below one second.

V. Conclusion

Applying program analysis during software development means that the analysis has to cope with evolving software and evolving analyzer configurations, especially in a model-based development process. To facilitate this we have developed a unique history-aware concept for the static analyzer Astrée which consists of two parts: The first component is an integrated revision system which works orthogonally to the standard revision control systems already in place. It not only provides the ability to backtrack and access previous versions of the analysis configuration, it can also automatically determine the differences between two analysis configurations and relate them to the correct source code versions. Users can explicitly create a revision, i.e. a snapshot of the analysis project; changes of the source code, analysis options, analysis directives and results in different revisions are automatically detected and highlighted. This revision system is complemented by a mechanism to automatically check the correctness of Astrée directives specified in the AAL language. A reference file is maintained which keeps track of relevant source code properties for each AAL annotation. When a pre-configured analysis is re-run on an updated version of the source code, users will be notified about potentially invalid analysis directives. Experimental results show that both approaches are applicable on large-scale industrial projects. This makes software verification applicable during the implementation stage, significantly reduces the effort to adapt the analyzer configuration to new source code versions, and makes analysis results on previous software versions easily reproducible.

Acknowledgements

The work presented in this paper has been supported by the German BMBF project FORTE.

References

[1] AbsInt. *The Static Analyzer Astrée– User Documentation for AAL Annotations*, 2013.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[4] D. Delmas and J. Souyris. ASTRÉE: from Research to Industry. In *Proc. 14th International Static Analysis Symposium (SAS2007)*, number 4634 in LNCS, pages 437–451, 2007.

[5] M. Dierkes and D. Kästner. Transferring Stability Proof Obligations from Model Level to Code Level. *Embedded Real Time Software and Systems Congress ERTS* [2], 2012.

[6] ISO/IEC International Standard. ISO/IEC 9899:1999 (E) Programming Languages – C. Second edition 1999-12-01, 1999.

[7] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the Absence of Runtime Errors. *Embedded Real Time Software and Systems Congress ERTS* [2], 2010.

[8] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *14th European Symposium on Programming ESOP'05*, number 3444 in LNCS, pages 5–20, 2005.

[9] A. Miné. Relational Abstract Domains for the Detection of Floating-Point Run-Time Errors. In *Proc. of the European Symposium on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, Barcelona, Spain 2004.

[10] Motor Industry Software Reliability Association (MISRA). MISRA-C:2004 - Guidelines for the use of the C language in critical systems, 2014.

[11] J. Pohland. Program Analysis on Evolving Software. Master's thesis, Saarland University, August 2014.

[12] Radio Technical Commission for Aeronautics. Formal Methods Supplement to DO-178C and DO-278A, 2011.

[13] Radio Technical Commission for Aeronautics. RTCA DO-178C. Software Considerations in Airborne Systems and Equipment Certification, 2011.

[14] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.