Enabling Certification through an Integrated Comprehension Approach

Raoul Jetley and S. Purushothaman Iyer
North Carolina State University

Safety is a primary concern for medical device software and systems (MDSS). Correspondingly, it becomes necessary to have a process of certification in place to ascertain the correctness of the software and to ensure that it meets all requirements and standards. Current certification methods for MDSS are process-oriented and based on guidelines for development and testing. Such an approach, though not entirely without its benefits, depends mainly on the results of test case executions for validation, and is unable to trace errors in the software to the source code, or the underlying design. In order to perform a more stringent and complete conformance check, one needs to have the means to track the source code to product specifications, and check the software for conformance to requirements.

To do this, the analyst performing the certification needs to have a good understanding of the software and of the underlying architecture. This understanding, often termed **software comprehension**, is obtained from the source code, test case results, and detailed documentation. However, given the scale and diversity of modern MDSS, it is virtually impossible for the analyst to manually navigate through the entire source code. The use of third party software and commercial off the shelf (COTS) components further complicates this task, as they typically offer the analyst little documentation and minimal support. As a result, the analyst ends up spending a significant amount of time and effort simply trying to understand the software. This effort can be greatly reduced though, if the analyst could have an efficient means to automatically abstract the high level descriptions from the software, and not have to rely on source code in order to do so.

Using slicing for comprehension. Program slicing has been effectively used as a tool to enable program comprehension [FR99]. Introduced by Weiser [Wei81] in 1981 as a method to facilitate understanding and debugging of programs by focusing on selected aspects of semantics, slicing uses a user-defined criterion to extract statements from a program. Any statement that does not satisfy the given slicing criterion is removed (or sliced away) from the original program. The resultant code fragment constitutes the slice for the program for the given criterion.

The analyst uses slicing as an abstraction technique by defining a slicing criterion based on his understanding of the system and application domain to obtain a slice. The criterion is subsequently refined in order to get finer, more concise slices that allow the analyst to trace requirements to actual (source) code.

In spite of being an effective tool, slicing does have its limitations. When used on large, strongly cohesive systems the resultant slices are often too big to be of any practical use. Using dynamic slicing, as opposed to the traditional static slicing, is not much help either, since it offers a restricted view of the control flow and is thus not very helpful for purposes of comprehension. Analysts often use iterative slicing to tackle this problem, relying on guesswork and estimates to define the initial slicing criteria; discarding slices that are based on incorrect hypotheses (are not very helpful for comprehension) and refining the correct ones iteratively. This is not a very efficient method for software understanding however, and still requires significant amount of effort for the analyst.

Moreover, slicing offers a top-down comprehension of the software only and is not well suited for the bottom-up or integrated models of comprehension [vMV95]. Such a top-down model necessitates that the analyst have a conceptual knowledge of the application domain, which may not always be possible. Lastly, slicing is best used to detect errors of commission in the software. To detect the more subtle errors of omission, a stronger, more formal framework needs to be used.

Integrating slicing with abstraction. As a solution to these problems, we suggest using model abstraction in conjunction with slicing. The concept of model abstraction derives from model checking - a popular static analysis technique. Model abstraction is based on the idea of viewing the analysis of a program as an abstraction of the program's behavior, and can be defined as the process of formally extracting the semantics of a program from the source code. The process of extraction uses a set of predefined abstract interpretations (AIs) to convert a given program to an abstract model based on the criteria defined [CC77].

Abstraction, by virtue of producing an abstract model similar to the program model, works in a manner that facilitates bottom-up comprehension. It thus forms a perfect foil for slicing by (automatically) providing program models to the analyst at specified levels of abstraction, that can be used to form the hypotheses upon which the slicing criteria are based upon. Further, the abstracted model can be used as a basis for model checking, to formally verify properties of the system and ascertain the validity of the program, or generate an error trace as a counter-example. This model checking provides the framework for detecting design faults and errors of omission that slicing cannot easily detect. A methodology for combining abstraction with slicing is proposed in [JI04].

Future Directions. To realize this integration, the analyst would need to be provided with an automated tool that combines slicing with model abstraction. The tool would ideally take as input a user-defined criterion and generate a corresponding program slice and abstract model for the same. The criterion for slicing and abstraction could be defined independently, or one could be derived from the other (Bandera [CDH$^+$00] and the Promela slicer [MT98] use temporal properties to derive the slicing criterion; a similar method could be developed for the other direction as well). The integration with abstraction could be extended to other slice refining techniques like conditioned and amorphous slicing as well. Using these techniques would result in smaller slices and correspondingly, more refined models; thus further improving the efficiency of the comprehension process.

Since a major advantage of abstraction is the ability to facilitate model checking, it would be desirable for the tool to produce a model that can be easily and efficiently verified using a model checker. Compliance with existing tools like Spin and SMV would be helpful in this respect. Another advantage presented by this feature would be the ability to compare abstract models generated for software provided by different vendors, or for different versions of the same software. This would allow the analyst to compare feature implementations for vendor software, and to track changes across different versions of the code. A notion of semantic equivalence could also be included to compare the different models generated.

Conclusion. Software comprehension is imperative for exhaustive software conformance checking and the certification process. An effective integrated comprehension model can be provided by combining the program slicing and model abstraction techniques. Automating this integration process and combining it with model checking and slice refining methods would provide the certification engineer with a powerful tool to ensure conformance of large-scale MDSS.

References

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CDH$^+$00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[FR99] Margaret Ann Francel and Spencer Rugaber. The relationship of slicing and debugging to program understanding. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 106, Washington, DC, USA, 1999. IEEE Computer Society.

[JI04] Raoul Jetley and S. Purushothaman Iyer. Post mortem analysis using abstraction and slicing. In *Proceedings of the workshop on Software Engineering Tools: Compatibility and Integration, Vienna, Austria. Publication pending subject to review*, 2004.

[MT98] L. Millett and T. Teitelbaum. Slicing promela and its applications to model checking, 1998.

[vMV95] Anneliese von Mayrhauser and A. Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society Press, 1981.

About the authors.

Raoul Jetley. Raoul is a doctoral student at the North Carolina State University. He received his Masters degree in Computer Science from NC State in December 2002, and a Bachelor's degree in Computer Science from the University of Mumbai in May, 1998. Raoul has experience working as an analyst for firms such as Unisys and Infosys Technologies Ltd.
e-mail: rpjetley@csc.ncsu.edu.
Work phone: (919)513-7309.

Dr. S. Purushothaman Iyer. Purush is a professor in the Computer Science department at North Carolina State University. He received his PhD from the University of Utah in 1986. Purush's research interests include programming and specification languages, software model-checking, probabilistic models of concurrency and probabilistic model-checking.
e-mail: purush@csc.ncsu.edu.
Work phone: (919)515-7291.