

A Representation for Describing and Analyzing Concerns in Source Code

Martin P. Robillard

Department of Computer Science
University of British Columbia
2366 Main Mall, Vancouver, BC
Canada V6T 1Z4
mrobilla@cs.ubc.ca

Many maintenance tasks address concerns, or features, that are not well modularized in the source code comprising a system. Sometimes, such non-localized concern code is the result of inadequate design. More often, it is the result of either unanticipated modifications or of a lack of expressibility in the technology available to the original designer to express interacting or overlapping concerns. The end result is that software developers must handle concern code scattered across a system's source when modifying the concern or implementing a new feature that interacts with the concern.

Existing approaches available to help software developers locate and manage the scattered implementation of concerns use a representation based on lines of source code. Because they do not explicitly express program structure, concern representations based on source code have inherent limitations when finding, describing, and analyzing concerns. First, since programmers tend to naturally navigate along program structures when looking for related code [2], working at the source code level requires mentally performing the first few steps of compilation to distill the program structures. Second, a representation of a concern based on source code does not explicitly describe the relationships between the various parts of a concern. Again, a developer must mentally process the code to derive these relationships. Finally, more sophisticated analyses, such as impact analysis techniques based on slicing [4], typically results in a large amount of fine-grained information about a program. Since, in most cases, it is only realistic to use a subset of this information when planning a change, the additional effort required to validate, interpret, and manage the results of sophisticated analyses might not always be warranted. To address these problems, we are trying to find a concern representation that captures as close as possible the amount and precision level of information developers need to efficiently plan a change, and that supports queries that can provide this information. As a proposed solution, we introduced the Concern Graph representation that abstracts the implementation details of a concern and makes explicit the relation-

ships between different parts of the concern. The Concern Graph abstraction has also been designed to allow an obvious and expensive mapping back to the corresponding source code. To investigate the practical tradeoffs related to this approach, we have built the Feature Exploration and Analysis tool (FEAT) that allows a developer to navigate over an extracted model of a Java program, to build up the subset of the model that corresponds to a concern of interest, and to analyze the relationships of that concern to the code base.

Because more details on the concept of Concern Graphs and the FEAT tool appear elsewhere in this volume [3], this research abstract focuses on future work and evaluation. Important directions for future work are:

- Tolerating inconsistency in the specification of concerns;
- Supporting the automatic inclusion of program elements into a Concern Graph;
- Supporting high-level queries on Concern Graphs;

The rest of this abstract addresses each of these issues in turn, and concludes with a discussion of the evaluation of the research.

Tolerating inconsistency

Because they describe fragments of an *existing* code base, Concern Graphs must rely on the names of global program entities as anchors between the concern representation and the corresponding program. This coupling requires Concern Graphs to be kept consistent with the code base. In practice, we observed that occasionally a developer will try to use a Concern Graph describing a concern that is slightly inconsistent with the corresponding program. An example of such an inconsistency could be a reference to a field that is no longer used. These inconsistencies arise naturally given the iterative nature of program maintenance, when developers alternate modifications to the program, testing, and querying the source code. Our goal is to support this process by extending the Concern Graph model to be able to support elements that are inconsistent with the rest of the code [1]. This way, it will be possible for developers to view and use inconsistent Concern Graphs, either to repair the inconsistencies, or to directly use the inconsistent elements in the query process, possibly to track down changes based on a description of an obsolete concern.

Automatic inclusion of concern elements

During an initial evaluation of the research, while observing developers using FEAT to find the concern code (see [3]), we noticed that there were situations in which the process of building concern descriptions could be facilitated through automatic inclusions of elements. For example, when a method was found to be entirely involved in implementing a concern, developers usually found useful to include all the call sites for this method in the Concern Graph. Similarly, when a field was identified as being used in a concern's implementation, sometimes all the methods accessing this field needed to be integrated into the concern description. We are currently working on a method for supporting the automatic identification and inclusion of program elements into a concern description. The work involves both determining a simple and effective way for users to specify inclusion rules, and a study of common idioms of element inclusion used during the creation of concern descriptions.

We believe that this feature will make it easier to effectively build concern descriptions, and that these descriptions will be more complete.

Supporting high-level queries

To help developers get the maximum benefit out of the effort they invest in building a Concern Graph, we plan to investigate ways to make Concern Graphs useful for more than one change task. A primary requirement for this goal is for Concern Graphs to be able to tolerate change in the underlying source code, as described above. Another important requirement is to be able to provide valuable information to a developer planning a change. We hypothesize that one way to provide this value-added information is through higher-level structural queries into the nature of the interactions between different Concern Graphs. This hypothesis is based on both our experience with FEAT and Concern Graphs, and in the results of a separate investigation, where it was observed that "crosscutting concerns tended to emerge as obstacles that the developer had to consider to make the desired change" [2].

Work in this area will involve devising support for concern overlap analyses through algorithms and heuristics comparing the use of program elements in different concerns. It is not clear at this stage whether this will be possible to accomplish without enhancing the Concern Graph model with additional information, such as constraint rules or ordering information.

Evaluation

Evaluating this research will involve determining whether Concern Graphs can help developers make more informed decisions when planning a change, and whether the effort involved in creating and querying Concern Graphs is a reasonable cost given the benefits. Because of the qualitative nature of these research questions, and the lack of control over behavioral events, the research method chosen for the evaluation of Concern Graphs is the case study.

Short-term case studies will be used to evaluate work in progress and to steer the research. A series of such case studies has already been performed to explore various aspects of Concern Graphs. These case studies are described in [3].

The short-term case studies should provide the insights and experience necessary to design a longitudinal case study of program maintenance using Concern Graphs. The study will involve introducing a set of participants to the Concern Graphs technology, and monitoring their use of the technology while they perform one or

more change tasks. The exact type of data we will collect will depend on the environment in which the case studies are performed. However, to help meet the research quality criterion of *construct validity* [5], we plan to collect both qualitative and quantitative data. The quantitative data will consist of values such as the amount of code changed, the concern graph descriptions used, the number and types of queries performed, etc. Depending on availability, the qualitative data will consist of usage patterns obtained through logs, interviews with the participants, and comments produced by the participants.

The contributions following from this work will be a description and working implementation of Concern Graphs, a novel way of describing scattered concerns for the purpose of program evolution. Specifically, Concern Graphs will give developers a conceptual tool and the supporting technology for:

- cost-effectively finding code implementing scattered concerns,
- documenting these concerns over time in a compact form that tolerates some degree of inconsistency, and
- analyzing the overlap between two concerns.

The contribution of the work will also include the analysis of empirical data evaluating the research.

Tool availability

The FEAT tool can be obtained for free evaluation by contacting the author.

ACKNOWLEDGMENTS

I would like to thank Gail Murphy for the guidance, help, and encouragement which have made this work possible.

REFERENCES

- [1] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, May 1991.
- [2] E. L. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. *Proceedings of the 1st Conference on Aspect-Oriented Software Development*, April 2002.
- [3] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, May 2002.
- [4] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [5] R. K. Yin. *Case Study Research: Design and Methods*, volume 5 of *Applied Social Research Methods Series*. Sage Publications Ltd., London, second edition, 1989.