Context-sensitive analysis of data interference for concurrent programs

Carlos Galindo, Marisa Llorens, Sergio Pérez and Josep Silva*

VRAIN, Universitat Politècnica de València, Camí de Vera s/n, E-46022 València, Spain

Abstract

Context-sensitive analyses enable the precise analysis of programs with procedures, such that only the relevant procedures will be selected, because the calling context is being taken into account. In threaded programs, interference dependence models the effects of concurrent assignments and access to shared variables, but current definitions of this dependence prevent context-sensitivity. Our work redefines interference dependence in order to make it compatible with known context-sensitive analyses, enabling more precise analyses.

Keywords

Interference dependence, dependence graphs, context-sensitivity, concurrent program analysis

1. Introduction and background

In the software engineering field of static analysis, *context-sensitivity* is a property that determines whether an analysis is able to distinguish the different calls made to a function, and only analyse the relevant ones. A context-sensitive analysis can produce faster and/or better results, by being able to ignore calls that are irrelevant to the question asked.

A common representation used to tackle software analyses are dependence graphs, which represent instructions in a program as nodes and the relationships or dependences between them as directed arcs. For example, *interference dependence* (denoted with $\dashrightarrow$) is a relationship that connects a variable definition and a variable usage in different parallel threads. It is useful in the analysis of shared-memory concurrent programs. It was originally defined for program slicing by Krinke [1], based on the concept of a *witness* (a possible execution of a program). This kind of dependence is not always transitive ($a \dashrightarrow b \land b \dashrightarrow c$ does not necessarily mean that $a \dashrightarrow c$), a valid witness must back up every sequence of two or more interference dependences ($(a \dashrightarrow b \land b \dashrightarrow c) \rightarrow (a \dashrightarrow c)$ if and only if there is a witness that includes an execution path $a, ..., c$).

Context-sensitive analyses can be performed in multiple ways. A simple but expensive method is to label each call and store the sequence of call labels that have already been analysed,

PNSE'23: International Workshop on Petri Nets and Software Engineering, June 26–27, 2023, Lisbon, Portugal

*Corresponding author.

cargaji@vrain.upv.es (C. Galindo); mllorens@dsic.upv.es (M. Llorens); serperu@dsic.upv.es (S. Pérez); jsilva@dsic.upv.es (J. Silva)

0000-0002-3569-6218 (C. Galindo); 0000-0002-2790-0055 (M. Llorens); 0000-0002-4384-7004 (S. Pérez); 0000-0001-5096-0008 (J. Silva)

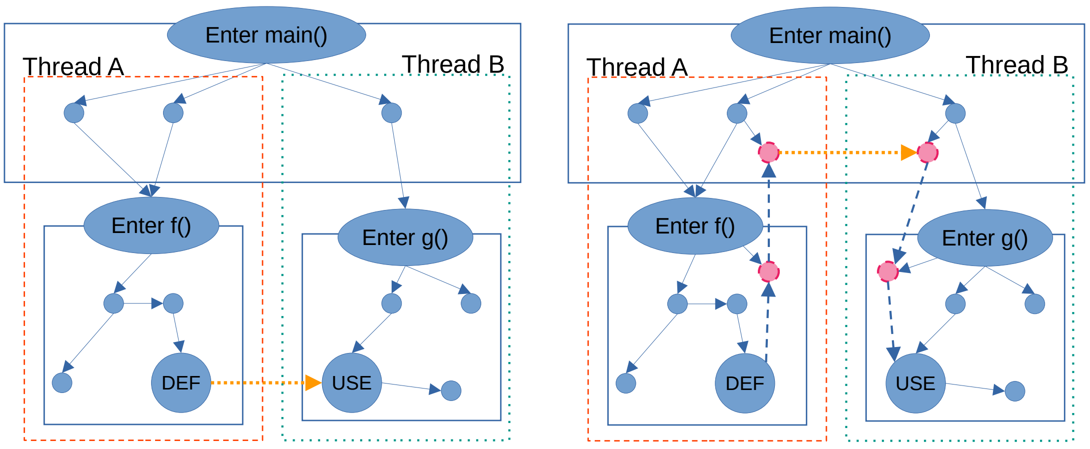

Figure 1: Two representations of interference dependences for a given program. Nanda and Ramesh's (left) features an interprocedural interference dependence, while our proposal (right) moves the dependence to the procedure calls, moving it outside the procedure and making it intraprocedural.

but this leads to an explosion of states which makes the analysis too expensive. An alternative is the use of *summary edges*, first proposed by Horwitz et al. [2]. They model the dependences inside each function's body at its calls, allowing the analysis to know the effect of the function without the need to analyse the function's body. The use of summaries restricts the dependences that need to be considered, preventing the analysis of unrelated functions (as the calls themselves are analysed through summaries).

In interference dependence, when a source of dependence is reached, all its dependences are transitively required (except for other interference dependences, which are analysed on a case-by-case basis), removing any restriction imposed by other aspects of the analysis, including context-sensitivity. Both requirements are conflictive, and thus the analysis can either consider interference or be context-sensitive.

Some authors, like Krinke [3], used the expensive call-labelling strategy, but was forced to make it less precise (limiting the length of the sequence of labels to 3-5 elements) in order to keep the time bound under control, and others, like Nanda and Ramesh [4], tried to merge summaries and interference dependence, but failed to account for interprocedural interference dependences that broke the calling context.

2. Context-sensitive interference dependence

Our approach is a redefinition of interference dependence, such that every instance of it appears between threads of the same procedure (removing all interprocedural jumps that break the calling context). However, interprocedural interference dependences must also be represented, and we designed a structure of dependences similar to the one generated by summaries. This allows us to use a system similar to Horwitz et al.'s, and produce context-sensitive interference dependences.

As an example, let's observe the structures in the left side of Figure 1, in which a `main` procedure spawns two threads (A and B). Thread A calls procedure `f`, while thread B calls procedure g. Concurrently, a variable is defined in f (thread A) and used in g (thread B). Thus, an interference dependence connects the definition to the usage (marked as DEF and USE in the graphs, respectively).

Nanda and Ramesh's approach place the interference dependence directly between the definition and the usage, making it an interprocedural dependence (shown as a dotted arc, it crosses the boundaries of a procedure, from f to g). As a consequence, since all the dependences of the definition are transitively required by the analysis, it is impossible to determine which calls to f are relevant and which are not, resulting in the analysis not being context-sensitive.

On the right side of Figure 1 we see our approach, which creates auxiliary nodes (pink and dashed) and arcs (vertical and dashed) to move the interference dependence from the real definition and usage to each of the calls to the procedures that perform these actions. The horizontal dotted arc represents the interference dependence, which is now intraprocedural. As a side-effect of this transformation, there is no doubt about which of the two calls to f in thread A produces the interference dependence, and we can take advantage of Horwitz et al.'s context-sensitivity framework.

3. Conclusions

In summary, our approach redefines interference dependence to be exclusively intraprocedural. It also generates new nodes that can trace the sequence of calls that link a concurrent definition-usage of a variable. With these two elements, our approach results in a context-sensitive analysis of concurrent programs, applying the framework of summaries laid out by Horwitz et al.

Acknowledgments

This work has been partially supported by the EU (FEDER) and the Spanish MCI/AEI under grant PID2019-104735RB-C41, and by the European Union's Horizon 2020 research and innovation programme under grant agreement No 952215 (Tailor).

References

[1] J. Krinke, Static slicing of threaded programs, SIGPLAN Not. 33 (1998) 35–42. URL: http://doi.acm.org/10.1145/277633.277638. doi:10.1145/277633.277638.

[2] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, in: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI '88, ACM, New York, NY, USA, 1988, pp. 35–46. URL: http://doi.acm.org/10.1145/53990.53994. doi:10.1145/53990.53994.

[3] J. Krinke, Context-sensitive slicing of concurrent programs, SIGSOFT Softw. Eng. Notes 28 (2003) 178–187. URL: https://doi.org/10.1145/949952.940096. doi:10.1145/949952.940096.

[4] M. G. Nanda, S. Ramesh, Interprocedural slicing of multithreaded programs with applications to java, ACM Trans. Program. Lang. Syst. 28 (2006) 1088–1144. URL: https://doi.org/10.1145/1186632.1186636. doi:10.1145/1186632.1186636.