Contract-based Slicing helps on safety Reuse

Sérgio Areias, Daniela da Cruz, Jorge Sousa Pinto
Departamento de Informática
Universidade do Minho
Braga, Portugal
{*danieladacruz,jsp*}*@di.uminho.pt*

***Abstract*—In this poster we describe a work in progress aimed at using a variant of specification-based slicing to improve the reuse of annotated software components, developed under the so called *design-by-contract* approach. We have named this variant as *contract-based* because we use the annotations, more precisely the pre and post-conditions, to slice programs intra and inter-procedures. The idea, expressed in the poster, is to take the pre-condition of the reused annotated component as slicing criterion, and slice backward the program where the component is called. In that way, we can isolate the statements that have influence on the variables involved on the pre-condition and check if it is preserved by that invocation, or not.**

***Keywords*-Component reuse; safety reuse; design-by-contract; slicing.**

I. Introduction

Reuse is a very simple an natural concept, however its practice is not so easy. According to the literature, selection of reusable components has proven to be a difficult task [1]. Sometimes this is due to the lack of maturity on supporting tools that should easily find a component on a repository or library [2]. Also, non experienced developers tend to reveal difficulties when describing the desired component in technical terms. Most of the times, this happens because they are not sure of what they want to find [2]. Another barrier is concerned with reasoning about component similarities in order to select the one that best fits in the problem solution; usually this is an hard mental process [1].

Integration of reusable components has also proven to be a not easy task, since the process of understand and adapt components is difficult, even for experienced developers[1]. Other challenge to component reuse is to certify that the integration of such component in a legacy system is correct. This is, to verify that the way the component is invoked will not lead to an incorrect behavior.

A strong demand for formal methods that help programmers to develop correct programs has been present in software engineering for some time now. The Design by Contract (DbC) approach to software development facilitates modular verification and certified code reuse. The contract for a component (a procedure) can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. So, a well defined annotation can give us most of the information needed to integrate a reusable component in a new system, as it contains crucial information about some constraints safely obtaining the correct behavior from the component.

In this context, if we use the annotations to verify the validity of every component's invocation, then we can guarantee that a correct system will be still correct after the integration of that component. This is the motivation for our research: find a way to help on the safety reuse of components. For such purpose we are currently developing a tool to identify when an invocation is violating the component's annotation, and display, whenever possible, a diagnostic or guidelines to correct it.

II. Safety reuse of software components

As referred in the previous section, our goal is to make easier the process of incorporate an annotated component into an existent system. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To assure this we need to verify a set of conditions with respect to the annotated component and its usage: verify its correctness with the respect to its contract; given a concrete call to the reusable component, verify if the concrete calling context preserves the precondition; given a concrete call and the postcondition of the component, verify if it is properly used in the concrete context after the call; given a reusable component and a set of calling points, specify the component body according to the concrete calling needs.

To verify the correctness of the component with respect to its contract, we have available multiple tools to do that validation (Esc/Java[1], Krakatoa, etc). These tools usually have as basis a Verification Condition Generator (VCGen), to generate from the contract a set of proof obligations that are then submitted to a theorem prover. For this step we intend to use a tool, developed in the context of a PhD, called GamaSlicer[2] [3].

To verify if the component invocations respect both the precondition and postcondition we will resort to slicing techniques. Traditional program slicing is a syntactic technique, but semantic forms have also been studied for over 10

[1] http://sort.ucd.ie/products/opensource/ESCJava2/

[2] http://gamaepl.di.uminho.pt/gamaslicer

years now, which combine slicing techniques with program annotation and verification, to identify synergies and take advantage of good practices on both sides[4], [5], [6].

In the context of GamaSlicer project, we have introduced a new concept: the *contract-based slice* of a program. A contract-based slice can be calculated by slicing the code of each procedure individually with respect to its contract (what we call an *open slice*), or taking into consideration the calling context of each procedure inside a program (which we call a *closed slice*).

Given an annotated procedure $\mathbf{p}_1$, we infer from a specific call (occurring in the body C of another annotated procedure $\mathbf{p}$) a specific postcondition that we use to slice the callee $\mathbf{p}_1$, keeping just the statements of its body C_1 that are actually relevant for that postcondition. The new procedure $\mathbf{p}'_1$, whose body is the slice C'_1, can be seen as a specialized version of the original $\mathbf{p}_1$. Actually, we go one step further and take into consideration all calls to the annotated callee $\mathbf{p}_1$ inside the current program. In this way, we obtain a closed slice that satisfies the conditions of all the calls but from which useless statements (with respect to that program) have been removed.

To illustrate our idea, please consider the Example 1 listed below that computes the maximum difference among student ages in a class. This component reuses the annotated component Min, defined also in Example 1, that returns the lowest of two positive integers.

Example 1 DiffAge

```
public int DiffAge() {
    int min = System.Int32.MaxValue;
    int max = System.Int32.MinValue;
    System.out.print(”Number of elements: ”);
    int num = System.in.read();
    int[] a = new int[num];
    for(int i=0; i<num; i++) {
        a[i] = System.in.read();
    }
    //calculating max and min
    for(int i=0; i<a.Length; i++) {
        max = Max(a[i],max); min = Min(a[i],min);
    }
    int diff = max − min;
    return diff;
}

 /*@ requires x >= 0 && y>=0;
   @ ensures (x > y)? \result == x : \result == y;
   @*/
   public int Min(int x, int y) \{
      int res = x − y;
      return (res > 0)? x : y);
   \}
```

The first step in this process, analyzes the Min invocation and builds a list with all the identifiers referred on it. In the second step, a backward slicing process is activated for all symbols in the list created in the previous step. Then, using the obtained slices, starts the detection of contract violations. For that, the precondition is back propagate along the slice to verify if it is preserved after each statement. Observing the slice for the array a, listed in the example 2 below, it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected. The third step consists in the notification of all the contract violations detected. In the example above, the user will receive an *warning* alerting to the possible invocation of Min with negative numbers (what does not respect the precondition). Besides the traditional error messages that should be issued at this point, we plan to include a visual aid. The tool will display a Labeled Control Flow Graph [7] exhibiting all problems identified.

Example 2 Backward Slicing for a[i]

```
int[] a = new int[num];
for(int i=0; i<num; i++) {
    a[i] = System.in.read();
}
for(int i=0; i<a.Length; i++) {
    max = Max(a[i],max); min = Min(a[i],min);
}
```

References

[1] N. A. M. Maiden and A. G. Sutcliffe, “People-oriented software reuse: the very thought,” in *Advances in Software Reuse - Second International Workshop on Software Reusability*. IEEE Computer Society Press, 1993, pp. 176–185.

[2] K. Sherif and A. Vinze, “Barriers to adoption of software reuse a qualitative study,” *Inf. Manage.*, vol. 41, no. 2, pp. 159–175, 2003.

[3] D. da Cruz, P. R. Henriques, and J. S. Pinto, “Gamaslicer: an online laboratory for program verification and analysis,” in *Proceedings of the 10th Workshop on Language Descriptions Tools and Applications (LDTA’10)*, 2010.

[4] J. J. Comuzzi and J. M. Hart, “Program slicing using weakest preconditions,” in *FME ’96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods*. London, UK: Springer-Verlag, 1996, pp. 557–575.

[5] I. S. Chung, W. K. Lee, G. S. Yoon, and Y. R. Kwon, “Program slicing based on specification,” in *SAC ’01: Proceedings of the 2001 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2001, pp. 605–609.

[6] C. Fox, S. Danicic, M. Harman, and R. M. Hierons, “Backward conditioning: A new program specialisation technique and its application to program comprehension,” in *IWPC*. IEEE Computer Society, 2001, pp. 89–97.

[7] J. Barros, D. da Cruz, P. R. Henriques, and J. S. Pinto, “Specification-based Slicing and Slice Graphs,” 2010, unpublished draft, available from http://alfa.di.uminho.pt/~danieladacruz/specificationSlice.pdf.