Automatic Inter-Procedural Test Case Generation

Karim-Cyril Griche
Laboratoire LSR-IMAG
BP 72, 38402 Saint Martin d'Hères Cedex, France
{karim-cyril.griche@imag.fr}

Abstract

This paper present a new approach to automatic structural test case generation in presence of called functions. It is based on constraint logical programming. Usual way of doing structural test in presence of called function is to replace these functions by stubs or, on the other side, to completely unfold the called functions. In this paper we propose a new way situated between the existent ones. It is based on the idea of functions simplification. We exposes a synthetic study of programme transformation techniques and then take advantage of them to define heuristics that will achieve functions simplification in the particular context of the test tool prototype Inka.

Introduction

During the last two decades, the way software is written has grown up. New programming languages and conception method have risen. A whole new development cycle is born.
Within this cycle, different phases can be seen: conception, realisation and verification. Our work fits in the last phase, verification; more precisely we test the software to check its correctness.
There is basically two different kind of test: static test, which is a code-based (structural) test and dynamic test, which use software executions. These tests method are completing each other's. We are going to focus on static tests method.

Our work is based on a new approach of the automatic structural test case generation problem defined in [GBR98]. It uses constraint logic programming (CLP) to try and solve the problem of generating test cases in order to attain the structural covering of a procedure. A test tool prototype, named Inka, has been developed. Inka is designed for automatic structural test case generation for C programmes. The operating cycle of Inka is cut in three parts:

- The first part translate a C program into it's equivalent in constraint logic program.
- The second module of Inka is used to minimize the set of constraints representing the C program. It uses the prolog solvers to propagate constraints until it reaches a fix point.
- Then the last module use the restraint set of constraints in order to find a test case reaching a node selected in the control flow graph of the procedure.

Tests realized by Inka are generated to achieve structural covering of a procedure. Usually this kind of test is done during the development of software, when all the procedures representing the software are not yet available. This means that, in order to achieve such a test, stubs must replace the missing procedures called. When all of the procedures representing a part of the software are available, some tests must be performed to assure that they work well together.

For now on, Inka is treating function calls by replacing the call by the set of constraints representing the called function. It unfolds systematically the called functions. While this method has the advantage to be transparent and to treat naturally the recursive functions, analyse of the whole unfold code is much too expensive in execution time.

Our work in the project is to find a way for Inka to treat large system comporting a lot of function calls without having to unfold systematically each call. Basically our approach of this problem is to find an alternative between replacing the called functions by stubs and unfold them systematically.
We want to use simplified called functions, in other terms we try to create realistic stubs for testing a procedure when knowing the called functions. By simplified function, we mean function that can decrease the cost of automatic structural test case generation in presence of called functions and function that are consistent (coherent) with the behaviour of the original functions.

In this paper we present two axes of investigations we are following. The first axis is theoretic: we report a bibliographic study made in order to determine which techniques could match with the idea of function simplification. The second axis is more empiric: we define our needs and determine some heuristics, which will be implemented and tested into prototype.

The first part of this paper is devoted to the state of art. It presents a synthetic study of methods of program's transformation and approximation. We show the matches between the idea of program simplification and the studied methods. The second part is devoted to

an empiric way of seeing program simplification. We approach this problem from different angles:

- How to create simpler function models
- Different possible configuration for using these models
- Experimental techniques to decrease complexity

In conclusion we sum up the results of our different approaches and describe the perspectives at middle and long range.

Programme Transformation Techniques

The goal of this section is not to present an overview of techniques intended to be directly applied to propose "realistic stubs". We try to inspire ourselves of the presented techniques in order to take profit of theses works to develop a solution based on heuristics to our specific problem.

Approximation is a concept based upon abstraction. It is present in every domain where calculus is used. (Mathematics, Physics…) The objective is to solve the complexity problem of a situation at cost of information and precision loss. In computer science, semantic analyse of complex programmes need to resort to approximation. One of the most famous techniques in this field is *abstract interpretation*.

Program transformation is a concept that can produce code by operating shifts in another programme's code. We are particularly interested into two types of shifts: *decomposition* and *specialisation*.

Decomposition is used to obtain a precise part of a programme, it's usually realised by cutting an original source code depending on a pre-established criterion. The technique operating such a structural transformation is named *Programme Slicing*.

Specialisation is a technique that will not alter the programme structure but it's behaviour. It's used to specialise the comportment of a programme to adapt it to a more precise environment. The resulting programme is an optimised version of the source programme designed to operate more accurately in particular cases. The method implementing this kind of optimisation is called *partial evaluation*.

Abstract Interpretation [Cou75] is a theory that deal with programming or specification, it is use to approximate their semantics. This technique has the ability to statically analyse dynamic programme proprieties. It can be considered as an extension to compilation techniques, which permit the software writer to predict future applications behaviour.

This technique formalises the idea that a programme's semantics can be more or less precise depending on the level of abstraction we are setting on. Abstract Interpretation is defined as a symbolic interpretation of a programme, which associates abstract values to variables instead of concrete's ones that are used during a programme execution. Afterwards each operation of the programme is interpreted with these abstract values.

A trivial example for this technique is illustrated by the sign rule, which allows predicting the sign of an arithmetic expression before even calculating it. Let say that x is a positive number and y a negative number. As you can see in the tables below, the abstraction of the expression "x*x + y*y" over the sign rule allow us to deduce that this expression is positive before calculating it.

Usually abstract interpretation of a programme is done in 2 phases: at first we build the control flow graph, then we apply the abstract interpretation algorithm on the graph. [Cou75] To each node of the control flow graph is associated an atomic abstract interpretation. The interpretation of the complete programme is a set of unitary interpretation on each node of the control flow graph.

Abstract interpretation is commonly used to statically analyse programmes. (Static verification of dynamic properties by using invariant calculus, Automatic differentiations between programme's semantics, Programme's transformation and optimisation)

Slicing is a structural decomposition technique. It "cuts" a function code and extracts it. Intuitively this technique correspond to a mask that an observer can put on a programme to focalise its attention on a particular point. It matches well with mental abstraction a developer can develop while debugging software.

In order to realise a slice of a function, a slicing criterion C=<p,v> must be defined. It consists of a point *n* in the function and of a set of variables *v*. The slice is realised by extracting a set of instructions, which might eventually affect the variables of *v* at the point *n*.

There are two levels of programme slicing: intra-procedural slice and inter-procedural slice. In its original definition, proposed by Weiser [Wei79], the intra-procedural slice is determined sequentially by calculating sets of instructions potentially impacting on our criterion. To determine which instructions may impact, we follow dependencies given by the data flow graph and the control flow graph. Inter-procedural programme slicing uses another graph, defined in [OO84], called programme dependency graph (PDG).

Partial Evaluation is based on the principal that say "if a programme can be executed with *n* parameters, it could be executed with *(n-m)* parameters if *m* arguments are statically known". This idea is based on a well-known Mathematics principal called *projection* or *restriction*. This technique is interesting as soon as you have spotted parameters that are often static. Once you have determined which parameters are static, using grounded values for these parameters will generate a *residual programme* that will give the same results than the original programme on a set of parameters (dynamic values).

The principal advantage of this technique is to produce a faster executing programme. It is used in

Compilation and Interpretation, Code optimisation, Meta programming.

Through this presentation of techniques, we wanted to establish a connexion between the concept of function simplification we had in mind and the different notions of programme's transformation and approximation. We wanted to get a clear view of what could be directly or indirectly applied in the context of our work. The next chapter formalises our needs for functions simplification. These needs are driven by the desire to adapt techniques presented to our particular problem.

Functions simplification

The theoretic study presented in the previous section allowed us to make a statement on what can be done to transform a function. We do not intend to directly use the techniques presented mainly for cost purposes. The gain in term of execution time does not seem to be sufficient to motivate a complete implementation in the Inka prototype. Moreover the internal architecture of Inka, based on CLP, forbids the direct use of the algorithm given in the studied papers. This conclusion brings us to try and define our own heuristics inspired of the techniques in order to produce simpler models of functions.

This section presents the approach we chose to define heuristics. It is separated in two parts. The first part presents our view of the different configuration possible for the models and advantages of each configuration. The second part defines the heuristics imagined so far and concludes on the gain achieved by implementing one of the heuristics.

Configuration Analyse

Studying transformation techniques showed us the necessity to have precise information on the context in which function models will be used. This brings us to define our needs to create models designed to replace the call to original functions. We define the context of our models as the information taken from the calling function. We separate this context, called *context of use*, in two categories: *calling context* and *objective for generation*.

Calling context. The calling context represents the state of the calling function just before the call to the called function. This state is defined as a set of domains concerning the variables and parameter used by the called function. It evolves following the calling function's comportment. This evolution changes the domains of parameters, which will then influence the called function.
The information contained in the calling context are useful essentially because of the influence they can have on the choice of control flow for our test objective (structural coverage). A typical example of calling context impact is a non-executable path in a function.

Objective for generation. This information intends to represent the consequence of a function call; it models the future use results given by the called function just after a call. Objectives could help to identify control flow needed to attain a structural goal. Being able to define and generate such an information as objective for generation could be an efficient way to exclude from the called function, paths which are not significantly useful for structural covering of the calling function.

Now that we have intuitively defined the context of use for models, we can answer to the questions implied by the modelling activity. The first question concerns the configuration and bounds surrounding models creation, which will determine the number of models needed. The second question is to describe the way to create models, we have envisaged two possibilities: static models and dynamic ones.

We have distinguished three different schemes for models creation depending of the use or not of the context of use:

- *One model for each called function*: this method does not need a context, we create one model per function and all calls to a function will be rerouted to the model. This scheme is basic and simple, we create very few models, one for each function susceptible to be called. We could create statically these models. Unfortunately this scheme does not take advantage of the context to approximate or specialise the function into a model.
- *One model for each pair of calling-called functions*: here we do the contrary of the previous scheme. We create a model for the called function each time it has a calling function. So for a called function we have as many models as there are calling functions to use them. The advantage of this scheme is to be more realistic by being specific to the calling function. Yet the models creating is far more complicated because of the constraints we have to gather concerning the calling function.
- *One model for each call to a function*: this scheme goes further into refining the models. It uses specificities of the calling function and more precisely the context of use for the model at the time it is called. Each function has as many models as this function is called. Each model is created with the maximum information we have about what it should be. Nevertheless this solution is more expensive and need particular tools allowing the calculus of context of use for each call.

For each model we have the possibility to create it statically in a pre-process or choose to create it on the fly at the time it is needed. In the static way we create a model for a function then we alter the calling function to make it call the model. On the contrary, the dynamic

way create a model depending on the state of the calling function, we do so each time a function is called.

We have three configurations and two modes for creation. At this time of the work, we can state that choices between two configurations depend on contradictory parameters. This choice is often a compromise between cost and expected gain from use. We can choose to create model statically or dynamically but some configuration implies the use of a particular model, for example "One model for each called function" is static by nature.

Gain from Transformation techniques

None of the studied method proposed a usable solution to our particular problem but each of them gives a few elements of solutions to certain parts of our problem. This section describes the way we intend to use each technique to solve a part of our general problem.

Abstract Interpretation, as described before, can be used to generate abstract context of a programme. A relation can be made between abstract contexts as defined in [Cou76] and calling context described earlier. In fact the abstract context of a calling function just before a call is constituted by a set of definition domain for variables. We are interested in global variables and parameters transmitted to the called function, this information can be found in the abstract context.

The idea is to use the abstract interpretation algorithm to create the abstract context of a function considering the call instant as our reference. An implementation of this method to build calling context could be:

-1 - Build the procedure's control flow graph
-2 - Generate abstract context for each edge of this graph
-3 - Extract calling context: the abstract context on the edge leading to a function call

Such a context could now be used in combination with a modified version of partial evaluation. We explained earlier that partial evaluation heavily depends on static parameters, which could be propagated into a function to specialise it. We don't have the possibility to use static parameters but a calling context can be propagated by the same way. The result of this process would not be a specialise function for parameters but a specialised function for a context. We have eliminated, in this function, control flows clearly not executable in the calling context.

Once we got rid of unnecessary control flow we can go further into control flows elimination. The created models must be able to produce realistic values; this means values that could be produced by the original function in the same situation. But our goal is structural covering of the calling function, so our idea is to say that we only need a sub set of all values that a called function can produce.

Actually we don't have the possibility of calculating the exact sub set needed. The attempt to do so is bound to the precise definition of the *Objectives for generation*. Until then we can suppose that the simplest path of the called function could produce the values needed by structural covering. This is a strong supposition, which allows us to try and eliminate, in the function, paths that are clearly complicating the automatic generation of structural test cases.

Transformation techniques were not directly applicable to function's simplification in Inka. They helped though to solve parts of the global problem. As it was not sufficient we propose another approach based on heuristics capable of simplifying the way Inka treats function calls.

Heuristics

In order to reduce complexity of called functions without completely loosing test cases pertinences, we define two heuristics simplifying the problem in given situations. They are based on two ideas: *Paths elimination* and *limitation on the call graph depth*.

Paths Elimination

This technique is using a binary evaluation of branches from the control flow graph. It is based on the supposition made in the precedent paragraph. We suppose achieving the structural covering of a calling function only needs a sub set of all values generated by a called function, so we try to eliminate branches judged to complicated.

To do so we use a weighting function, which gives a weight to each instruction of a function. The rules defining weights are defined to reflect complexity of each instruction in the Inka's solving process. A first basic weighting process could give a higher weight to branches comporting function calls or loops, as they are constraints generating operators in Inka. Of course the only way to define a good weighting function and to compare them is experiences.

This technique will be implemented in two modules. The first one is static and gives weight to branches. The second one is dynamic; it alters Inka working process in order to force it to prefer lighter branches when it is possible.

Limitation on the call graph depth

We have seen at the beginning of this document that currently Inka was dealing with function calls by unfolding them completely. This is a problem when the number of calls in a system under test is too important. This lead us to define a new politic for function's

unfolding onto Inka. To do so we have doted the prototype of a way to stop function's unfolding once it reaches a defined deepness in the calling graph of a system.

This new operator is motivated by the idea that the time and complexity gained by stopping unfolding will be greater than the information loosed by not propagating more constraints. Technically this is a dynamic method, which create on the fly models for a function. Here models are not explicitly created but this heuristic can be seen as an approximation for the first function called.

Its implementation needs to interfere with Inka but it has the advantage not to need any static pre-processing. This heuristic is implemented and has been tested on examples. So far we can say that we know what can be gained at best. Further experiences must be taken to have a clear view of the gain in best and worst situation for this heuristic.

Conclusions and perspectives

The work presented along this paper intends to extend a test tool in order to make it capable of generation structural test cases in presence of called functions. Usual solutions to this problem propose the use of stubs or unfold the called functions. We have exposed the reason why these methods are limited and don't apply well to our problem.

In order to solve this difficulty our work objective is to propose an alternative between stubs and unfolding. Our idea is to create simplified version of the called function. These models must be simpler than the original but still accurate with our goal of structural covering.

The work presented in this paper consists firstly on a study of different programme transformation techniques in order to evaluate their pertinence with our problem. We analysed the possibility of direct or indirect use of any of these techniques in the idea of achieving functions simplification as we defined it.

The theoretic study leads us to classify the techniques on the axis of transformation and approximation. We use this classification to identify possible gain from the use of any of these techniques. The interest of such a classification is to propose a conceptual and methodological framework to elaborate a better solution.

Despite the specificity of our problem we noted few direct use of abstract interpretation in the calling context calculus and of slicing to extract paths from control flow graphs. We intend to adapt partial evaluation to make it work with domain of values, as it would be a great gain to our idea of functions simplification.

On the practical field we propose an approach based on two heuristics. One of these has been implemented into the prototype and early results in term of execution time are really encouraging.

In the near future, our work will focus on the implementation of the other heuristic. Then further experiences will be taken to precisely calibrate both heuristics to make them cooperate into Inka. The next step would work toward a better use of the studied techniques by implementing part of them into heuristics, particularly calling context calculus and propagation of this context in the called function through adapted partial evaluation.

References

GBR98 A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. *Software Engineering Notes*, 23(2):53-62, Mar. 1998. disponible at http://www.esi.fr/~rueher

Cou75 Cousot P., Cousot R., *''Vérification statique de la cohérence dynamique des programmes''*, La boratoire d'Informatique, U.S.M.G., Grenoble, Research Report of contract IRIA-SESORI 75-035, Sept. 1975.

Cou76 Cousot P. et Cousot R. '' *Static determination of dynamic properties of programs''*, In Proceedings of the Second International Symposium on Programming, p. 106-130, Dunod, Paris, 1976.

OO84 Ottenstein K., Ottenstein L., '' *The program Dependence graph in Software development environments*'', In proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, p. 177-184, May 1984.

Wei79 Weiser M., '' *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*'', PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.