Component Identification Through Program Slicing

Nuno F. Rodrigues [1,2]

Departamento de Informática
Universidade do Minho
Braga, Portugal

Luís S. Barbosa [1,3]

Departamento de Informática
Universidade do Minho
Braga, Portugal

Abstract

This paper reports on the development of specific slicing techniques for functional programs and their use for the identification of possible coherent components from monolithic code. An associated tool is also introduced. This piece of research is part of a broader project on program understanding and re-engineering of legacy code supported by formal methods.

Key words: Program Slicing, Static Analysis, Component Identification.

1 Introduction

A fundamental problem in system's re-engineering is the identification of coherent units of code providing recurrently used services. Such units, which are typically organised around a collection of data structures or inter-related functions, can be wrapped around an interface and made available as software components in a modular architectural reconstruction of the original system. Moreover they can then be made available for reuse in different contexts.

This paper proposes the use of software *slicing* techniques to support such a component's identification process. Introduced by Weiser [16,14,15] in the

[1] The research reported in this paper is supported by FCT, under contract POSI/ICHS/44304/2002, in the context of the PURe project.

[2] Email: `nfr@di.uminho.pt`

[3] Email: `lsb@di.uminho.pt`

late Seventies, program slicing is a family of techniques for isolating parts of a program which depend on or are depended upon a specific computational entity referred to as the *slicing criterion*. Its potential for service or component identification is therefore quite obvious. In practice, however, this requires

- A flexible definition of what is understood by a *slicing criterion*. In fact, Weiser's original definition has been re-worked and expanded several times, leading to the emergence of different methods for defining and computing program slices. Despite this diversity, most of the methods and corresponding tools target either the imperative or the object oriented paradigms, where program slices are computed with respect to a variable or a program statement.
- The ability to extract actual (executable) code fragments.
- And, of course, suitable tool support.

All these issues are addressed in this paper. Our attention, however, is restricted to *functional* programs [2]. Such focus is explained not only by the research context mentioned below, but also because we deliberately want to take an alternative path to mainstream research on slicing where functional programming has been largely neglected. Therefore our research questions include the definition of what a slice is for a functional program, how can program data be extracted and represented, what would be the most suitable criteria for component identification from functional monolithic code. There is another justification for the qualificative *functional* in our title: the tool that supports the envisaged approach was entirely developed in Haskell [2].

The context for this research is a broader project on *program understanding and re-engineering* of legacy code supported by formal methods. A number of case-studies in the project deal with functional code, even in the form of executable specifications [4]. Actually, if forward software engineering can today be regarded as a lost opportunity for formal methods (with notable exceptions in areas such as safety-critical and dependable computing), reverse engineering looks more and more a promising area for their application, due to the engineering complexity and exponential costs involved. In a situation in which the only quality certificate of the running software artefact still is life-cycle endurance, customers and software producers are little prepared to modify or improve running code. However, faced with so risky a dependence on legacy software, managers are more and more prepared to spend resources to increase confidence on — *i.e.*, the level of understanding of — their code.

The paper is organised as follows. Section 2 reviews basic concepts in program slicing and introduces introduces functional slicing, specifying a new representation structure — the *FDG* (*Functional Dependence Graph*) — and the slicing operations over it. The corresponding prototype tool (*HaSlicer*)

[4] Specification understanding is not so weird as it may look at first sight. Actually, the authors became aware of the amount and relevance of *legacy specifications* in the context of an industrial partnership on software documentation.

is described in section 3. Section 4 discusses how these techniques and tool can be used for 'component discovery' and identification. A small example is included to illustrate the approach. The paper ends with a small section on conclusions and future work.

2 Functional Program Slicing

2.1 Program Slicing

Weiser, in [15], defines a program slice S as *a reduced executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P*. A complementary definition characterizes program slices as fragments of a program that influences specific computational result inside that program [13]. The computation of a program slice is called *program slicing*. This process is driven by what is referred to as a *slicing criterion*, which is, in most approaches, a pair containing a line number and a variable identifier. From the user point of view, this represents a point in the code whose impact she/he wants to inspect in the overall program. From the program slicer view, the slicing criterion is regarded as the *seed* from which a program slice is computed. According to Weiser original definition a slice consists of an executable sub-program including all statements with some direct or indirect consequence on the result of the value of the entity selected as the slicing criterion. The concern is to find only the pieces of code that affect a particular entity in the program.

Weiser approach corresponds to what would now be classified as a *backward*, *static* slicing method. A dual concept is that of *forward slicing* introduced by Horwitz et al [5]. In forward slicing one is interested on what depends on or is affected by the entity selected as the slicing criterion. Note that combining the two methods also gives interesting results. In particular the union of a backward to a forward slice for the same criterion n provides a sort of a selective window over the code highlighting the *region* relevant for entity n.

Another duality pops up between *static* and *dynamic* slicing. In the first case only static program information is used, while the second one also considers input values [6,7] leading frequently, due to the extra information used, to smaller and easier to analyse slices, although with a restricted validity.

Slicing techniques are always based on some form of abstract, graph-based representation of the program under scrutiny, from which dependence relations between the entities it manipulates can be identified and extracted. Therefore, in general, the slicing problem reduces to sub-graph identification with respect to a particular node. Note, however, that in general slicing can become a highly complex process (*e.g.*, when acting over unstructured control flow structures or distributed primitives), and even, in some cases undecidable [12].

2.2 Functional Program Slicing

As mentioned above, mainstream research on program slicing targets imperative languages and, therefore, it is oriented towards particular, well characterised notions of computational variable, program statement and control flow behaviour. Slicing functional programs requires a different perspective. Functions, rather program statements, are the basic computational units and functional composition replaces statement sequencing. Moreover there is no notion of assignable variable or global state whatsoever. Besides, in modern functional languages encapsulation constructs, such as Haskell [2] *modules* or Ml [4] *abstract data types*, provide powerful structuring mechanisms which can not be ignored in program understanding. What are then suitable notions of slicing for functional programs? More specifically: suitable with respect to the component identification process? Such is the question set in this section.

2.3 Functional Dependence Graphs

As mentioned above slicing techniques are always based on some kind of *dependence graph*. Typical such structures are *control flow graphs* (CFG) and *program dependence graphs* (PDG).

For a program P, a CFG is an oriented graph in which each node is associated with a statement from P and edges represent the corresponding flow of control between statements. These kind of graphs rely entirely on a precise notion of program statement and their order of execution inside the program. Since functional languages are based on expression rather than statements, CFG's are not immediately useful in performing static analysis over functional languages.

A PDG is an oriented graph where the nodes represent different kinds of entities in the source code, and edges represent different kinds of dependencies. The entities populating the nodes can represent functions, modules, datatypes, program statements, and other kind of program structures that may be found in the code. In a PDG there are different sorts of edges (*e.g.*, loop-carried flow edges, loop-independent flow edges, control dependence edges, etc) each representing a different kind of dependency between the intervenient nodes.

Adapting the definition of PDG's to the functional paradigm, one may obtain a structure capturing a variety of information that, once combined, can form the basis of meaningful slicing criteria. This leads to the following definition.

Definition 1 (Functional Dependence Graph) *A Functional Dependence Graph (FDG) is a directed graph, $G = (E, N)$ where N is a set of nodes and $E \subseteq N \times N$ a set of edges represented as a binary relation between nodes. A node $N = (t, s, d)$ consists of a node type t, of type $NType$, a source code location s, of type $SrcLoc$ and a description d of type $Descr$.*

A source code location is simply an index of the node contents in the actual source code.

Definition 2 (SrcLoc) *The type SrcLoc is a product composed by the source file name and the line-colunm code coordinates of a particular program element,* i.e., $SrcLoc = SrcFileName \times SrcBgnLine \times SrcBgnColumn \times SrcEndLine \times SrcEndColumn$.

More interesting is the definition of a node type which captures the information diversity mentioned above and is the cornerstone of FDG's flexibility.

Definition 3 (NType) *the type of a FDG node is given by the following enumeration of literals*

$$\begin{aligned} NType \; = \; & N_m(module) \mid N_f(function) \\ & \mid N_{dt}(data\ type) \mid N_c(constructor) \\ & \mid N_d(destructor) \end{aligned}$$

Let us explain in some detail the intuition behind these types.

Nodes bearing the N_m (Module) type, represent software modules, which, from the program analysis point of view, corresponds to the highest level of abstraction over source code. Note that Haskell has a concrete definition of module, which makes the identification of N_m nodes straightforward. Modules encapsulate several program entities, in particular code fragments that give rise to other FDG nodes. Thus, a N_m node depends on every other node representing entities defined inside the module as well as on nodes corresponding to modules it may import.

Nodes of type N_f represent functions, *i.e.*, abstractions of processes which transform some kind of input information (eventually void) into an output (eventually void too). Functions are the building blocks of functional programs, which in most cases, decorate them with suitable type information, making extraction simple. More complex is the task of relating a function node to the nodes corresponding to computational entities in its body — data type references, other functions or what we shall call below *functional statements.*

Constructor nodes (N_c) are specially targeted to functional languages with a precise notion of explicit type constructors (such as the ones associated to *datatype declarations* in Haskell). Destructor nodes (N_d) store datatype selectors, which are dual to constructors, and again specific to the functional paradigm [5] .

This diversity of nodes in the FDG is interconnected by arcs. In all cases an edge from a node n_1 to a node n_2 witnesses a dependence relation of n_2 on n_1. The semantics of such a relation, however, depends on the types of both

[5] A similar notion may, however, be found in other contexts — *e.g.*, the C selector operator "." which retrieves specific fields from a `struct` construction. Object oriented languages also have equivalent selector operators.

Target NType	**Source NType's**	**Edge Semantic**
N_m	$\{N_m\}$	Target node imports source node
N_m	$\{N_f,\ N_c,\ N_d,\ N_{dt}\}$	Source node contains target node definition
N_f	$\{N_{st}\}$	Statements belong to function definition
N_f	$\{N_c, N_d,\ N_{dt}, N_f\}$	Function is using target node functionality
N_{dt}	$\{N_{dt}\}$	Source data-type is using target data-type
N_{dt}	$\{N_c\}$	Data-type is constructed by target node
N_{dt}	$\{N_d\}$	Data-type is destructed by target node

Table 1
FDG Edge Description

nodes. For example, an edge from a N_f (function) node n_1 to a N_m (module) node n_2 means that the module represented by n_2 depends on the function associated to n_1, that is, in particular, that the function in n_1 is defined inside the module in n_2.

On the other hand, an edge from a node n_3 to n_4, both of type N_f, witnesses a dependence of the function in n_4 on the one in n_3. This means, in particular, the latter is called by the former. Notice the difference from the N_m, N_f case where dependence means definition inside the module.

Table 1 introduces the intended semantics of edges with respect to the types of nodes they connect. Also note that a FDG represents only *direct* dependencies. For example there is no node in a FDG to witness the fact that a module uses a function defined elsewhere. What would be represented in such a case is a relationship between the external function and the internal one which calls it. From there the indirect dependence can be retrieved by a particular slicing criterion.

2.4 The Slicing Process

Program slicing based on *Functional Dependence graphs* is a five phase process, as illustrated in figure 1. As expected, the first phase corresponds to the parsing of the source code, giving origin to an *abstract syntax tree* (AST) instance t. This is followed by an abstraction process that extracts the relevant information from t, constructing a FDG instance g according to the different types of nodes found.

The third phase is where the actual slicing takes place. Here, given a

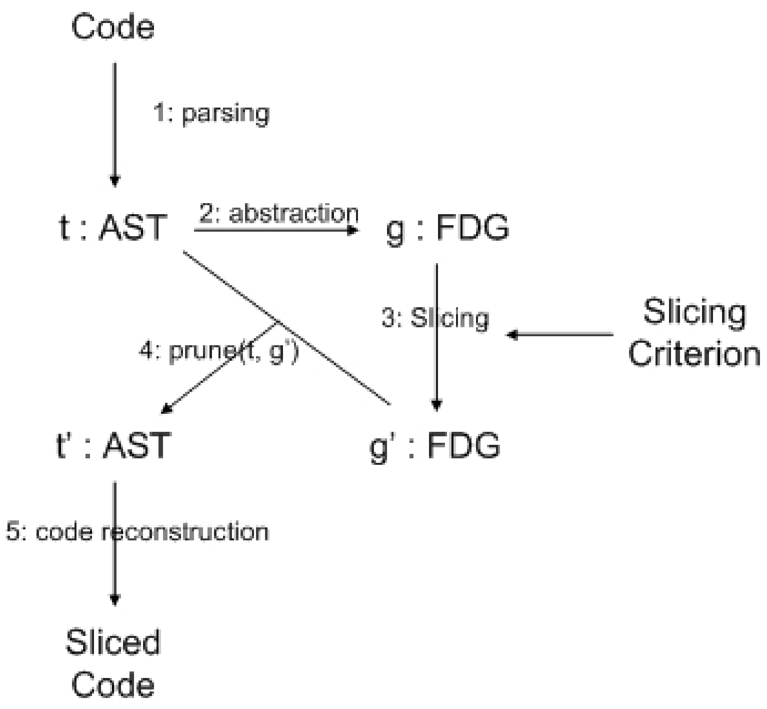


Fig. 1. The slicing process

slicing criterion, composed by a node from t and a specific slicing algorithm, the original FDG g is sliced, originating a subgraph of g which is g'. Notice that, slicing takes place over the FDG, and that the result is always a subgraph of the original graph.

The fourth phase, is responsible for pruning AST t, based on the sliced graph g. At this point, each program entity that is not present in graph g', is used to prune the correspondent syntactic entity in t, giving origin to a subtree t' of t. Finally, code reconstruction takes place, where the pruned tree t' is consumed to generate the sliced program by an inverse process of phase 1.

In [9] a number of what we have called *slicing combinators* were formally defined, as operators in the relational calculus [1], on top of which the actual slicing algorithms, underlying phases three and four above, are implemented. This provides a basis for an algebra of program slicing, which is, however, out of the scope of this paper.

3 The HaSlicer Prototype

HaSlicer[6] is a prototype of a slicer for functional programs entirely written in Haskell built as a proof-of-concept for the ideas discussed in the previous section. Both forward, backward and forward dependency slicing are covered.

[6] The prototype is available for testing at http://wiki.di.uminho.pt/wiki/bin/view/Nuno

Node Color	Node Type
🟥	N_m
🟦	N_f
🟩	N_{dt}
🟨	N_c
⬜	N_d

Table 2
FDG Edge Codes

In general the prototype implements the above mentioned slicing combinators [9] and addresses two other issues fundamental to component identification: the definition of the *extraction* process from source code and the incorporation of a *visual* interface over the generated FDG to support user interaction. Although its current version accepts only Haskell code, plug-ins for other functional languages as well as for the Vdm-Sl metalanguage [3] are currently under development.

Figure 2 shows two snapshots of the prototype working over a small Haskell program. Screenshot 2 (a), shows the visualization of the entire FDG loaded in the tool. Notice that the differently coloured nodes indicate different program entity types according to Table 2.

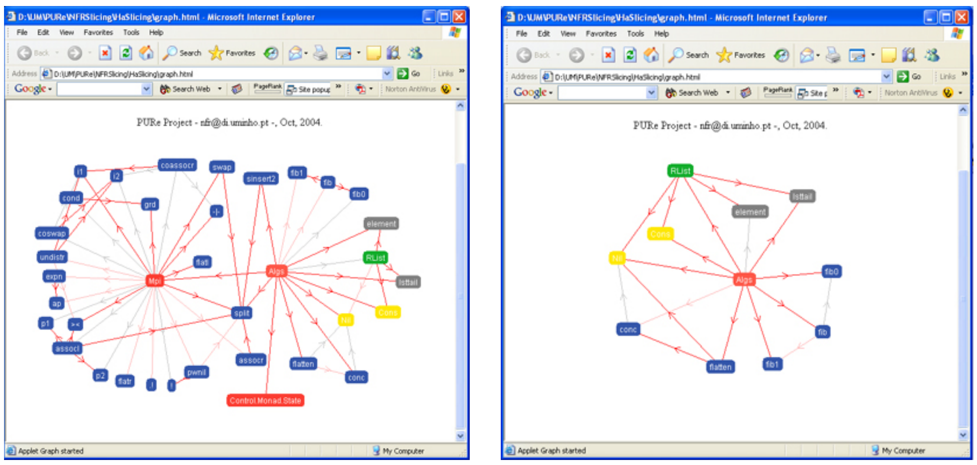
Figure 2.(a) reproduces the subgraph resulted from performing slice over one of the nodes of the graph from 2.(b). Once a slice has been computed, the user may retrieve the corresponding code. The whole process can also be undone or launched again with different criteria or object files.

4 Component Discovery and Identification

4.1 Two Approaches

There are basically two ways in which slicing techniques, and the HaSlicer tool, can be used in the process of component identification: either as a support procedure for manual component identification or as a 'discovery' procedure in which the whole system is searched for possible *loci* of services, and therefore potential components. In this section both approaches are briefly discussed.

The first approach deals with manual component identification guided by analysing and slicing some representation of the legacy code. In this context, the FDG seems to provide a suitable representation model. Through its analysis, the software architect can easily identify all the dependencies between



(a) (b)

Fig. 2. Slicing with HaSlicer

the code entities and look for certain architectural patterns and/or undesired dependencies in the graph.

One of the most interesting operations in this category is component identification by service. The idea is to isolate a component that implements a specific service of the overall system. The process starts in a top-down way, looking for the top level functions that characterise the desired service. Once these functions are found, *forward dependency slicing* is applied starting from the corresponding FDG nodes. These produces a series of sliced files (one per top level function), that have to be merged together in order to build the desired component. Note that a forward dependency slice collects all the program entities which each top level function requires to operate correctly. Thus, by merging all the forward dependency slices corresponding to a particular service one gets the least (derived) program that implements it.

This process leads to the identification of a new component which, besides being reusable in other contexts, will typically be part of the (modular) reconstruction of the original legacy system. But in what direction should such system be reorganized to use the identified service as an independent component? This would require an operation upon the FDG which is, in a sense, dual to slicing. It consists of extracting every program entity from the system, but for ones already collected in the computed slices. Such operation, which is, at present, only partially supported by HaSlicer, produces typically a program which cannot be immediately executed, but may be transformed in that direction. This amounts basically to identify potential broken function calls in the original code and re-direct them to the new component's services.

The second use of slicing mentioned in the beginning of this section under the designation of 'component discovery' relies on slicing techniques for the automatic isolation of possible components. In our experience this was found particularly useful at early stages of component identification. Such procedures, however, must be used carefully, since they may lead to the identification of both false positives and false negatives. This means that there might be good candidates for components which are not found as well as situations in which several possible components are identified which turn out to lack any practical or operational interest.

To use an automatic component 'discovery' procedure, one must first understand what to look for, since there is no universal way of stating which characteristics correspond to a potential software component. Thus, one has to look for components by indirect means, that certainly include the identification of certain characteristics that components usually bear, but also some filtering criteria.

A typical characteristic that is worthwhile to look for concerns the organization of a bunch of functions around a common data type structure. Therefore a possible criteria for component 'discovery' is based on the data types defined on the original code. The idea is to take each data type and isolate both the data type and every program entity in the system that depends on it. Such an operation can be accomplished by performing a *backward slicing* starting from each data type node in the FDG.

A second well known characteristic, identified by the object-orientation community, relates to the fact that 'interesting' components typically present a low level of coupling and a high level of cohesion[18]. Briefly, *coupling* is a metric to assess how mutually dependable two components are, *i.e.*, it tries to measure how much a change in one component affects other components in a system. On the other hand, *cohesion* measures how internally related are the functions of a specific component. Generally, in a component with a low cohesion degree errors and undesirable behaviour are difficult to detect. In practice if its functions are weakly related errors may 'hide' themselves in seldom used areas and remain invisible to testing for some time.

The conjunction of these two metrics leads to a 'discovery' criteria which uses the FDG to look for specific clusters of functions, *i.e.*, sets of strongly related functions, with reduced dependencies on any other program entity outside this set. Such function clusters cannot be identified by program slicing techniques, but the FDG is still very useful in determining this clusters. In fact these kind of metrics can be computed on top of the information represented in the FDG. The HaSlicer tool, in particular, compute their combined value through

$$\begin{aligned} Coupling(G, f) &\triangleq \sharp\{(x, y) \mid \exists x, y.\ yGx \wedge x \in f \wedge y \notin f\} && (1)\\ Cohesion(G, f) &\triangleq \sharp\{(x, y) \mid \exists x, y.\ yGx \wedge x \in f \wedge y \in f\} && (2)\\ CCAnalysis(G) &\triangleq \{(Coupling(G, f), Cohesion(G, f)) \mid \forall f \in \mathcal{P}F\} && (3) \end{aligned}$$

where G is a FDG and F a set of functions under scrutiny. Depending on how liberal or strict one wants the component discovery criteria to be, different

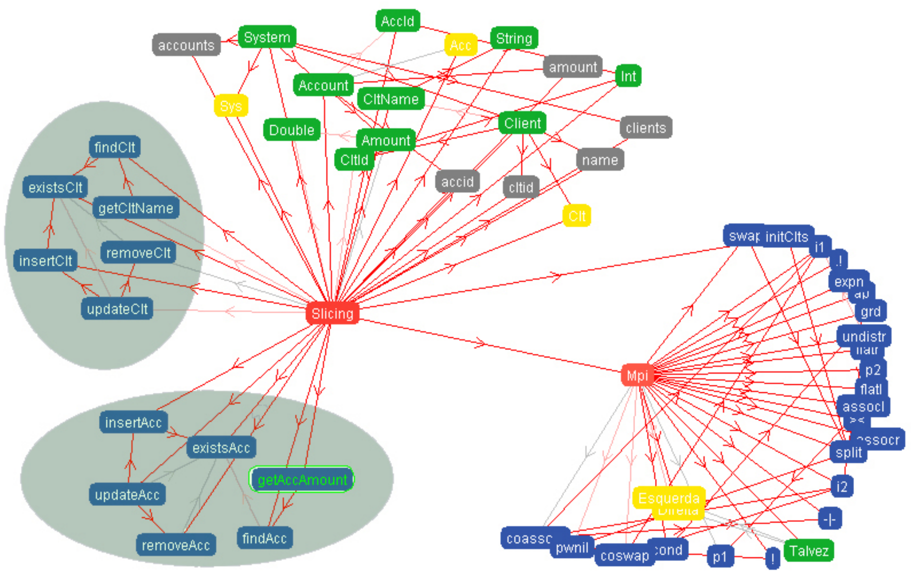


Fig. 3. FDG for the Toy Bank Account System

acceptance limits for coupling and cohesion can be used. This will define what clusters will be considered as *loci* of potential components. Once such clusters are identified, the process continues by applying *forward dependency slicing* on every function in the cluster and merging the resulting code.

4.2 A Toy Example

To illustrate the use of slicing for component identification, consider the Haskell code for a toy bank account system, shown in Appendix A. The corresponding FDG, as computed by HaSlicer is depicted in Figure 3.

If one tries to apply an automatic component 'discovery' method to this code, based, for example, in the combined cohesion-coupling metric, the number of cases to consider soon becomes very large. This occurs because the algorithm iterates powerset over the set of functions. Nevertheless, a simple filter based on both coupling, cohesion and cardinality of the sets under analysis largely decreases the number of cases to consider. The idea is to tune the 'discovery' engine to look for high cohesion values combine with both a low value of coupling and, what is most important, a reduced number of elements in the set being analysed. The results of applying such a filter to the example at hands are reproduced in Table 3.

Clearly, two components have been identified (corresponding to the gray area of the FDG in Figure 3): a component for handling Client information and another one for managing Accounts data. As mentioned above, the process would continue by applying forward dependency slicing over the nodes corresponding to the functions in the identified sets, followed by slice merging.

Functions' Clusters	Coh	Cou
getAccAmount findAcc existsAcc insertAcc updateAcc removeAcc	7	0
getCltName findClt existsClt insertClt updateClt removeClt	7	0

Table 3
Cohesion and Coupling Metric for Example 3

5 Related Work

The FDG definition used in our approach is closely related to the notion of Program Dependence Graph defined by Ottenstein and Ottenstein in [8], though we have specialized the graph to face the functional paradigm and introduced new semantics to the node relations.

Our methodology for component identification is based on the ideias first presented by Schwanke et al [11] [10], where design principles like coupling and cohesion are used to identify highly cohesive modules. Here we diverge from the existing approaches, by making use of the lazy properties of Haskell in order to obtain answers in an acceptable time.

A second difference between our approach to component identification and other techniques, which are usually included in the boarder discipline of software clustering [17], is that we are working with functional languages with no aggregation units other then the module itself. In contrast to this, most of the software clustering algorithms are oriented to the OO paradigm, and as a consequence, they are often based on the notion of class which is itself an aggregation construct. Thus, we have to cope with a much smaller granularity of programming units to modularize.

6 Conclusions and Future Work

Under the overall *motto* of functional slicing, the aim of this paper was two-fold. On the one hand a specific dependence graph structure, the FDG, was introduced as the core graph structure for functional slicing and a corresponding prototype developed. On the other hand it was shown how slicing techniques can be used to identify software components from (functional) legacy code, either as a support tool for the working software architect or in an automatic way in a process of component 'discovery'. The latter is particularly useful as an *architecture understanding* technique in the earlier phases of the re-engineering process.

What makes FDG a suitable structure for our purpose is the introduction of an ontology of node types and differentiated edge semantics. This makes possible to capture in a single structure the different levels of abstraction a program may possess. This way a FDG captures not only high level views of a software project (*e.g.*, how modules or data-types are related), but also low level views (down to relations between functional statements inside function's bodies, not discussed here but see [9]). Moreover, as different program abstraction levels are stored in a single structure, it becomes easy to jump across views according to the analyst needs. Finally, notice that the FDG structure is flexible enough to be easily adapted to other programming languages and paradigms.

An area of future research is the adaptation of graph clustering techniques, already available in the literature, to the discovery of components over FDG instances. Concerning this aspect, we have already carried out some experiences with adjacency matrixes algorithms which point to a significant reduction in the time to compute component candidates.

As mentioned in the Introduction, this research is part of a broader agenda. In such a context, current work includes:

- The generalization of slicing techniques to the software architecture level, in order to make them applicable, not only to architectural specifications (as in [19]), but also to the source code level of large *heterogeneous* software systems, i.e. systems that have been programmed in multiple languages and consists of many thousands of lines of code.
- The research on the interplay between component identification based on slicing, as discussed in this paper, and other analysis techniques (such as, *e.g.*, *type reconstruction*) also based on graph analysis.

References

[1] R. C. Backhouse and P. F. Hoogendijk. *Elements of a relational theory of datatypes.* In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.

[2] R. Bird. "Functional Programming Using Haskell". Series in Computer Science. Prentice-Hall International, 1998.

[3] J. Fitzgerald and P. G. Larsen. "Modelling Systems: Pratical Tools and Techniques in Software Development". Cambridge University Press, 1998.

[4] R. Harper and K. Mitchell. *Introduction to standard* MLA. Technical Report, University of Edimburgh, 1986.

[5] S. Horwitz, T. Reps, and D. Binkley. *Interprocedural slicing using dependence graphs.* In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conf. on Programming Usage, Design and Implementation*, pages 35–46. ACM Press, 1988.

[6] B. Korel and J. Laski. *Dynamic program slicing. Inf. Process. Lett.*, 29(3):155–163, 1988.

[7] B. Korel and J. Laski. *Dynamic slicing of computer programs. J. Syst. Softw.*, 13(3):187–195, 1990.

[8] K. J. Ottenstein and L. M. Ottenstein. *The program dependence graph in a software development environment.* In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering posium on Practical software development environments*, pages 177–184. ACM Press, 1984.

[9] N. Rodrigues. *A basis for slicing functional programs.* Technical report, PURe Project Report, DI-CCTC, U. Minho, 2005.

[10] R. W. Schwanke. *An intelligent tool for re-engineering software modularity.* In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 83–92, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[11] R. W. Schwanke and S. J. Hanson. *Using neural networks to modularize software. Mach. Learn.*, 15(2):137–168, 1994.

[12] A. M. Sloane and J. Holdsworth. *Beyond traditional program slicing.* In *the International Symposium on Software Testing and Analysis*, pages 180–186, San Diego, CA, 1996. ACM Press.

[13] F. Tip. *A survey of program slicing techniques. Journal of programming languages*, 3:121–189, 1995.

[14] M. Weiser. *Program Slices: Formal, Psychological and Practical Investigatios of an Automatic Program Abstraction Methods.* PhD thesis, University of Michigan, An Arbor, 1979.

[15] M. Weiser. *Programmers use slices when debugging. Commun. ACM*, 25(7):446–452, 1982.

[16] M. Weiser. *Program slicing. IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[17] T. A. Wiggerts. *Using clustering algorithms in legacy systems remodularization.* In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, USA, 1997. IEEE Computer Society.

[18] E. Yourdon and L. Constantine. "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design". Prentice-Hall, 1979.

[19] J. Zhao. *Applying slicing technique to software architectures.* In *Proc. of 4th IEEE International Conferencei on Engineering of Complex Computer Systems*, pages 87–98, August 1998.

A Toy Bank Account System

```
module Slicing where

import Mpi
```

```
data System = Sys { clients  :: [Client],
                    accounts :: [Account] } deriving Show

data Client = Clt { cltid :: CltId,
                    name  :: CltName } deriving Show

data Account = Acc { accid  :: AccId,
                     amount :: Amount } deriving Show

type CltId   = Int
type CltName = String
type AccId   = Int
type Amount  = Double


initClts :: [((CltId, CltName), (AccId, Amount))] -> System
initClts = (uncurry Sys) . split (map ((uncurry Clt) . fst))
                                 (map ((uncurry Acc) . snd))

findClt :: CltId -> System -> Maybe Client
findClt cid sys =
    if (existsClt cid sys) then Just . head . filter ((cid ==) . cltid) . clients $ sys
                           else Nothing

findAcc :: AccId -> System -> Maybe Account
findAcc acid sys =
    if (existsAcc acid sys) then Just . head . filter ((acid ==) . accid) . accounts $ sys
                            else Nothing

existsClt :: CltId -> System -> Bool
existsClt cid = elem cid . map cltid . clients

existsAcc :: AccId -> System -> Bool
existsAcc acid = elem acid . map accid . accounts

insertClt :: (CltId, CltName) -> System -> System
insertClt (cid, cname) (Sys clts accs) =
    if (existsClt cid (Sys clts accs)) then error "Client ID already exists!"
                                       else Sys ((Clt cid cname) : clts) accs

insertAcc :: (AccId, Amount) -> System -> System
insertAcc (acid, amount) (Sys clts accs) =
    if (existsAcc acid (Sys clts accs)) then error "Account ID already exists!"
                                        else Sys clts ((Acc acid amount) : accs)

removeClt :: CltId -> System -> System
removeClt cid (Sys clts accs) =
    if (existsClt cid (Sys clts accs)) then Sys (filter ((cid /=) . cltid) clts) accs
                                       else Sys clts accs

removeAcc :: AccId -> System -> System
removeAcc acid (Sys clts accs) =
    if (existsAcc acid (Sys clts accs)) then Sys clts (filter ((acid /=) . accid) accs)
                                        else Sys clts accs

updateClt :: (CltId, CltName) -> System -> System
updateClt (cid, cname) sys =
    if (existsClt cid sys) then insertClt (cid, cname) . removeClt cid $ sys
                           else insertClt (cid, cname) sys

updateAcc :: (AccId, Amount) -> System -> System
updateAcc (acid, amount) sys =
    if (existsAcc acid sys) then insertAcc (acid, amount) . removeAcc acid $ sys
                            else insertAcc (acid, amount) sys

getCltName :: CltId -> System -> Maybe CltName
getCltName cid sys = case findClt cid sys of
```

```
                        Just clt -> Just . name $ clt
                        Nothing  -> Nothing

getAccAmount :: AccId -> System -> Maybe Amount
getAccAmount acid sys = case findAcc acid sys of
                           Just acc -> Just . amount $ acc
                           Nothing  -> Nothing
```