Proceedings

1st International Workshop on Software Evolution Transformations

SET 2004

November 9, 2004
Delft, the Netherlands

Held in conjunction with
The 11th Working Conference on Reverse Engineering (WCRE)

Sponsored by
Queen's University, Kingston, Ontario, Canada

Proceedings

1st International Workshop on Software Evolution Transformations (SET 2004)

November 9, 2004
Delft, the Netherlands

Edited by
Ying Zou
James R. Cordy

Sponsored By
Department of Electrical & Computer Engineering
School of Computing
Queen’s University, Kingston, Ontario, Canada

Held in conjunction with
The 11th Working Conference on Reverse Engineering
(WCRE 2004)

Message From Workshop Chairs

SET 2004

Welcome to SET 2004, the first international workshop on Software Evolution Transformations. SET 2004 is intended to bring together researchers and practitioners to investigate state-of-the-art techniques, tools and methodologies to establish engineered source code transformation processes, to capture reusable transformation patterns for continuous code evolution, and to consider how to incorporate novel code transformation techniques from research and academia into industrial practice. We expect the presentations and discussions will identify challenges, develop ideas and provide approaches to help maintain legacy systems at satisfactory levels using code transformation techniques.

We received eighteen submissions from nine countries. After a review process including reviews and recommendations by at least two members of the program committee for each paper, eleven excellent papers were chosen for publication. These selected papers have been chosen for presentation to spur discussion of key concepts.

We wish to thank the program committee and all the authors who submitted position papers for the workshop. We would like to thank our key sponsors: the Department of Electrical and Computer Engineering and the School of Computing at Queen's University at Kingston, Ontario, Canada, for their support of the Workshop. Finally, we would like to thank the WCRE 2004 organizing committee for their effort and support in making this workshop possible.

Ying Zou and James R. Cordy
SET 2004 Workshop Chairs

Program Committee

SET 2004

Contents

International Workshop on Software Evolution Transformations

SET 2004

Session 1: Refactoring Process

Enabling Dynamic Software Evolution through Automatic Refactoring

Peter Ebraert
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
Email: pebraert@vub.ac.be

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
Email: tjdhondt@vub.ac.be

Tom Mens
University of Mons-Hainaut
Avenue du Champ de Mars 6
B-7000 Mons, Belgium
Email:tom.mens@umh.ac.be

Abstract— **Many software systems must always stay operational, and cannot be shutdown in order to adapt them to new requirements. For such systems, dynamic software evolution techniques are needed. In this paper we show how we can exploit automated refactorings to improve a software the component structure of a software system while the system is running in order to facilitate future evolutions. We report on some experiments we performed in Smalltalk to achieve this goal.**

I. Introduction

People always say that you should never change a system that is working ne. However, even if a software system seems to work properly from a user's point of view, it may be dif cult to maintain or adapt from a developer's point of view. As such, it may be very cumbersome to evolve the system by adding a new feature, xing a bug or porting the system to a new environment.

In all these situations where a software system is not exible enough to allow for a certain change, the technique of *software refactoring* can be used. According to Fowler [1], a refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." Refactorings can be used to simplify the structure of a software system in order to prepare it for a certain evolution step.

Now suppose we have a running system, and we would like to evolve it without shutting it down. This is a much bigger challenge since there are considerably more constraints on the running system. Refactoring techniques would be very useful here too. For example, by reducing the coupling between components in a running system, we could at the same time increase the system performance (from a user point of view) and its understandibility and evolvability (from a developer point of view).

Until now, refactorings have only been investigated in the context of source code restructuring. The main contribution of this paper is to show the use and feasibility of applying *dynamic refactorings*, i.e., refactorings that modify a running system.

II. Experimental Setup

In order to be able to apply refactorings to a running system, we need to be able to dynamically modify the structure and behavior of the software that is being executed. To this extent we need a software system that has full re ective capabilities. According to Pattie Maes [2], re ection is the ability of a program to manipulate as data, something that is representing the state of the program during its own execution.

A re ective system is able to reason about itself by the use of *metacomputations* – computations about computations. For permitting that, such a system is composed out of two levels: the *base level*, housing the base computations and the *metalevel*, housing the metacomputations. Both levels are said to be *causally connected*. This means that, from the base level point of view, the application has access to its representation at the metalevel and that, from the metalevel point of view, a change of the representation will affect ulterior base computations.

Because the focus of this paper is on refactorings, we restrict ourselves to class-based object-oriented languages. Object-oriented languages like Java are excluded because of the limitations of their re ective capabilities. Smalltalk, on the other hand, is fully re ective: everything is an object, and can thus be taken apart, queried for information and possibly be modi ed. Even messages are objects, and can thus be monitored and modi ed when they are sent or received [3].

III. Proposed solution

A. Terminology

In order to provide a sound basis for starting a discussion on how we plan to do dynamic evolution, we rst need to establish a common vocabulary that will be used throughout this paper.

A **software system** is assumed to consist of a set of processing components with directed connections indicating the communication that occurs between the components.

A **component** is a processing entity that can request and provide services. In class-based OO languages, components typically consist of interrelated classes, which are communicating through message passing.

A **connection** is a directed communication from one component - the *initiator* of the communication to another component - the *recipient*. This connection contains all important messages sent between the classes contained in the initiator component and the recipient component. Connections indicate

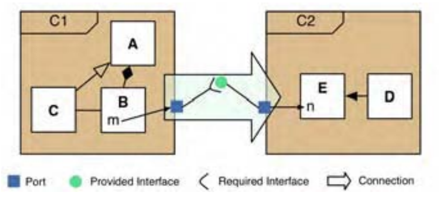


Fig. 1. A base level application architecture

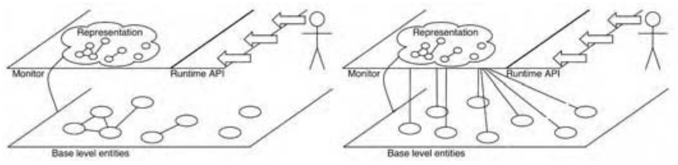


Fig. 2. Runtime evolvability by means of a two layered architecture: inter-component communications (left) are indirected to the monitor (right).

that the component might initiate transactions.

A **transaction** is an exchange of information, by means of message sends, between two and only two components, initiated by one of the components. Transactions are the means by which the state of a component is affected by other connected components in the system. Transactions consist of a sequence of one or more message exchanges between the two connected components. It is assumed that transactions complete in bounded time and that the initiator of a transaction is aware of its completion. The completion of transactions at the initiator is required to ensure correct termination of the change management protocol that will be described later.

Figure 1 introduces the notation that we use to show what the structure of a base-level software system looks like. In this notation, the system is represented as a directed graph of components and connections. The big edge in the directed graph represent a connection between two components.

B. Detecting the dynamic component structure

Most of the time, we do not know the dynamic component structure of an application described as a directed graph of nodes (components) and edges (connections), but instead we only have the source code of the application. The dynamic component structure re ects the way objects need to be grouped into components based on their communication patterns. To increase cohesion and reduce coupling between components, objects that exchange many messages between one another should be clustered into the same component.

In order to detect which objects can be clustered together, static source code analysis does not suf ce. First of all, static analysis only provides an approximation of the possible runtime behaviour. Second, since we reason about object-oriented code, we need to take into account dynamic information such as late binding and polymorphism. Third, since Smalltalk is a dynamically typed programming language, we need to do at least some amount of dynamic type inferencing.

Therefore, we decided to resort to a dynamic analysis of the program. We use statistical information that we obtain by monitoring execution traces of the application at runtime. This can be achieved by using a two-level architecture. While the application is running normally at the base level, a monitoring application will be located at the metalevel. Such applications already exist [4], but do not meet all our needs. That is why we developed our own monitoring application that collects information on running applications. That information is used to cluster classes that are very related together into components. The messages that go from a class in a certain cluster to a class in another cluster, will be captured in connections.

C. The dynamic evolution framework

In [5], [6] we presented a rst version of an evolution framework [7]. It shows how we apply a two-layered architecture to allow the modi cation of the behavior of running applications. For doing so, we instrument the base-level application with calls to a metalevel monitor at every point where communication between components occurs. During execution, the monitor passes control to the concerned components (following the metalevel representation of the application), making its presence unnoticeable. This is illustrated in gure 2.

In order to evolve the application, the user has to change the application's representation in the monitor. To that extent, a runtime API is included so that on-line interaction with the monitor becomes possible. The functionalities of the API include the addition, the removal and the modi cation of system components, and their relations. Each of these operations have their pre-conditions, which have to hold before the operation is actually carried out by the framework [8]. When modifying an existing component, problems could rise concerning state consistency [9].

D. Dynamic refactoring and evolution

1) Detecting possible refactorings: In order to decide which part of the code needs to be refactored to facilitate future evolutions, we use the information obtained from the monitoring application to verify at regular time intervals whether the current component structure should be modi ed. It this happens to be the case, the proper refactorings will be suggested to be applied with our without user interaction.

2) Change Management Protocol: An evolution or refactoring typically replaces a set of components C_1, C_2, C_3, ... by their new versions C'_1, C'_2, C'_3,... We use the notation ΔC_i to denote the difference between C_i and C'_i. In this section, we rst de ne a set of atomic steps - *change transactions* - that can be used to compute ΔC in an automated way.

In object-oriented programs, components typically consist of some related objects, that on their turn consist of instance variables and methods. Most of the relations between the objects are caught in the methods and instance variables. This is why our meta-object protocol currently implements the following set of atomic change transactions. (In the future we intend to extend this set to cover a more realistic set of applications.)

chaName Changes the name of a class.

addMethod Adds a method to a class. As a result, all objects that are instance of this class will automatically understand this new method thanks to Smalltalk's method lookup mechanism.

remMethod Removes a method from a class. As a result, all instances of this class may no longer understand this method. Hence, one should be very careful with this operation as it can give rise to runtime exceptions. We will explain how to deal with this later in this section.

chaMethod Modi es the implementation of a method in a class. Again, this will have an impact on all objects that are instance of this class or one of its subclasses.

addInstVar Adds an instance variable to a class. As a result, all objects that are instance of this class have a new variable they can use to store values. By default, the value will be set to `nil`.

remInstVar Remove an instance variable from a class.

chaSuper Changes the parent of a class. This will change the inheritance hierarchy, and as a result the methods that any object of this class, or of any of its children, can understand.

deactivate Deactivates a component – all the classes and instances it contains – to make sure that no transaction will occur in the component. When a component is deactivated, all transactions are put in a waiting list.

activate Allows the component – all the classes and instances it contains – to resume its execution. All waiting transactions will be processed upon activation.

By monitoring the users' actions when evolving on of ine copy of C to C', we can automatically obtain ΔC. Afterwards, the operations that need to be performed, in order to comply with possible preconditions of certain actions, are inserted in order to obtain the *change transaction sequence*; the sequence of all atomic change steps. That sequence is then used to evolve the online C to C'.

The main bene ts of this approach are the preservation of the state and object identity, as we will keep on working on the same (already existing) component C. Replacing an entity C would involve the creation of C', the swapping of all relations from C to C', the deletion of C and the mapping of the state from C to C'. Evolving the existing C component to C' only involves the creation of C' and the propagation of the changes on C. This implies that there will be no more relation swapping problems and less state mapping problems.

A second bene t is the possibility to test the new version of the component of ine. In order to validate C', we need to perform some tests. First we need to perform *component testing* (testing the internal behavior of the component), and then we need to do *system testing* (testing the external behavior of the component - its relation with the other components).

Every one of the atomic change transactions has some speci c requirements that need to comply before the transaction can be carried out. For instance, we cannot remove a certain method m that is still called somewhere. So before removing m, the methods in which m is called, must be modi ed so that they do not call m anymore. For including these preconditions in the rules, we have to introduce states. A method or instance variable is in a *removable* state if it is not called anymore. For modifying methods or instance variables, we need to make sure that the components that use them, are deactivated, so that no inconsistencies are introduced while changing the component. So we can say that instance variables or methods are in a *modi able* state, if they cannot be called while the evolution is occurring. Adding a method or instance variable does not have any precondition, so we say that the precondition for adding always holds true.

When the backtracking algorithm nishes, it outputs the best set of atomic transactions that can be followed in order to bring C to C'. Note that it is guaranteed that a valid path will be found as C and C' are both composed of objects that are in their turn composed of methods and instance variables. In the worst case, all methods and instance variables of all objects in C can be removed, and the methods and instance variables of all objects in C' then added. From the moment the best path is found, the change transaction set is established.

3) Managing consistency: Informally, we can say that a consistent application state is a state from which the system can continue processing normally rather than progressing towards an error state. A system is viewed as moving from one consistent state to the next, as the transaction processing continues. In fact, application transactions modify the state of the application, and, while in progress, have transient state distributed in the system. While transactions are in progress, the internal states of nodes may be mutually inconsistent. So we should also avoid the loss of application transactions and make sure that we achieve a consistent state after carrying through a change. This consistent state requires that there is no communication in progress between the affected components nor with their environment.

For making sure that no communication is occurring while a certain component is being modi ed, we introduce the concept op *deactivated* components. Such components will queue all incoming transaction requests, and carry them out whenever they get *activated*. The evolution framework that we proposed in section III-C, offers the functionality of activating and deactivating system components. Note that the current implementation goes out from an asynchronous messaging system, and that transactions that are queued, will not proceed. Such an approach will make sure that consistency is preserved, but will also make the entire application stop, unless it is developed in a distributed way (with multiple threads). In class-based systems, deactivating a component means that we will deactivate all its class and all living instances of that class and bring them in a modi able state.

4) Carrying out refactorings and evolutions: From the moment we have the change transaction sequence, we can start implementing these changes on the running system. In this phase, the atomic changes of the transaction set will be implemented in the system one by one. While most of them can be done transparently, some may require the programmer's interference. This is the case when there is a state involved, that needs to be preserved. Concretely, when an instance variable is deleted or modi ed, the question arises what has to

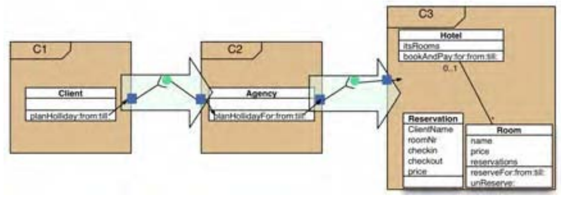


Fig. 3. The original architecture of the travel example; providing 1 interface.

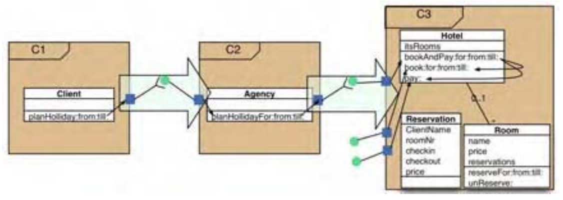


Fig. 4. The adapted architecture of the travel example; providing 3 interfaces.

happen with the value of that instance variable. Either the value can be ignored, or its is needed later in a new instance variable that will be added. Consequently, when an instance variable is added, the programmer is also requested to interfere, and to tell wether the variable should be initialized with a certain value. For example, using Euros instead of Belgian Francs in our bank accounts requires us to use the following formula: 'take the old value and multiply it by 40,3399, and use it as the new value'.

For methods in class-based systems, things are much simpler. Because methods are only referenced through the class itself, adapting them on the class level does the job.

IV. Example

We validate our approach by working out the simple example that we shortly introduced above. In that example we have a software system with clients, travel agencies and hotels, as the basic components (gure 3). When a certain client wants to go on a holiday, he contacts an agency he knows. That agency can then book a room in a hotel, by using the `bookAndPay:for:from:till:` interface that is provided by the Hotel component. This service makes sure a room is booked and payed for.

Imagine that, at a certain time, we might want to allow reservations that are not payed at the same time than making reservations. For adding that functionality to the system, we need to split up the `bookAndPay:for:from:till:` service in two different services: `book` and `pay`. For not affecting existing agencies, that still might want to use the `bookAndPay:for:from:till:` service, that service must still be provided by the Hotel component. For not having code duplication, that service will invoke the book and pay service in its time, as we can see in gure 4.

Place	Atomic change	Parameters
C3	deactivate	
C3> $Hotel$	chaMethod	"bookAndPay:for:from:till:"[meth.Body]
C3> $Hotel$	addMethod	"book:for:from:till" [meth.Body]
C3> $Hotel$	addMethod	"pay:" [met.Body]
C3	activate	

TABLE I
THE ATOMIC CHANGE SET FOR METHOD EXTRACTION

Table I shows the atomic changes that are needed for performing this evolution. This speci cation is passed to the Evolution Framework, that handles all those steps, and thus carries out the evolution. As, in this case, there is no state modi cation involved, the user won't have to interact with the system anymore and the evolution gets carried out totally automatic. The resulting architecture can be seen in gure 4.

V. Conclusion

In some cases, software systems can not be turned off for carrying out an evolution. This triggers the need for a framework that supports dynamic evolution. We suggested to apply dynamic refactorings to improve the runtime component structure of object-oriented software systems. The approach relies on the re ective properties of the underlying programming language in order to modify the application's behavior.

Our framework uses a monitor at meta-level that keeps track of the base-level application. This monitor detects the dynamic architecture of the application by collecting statistical information about the messages sent between objects. This information is used to determine the dynamic software architecture in terms of components and connections. Whenever this structure changes, dynamic refactorings can be triggered to evolve the component structure at runtime.

Currently, we are in the process of implementing the above framework in Smalltalk. The framework allows the addition, removal, or modi cation of base-level components while the system is running. The framework provides facilities to establish atomic change sequences, manage consistency, preserve object identity and state, and apply dynamic refactorings and evolutions.

References

[1] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[2] P. Maes, "Computational re ection," Ph.D. dissertation, Arti cial Intelligence Laboratory, Vrije Universiteit Brussel, 1987.
[3] S. L. Messick and K. L. Beck, "Active variables in smalltalk-80," in *Technical Report CR-85-09*. Computer Research Lab, Tektronix, 1985.
[4] R. Wuyts, "Smallbrother - the big brother for smalltalk," Université Libre de Bruxelles, Tech. Rep., 2000. [Online]. Available: http://homepages.ulb.ac.be/~rowuyts/SmallBrother/index.html
[5] P. Ebraert and E. Tanter, "A concern-based approach to dynamic software evolution," in *The proceedings of the Dynamic Aspects Workshop in conjunction with AOSD*, Lancaster, UK, march 2004.
[6] P. Ebraert and T. Tourwe, "A re ective approach to dynamic software evolution," in *The proceedings of the Workshop on Re ection, AOP and Meta-Data for Software Evolution in conjunction with ECOOP*, Oslo, Norway, June 2004.
[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer Society*, vol. 37, no. 10, pp. 46–54, october 2004.
[8] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, November 1990.
[9] Y. Vandewoude and Y. Berbers, "Fresco: Flexible and reliable evolution system for components," in *Electronic Notes in Theoretical Computer Science*, 2004.

Search based structure improvement

Olaf Seng and Gert Pache
FZI Forschungszentrum Informatik, 76131 Karlsruhe, Germany
{seng,pache}@fzi.de

Abstract

A software system's structure degrades increasingly during its lifetime and therefore it gets harder and harder to maintain. As a result the structure needs to be reconditioned from time to time. The problem is to change the structure at the appropriate places, since an improvement at one spot might lead to a degradation at another. This paper describes a new methodology that, starting from an initial structure, computes a behaviorally equivalent structure, having better metric and heuristic values being used to assess the structure's quality. This new structure can then be used to derive the refactorings needed to transform the existing source code. The main idea is to treat the problem as a search problem and to solve it using a genetic algorithm.

1 Introduction

The structure of a software system inevitably degrades during its life cycle, because the system and therefore its structure has to be adapted to new and changing requirements, that are unforeseen and have to be implemented within a tight time frame. Since the structure has a major impact on the costs of system maintenance, it has to be reconditioned from time to time.

The difficult task is to change the structure at the appropriate places, since an improvement at one spot might lead to a degradation at another. In order to change the structure, existing behaviorally preserving refactorings [7] can be used. To determine whether a change has been successful the quality of both structures can be measured using known metrics and heuristics [3], like e.g. low coupling between subsystems or as few god classes[3] as possible in a system.

Consider for example the structure shown in figure 1. Placing *Class1* and *Class2* into *Subsystem1* and *Class3* and *Class4* into *Subsystem2* would certainly lead to reasonable coupling and cohesion values, however it would also introduce a direct cyclic dependency between the two subsystems.

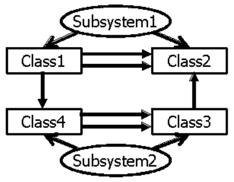


Figure 1. Example structure

The goal of this work is to develop a methodology for object-oriented systems that, starting from an existing system structure, determines a structure with better metric values and fewer violations of good object oriented design principles.

Such a methodology must at first be *global*: the overall structure has to be improved, since a local improvement can lead to a degradation at several other places. This also means that several metrics and heuristics should be considered. At second, it should be *semi-automatic:* User input should be requested only before and after the procedure is applied. And at third, it has to be *practical*, which means that the externally observable behavior must not be changed and it needs to be able to process real life systems.

2 Approach

Since the used metrics and heuristics can have a contradicting influence on each other, the general idea is to consider the task as a search problem. Metrics and heuristics have to be formulated as a cost function and well established techniques like genetic algorithms or hill-climbing can be used to find a solution. In this case the search space consists of a number of behaviorally equivalent systems. Since real life systems consist of thousands of structural elements and a large number of behavior preserving refactorings [7] are known, it is quite clear, that the search space is very large. Therefore for the moment we limit ourselves to the following three types of refactorings, thus drastically reducing the search space: *move element* (e.g. moving a class from one subsystem to another one), *split up element* (e.g. splitting up a class) and *merge existing elements* (e.g creating a new class by combining two existing classes). Of course an element can be deleted, if all its members have been moved to other elements.

Even though the number of possible types of refactorings is limited to three atomic ones, they can have an impact on quite a large number of metrics and heuristics for subsystems and classes. If a *desired architecture* is given, e.g. a layered architecture, and there are several violations, the approach can try to determine another system that complies to the given architecture by moving classes around. Applying these refactorings can improve *coupling*, *cohesion* and *complexity* metrics. Coupling values between subsystems or classes should be as low as possible [6]. Cohesion inside subsystems or classes should be as high as possible. Heuristics like e.g. god class or data class[11], address the problems that entities contain too many or too few elements or are too complex with respect to e.g. control flow. Examples for other heuristics are *Cyclic dependencies* and *Bottlenecks*. Cyclic dependencies between two or more entities are typically undesired. Bottlenecks are entities, which know about and are known by too many other entities [3].

Further metrics and heuristics can be found in [11]. All theses metrics and heuristics are related to size and placement of elements. We consider this as a two phased process. The first phase deals with proper placement and size of elements making the system more understandable, while the second phase tackles the problem of improving inheritance hierarchies. For solving this optimization problem, techniques such as hill-climbing can be used. However, our preferred optimization technique is a genetic algorithm, since it allows a more efficient traversal of the search space and makes it easier to avoid local extrema.

2.1 Overview

Figure 2 shows the general steps of the approach using a genetic algorithm. Optimization is not done on source code directly, but on an abstract structure model, on which the refactorings can be simulated and the fitness function can be calculated. During a precalculation step, the existing structural elements need to be classified, since not all elements of one type can be treated in the same manner. Utility subsystems for example do not need to have low coupling values to other subsystems, because they provide functionality, that should be used elsewhere.

Several potential solutions are considered at a time. In this case the existing structure model is copied n times. With the help of the genetic operators mutation and crossover, the n solutions are randomly modified or new solutions are created by taking some parts of two existing solutions. Using the fitness function and a tournament selection strategy [2] the most promising n elements are selected. Before the algorithm starts, the user can define the fitness

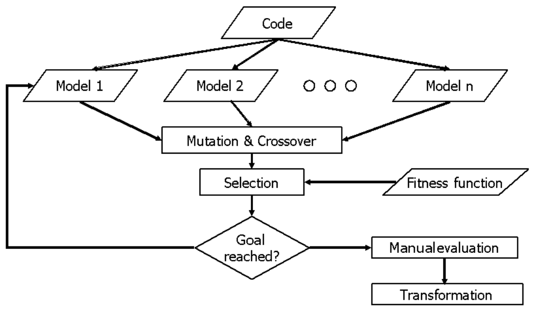


Figure 2. Algorithm

function manually, by selecting several metrics or heuristics as well as by changing thresholds. The optimization goal is reached after a certain number of evolution steps. Then the new structure is presented to the user, who can carry out the actual transformations either manually or by means of a tool. The list of refactorings to be done can be easily back traced by comparing the initial structure of the system with its final structure and finding out which elements have been moved, created or deleted. Since optimizing a whole system at once might be a little bit confusing for the developers, our approach is designed to be carried out in two steps. In the first step, one should improve the subsystem design and in the second step class design inside one ore more subsystems can be optimized. It might be useful to carry out step one again, in order to benefit from the better class design.

2.2 Subsystems

The usually needed pre- and postcondition checks can be left out when optimizing subsystem design. All possibly occurring violations are visibility problems or name clashes, that can be resolved by the user afterwards. However, it is important to come up with a proper model and representation, a fitness function and suitable operators for the genetic algorithm.

Our model is a graph with directed edges, whose nodes can be either subsystems or classes. Elements inside classes like methods are collapsed into the class they belong to. The relations between those elements e.g. method calls are moved up to relations between the classes. All relations leading to dependencies between classes have to be modeled, e.g. method calls or attribute declarations. In our representation genes are sequences of subsystems and therefore they can have different lengths [2]. The mapping between subsystems and classes is stored separately. Since most of our operators work on subsystems, we preferred this representation instead of the conventional one, in which genes are sequences of classes.

Our mutation operators are: *move a class from one subsystem to another*, *split a subsystem into two different sub-*

systems, *merge two different subsystems into a subsystem* and *delete an empty subsystem*. These mutations are not applied completely randomly. The current metrics and heuristic values are taken into account. For example, classes are not moved to a subsystem, that is completely independent from the current one. The advantage of this limited randomness is, that a better solution can be found faster, since only promising mutations are chosen. Our crossover operator tries to conserve as many subsystems as possible [2]. At first, whole subsystems inherited from two parents *one* and *two* are copied to a new child. The subsystems taken from parent *one* which contain classes belonging to more than one subsystem are removed. In a second step all classes not belonging to a subsystem need to be assigned to one of the existing subsystems.

Our fitness function is a so called multi modal fitness function. Each of the individual functions calculates a value between one and zero.

$$fitness \begin{cases} cohesion & : \dfrac{\sum_{i=1}^{\#s} \sum_{j=1}^{\#c(s_i)} \frac{\#k(c_j)}{\#c(S_i)^2}}{\#s} \\ coupling & : \sum_{i=1}^{\#s} \left(\left(\frac{\#rO(s_i)}{\#r(s_i)} \right) * \frac{\#r(s_i)}{\#r} \right) \\ complexity & : \sum_{i=1}^{\#s} \left(com(s_i) * \frac{\#c(s_i)}{\#c} \right) \\ cycles & : \dfrac{\sum_{i=1}^{n} size(scc[i])^k}{\#s^k} \\ bottlenecks & : \sum_{i=1}^{\#s} \frac{min(inDeg(s_i), outDeg(s_i))}{maxDeg} \end{cases}$$

Currently we are using standard coupling and cohesion metrics as parts of our fitness function [1]. To measure cohesion for a system s, we sum up for each subsystem s_i and for each class c_j inside s_i the number of different classes inside s_i known by c_j ($\#k(c_j)$) and divide this by the square of the number of classes in s_i ($\#c(s_i)$). This value is then normalized by dividing the overall sum by the number of subsystems (#s).

Our coupling value for a subsystem s_i is calculated by counting the number of relations going to other subsystems s_j ($\#rO(s_i)$) divided by the overall number of relations s_i is involved in ($\#r(s_i)$). This number is weighted by multiplying it with $\#r(s_i)$ divided by the overall number of relations (#r).

The complexity function requires four parameters. Complexity is considered to be optimal (1) if it is inside the interval $[minO, maxO]$. The value is linearly interpolated between zero and one inside the intervals $[minU, minO]$ and $[maxO, maxU]$. Figure 3 shows how such a fitness function looks like. Size can be measured using for example number of classes or control flow complexity.

The fitness value for cycles is calculated by summing up the size of strongly connected components. To give higher

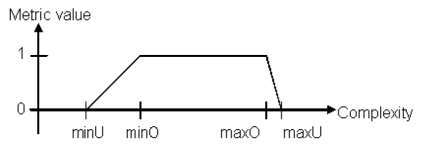


Figure 3. Shape of complexity fitness functions

penalties to cycles with more than two entities involved, we usually use a factor $k \geq 1$. To normalize the fitness value, the sum is divided by the number of subsystems. For calculating the bottleneck metric for a subsystem we measure the in-degree and out-degree and divide the minimum of the two by the highest in- or out-degree (*maxDegree*) value currently found.

2.3 Classes

The goal of this next step is to improve the class design by moving attributes and methods around and splitting or merging classes. Behavior preservation is more difficult to guaranty in this case, since changing a class can change the objects created at runtime.

When moving methods around the biggest challenge is updating the call sites, since one needs a reference to an object of the type the method has been moved to. Adapting this so called access-chains is usually considered a manual task. But in our case, since we want to simulate multiple movements one after the other, we have to make sure, that a new reference can be calculated. Since checking pre- and postcondition is language specific, we started creating a methodology for Java and plan to extend it later.

During a first step, we select the methods and attributes that can be moved around. E.g. polymorphic methods, that implement or override a method of a superclass cannot be moved. Methods that can be moved around easily are e.g. static methods. The model used here is an extension of the model used for subsystem optimization. New model elements are e.g. parameters, local variables and of course the access chains to attribute accesses and method calls.

Currently, we are using for types of transformations: We can move static attributes and methods. Methods can be moved to a class, when the method has a parameter of the target class, or the containing class has an attribute whose type is the target class. These restrictions allow us to adapt access-chains easily. Attributes can be moved, when the containing class and the target class have a one to one relationship between corresponding objects at runtime. Classes can be split and merged if their objects have a one to one relationship at runtime.

Our fitness function for this step has not been completely defined yet, but of course we intend to use coupling, cohesion and size metrics similar to the metrics used for subsystem optimization. Additionally we plan to use heuristics like *god class, data class, superclass knows subclass* [11].

3 First experiments

We have developed a first prototype that can be used to improve the partition of subsystems. At the moment, we are able to process systems written in Java and C++. Our first case study is an application developed in-house, consisting of about 80000 LOC, 200 classes and 3000 methods. Development started about seven years ago, and until recently, no explicit subsystem boundaries have been defined. But as the system grew, we wanted to come up with a proper modularization of the system.

Using our approach, that takes into account - in addition to coupling, cohesion and complexity - cycles and bottlenecks, we received a satisfying modularization. Average subsystem size was about 8 classes, there were no cycles between them, only a few bottlenecks and acceptable coupling and cohesion values. One important observation is, that we found results with similar coupling and cohesion values, but different number of cycles and bottlenecks. Concerning runtime, one execution of the optimization process with a population size of 100 and 250 evolution steps took about eight minutes on a Pentium 4 at 1,4 Ghz with 512 MB of RAM.

4 Related Work

The idea to generally treat software engineering as a search problem is described in [4]. Our work focuses on the aspect of optimizing the structure of a system.

In the area of design improvement there are several people dealing with the modularization of software systems [9] [1][8]. Some of them use genetic optimization techniques as well, but their fitness function is usually limited to coupling, cohesion and complexity. They only try to improve the modularization into subsystems and do not consider class design.

An approach using a similar technique to ours can be found in [10]. However the goal of this approach is a different one. The authors want to improve the design of inheritance hierarchies. As far as we can tell, they do not consider pre- and postconditions for their refactorings.

The approach described in [12] tries to improve the system structure as well, but only locally for one bad structure at a time.

Another approach for improving subsystem structures can be found in [5]. User input is necessary while applying the procedure and not like we demanded only beforehand and afterwards.

5 Conclusion and next steps

The goal of the methodology in this paper is to find a behaviorally equivalent structure for a given software system, which is better with respect to the used metrics and heuristics. The idea is to treat the task as an optimization problem, where metrics and heuristics are formulated as a fitness function.

One of our next steps is to develop a complete fitness function for evaluating class designs. The method itself has to prove its usefulness in further case-studies, for improving subsystem and class design. To make the approach even more powerful the question of modifying inheritance relationships has to be addressed, since many heuristics assess the quality of inheritance hierarchies.

References

[1] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. In *CSMR 2004*, 2004.

[2] M. Biehl. Zerlegung von Softwaresystemen mit genetischen Algorithmen, 2004.

[3] O. Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Technology of Object-Oriented Languages and Systems - TOOLS 30*, 1999.

[4] J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. S. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *Journal of IEE Proceedings - Software*, pages 161–175, 2003.

[5] H. Fahmy, R. Holt, and J. Cordy. Wins and Losses of Algebraic Transformations of Software Architectures. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, 2001.

[6] N. E. Fenton and S. L. Pfeeger. *Software Metrics A Rigorous and Practical Approach*. PWS Publishing, 1997.

[7] M. Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, 1999.

[8] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for searchbased optimization of software modularization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002.

[9] B. Mitchell. *A Heuristic Search Approach to Solving the Software Clustering Problem*. PhD thesis, Drexel University, Philadelphia, 2002.

[10] M. O'Keeffe and M. O. Cinneide. Towards automated design improvement through combinatorial optimisation. In *Proceedings of the Workshop on Directions in Software Engineering Environments*, 2004.

[11] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[12] A. Trifu, O. Seng, and T. Genssler. Automated design flaw correction in object-oriented systems. In *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR 2004)*. IEEE Computer Society, 2004.

Document-Oriented Source Code Transformation using XML

Michael L. Collard, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
collard@cs.kent.edu, jmaletic@cs.kent.edu

Abstract

The paper takes a document-oriented view of source-code transformation and describes how an underlying XML representation for source code can be used to support refactorings. The method preserves all documentary structure of the source and is applied in a straightforward manner.

1. Introduction

We view the transformation of source code, for the purpose of software evolution, as a heavily document-oriented activity. Transformations, such as refactoring, are a mapping from source-code documents to source-code documents. Automating these types of transformations is integral to supporting the evolutionary process since this can be a time consuming and error prone task if done manually. However, the final result of the transformation must be in the form readable by the developer for it to be useful for continued evolution. A successful software-evolution transformation must easily support all aspects of the document including its lexical, structural, syntactic, and documentary nature.

The transformation must preserve the programmer's view of the document. Not doing so may mean the rejection of the system, as described in a number of studies on real world projects [4, 8]. In [4] examples are given on large projects where any potential changes to the system had to be presented to the programmers in the exact view of the source code that they were familiar with. If not, the proposed changes were rejected.

In [8] the concept of the *documentary structure* of source code, whose elements include all white space and comments, is presented. It is described as what a programmer places in the source code for the sole purpose of assisting whoever is reading the program. Examples given show that white space, such as line breaks and indentation can be more important than comments and that the notion of a single comment is not well defined.

This documentary structure is often at odds with the linguistic structure of the program. Unfortunately for many parse-tree-based approaches, this documentary structure is completely lost. Attempts to preserve these ties often result in the documentary structure not being easily integrated back into the representation. There are exceptions to this problem however and one notable example is the DMS systems by Baxter [1]. The DMS project has gone to great lengths to address this specific issue by storing important textual items within the underlying abstract-syntax graph.

Any transformation approach for software evolution must preserve documentary structure and must allow it to be a first-class part of the document. A successful analysis and transformation system will allow a combination of changes to any of the text, whether or not it is part of the formal syntactic structure.

In contrast to these requirements, software-development tools typically take a totally compiler-centric approach of representing the source code as a syntax tree according to the formal linguistic structure of the program. It has been observed that these compiler-centric approaches are often not a good match to the problems that they are trying to solve [6, 8].

Our approach takes a very document-oriented XML approach to the transformation of source code. We developed a robust parser to markup C++ source code in a lightweight representation. This allows us to leverage XML technologies for tasks such as fact extraction and transformation.

In order to realize document-oriented source-code transformation, a sophisticated underlying infrastructure is necessary. As such, the approach is built on top of an XML representation of the source code developed to support program-analysis tasks, namely srcML [2, 3, 7]. This representation explicitly embeds high-level syntactic information within the source code in such a way that it does not interfere with the textual/documentary context of the source code. The representation is unique in that it preserves the programmer’s view of the source code while at the same time explicitly adding parts of the

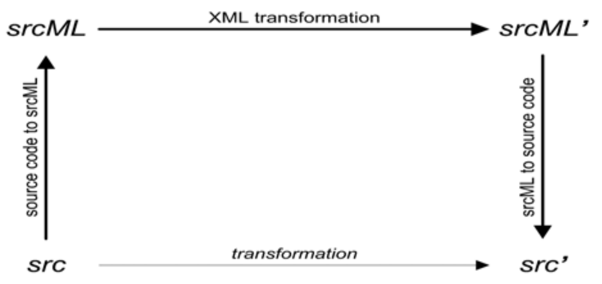


Figure 1. With srcML source-code transformations, such as refactoring, are raised to the level of XML transformations. The source code is translated into equivalent srcML, the srcML undergoes XML transformation, and the resulting srcML is translated back to source code.

abstract syntax to the source. srcML directly supports such tasks as lightweight fact extraction, source-code transformations, and the integration of source models (e.g., call graphs) using XML tools and standards. It also supports the embedding of meta-information into the source (e.g., hyperlinks).

We will describe our underlying approach and demonstrate via a simple example how it can be used to transform source code (i.e., refactor) in a document and syntactic preserving manner. Lastly we discuss our future plans and direction for this research.

2. Transformation of Source Code

srcML is an XML representation of the complete contents of a source-code file. All text (including white space) from the original source-code file is preserved in its original ordering. XML meta-characters, including '<', '>', and '&' are stored in an escaped form, i.e., '&lt;', '&gt;', and '&amp;' respectively. A special representation is used for the form feed character that is not able to be directly represented or escaped in XML. Per the XML standard, line ends in the source-code file are normalized to a single character. Generating the original contents of the source-code file from the srcML representation is straightforward.

The XML elements of srcML surround the text that they describe including syntactic structures e.g., elements *class*, *function*, *if*, *while*, etc., documentary structure, e.g., element *comment*, and preprocessor statements, e.g., element *cpp:directive*. The srcML markup stops at the expression level with elements for identifiers (element *name*) and function calls (element *call*).

The srcML representation is relatively compact with a 5 to 7 times increase in file size over the source-code document. A robust source code to srcML translator exists with the ability to translate at over 7500 LOC per second. The translator currently handles the C/C++ languages. With srcML, XML transformations can be used to perform non-intrusive transformations on source code. Transformation can be performed on any selected element of the syntactic or documentary structure, while simultaneously preserving all other elements.

The transformation process is shown in Figure 1. The source-code document, *src*, is converted by the srcML translator into an equivalent srcML document, *srcML*. XML transformations of the srcML document convert *srcML* to *srcML'*. Afterwards, the transformed srcML document, *srcML'*, is translated back to a source-code document, *src'*.

We carefully define srcML so that srcML documents can be transformed using any desired XML transformation approach, e.g., SAX, DOM, XSLT. This is different than a *data-oriented* approach (e.g., GXL) that has little or no connection to the original document.

A srcML transformation can be performed non-intrusively. All elements of the source-code document, including documentary structure, such as white space, preprocessor directives, etc., can be passed unaltered through the transformation. An identity XML transformation at the srcML level, i.e., *srcML* is identical to *srcML*', is equivalent to an identity transformation at the textual level, i.e., *src* is identical to *src*' (with the possible exception of a change in end-of-line character). Only the source-code elements that require changes are altered in any way. Of special note is the preservation of white space. Some XML processors consider white space in elements insignificant and by default normalizes them. However, our experience has found that careful use of these tools allows the process to preserve these elements.

A specific transformation of current interest is refactoring, a source-to-source transformation that preserves program behavior. The purpose of refactoring is to improve the overall structure of the code so that it can be more easily extended, repaired, and to increase comprehension.

Current refactoring tools have difficulty with the preservation of documentary structure, and in languages such as C++, only a limited number of refactorings from the catalog are supported with automation. In addition, tools that can perform refactorings are very carefully constructed for specific refactorings. Most tools cannot be easily enhanced or extended, and they do not provide a general purpose format and/or language that can be used by a programmer to adapt them to their own use, or to write additional transformations [8].

The rest of this section will demonstrate a non-intrusive refactoring of source code via using an XML transformation. The example shown is a refactoring from Fowler's [5] titled "*Replace Nested Conditional with Guard Clauses*". This refactoring replaces a conditional statement wrapped around normal (i.e., non-error) processing with a guard clause placed before normal processing. The guard clause performs an early return from the method when an error occurs. Fowler argues for the removal of the nested conditional because it hinders the comprehension of the normal execution path. An example of a function undergoing the refactoring is shown in Figure 2 and Figure 3.

In order to perform the refactoring an XSLT program was written. The XPath template matching of XSLT is particularly well suited to XML document transformation. We will now describe important parts of the XSLT program.

In order to limit changes only to the parts of the document that is desired we use a default copy template so that, unless otherwise specified, the elements and text will go through the refactoring unaltered. This includes all elements and text. Doing so preserves all documentary structure, statements, and preprocessor statements.

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper values
  if (is_non_negative(n)) {

    // calculate factorial
    product = 1;
    int i = 1;
    while (i <= n) {

      // update the product
      product *= i;

      // next value
      ++i;
    }
  }

  //now the factorial
  return product;
}
```

Figure 2. The original function uses a nested conditional to check the status of the input parameter.

The first issue is how to identify the nested conditional that will be replaced. srcML allows the formation of XPath expressions to refer to particular statements in the source code. For this refactoring a specific parameter, *$cname*, was provided to the XSLT program. The main XSLT template matches the `if`-statement that forms the nested conditional using the XPath expression:

```
src:if[src:condition/src:expr//src:name=$c
name]
```

This expression matches all *src:if* elements whose condition contains the same name as the parameter *$cname*. This parameter contains the name of a function call that this example uses to identify a nested conditional. After matching the nested conditional the main template generates the guard clause and converts the former contents of the guard clause so that they are at the top level of the function. In addition, it fixes the indentation on these contents so that their indentation lines up with the rest of the program. We will now look at how a document view can be used to support these two parts of the transformation.

The main template uses another template, with the mode attribute "guard", to construct the guard clause. An *if*-statement embeds the condition from the nested conditional and inserts a *return*-statement as below:

```
if (!<xsl:value-of
select="src:condition/src:expr"/>)
return -1;
```

```
int factorial(int n) {

  // factorial value
  int product = -1;

  // check for proper values
  if (!is_non_negative(n))
    return -1;

  // calculate factorial
  product = 1;
  int i = 1;
  while (i <= n) {

    // update the product
    product *= i;

    // next value
    ++i;
  }


  //now the factorial
  return product;
}
```

Figure 3. The transformed, refactored function on the right uses a guard clause and preserves the normal processing.

This demonstrates an advantage of the transparency of srcML. It is not necessary to markup all new, generated code in srcML. Explicit source-code text can be used in the transformation program interspersed with XML programming statements. If full srcML markup is needed, this partial srcML document can be converted to source code, and then processed by the srcML translator to generate a fully-marked srcML document. After the creation of the guard clause the main template calculates the previous indentation of the if-keyword using the XPath: `preceding-sibling::text()[1]`.

The remainder of the main template handles the contents of the nested conditional and "un-nests" the contents. The entire contents of the block are converted to a single string. The braces at the start and end of the block are removed by extracting a substring of the block that removes the first and last characters.

This allows the transformation to work at multiple levels. We are able to identify where the block is by using the syntactic elements. Even though the contents of the block contain syntactic elements, we are able to ignore the syntactic elements and process the contents of the block as a string.

This string is passed to a template that removes these characters from the beginning of each line. The template used for this is not shown, but is a recursive template that splits the strings at the new line and checks for needed trimming of the white space.

The current form of our refactoring has some limitations. First, the identification of a nested conditional (versus normal processing) is limited to a special function name. The user would have to indicate, perhaps by location, the nested conditional. Second, the `return`-statement has a fixed return value. The value could be indicated by the user, e.g., via a parameter, or deduced from the context. Improvements can also be made to the XSLT program to make it simpler and more straightforward by defining special XPath functions for the string handling, e.g., indentation.

This example demonstrated a flexible, non-intrusive approach to source-code refactoring. By using a source-code representation appropriate to the problem, all levels of source-code information were available for transformation. Although XSLT was used in the example, any XML transformation language or API could have been used.

3. Future Directions

Our current work continues to expand the example shown to the other refactorings in Fowler's catalog. Even with the example shown, there exist open issues regarding the specification of the location of the refactoring and the matching of the documentary structure. We also plan to compare our method with other transformational/refactoring approaches.

The presented refactoring involved a single source-code file. Refactorings that require more than one source-code file will involve the formation of complex srcML documents, i.e., a single srcML document that contains the representation of multiple source-code files. The srcML format already provides for this by allowing the main element of a source-code file, i.e., the *unit* element, to be nested. The element *unit* also already contains attributes to store directory and file information. Therefore a merge and split can be used to perform a single refactoring on multiple files.

A disadvantage of a direct document-oriented source-code representation is the lack of higher-level abstractions. These higher-level models are associations between potentially separate parts of the source code. These source models include call graphs, logical class structures, and task-specific abstractions. The document representation can only mark what is in the code and is unable to directly mark higher level associations.

The representation allows for the integration of the source-model information with the source code. In effect, we actually allowed one to imbed any type of meta-information into the source. XPath expression involving the source model can be used in the location of a particular element in the source code.

This work was supported in part by a grant from the National Science Foundation (CCR-02-04175).

4. References

[1] Baxter, I. D., Pidgeon, C., and Mehlich, M., "DMS: Program Transformations for Practical Scalable Software Evolution", in Proceedings of 26th International Conference on Software Engineering (ICSE04), Edinburgh, Scotland, UK, May 23 -28 2004, pp. 625-634.

[2] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[3] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[4] Cordy, J. R., "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 196-206.

[5] Fowler, M., Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[6] Klint, P., "How Understanding and Restructuring Differ from Compiling - A Rewriting Perspective", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 2-12.

[7] Maletic, J. I., Collard, M. L., and Marcus, A., "Source Code Files as Structured Documents", in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), Paris, France, June 27-29 2002, pp. 289-292.

[8] Van De Vanter, M. L., "The Documentary Structure of Source Code", Information and Software Technology, vol. 44, no. 13, October 1 2002, pp. 767-782.

Session 2:
Source Code Paring and Fact Extraction

Application of Island Grammars in a Toolkit for Software System Analyses

Rob van der Leek[*]
Faculty EWI, Delft University of Technology
Delft, The Netherlands
r.c.vanderleek@ewi.tudelft.nl

Abstract

This paper demonstrates the use of island grammars as a method for source fact extraction applied in a toolkit for software system analyses. It focuses mainly on practical aspects, by presenting the tools and techniques that are used in this process and by providing a small use case. The first section shows the need for agile parsing techniques in the exploration of legacy software systems. Section 2 contains an overview of the resources used in the island grammar construction process. In section 3 a toolkit for software analyses is presented. To stress the practical aspect of this paper, the toolkit and an analysis based on an island grammar are used in a small use case in section 4. The last section of this paper reflects on the experience of applying island grammars as a technique for fact extraction in software analyses and references related work.

1. Introduction

Software Exploration Exploring legacy software systems can be a daunting task. It is not uncommon to encounter systems with a size in the order of millions of lines of code, consisting of a mix of legacy business languages, domain specific languages, proprietary languages and rare language dialects. Moreover, incomplete sources and embedded languages (such as embedded SQL) present a serious obstacle to explore these systems with traditional parsing tools.

Fact Extraction Lexical analysis techniques often provide a first help to explore the aforementioned software systems. However, not all facts can be extracted by source code scanning alone. As the complexity of source code analyses grow the lexical approach quickly looses its strength since it requires a growing amount of support code. Lexical approaches are usually also targeted at a specific problems which makes them harder to be reused in different situations.

Island Grammars An island grammar [6, 7] is a grammar that consists of two parts: (i) detailed productions describing certain constructs of interest (the *islands*), and (ii) liberal productions that catch the remainder of the input (the *water*). The grammar is a partial grammar since it contains only a subset of the productions of a complete grammar. Island grammars meet the extremes of lexical analysis and traditional parsing in the middle. It provides the tolerance and robustness of lexical analyzing, yet it keeps the accuracy and structured approach of context-free parsing. Island grammars therefore are very suited for the exploration of legacy software systems.

Software Analyses The Software Improvement Group (SIG) is a Dutch company where the exploration of legacy software systems is one of the main business activities. The SIG offers software assessments that provides companies insight into their legacy code bases. All assessments are done on the actual source code of the system. An assessment usually consists of a number of fact extracting source code analyses. Due to the limited period of time available for these assessments it is crucial for the company to have software exploration tools that are flexible enough to be quickly adapted for new software systems. In the course of an assessment there is no time to construct new parsers, or even to modify existing parsers.

The SIG has developed its own framework ap-

[*]This paper was written with support of the Software Improvement Group, Diemen, The Netherlands, `http://www.sig.nl`

plication for doing analyses on software systems. An integration of agile parsing techniques, such as island parsing, into this framework enables the company to do extensive software assessments in a relative short amount of time.

2. Island Grammar Construction

Island grammars do not force the use of a particular grammar syntax formalism or parsing technique. This section presents the combination of syntax formalism, parser generator, and grammar development environment that was used for the construction of an island grammar in section 4 of this paper.

2.1. Grammar syntax formalism

A grammar syntax for island grammars preferable should be concise, free of any parsing specific elements and should promote structured grammar design and grammar reuse. SDF [9] (the Syntax Definition Formalism) supports all of these features. SDF has a pure, EBNF-like syntax. It allows the combination of lexical and context-free grammars and its grammar definitions can be modular. Listing 1 in section 4.1 shows an example of an SDF grammar module. Each production of the module consists of a left hand side with mixed lexical/context-free symbols that are injected into a non-terminal symbol on the right hand side.

2.2. Parser generator

Island grammars are inherently ambiguous due to their tolerant behavior. Ambiguities that occur during parsing can not always be resolved with the help of grammar productions alone. Island grammars can produce multiple valid parse trees for the same input (here we speak of a *parse forest*). It is the parser's responsibility to select the most optimal tree from this forest based on priorities between rules in the given grammar.

The above mentioned island grammar properties exclude most traditional LR and LL parser generator implementations as candidates for generating island parsers. For the use case in this paper a Scannerless GLR parser [8] (SGLR) was used. GLR (Generalized LR) parsers accept any context-free grammar and support the creation of ambiguous parse trees during parsing. SGLR uses parse tree filters to prune the parse forest afterward.

2.3. Grammar development environment

A grammar development environment enables rapid grammar engineering through a round-trip process of grammar specification, parser construction and parser testing. The environment used for the examples in this paper is the Meta-Environment [1]. The Meta-Environment uses SDF as its core syntax definition formalism and employs SGLR as a parser generator.

3. The Analysis Toolkit

The System Analysis Toolkit (SAT) is the framework application for source code analyses used at the SIG. The software is written in Java, as are all of the analyses. The toolkit has a modular design that allows it to be be extended with new analyses or tailored versions of existing analyses.

A source code analysis is done with SAT by creating an analysis network. The analysis network is a directed graph where each node is a functional element of the analysis and each edge controls the flow of data through the network. A node can have multiple incoming and outgoing edges. Analysis networks are built in a graphical environment, and can be executed directly from that environment or in a batch-like manner. Examples of typical analyses performed with SAT are a dependency graph analysis and various software metric analyses (e.g. fan-in/fan-out, cyclomatic complexity, etc.)

Figure 1 shows an example of a very basic SAT network for counting the number of lines in a Java source. The analysis reports both the number of lines with comments included and without comments. The two ViewDataNode boxes at the right show the respective results of that count.

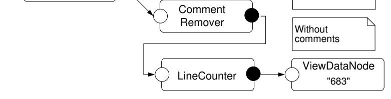


Figure 1. A basic SAT network

4. A Use Case

The application of island grammars in the system analysis toolkit is demonstrated by means of a short use case. The use case shows how a representative analysis, a Java method extraction analysis, can be implemented with an island grammar and how it can be integrated into the toolkit. The method extraction analysis is considered to be representative for the following reasons: (i) most imperative programming languages support some notion of methods (albeit subroutines, procedures, functions, etc.) (ii) the results of this analysis are used in various other analysis, such as fan-in/fan-out, call graph and cyclomatic complexity, (iii) it is not straightforward to express this analysis with lexical techniques (such as regular expressions).

The purpose of this use case is to demonstrate how a widespread, mature programming language like Java can be explored using island grammars. Since complete grammars are available for Java, it may seem odd to choose the island grammar approach. However, even a modern language like Java has its language extensions (e.g. AspectJ, Generic Java) and dialects that require a specialized grammar (see [4] for a discussion of agile parsing techniques based on grammar modifications).

4.1. The island grammar

Each island grammar construction process starts by defining the *constructs of interest*. In the use case these are the grammar productions for the method header and the method body. Listing 1, extracted from the Java SDF grammar in the SDF grammar base [1], shows the relevant grammar productions.

```
context-free syntax
  MethodHeader MethodBody                     -> MethodDeclaration
  Modifier* Type MethodDeclarator Throws? -> MethodHeader
  Identifier "(" {FormalParameter ","}* ")"
                                              -> MethodDeclarator
  MethodDeclarator "[" "]"                    -> MethodDeclarator

  PrimitiveType                               -> Type
  ReferenceType                               -> Type
  Identifier                                  -> VariableDeclaratorId
  VariableDeclaratorId "[" "]"                -> VariableDeclaratorId
  Type VariableDeclaratorId                   -> FormalParameter

  "throws" {ClassType ","}+                   -> Throws
  Block                                       -> MethodBody
  "{" BlockStatement* "}"                     -> Block
  Block                                       -> BlockStatement
```

Listing 1. Relevant grammar productions

In the island grammar, the production for the non-terminal BlockStatement becomes a catch-all production that consumes all content between a pair of balanced curly braces.

[1] http://www.program-transformation.org/gb

To create an island grammar from this set of productions only a few extra steps have to be taken. First the standard island grammar modules for Layout and Water are included to make the parser discard superfluous whitespace and to make it tolerant for input that is not matched by the constructs of interest productions. Secondly a Comment module is included to prevent false positives inside Java comments. Productions for string literals are placed in the Lexical module. Figure 2 shows the complete grammar module hierarchy for the method extraction island grammar. The numbers behind the module names represent the number of productions each module contains.

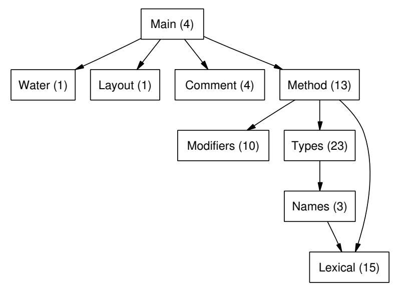


Figure 2. The module hierarchy

The modules Modifiers, Types and Names are included to increase the detail of the parse tree at the method header level. Figure 3 contains a part of the parse tree at this level.

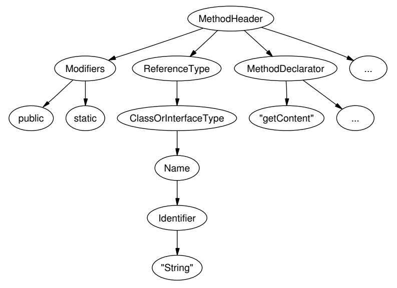


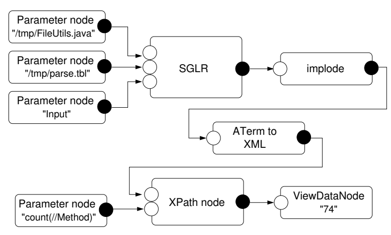
Figure 3. The parse tree at the method header level

4.2. The parse tree interface

The parse tree interface defines the way data is extracted from the parse tree produced by the island parser. A successful applied method in other analyses is based on tree traversal [5]. However, this approach requires extra code in the form of a tree walker implementation and it's behavior is not easy to modify once it is implemented for a specific analysis. In the use case a more dynamic parse tree interface is used, based on a structured data query language (XPath). Fact extraction expressions can be entered interactively in the analysis network editor of the SAT.

4.3. The analysis network

Figure 4 shows an analysis network for counting the number of methods in a Java class. The SGLR module is a wrapper for the SGLR parser. This module takes as input a parse table, the name of the start production (which is Input in this case) and the Java input source. The SGLR module produces a parse tree in the ATerm [2] format which is imploded and converted to XML in the next two steps of the network. The XPath module receives this data structure and queries it with the expression shown in the left bottom node (count(//Method)). Finally, the result of this query is displayed on the right by the ViewDataNode.



Parameter node "/tmp/FileUtils.java"
Parameter node "/tmp/parse.tbl"
Parameter node "Input"
SGLR
implode
ATerm to XML
Parameter node "count(//Method)"
XPath node
ViewDataNode "74"

Figure 4. A network for Java method counting

The setup in figure 4 only demonstrates a trivial usage scenario. Another example of a dynamic query that can be done with this network is for example the extraction of all method names (//Method//MethodDeclarator/text()).

5. Concluding Remarks and Related Work

With a practical example this paper has shown that island grammars can be a valuable aid in the exploration of (legacy) software systems.

The island grammar approach has clear advantages over conventional lexical techniques: analyses based on small grammar definitions are easier to maintain, extend and reuse than those built out of a mix of regular expressions and support code.

In combination with an environment for grammar development, a framework for software analyses and light-weighted parse tree extraction methods, a source fact extraction can be realized using island grammars in a short period of time.

For reasons of brevity, we refer to [3, 6, 7] for a detailed discussion of related work.

References

[1] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proceedings of Compiler Construction 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[2] M.G.J. van den Brand, H.A. de Jong, Klint P., and Olivier P. A. Efficient annotated terms. *Software Practice and Experience*, 30(3):259–291, 2000.

[3] C. Clarke and A. Cox. Syntactic approximation using iterative lexical analysis. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC 2003)*, pages 154–164. IEEE Computer Society Press, May 2003.

[4] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Agile parsing in TXL. *Journal of Automated Software Engineering*, 10(4):311–336, October 2003.

[5] T. Kuipers and J. Visser. Object-oriented tree traversal with JJForester. In *Proceedings of the Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44, pages 28–52. Elsevier Science Publishers, 2001.

[6] L. Moonen. Generating robust parsers using island grammars. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society Press, October 2001.

[7] L. Moonen. Lightweight impact analysis using island grammars. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC 2002)*, pages 219–228. IEEE Computer Society Press, June 2002.

[8] E. Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, University of Amsterdam, Programming Research Group, August 1997.

[9] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Amsterdam, 1997.

Software Factbase Extraction as Algebraic Transformations: FEAT

Yuan Lin, Richard C. Holt
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON N2L 3G1, Canada
{y3lin, holt}@waterloo.ca

Abstract

Fact extractors provide information about target software systems for use in various reverse engineering and program comprehension tools. Fact extractors are too often *ad hoc*, incomplete and not well documented. Consequently, it is useful to formalize the requirements for fact extraction and how these requirements can be met. Some modern compilers, GCC in particular, provide reliable facts about source code in typed graphs. We propose that software Factbase Extraction as Algebraic Transformations (FEAT) on these typed graphs can support the needs of various reverse engineering and program comprehension tools. In this paper, we argue that relational algebraic transformations are reliable, expressive and easy to expand.

1. Introduction

Fact extractors provide information that is used by reverse engineering and program comprehension tools. Writing a fact extractor is far from trivial, and it is not easy to keep an extractor up to date. As new tools emerge they demand new kinds of information about the target software. Fact extractors must be able to provide different views of the legacy code and must satisfy various schemas describing extracted facts. Unfortunately, as several authors pointed out [4], too often fact extractors are not very reliable. Two obstacles to producing a high quality fact extractor are:

(1) languages like C and C++ are complicated and ambiguous, making parsing and semantic analysis of source code difficult, and

(2) fact extractors are usually written in an *ad hoc* manner [4], with little documentation on the extractor's data structure, algorithm and schema of extracted facts. An *ad hoc* approach is suitable for solving simple problems but troublesome in the face of changing user requirements.

Combining advancement in compiler technology and binary relational algebra, we propose software Factbase Extraction as Algebraic Transformations (FEAT) as a solution to the quality problem of fact extractors. Some modern compilers, GCC in particular, can dump facts about source code in a typed graph [4], called an Abstract Semantic Graph (ASG). The graph includes an Abstract Syntax Tree of the source code, typing information and resolution of names. The graph is relatively easy to understand. Generally, there is no ambiguity in this ASG, because any potential ambiguity has already been resolved by the compiler. In particular, we have analyzed GCC ASGs of source code, not the source itself. As a result, we avoid parsing and semantic analysis of C++ programs. We suggest this is a good way to overcome the first obstacle to improving fact extractor's quality.

To overcome the second obstacle, we use binary relation algebra, not programming language to process GCC ASG and generate answers. Binary relation algebra is a set of well-defined mathematical notation, axiomatized by Tarski and Schmidt [12]. We can consider that it has only one data structure—tuples. Binary algebraic transformations are written as formulae composed using algebraic operations. Equivalent logic written in a C program would be many lines of complex code [5,7]. These transformations allow the developer to work at a very high level, in particular, at the level of algebraic manipulations of graph structures [1,3,6,10]. Our position is that this approach holds the promise of low cost, agile, accurate and responsive way to analyze legacy source code.

The rest of the paper is organized as follows. Section 2 discusses the GCC ASG. Section 3 gives a brief introduction of binary relational algebra [12]. Section 4 presents how to build a fact extractor based on binary relational algebra. Section 5 lists our conclusions.

2. GCC Abstract Semantic Graphs

GCC uses a typed graph [4] as an ASG to represent syntactical and semantic information from the source code. Such a *typed graph* has typed nodes, e.g., nodes for functions and nodes for variables, and typed edges, e.g., edges for function "call" and for data "reference". In the ASG, nodes represent type declarations, statements and expressions. A node can have attributes. For example, an identifier node may have a *name* attribute that is a text string, and an integer constant node may have a *value* attribute. The types of edges help give the meaning of the graph. For example, a function declaration node may have a *name* edge pointing to the identifier node storing the name string and a *body* edge pointing to the body of the function.

We will illustrate ASGs using the simple C program in Figure 1.

```
main() {
        f(1);
}
```

Figure 1. Simple C program.

GCC compiles this example program and dumps its ASG into an ASCII file as shown in Figure 2 (some details are omitted).

```
@1    function_decl
            name: @2      body: @5
@2    identifier_node
            strg: main
@5    compound_stmt
            body: @14
…
```

Figure 2. Partial ASG for Figure 1.

In this ASG, node 1 has the class *function_decl*. Its *name* is stored in node 2 and its *body* is in node 5. Node 2 is an *identifier_node*, whose *strg* attribute has the value *main*. Node 5 is a *compound_stmt* node, whose *body* is located via node 14. Figure 3 illustrates of this partial ASG.

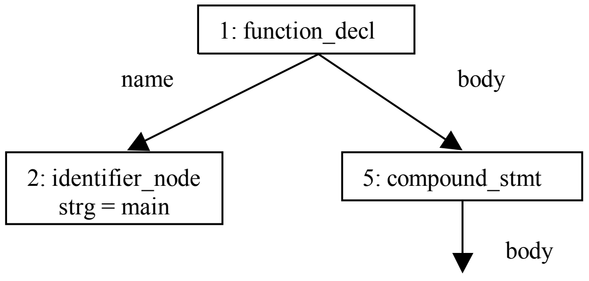


Figure 3. Diagram for the partial ASG.

3. Binary relational algebra and Grok

Before we explain how to analyze factbases with binary algebraic transformations, we will give a brief introduction of binary algebra and Grok [7]. Binary algebra defines operations on binary relations. Grok is a binary relation calculator that inputs a binary algebraic formula (really, a script containing formulae) along with an initial graph, and outputs the transformed graph.

3.1. ASG in TA format

ASGs like the one in Figure 2 can be represented as binary relations, and TA format [7] (which is much like the RSF notation [7]) is a popular format from binary relations. In TA format, all elements of an ASG (nodes, edges and attributes) can be represented as tuples <R x y>, where R is the relation and x and y are values. A tuple <R x y>, can also be thought of as an edge from x to y of type R. We are using the terms tuple and edge interchangeably in this paper.

The ASG in Figure 2, as written in TA, is shown in Figure 4.

```
$INSTANCE        @1      function_decl
$INSTANCE        @2      identifier_node
$INSTANCE        @5      compound_stmt
name             @1      @2
body             @1      @5
strg             @2      main
body             @5      @14
…
```

Figure 4. Partial ASG in TA format

Note that the order of tuples in TA is immaterial, i.e., re-ordering has no effect on the graph.

3.2. Binary relational algebra operators

Binary relational algebra [7,12] has set operators and relational operator. Set operators are the same as that in set theory. Relational operators include the following:

Relational composition. $R_1 \circ R_2$ is the relation $\{<x, z> \mid \text{there is } y \text{ such that } <x, y> \in R_1, <y, z> \in R_2\}$. It can be thought of as a path from x to y to z.

Transitive closure. R^+ is the relation defined by

$$R + (R \circ R) + (R \circ R \circ R) + \ldots$$

until a fixed point is reached, where infix + means the union of relations. It is essentially a path composed of one or more edges in relation R.

Reflexive transitive closure. R^* is defined as $ID + R^+$ where ID is the identity relation $\{ <x, x> \}$. It is essentially to a path composed of zero or more edges in relation R.

Set projection. S . R, where S is a set, is the set $\{x \mid y \in S \text{ and } <y, x> \in R\}$. In other words, it is a set of nodes that are connected to nodes in S via edges in relation R.

4. A simple fact extractor

In this section, we will build a simple fact extractor to illustrate FEAT. This will be done by writing the Grok script, line by line, that implements the extractor. This fact extractor creates the call graph for C source program. This is a common task for fact extractors [4,8,9]. The call graph will be constructed as a *call* relation, in which each tuple <*call* x y> means that function definition x contains a call to function y. To find all these tuples, our Grok script will do the following:

1) find each function definition (e.g., function x),
2) find each function call (e.g., call to y), and
3) determine which function definition contains which function call.

4.1. Find each function definition

In GCC ASGs, each function definition is represented by a *function_decl* node with a *body* edge pointing to the subtree representing the function's body (see Figure 5). In lines 1 of our Grok script (see below), we select *nd* nodes as those nodes that are instances of class *function_decl*. Then line 2 traces (by relational composition) from each *fd* node across a *body* edge to the sub-tree. (Technically speaking in line 2, the set *nd* is implicitly promoted to be *id(nd)*, which is the identity relation over set *nd*.)

```
nd := $INSTANCE . function_decl          (1)
fd := nd o body                          (2)
```

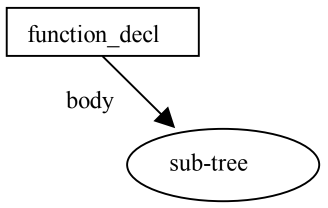


Figure 5. A function definition in ASG

4.2. Find each function call

In GCC ASGs, each function call in the source code has three corresponding nodes in ASG: a *call_expr* node for identifying the function call, an *addr_expr* node for representing the memory location of the function and the *function_decl* node of the function being called. This *function_decl* node is shared among all calls to that function. *A fn* edge connects the *call_expr* node to the *addr_expr* node, and an *op0* edge connects the *addr_expr* node to the *function_decl* node. The Grok lines to create an *fc* edge that links the *call_expr* node directly to the *function_decl* node are given here (note the similarity to lines 1 and 2 above):

```
n2 := $INSTANCE . call_expr              (3)
fc  := n2 o fn o op0                     (4)
```

4.3. The *call* relationship

To create the *call* relation, we need to determine, for each function x, all those calls that are contained in x. To do this, we will create a relation called *contain,* which has the following meaning. A function definition *contain*s all statements inside the definition, an expression statement *contain*s the actual expression, an expression *contain*s operands and a function call *contain*s its arguments. Consequently, tuples in the *contain* relation form a path from a function definition to its contained function calls. We will now translate these ideas into Grok.

4.3.1. Building contain relation

We use the following 4 lines to compute the *contain* relation. For clarity, we write this as 4 lines, but these lines could as well be combined into a single line.

```
contain := op0 + op1                        (5)
contain := contain + expr                   (6)
contain := contain + args o (chan *)        (7)
contain := contain + body o (next *)        (8)
```

In line 5 (above), o*p0* and *op1* are edges from a (binary) expression to its two operands and are part of the *contain* relation. In line 6, *expr* edges connect an expression statement to the actual expression and so these are also part of our *contain* relation.

Line 7 deals with arguments of a function call as will now be explained. As illustrated in Figure 6, in the GCC ASG, an *args* edge connects a function call to its first argument. Then, *chan* (short for chain) edges connect the first argument to the second, the second to the third, and so on. In the figure, dashed arrows indicate *contain* edges created by paths composed of *args* and *chan* edges. Line 7 computes *contain* edges from a function call to all of its arguments that are equivalent to paths of one *args* edge followed by zero or more *chan* edges.

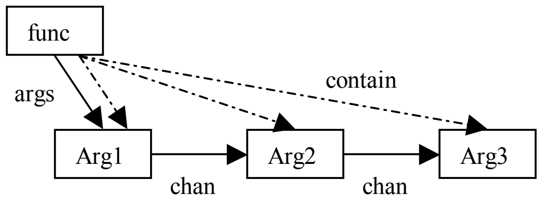


Figure 6. Arguments of a function call.

Similarly, line 8 computes *contain* edges from compound statements to the statements inside its body. The *body* edge points to the first statement, the *next* edges connecting the first statement to the rest, and we build *contain* edges based on the paths of one *body* edge followed by zero or more *next* edges.

4.3.2. Matching the caller with the callee

Finally, we will match each function definition with the function calls inside it. Based on the *contain* relation constructed by lines 5-8, we know function definitions *contain* statements, statements *contain* statements and expressions, expressions *contain* operands and function calls *contain* arguments. We know from the structure of the GCC ASG that a function definition x contains a function call exactly when there is a path from x's definition node to the function call via one or more *contain* edges. The Grok code for this is:

```
        call := fd o (contain +) o fc        (9)
```

Recall that *fd* is the set of all function declarations, and *fc* is the set of all called functions. Hence, lines 1-9 compute the call graph.

We simplified our presentation of a call graph extractor by ignoring certain aspects. We omitted logic such as removing the parts of the graph that are not part of the call graph. We did not explicitly handle the problem of linking the names of functions to the caller and callee nodes. Also, we ignored questions about static functions and function pointers. These can be dealt with by an expanded Grok script. Our point has been to illustrate how straightforward it is to transform an ASG to the sort of graph required to support fact extraction.

A visualization tool such as *lsedit* [11] can draw a diagram for the call graph based on *call* relation that our extractor produces.

5. Conclusions.

The FEAT (Factbase Extraction as Algebraic Transformations) approach provides a number of advantages over other languages that can be used to implement fact extraction. The advantages include the following.

FEAT is more reliable. It is challenging to write a parser and semantic analyzer for C and C++ because such languages are complicated and ambiguous. However, the front end of a modern compiler such as GCC does a very good job in parsing and eliminating ambiguities in C and C++ source code. The GCC ASG is a reliable source of information about the source code and when dumped provides a convenient basis for fact extraction [4].

Binary relation algebra is defined in a mathematical way and can be proved or refuted in a mathematical fashion. The formulas are short, compared with code in a typical programming language, so their correctness can be more easily assured by careful inspection [1,3,8,10].

FEAT is more expressive. A very high level language such as Grok supports a short development cycle, relatively easy maintenance and fast adaptation to meet new requirements [1,3,9,10]. Grok's algebraic expressions are at a high level of abstraction, so the programmer of the transformer can more easily understand the transformer while ignoring low level details such as data structures implemented in C. The algebraic expressions are shorter, commonly by an order of magnitude, than the corresponding C code, so they are easier to validate than C code. Therefore, it takes less time to write a new fact extractor or modify an old one for a new output fact schema.

FEAT is easier to expand. Expanding an extractor written in programming language such as C commonly involves changing the extractor's data structures and algorithms [5,6,7]. However, that is not the case in FEAT, in that the data, from the point of view of the programmer, remains a graph (not a data structure), represented externally in TA format. The internal representation and data structures inside the Grok engine are hidden from the programmer, allowing the programmer to work solely at the high level of graph structures. In other words, when working at the level of algebraic relational operators, the programmer of a fact extractor can think and program at a level that is natural for the problem at hand.

6. References

[1] Guilleermo Arango, Ira Baxter, Peter Freeman and Christopher Pidgeon, "TMM: Software Maintenance by Transformation", pp27-39, IEEE Software, 3(3), May, 1996.

[2] Ira Baxter, I. Christopher Pidgeon, "Software Change Through Design Maintenance". Proceedings of the International Conference on Software Maintenance '97, 1997, IEEE Press.

[3] J.R. Cordy, T.R. Dean, A.J. Malton and K.A. Schneider, Software engineering by source transformation - experience with TXL. In Proceedings of SCAM'01 - IEEE 1st International Workshop on Source Code Analysis and Manipulation. Florence, November 2001.

[4] T. R. Dean, A. J. Malton, and R. C. Holt. Union schemas as a basis for a C++ extractor. In 2001 Working Conference on Reverse Engineering. Stuttgart, October 2001.

[5] H.M. Fahmy, R.C. Holt, and J.R. Cordy, "Wins and Losses of Algebraic Transformations of Software Architecture", Automated Software Engineering, San Diego, California, November 26-29, 2001.

[6] H. Fahmy, R. C. Holt, S. Mancoridis, "repairing Software Style Using Graph Grammars", In the IBM Proceedings of the Seventh Centre for Advanced Studies Conference (CASCON'97), Toronto, Ontario, Canada, November, 1997.

[7] R. C. Holt. "Introduction to the Grok Programming Language". From http://plg.uwaterloo.ca/~holt/papers/grokintro.doc

[8] Ralf Lämmel, Semantics of Method Call Interception, AOP'01, Paderborn, May 3-4, 2001

[9] Ralf Lämmel, "Generic Type-preserving Traversal Strategies", ENTCS, Volume 57 Also In: Proc. WRS'01, Utrecht, May 26, 2001

[10] Leon Moonen, "Exploring software systems", In Proceedings of the International Conference on Software Maintenance (ICSM 2003). IEEE Computer Society Press, September 2003.

[11] SWAG lab, Univ. of Waterloo, Software Architecture Toolkit, http://www.swag.uwaterloo.ca/swagkit/

[12] A. Tarski, "On the Calculus of Relations", Journal of Symbolic Logic, Vol. 6, No. 3, 1941, pp. 73-89.

Transformation Systems For Real Programming Languages

Preprocessing Directives Everywhere

Michael Mehlich

Semantic Designs, Inc.
12636 Research Blvd, C214
Austin, TX 78759

mmehlich@semanticdesigns.com

Abstract

Many successful programming languages contain preprocessing directives. Transformation systems for evolving source code in such languages therefore need to be able to deal with such directives during analysis and transformation and must preserve them as appropriate in the resulting code.

This paper discusses challenges arising from these requirements including issues related to parsing source code, performing name and type resolution, and applying transforms.

1 Introduction

Dealing with the full complexity of programming languages like C++ or Verilog is hard. Many tools avoid this complexity by extracting only simple information about the source code and supporting only correspondingly simple transformation tasks reliably. Other tools extract approximations of more complex information but rely on human engineers to validate the results.

While such tools provide valuable assistance to human engineers, more advanced tools for evolving large legacy software systems, e.g. for applying semantics-preserving refactorings [1] or for migrating source code to other languages or operating system platforms, must have the capability for deep understanding of the syntactic and semantic details of the source code as imposed by the underlying programming languages.

As the source code resulting from applying these advanced tools is to be used as the basis for continuing maintenance of the evolved software system, it is essentially for them to be able to deal with source code as the human engineer sees (and understands) it. For languages with preprocessors, e.g. for C, C++, C#, or Verilog to name but a few, this requires dealing with the presence of macros and preprocessing conditionals in the source code [6]. Expanding the preprocessing directives is not an option such tools can choose.

In order to ensure correctness of applied transformations, such tools also need to deal with semantic issues, with the meaning of symbols, including the relationship between declarations and uses of entities in the source code, being at the forefront. Such analyses can be hard to implement for complex languages like C++ [12] and are further complicated by the requirement disallowing expansion of preprocessing directives. In particular, analyses have to take into account preprocessing conditionals.

Finally, transformations have to be applied reliably to the source code in order to achieve the desired effect. In general, such transformations need easy access to semantic information that has been pre-computed by some analyses, but also need to be able to call on analyses that are to be performed on the fly.

Application of transformations is complicated by the presence of preprocessing conditionals, which may impose the needs to transform a single code fragment in different ways depending on possible configurations[1] of the software system. Similarly, macros might have to be translated in multiple ways depending on the various contexts they are called from, even if no preprocessing conditionals are present.

2 Syntax

The basic context free syntax of many programming languages often cannot be captured easily by LL(1) or LR(1) grammars. Even when possible, it is generally not advisable to develop a grammar just to satisfy such restrictions since this might obscure the clarity of the language definition. Rather, the grammar should follow

[1] A configuration is a combination of (possibly unknown) facts about macros that may be true at the beginning of a translation unit, and potentially influences the result of the compilation process.

the conceptual syntactic structure of the language, as such a structure supports formulating source-to-source transformations based on the native syntax [10] of the respective programming language.

In general, such context free grammars contain local ambiguities, i.e. they allow a single phrase in the language to be parsed in different ways. While these ambiguities can often be resolved by interleaving parsing and name resolution, it is rather good practice to separate the two concerns of context free analysis and static semantic analysis.

In the presence of preprocessing conditionals, it might even be necessary to preserve multiple alternatives and produce an ambiguous parse tree, reflecting different variants dependent on possible configurations of the software system. Consider the following (contrived) source code fragment in C:

```
#ifdef CONFIG
  typedef int x;
#else
  int x(int);
#endif
int y;
int f() { x(y); }
```

Depending on whether the macro `CONFIG` is defined or not, the statement `x(y);` either is a declaration statement or is an expression statement. As there is no explicit `#define` or `#undef` for `CONFIG` in the given code fragment, both are possible outcomes when processing the source code.

Another complication is the need for the syntax trees produced by the parser to represent the placement of preprocessing conditionals in the source code. For well-structured occurrences, this can be achieved e.g. by adding grammar rules that allow these directives around designated language constructs [4]. Unstructured occurrences, the scopes of which cross language concept boundaries, cause more of a problem, however. For example, in the following fragment, the `#ifdef` construct encapsulates only a part of the conditional statement, creating an overlap of syntactic forms.

```
#ifdef CONFIG
  if (c1) {
#else
  if (c2) {
#endif
  ...
  }
```

In principle the grammar can be made to accommodate this form. However, it is practically impossible to cover all conceivable forms of occurrences in advance. A possible approach to alleviate this problem is to expand (or shrink) the extent of preprocessing conditionals by including parts of their respective contexts until they become well-structured occurrences [8], e.g. in the above example the `#endif` directive could be moved to the end of the statement by a special preprocessor. Alternatively, the `#ifdef`, `#else`, and `#endif` directives could be moved to only enclose `c1` and `c2`, respectively.

Further problems arise in the presence of macros, information about which has to be included into the syntax tree. In principle, this could be achieved by expanding macro calls and keeping track of the source of fragments of the syntax tree created during parsing (e.g. similar to the fold representation described in [9]), though this can become tedious if the preprocessor allows manufacturing tokens from token fragments. Alternatively, for well-structured macros, macro calls could be preserved and directly represented as such in the syntax tree using a dedicated node referring the macro definition, with subtrees representing the actual arguments and optionally with a subtree representing the expansion of the macro call. Unstructured macros can then be dealt with by expanding (or shrinking) the extent of the respective macro calls.

Combining macros and preprocessing conditionals allows the conditional definition of macros; i.e. macros may be defined in different ways or may be defined or not defined dependent on the configuration under which the software system is compiled. Such variations may cause several alternatives in the syntax tree to be produced that can be represented e.g. by injecting auxiliary preprocessing conditionals, the branches of which reflect the different variants [8]. However, in order to be able to reproduce the original source code, these conditionals should be distinguishable from the conditionals explicitly occurring in the source code.

3 Semantics

Many transformations rely on semantic information as part of their respective application condition in order to ensure that their application preserves or achieves desired properties of the source code. In general, these conditions rely on semantic information of the underlying programming language and on analyzers extracting information of interest from the source code respectively syntax tree, thus allowing to determine whether the respective condition is satisfied.

An important fundamental basis of such analyses often is a symbol table containing core information about declared entities in the program and a mapping from symbols, a.k.a. identifiers, in the source code to the respective symbol table entries.

Implementing a name and type resolver to create a symbol table for source code written in a complex language like C++ [12], with its plethora of informally described rules, can be a daunting task. However, it is a necessity in order to be able to reliably apply sophisticated transformations, e.g. for refactorings, to source code in such languages.

Attribute grammars allowing side effects to update large data structures, and in particular symbol tables, combined with a powerful pre-defined symbol table management subsystem have been demonstrated to provide suitable means for realizing name and type resolution [5]. Such attribute grammars can be compiled into parallely executable code based on the partial order dependencies between attribute evaluation rules.

Macro calls can easily be incorporated into this approach by performing name and type resolution on their respective expansions. This suggests but does not require the expansions to be explicitly represented in the syntax tree.

However, preprocessing conditionals impose additional challenges for symbol table construction. In particular, entries in the symbol table must be conditional themselves. Consider the following example:

```
#ifdef CONFIG
  typedef int t;
#else
  typedef float t;
#endif
  t x;
```

Provided types are stored in the symbol table, a conditional entry for `x` is needed, stating that `x` is of type integer if `CONFIG` is a defined macro and is of type float otherwise. The symbol table should include both alternative types and the conditions describing which type is selected under which configuration [2], [7].

Preprocessing conditionals may not only cause conditional entries in a symbol table, but also conditional relationships between lexical scopes in a symbol table. Consider a class that inherits from a base class only in a certain configuration but does not do so in other configurations as described in the following example:

```
int x;
class A { int x; };
class B
  #ifdef CONFIG
     : A
  #endif
  { int f() { x; } }
```

A symbol table lookup for the symbol `x` in the method `f` now must find `x` in class `A` if the macro `CONFIG` is defined, but has to find the global declaration of `x` if the macro is not defined. As the definedness of the macro `CONFIG` is not determined, the lookup actually has to produce both answers. The symbol table management subsystem therefore needs to support tracking conditional relationships and looking up symbols across such relationships.

Creating a symbol table with conditional entries and relationships during attribute evaluation in general requires keeping track of the actual configurations under which attributes have certain values. The attribute evaluator generator of the DMS Software Reengineering Toolkit [5] supports generating code for this purpose by using a simple generation time flag indicating whether the attribute evaluator needs to keep track of configurations and, if so, automatically produces proper code to keep track of attribute value-condition pairs. Generated attribute evaluators are capable of calling user-defined functions with the current configuration as an additional argument, calling a function multiple times with different configurations if necessary, and of receiving and integrating sets of value-condition pairs that are provided as results of such functions.

Further analyses, e.g. control- and data-flow analyses, should support conditionals in a similar way, i.e. should produce multiple answers dependent on possible configurations of software systems.

4 Transformations

The syntactic and semantic information about source code collected without expanding preprocessing directives provides a strong foundation for correctness preserving transformations on source code using such directives. If the result of applying transformations is to be used for further maintenance of the system, as is the case e.g. for refactorings and for migrations, preserving preprocessing directives is required, as it would be unacceptable to lose the configuration information represented by preprocessing conditionals or the abstract idioms represented by macros.

Preserving such directives also enables transformations that would be impossible otherwise. E.g., obsolete configurations could be automatically removed from source code [4], macros representing constant expressions could be replaced by constant declarations or enumeration constants [3], or refactorings could involve preprocessing conditionals and macros [6].

Explicit inclusion of preprocessing conditionals into the grammar allows writing many such transforms in the native syntax of the underlying programming language. E.g., the following rule removes a preprocessing conditional the condition of which is always false:

```
rule simplify_pp_if(e,s1,s2)
      :statement->statement
  = "#if e
       \s1
     #else
       \s2
     #endif"
  -> s2
  if is_false(e).
```

On the other hand, rules for which preprocessing conditionals are of no interest are best formulated without regards to them. However, this might result in an otherwise appropriate rule not being applicable due to a preprocessing conditional in the source code that is badly placed with respect to the rule. Consider the following example:

```
x
#ifdef CONFIG
  +0
#endif
```

A simple rule rewriting `x+0` to `0` for eliminating the addition will not apply due to the presence of the preprocessing conditional around `+0`.
This can be alleviated by moving preprocessing conditionals such that they do not prevent application of a transform, e.g. by adding explicit rules. However, this could result in a proliferation of rewrite rules.

Alternatively, with sufficient information about preprocessing conditionals a rule application engine could automatically move the conditional if (and only if) necessary in order to apply a rule.

Additional problems arise when the applicability of a rule depends on semantic information, e.g. the type of a variable, which itself is dependent on the configuration of the software system. Then the rule application engine will have to add a proper preprocessing conditional on the fly for a code fragment that might not have contained any preprocessing conditionals and rewrite only the branch for which the variable has the proper type.

For languages supporting macros, transformations will frequently be applicable to code fragments involving expansions of macro calls. This implies, that transformations will require the respective macro definitions referred to by the macro calls to be transformed too, in order to achieve the desired effect on the source code.

For purely syntactic transforms that do not cross macro boundaries, it might be sufficient to apply them to a syntax tree representing the macro body of the macro definitions. However, in general, different macro calls referring the same macro definition will require different "transformed" macro definition in order to deal with rules crossing macro boundaries or with rules the applicability of which depend on semantic conditions.

This problem suggests that macros should initially be "disregarded" during the transformation process itself, i.e. the transformations are applied using expansions of the macro calls, except for keeping track of locations of macro calls and macro arguments.

In a follow-up step, new macro definitions can be constructed starting from the transformed expansions of macro calls by deriving a candidate macro definition for each macro call and then combining the resulting set of macro definitions. In general, this may result in multiple macro definitions for a single macro definition in the original source code, which can be handled by generating variant names for the macros.

This approach has been applied successfully in a JOVIAL [11] to C translator developed by Semantic Designs.

5 Conclusions

Preprocessing directives are ubiquitous in many successful programming languages. As has been discussed above, such directives impose unique challenges for developing transformation systems that are capable of evolving source code in such languages without losing the configuration information or abstractions represented by the preprocessing directives.

The DMS Software Reengineering Toolkit [5] provides a foundation for meeting these challenges. It has been used among others to implement a JOVIAL to C translator that preserves well-structured macros, C and C++ parsers that preserve preprocessing directives, and a name and type resolver for C++. This analyzer is currently being extended in order to support preprocessing directives. Though further work is necessary, the current state suggests that dealing with preprocessing directives within parsers and analyzers is indeed feasible.

The DMS Software Reengineering Toolkit is still lacking support for native syntax rules inducing preprocessing conditionals when transforming source code. However, Semantic Designs intends to implement such an extension in the near future.

References

[1] R.L. Akers, I.D. Baxter, and M. Mehlich. *Re-Engineering C++ Components via Automatic Program Transformation.* Symposium on Partial Evaluation and Program Manipulation, pp. 51-55. ACM Press, 2004.

[2] L. Aversano, M. Di Penta, and I.D. Baxter. *Handling Preprocessor-Conditioned Declarations.* International Workshop on Source Code Analysis and Manipulation, pp. 83-92. IEEE Computer Society, 2002.

[3] G.J. Badros. *PCp^3: A C Front End for Preprocessor Analysis and Transformation.* Master's Thesis. University of Washington, Department of Computer Science and Engineering, 1997.

[4] I.D. Baxter and M. Mehlich. *Preprocessor Conditional Removal by Simple Partial Evaluation.* Working Conference on Reverse Engineering, pp. 181-290. IEEE Computer Society, 2001.

[5] I.D. Baxter, C. Pidgeon, and M. Mehlich. *DMS: Program Transformation for Practical Scalable Software Evolution.* International Conference on Software Engineering, pp. 625-634. IEEE Computer Society, 2004.

[6] A. Garrido and R. Johnson. *Challenges of Refactoring C Programs.* International Workshop on Principles of Software Evolution, pp. 6-14. ACM Press, 2002.

[7] A. Garrido and R. Johnson. *Integrating Preprocessing Conditionals in CRefactory.* Submitted to: International Conference on Software Engineering, 2005.

[8] A. Garrido and R. Johnson. *Refactoring C with Conditional Compilation.* International Conference on Automated Software Engineering, pp. 323-326. IEEE Computer Society, 2003.

[9] B. Kullbach and V. Riediger. *Folding: An Approach to Enable Program Understanding of Preprocessed Languages.* Working Conference on Reverse Engineering, pp. 3-12. IEEE Computer Society, 2001.

[10] A. Sellink and C. Verhoef. *Native Patterns.* Working Conference on Reverse Engineering, pp. 89-103. IEEE Computer Society, 1998.

[11] *JOVIAL (J73).* United States Air Force, Military Standard MIL-STD-1589C, 1984.

[12] *Programming Languages – C++.* International Standard ISO/IEC 14882, 1998.

Session 3: Software Transformations

Coupled Software Transformations

— Extended Abstract —

Ralf Lämmel
VUA & CWI, Amsterdam, The Netherlands

Abstract

We identify the category of coupled software transformations, which comprises transformation scenarios involving two or more artifacts that are coupled in the following sense: transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished. We describe the essence of coupled transformations. We substantiate that coupled transformation problems are widespread and diverse.

1. De nition of the subject matter

Everyone is used to two common categories of software transformations: type-preserving transformations (also called rephrasing in [15], but other terms are around as well) vs. type-changing transformations (mostly called translations in the literature). We use the term *type* as a placeholder for format, grammar, schema, language, meta-model, and others. (Here, we do not necessarily restrict ourselves to context-free structure.) A *type-preserving transformation* asserts the same type for input and output. For instance, program optimisers and program normalisers are of that kind. A *type-changing transformation* maps data according to one type to data according to another type. For instance, language compilers, application generators, and PIM-to-PSM transformations in MDA are of that kind.

Many transformation scenarios are of a more complex shape. Most notably, one often ends up wanting coupled transformations — the subject of this paper. By this, we mean that ...

> ... *two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished.*

We note that any kind of coupled transformation problem must instantiate the notion of *consistency* for the involved kinds of artifacts. A coupled transformation starts from and nishes with a consistent conglomeration of artifacts. Any kind of coupled transformation problem must also instantiate the notion of *reconciliation*, which de nes how transformations of one artifact affect all the other artifacts.

We emphasise that we use the term *software transformation* rather than the more restrictive term program transformation. That is, we do not restrict ourselves to the transformation of source code. Hence, transformations of data, non-executable speci cations, grammars, meta-models, documentation, and other artifacts are included.

We also emphasise that we do not restrict ourselves to information-preserving (or semantics-preserving) transformations because enhancements or reductions should be included as well. For instance, *evolutionary transformations* require such generality; see the various transformation properties for (rule-based) programs in [19].

2. An example related to software evolution

Consider an information system that uses a relational database for data management, and all functionality is implemented in 2nd – 4th generation languages ("2–4GLs"). We are faced with artifacts such as the following: the relational model, which is implemented as the database schema in the database; 4GL forms for user-interface components; 4GL reports; SQL or PL/SQL fragments that contribute to 4GL sources; embedded SQL code in the 2–3GL code; 2–3GL programs with data structures that rehash some of the relational model.

There are various evolution scenarios that call for coupled transformations. For instance, we might face a change request that is phrased as a modi cation at the level of the database schema. This primary modi cation must be completed by a database instance mapping. Furthermore, all the 2–4GL code is likely to require modi cation as well. So we have to adapt SQL and PL/SQL snippets, we have to adapt embedded SQL code, and even native 2–4GL code because of the way it commits to the database schema.

All these adaptations are coupled. Consistency is here about the use of the *same* relational model in the different code artifacts. So this is de nitely a coupled transformation *problem*. It is a different question whether or not one succeeds to provide an effective implementation of the scenario, which would need to include an operational reconciliation for all the code artifacts relative to an evolving database schema.

3. The purpose of this text

The general category of coupled software transformations has not been identi ed previously — even though speci c transformation techniques do exist, and quite some amount of related research is being pursued; see, e.g., [5, 11, 26, 29, 22, 17, 10]. So giving nally a name to this category is perhaps useful as such. We go further than that: in Sec. 4, we will describe the essence of coupled transformations.

Coupled transformation problems are ubiquitous; they are encountered in various disciplines of computer science, e.g., in language processing, generative programming, automated software engineering, software re-engineering, model-driven architecture, and database re-engineering. In Sec. 5, we will enumerate some problem domains in which coupled transformations are relevant. The understanding of coupling differs radically for these domains.

4. The essence of coupled transformations

Without loss of generality, we will consider reconciliation for *two* artifacts. Let A and B be the types of the artifacts. We assume a consistency relation c on A and B. We are given two concrete artifacts $a : A$ and $b : B$ such that $c(a, b)$ holds. We consider a type-preserving transformation on A, denoted by g, and we apply this transformation to a such that we obtain $a' = g(a)$. Then, the reconciliation issue is about determining a suitable b' such that $c(a', b')$ holds. We summarise selected reconciliation options in Fig. 1; continuous arrows visualise transformations; dashed arrows visualise consistency claims.

The rst option in the gure, *no reconciliation*, is merely there to provide a good starting point for the discussion. If g is known to be restricted such that a is changed without challenging consistency, then we can just keep b — as is. For instance, using SQL's data manipulation language on a database instance does not affect the underlying database schema. So this sort of restricted instance transformation does not trigger a schema transformation. Clearly, the inverted situation, where the database schema is transformed, is not covered by this trivial option.

The second option in the gure, *degenerated reconciliation*, is still trivial. We assume that the concrete artifacts of type B are derivable from the concrete artifacts of type

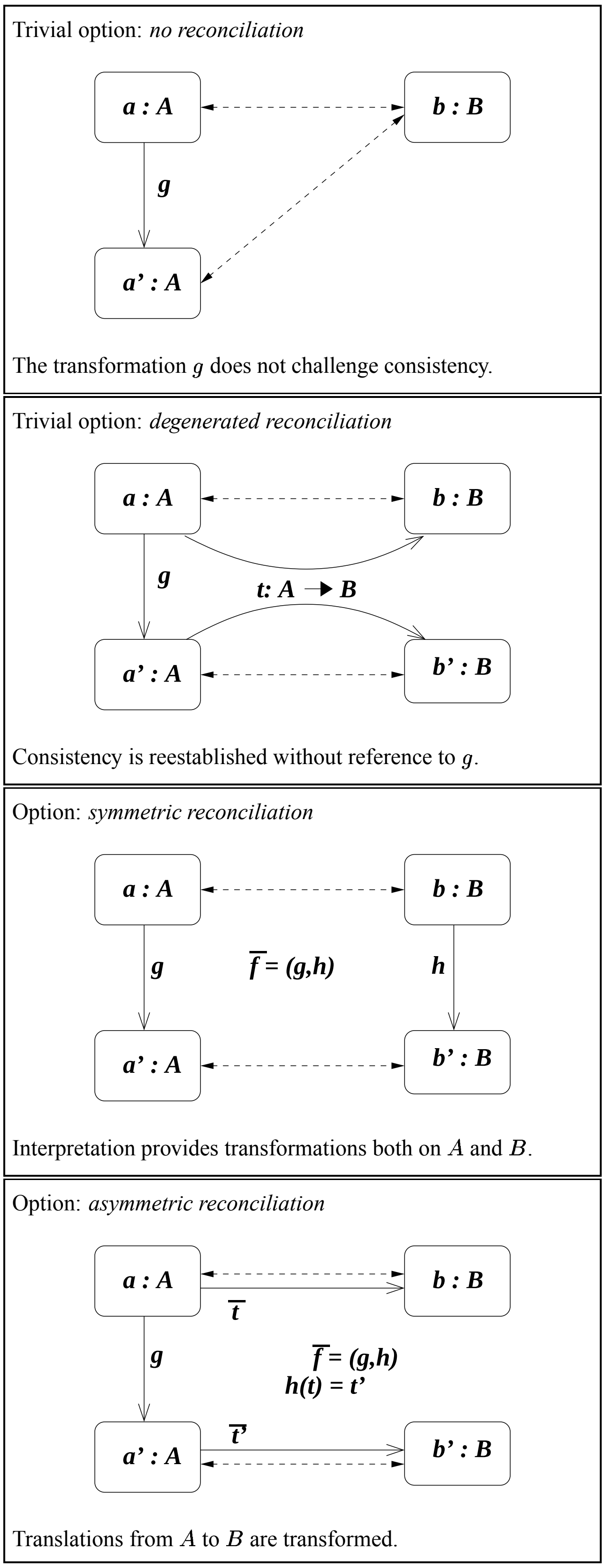

Trivial option: no reconciliation
a : A
b : B
g
a' : A
The transformation g does not challenge consistency.
Trivial option: degenerated reconciliation
a : A
b : B
g
t: A → B
a' : A
b' : B
Consistency is reestablished without reference to g.
Option: symmetric reconciliation
a : A
b : B
g
f = (g,h)
h
a' : A
b' : B
Interpretation provides transformations both on A and B.
Option: asymmetric reconciliation
a : A
b : B
t
g
f = (g,h)
h(t) = t'
t'
a' : A
b' : B
Translations from A to B are transformed.

Figure 1. Selected reconciliation options

A — by means of a translation t. For instance, consider the implementation of a domain-speci c language (DSL) via code generation. In a simple case, we can transform the DSL code, and the generated code is simply regenerated. However, if the generated code can be customised, and such adaptations are required to survive regeneration, then proper reconciliation has to be faced.

The third option in the gure, *symmetric reconciliation*, covers a meaningful subcategory of coupled transformations. Here we start from a transformation *description* f, which is phrased in a transformation language. The interpretation of the description f, denoted as $\overline{f}$, provides two actual transformations, one on A, and another on B. For instance, the reconciliation of a database instance in reply to adaptations of a database schema can be covered by this option [12]. (Likewise, XML documents must be updated when the underlying DTDs or XML schemas are adapted [21].) The applicability of this option requires a genuine de nition of the transformation language. A critical issue is the inclusion of all controls into f that are eventually needed for either A or B. For instance, transforming a database schema, such that a NOT NULL column is added, requires the speci cation of a default value — even though this value is not essential for the schema transformation, but only for the instance mapping.

The fourth option in the gure, *asymmetric reconciliation*, is based on the assumption that we have access to an actual translation of a to b. In fact, we require again a *description* t of this translation in terms of a dedicated transformation language. Then, the actual translation from a to b is the interpretation $\overline{t}$ of the description t. The description t can be seen as a means to capture the derivation history of b when related to a. Furthermore, this translation t manifests the consistency of a and b. As in the case of symmetric reconciliation, f is again phrased in a transformation language. This time, the interpretation of f, provides an actual transformation on A and a transformation on translation descriptions (such as t). The aforementioned survival problem of customisations in code that is generated from a DSL program is handled by this option. That is, we use a translation description to maintain a link between DSL code and generated code. The translation description records the customisation of the generated code. When the DSL program is adapted, the recorded customisations can be re-executed (modulo adaptation) on the regenerated code. This option involves two critical issues. Firstly, we have to identify a language for translation descriptions that is t for modelling the derivation of artifacts according to the problem domain at hand. Secondly, we have to de ne the primary transformation language such that its effect can also be transposed to the translation descriptions.

We make no claim of completeness regarding this list of options. For instance, one can think of *reconciliation by matching*, where the mere consistency relation is complemented by a metric that measures the consistency "distance" between two artifacts. That is, given a', and its consistency distance from b, we could gradually adapt b until full consistency is reestablished. We favour symmetric and asymmetric reconciliation because these options inherently employ the structure of the primary transformation. However, even this criterion might be amenable to other realisations. Also, there are presumably a number of possible re nements for the two favoured options, and they might also be mixed. Future work is needed to deliver a comprehensive set of more detailed options.

5. Typical problem domains

We will now list scenarios for coupled transformations. We identify the artifacts and transformations of interest as well as the relevant instances of the notions consistency and reconciliation. This list is by no means complete. The degree of the formal and technical mastery of coupled transformations differs very much per problem domain. An example of a well-understood kind of coupled transformation is the joint transformation of database schema and database instance during database re-engineering [12]. By contrast, the techniques for updating language processors for programming languages in reply to an evolving syntax or semantics are still to be discovered; see [20] for some ideas.

A list of coupled transformations scenarios follows.

Consistency maintenance in cooperative editing Distributed editing of the same content requires synchronisation [7, 28]. Depending on details, either the local copies of the content or the local session state are considered the artifacts subject to coupled transformation. There can be any number of such artifacts, but they happen to be all of the same type. Editing actions de ne the primary transformation language. Reconciliation of a given user view means to incorporate all pending editing actions that were emitted by other users. Consistency means that all views agree on the content. A speci c challenge is that remote editing actions should be incorporated only at de nite points in time, when their effect on the local view and any necessary con ict resolution will be acceptable for the user.

Consistency maintenance in software modelling In software modelling with UML one uses different kinds of structural and behavioural diagrams such as use case diagrams, class diagrams, state diagrams, and sequence diagrams. Consistency of a multi-diagram software model means that the different diagrams do not disagree on each other in those areas where they overlap [18, 14]. Consistency must be maintained along the evolution of UML models. As a simple form of transformation, one can consider refactorings of UML class diagrams. (Refactorings were originally introduced for the transformation of object-oriented programs, but they have been instantiated for UML diagrams as well [4, 25].)

Co-evolution of design and implementation The system design and the actual implementation should be coupled throughout continuous system maintenance and enhancement. The idea of a methodology and technology for co-evolution is that the coupling must be operationalised or at least checked [6, 31, 8] because design and implementation diverge otherwise. One way to operationalise the coupling is to generate the implementation from the design, modulo provisions for allowing editing at the source-code level and for pushing back implementational changes into the design. The operationalisation can also work the other way around if the additional design information becomes an integral part of the actual code.

View-update translation When updates are allowed at the level of database views, then such updates need to be translated back to the underlying database [2, 9]. Such view-update translation is clearly an instance of asymmetric reconciliation. The view-update problem for databases has been generalised in the recent work on bidirectional transformations between data representations of different levels of abstraction [10, 13] (cf. concrete and abstract views). The view-update problem has also been encountered, in some form, in functional programming, when pattern matching and building is to be provided for abstract datatypes rather than concrete algebraic datatypes [30, 3, 23].

Intentional programming and uid AOP In Simonyi's intentional programming [27, 1], the programmer can specify domain-speci c abstraction forms, while simultaneously recording domain-speci c optimisations that may apply to such new abstractions. Programmers can browse and edit (or transform) programs using different syntaxes enabled by the competing abstraction forms. An intentional programming system would take care of the coupling between the external syntaxes and the internal abstract syntax. A similar situation applies to uid AOP [16] — a strong form of aspect-oriented programming, where programs cannot just be edited to ful l crosscutting concerns, but programs can even be re-sliced according to different views.

Reconcilable model transformation According to OMG's model-driven architecture (MDA [24]), software development starts from a platform-independent software model (PIM), which is then re ned into a platform-speci c model (PSM) by semi-automatic transformations. To this end, model-driven approaches employ meta-models for platform-independent models, for platforms, for platform-speci c models. MDA approaches also tend to employ annotations for driving the the PIM-to-PSM mappings. The basic MDA approach emphasises the operationalisation of the PIM-to-PSM mapping, which is a type-changing transformation. A strong version of MDA would require coupling between all involved models and meta-models [8]. For instance, the modi cation of a platform model should allow for the reconciliation of all actual PIMs that refer to this platform.

Representations in software re-/reverse engineering Software reverse engineering employs problem-oriented abstraction layers, starting from a low-level source-code model, with less code- or language-speci c representations in between, and possibly complemented by high-level architectural descriptions at the top. Coupling concerns the mappings between the layers. These mappings need to be traceable in order to enable the navigation between layers. Likewise, software re-engineering can take advantage of extra intermediate program representations, e.g., PDG or SSA for control o w or data o w dependencies. When re-engineering transformations are expressed at the level of intermediate formats, then these transformations still need to be mapped back to the concrete source code. Yet other forms of coupling deal with the preservation of preprocessing directives, and other low-level source-code properties.

6. Final remark

We have identi ed the notion of coupled software transformations. We have collected and integrated some basic material on the subject, while we have refrained from a discussion of technical details as they arise in speci c coupled transformation scenarios and speci c conceptual frameworks for coupled transformations.

It is very rewarding to understand that software transformations can exhibit more structure than being organised in terms of type-preserving or type-changing functions. We can have transformations on transformations on ...

Acknowledgement

The author gratefully acknowledges inspiring discussions with James R. Cordy and Andreas Winter in the context of designing the Dagstuhl seminar 05161 "Transformation techniques in software engineering".

References

[1] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional

programming. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 114–123. IEEE Computer Society Press, 1998.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[3] F. Burton and R. Cameron. Pattern Matching with Abstract Data Types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[4] G. Butler and L. Xu. Cascaded refactoring for framework. In *Proc. Symposium on Software Reusability*, pages 51–57. ACM Press, 2001.

[5] A. van Deursen, P. Klint, and F. Tip. Origin Tracking. *Journal of Symbolic Computation*, 15:523–545, 1993.

[6] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of Object-Oriented Software Design and Implementation. In *Proc. International Symposium on Software Architectures and Component Technology 2000*, 2000.

[7] C. Ellis, S. Gibbs, and G. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[8] J.-M. Favre. Meta-models and Models Co-Evolution in the 3D Software Space. In *Proc. International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA'03)*, 2003.

[9] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, 1988.

[10] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, 2003. Revised April 2004.

[11] J.-L. Hainaut, C. Tonneau, M. Joris, and M. Chandelon. Schema Transformation Techniques for Database Reverse Engineering. In *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, 1993. E/R Institute.

[12] J. Henrad, J.-M. Hick, P. Thiran, and J.-L. Hainaut. Strategies for Data Reengineering. In *Proc. Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Computer Society Press, 2002.

[13] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Proc. ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189. ACM Press, 2004.

[14] Z. Huzar, L. Kuzniarz, G. Reggio, J. Sourrouille, and M. Staron. Consistency Problems in UML-based Software Development II, 2003. Workshop proceedings; Research Report 2003:06.

[15] M. d. Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. v. d. Brand and D. Parigot, editors, *Proc. Workshop on Language Descriptions, Tools and Applications (LDTA'01)*, volume 44 of *ENTCS*. Elsevier Science, 2001.

[16] G. Kiczales. The Fun has Just Begun. AOSD'03 Keynote Address, available from `http://www.cs.ubc.ca/~gregor`, 2003.

[17] J. Kort and R. Lämmel. Parse-Tree Annotations Meet Re-Engineering Concerns. In *Proc. International Workshop on Source Code Analysis and Manipulation (SCAM'03)*, Amsterdam, 2003. IEEE Computer Society Press.

[18] L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar. Consistency Problems in UML-based Software Development, 2002. Workshop proceedings; Research Report 2002:06.

[19] R. Lämmel. Evolution of Rule-Based Programs. *Journal of Logic and Algebraic Programming*, 60–61C:141–193, 2004. Special Issue on Structural Operational Semantics.

[20] R. Lämmel. Evolution scenarios for rule-based implementations of language-based functionality. In L. Aceto, W. Fokkink, and I. Ulidowski, editors, *Proc. Workshop on Structured Operational Semantics (SOS'04)*, ENTCS. Elsevier, 2004. 20 pages. To appear.

[21] R. Lämmel and W. Lohmann. Format Evolution. In J. Kouloumdjian, H. Mayr, and A. Erkollar, editors, *Proc. Re-Technologies for Information Systems (RETIS'01)*, volume 155 of *books@ocg.at*, pages 113–134. OCG, 2001.

[22] A. Malton, K. Schneider, J. Cordy, T. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proc. International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society Press, May 2001.

[23] G. S. Novak Jr. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, 1995.

[24] OMG. Model Driven Architecture, 2001–2004. web portal `http://www.omg.org/mda/`.

[25] K. Rui and G. Butler. Refactoring use case models: the metamodel. In *Proc. Twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 301–308. Australian Computer Society, Inc., 2003.

[26] A. Schürr. Speci cation of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop*, volume 903 of *LNCS*, pages 151–163, Herrsching, Germany, 16–18 June 1994. Springer-Verlag.

[27] C. Simonyi. The death of programming languages, the birth of intentional programming. Technical report, Microsoft, Inc., Sept. 1995. Available from `http://citeseer.nj.nec.com/simonyi95death.html`.

[28] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[29] E. Visser. Strategic pattern matching. In P. Narendran and M. Rusinowitch, editors, *Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *LNCS*, pages 30–44, Trento, Italy, July 1999. Springer-Verlag.

[30] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proc. Principles Of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.

[31] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

Automatically Transforming GNU C Source Code

Christopher Dahn, Spiros Mancoridis
Department of Computer Science
Drexel University, Philadelphia, PA, USA
{chris.dahn}@computer.org
{mancors}@drexel.edu

Abstract

To perform automated transformation techniques on production quality GNU C source code, non-trivial normalizations must occur. The syntax of GNU C contains inherent ambiguity that must be overcome. The techniques used by an automated transformation tool, Gemini, are presented.

1. Introduction

The process of performing automated source code transformation on GNU C code is complicated. GNU C is a flexible language that allows developers to write code in a variety of ways that are all semantically equivalent. This flexibility imposes challenges to software maintainers who wish to perform source transformation automatically.

In this paper we look at techniques used by the Gemini [4, 3] tool, developed at Drexel University, which performs source code transformation, automatically, on GNU C source code. The tool transforms GNU C arrays into pointers. The transformation must take into account the presence of non-C input (e.g., preprocessor statements) and an ambiguous language definition, while maintaining semantic equivalence of the input and output source code.

Gemini uses the GNU C Compiler (GCC) to perform preprocessing of GNU C source code, and TXL [1, 2, 5] to perform the automated transformation.

The remainder of this paper is structured as follows: Section 2 discusses some of the specific challenges posed by GNU C, Section 3 discusses the details of the TXL transformations used to normalize the source code, and Section 4 discusses the conclusions we can draw from the study.

2. Challenges of GNU C

2.1. Preprocessor Input

Gemini cannot transform GNU C source code that contains macros, since macros are resolved at compile-time. Therefore, Gemini passes the source code through the GCC preprocessor before transformation.

Gemini's preprocessing performs two primary tasks. First, it pre-processes the original source code with the GCC preprocessor. Second, it removes all of the preprocessor statements from the result.

The output from the GCC preprocessor contains preprocessor statements which must be removed. These statements are used by GCC to reconstruct the original file scopes to resolve duplicate global declarations. It isn't possible to define a grammar that includes GNU C preprocessor statements everywhere they may occur. Generally, these statements can occur anywhere in the input. Creating an ambiguous grammar to describe them leads to parser generation problems.

2.2. C Declaration Ambiguity

GNU C is a flexible language. Its flexibility results in a complicated grammar. One of the complications found in GNU C grammars is embodied by an ambiguity in how declarations and statements are parsed.

In C, users can alias types using a `typedef` declaration. The presence of user-defined types causes GNU C grammars to be ambiguous about how to parse statements and declarations. Figure 1 and Figure 2 show possible grammars for a GNU C declaration and statement, respectively.

An example of a declaration described by the first grammar is, `my_type (foo);`. The statement grammar describes function calls with one argument, such as, `printf (foo);`. In the example declaration, the declarator type, `my_type`, is a user defined type alias (i.e., previously defined with a `typedef`).

```
<declaration> ::= <type> <declarator> ';'
<type> ::= <nativetype> | <typealias>
<nativetype> ::= int | char | void | float | double
<typealias> ::= <identifier>
<declarator> ::= <identifier> | '(' <identifier> ')'
```

Figure 1. BNF for one possible declaration grammar.

```
<statement> ::= <expression> ';'
<expression> ::= <postfixexpression>
                 <postfixextension>
<postfixexpression> ::= <identifier>
<postfixextension> ::= '(' <identifier> ')'
```

Figure 2. BNF for one possible statement grammar.

At the syntax level, both of these GNU C constructs have the same parse tree. A compiler would differentiate the former as a declaration and the latter as a statement during semantic analysis of the parse tree. However, TXL only operates on input syntax, hence the transformation that Gemini performs must first take steps to overcome this grammatical ambiguity.

3 TXL Transformation

The TXL transformation is composed of eleven steps. The relevant steps are outlined in this paper. Only one of the steps actually performs the array-to-buffer transformation, the other steps are required to normalize the way GNU C declarations are parsed (due to the flexibility of GNU C syntax), to account for the declaration/statement ambiguity previously discussed, and to ease the final array-to-buffer transformation.

Figure 3 shows an example GNU C program. This program is used to illustrate the eleven transformation steps. The lines affected by the current transformation step will be highlighted.

Step 1: Expand Shorthand Types. The first step is to find all declarations in the GNU C program that have been declared using a shorthand form of the `int` type. GNU C allows the integer qualifiers `long`, `short`, `unsigned`, and `signed` to be used as the declarator type. In this case, the GNU C grammar used with TXL will cause it to parse the declaration without a type. Hence, the parse tree for the declaration will have only a declaration type qualifier. In order to normalize declaration types, all declarations with

```
#include <stdio.h>

typedef struct
{
    char c;
} (char_buf)[5], char_st;

int main(int argc, char **argv)
{
    char *test = "test";
    char_buf (buffer) =
    {
       [0].c='t', [1].c='e', [2].c='s',
       [3].c='t', [4].c='\0'
    };
    char_buf buffer2;
    struct st
    {
       unsigned i;
    } st = {42};
}
```

Figure 3. Sample source code to be trans formed.

```
struct st
{
    unsigned int i;
} st = {42};
```

Figure 4. Step 1: Expand shorthand types.

only type qualifiers are expanded to have an explicit type of `int`, as shown in Figure 4.

Step 2: Name Anonymous Elaborated Types. Elaborated types, or enumerated types, are user-defined types. In C, an elaborated type is either a `struct`, `union`, or `enum`. Figure 6 shows the two ways that an elaborated type can be declared.

In Figure 6, the type of the variable `foo` is an anonymous elaborated type. These types can only be used for a single declaration. That is, no other declaration can have the same type as `foo`, even if the type definitions of both declarations are identical.

The declaration of `foo2` defines a named elaborated type. This form is often used when the programmer intends to declare other variables with the same type. A subsequent declaration cannot declare the body of the `struct` again, rather, it references the `struct` by name (e.g., `struct st bar;`). In this case, the types of `foo2` and `bar` are equivalent.

```
typedef struct txl_WasAnonElabType1
{
    char c;
} (char_buf)[5], char_st;
```

Figure 5. Step 2: Name anonymous elabo rated types.

```
struct
{
    int i;
} foo;

struct st
{
    int i;
} foo2;
```

Figure 6. Declaration of elaborated type.

In this step, a unique name is assigned to every anonymous elaborated type. Each name is unique in the namespace of the file, as shown in Figure 5. That is, the generated name will not be found anywhere else in the file undergoing transformation. Assigning a unique name to an anonymous elaborated type does not change the semantics of the program, since a unique name guarantees that the type has not been used anywhere else.

Step 3: Expand Declarator Lists. After all anonymous elaborated types in the program have been named, lists of declarators can be expanded, as shown in Figure 7. This is performed after naming anonymous elaborated types so that the proper type is distributed to each expanded declarator. If the anonymous elaborated type was distributed, then none of the new declarators would be equivalent in type which could cause type mismatch errors at compile time.

Step 4: Remove Extraneous Parentheses. Removing unnecessary parentheses from declarators, as shown in Figure 8, allows the assumption that a valid (i.e., transformable) declaration will never contain parentheses.

Generally, any number of parentheses may be added around a declarator without changing the semantics of the declaration. However, each set of parentheses creates a unique parse tree for the declaration. In particular, adding extraneous parentheses to declarations is the cause of the grammar ambiguity described in Section 2.2.

Due to the ambiguity, it is not possible to determine automatically if TXL has erroneously parsed a statement as a declaration. Removing the parentheses from a function call will always produce a syntax error at compile time. Hence,

```
typedef struct txl_WasAnonElabType1
{
    char c;
} (char_buf)[5];
typedef struct txl_WasAnonElabType1
    char_st;
```

Figure 7. Step 3: Expand declarator lists.

```
typedef struct txl_WasAnonElabType1
{
    char c;
} char_buf[5];

int main(int argc, char **argv)
{
    char *test = "test";
    char_buf buffer =
    {
       [0].c='t', [1].c='e', [2].c='s',
       [3].c='t', [4].c='\0'
    };
```

Figure 8. Step 4: Remove extraneous paren theses.

this process must be conservative in its selection. Specifically, if a declaration's type is an identifier, then it is assumed to be a statement (i.e., erroneously parsed) and the parentheses are not removed. This conservative approach allows the transformation to normalize all `typedef` declarations for Step 6. Subsequent steps will allow us to perform this step again to remove remaining false negatives (i.e., variable declarations that should have matched but were skipped).

Step 5: Unique Local Elaborated Types. This step, shown in Figure 9, ensures that when Step 6 removes all `typedef` aliases, local declarations will not have types that alias a globally defined elaborated type. This can lead to errors, since `typedef` flattening will invalidate scope protection of type declarators.

To make the types unique, the identifier of each elaborated type defined in the body of a function is replaced with a new, unique identifier. This takes into account forward declarations of elaborated types. Once a unique identifier has been chosen, all references to the previous type are replaced with references to the new, unique type.

Step 6: Flatten `typedef` aliases. This step is critical in determining which declarations to transform. This step also resolves any of the remaining ambiguities described in Section 2.2 and Step 4. The ambiguity is resolved since this

```
struct st1
{
    unsigned int i;
} st = {42};
```

Figure 9. Step 5: Unique local elaborated types.

```
struct txl_WasAnonElabType1 buffer[5] =
{
    [0].c='t', [1].c='e', [2].c='s',
    [3].c='t', [4].c='\0'
};
struct txl_WasAnonElabType1 buffer2[5];
```

Figure 10. Step 6: Flatten `typedef` aliases.

step replaces all identifiers that are type aliases with the type that they alias. Hence, all type aliases are resolved to the native or elaborated types they alias, as shown in Figure 10.

To flatten the `typedef` aliases, each `typedef` is visited once, starting at the top of the file being transformed. It can be assumed that the very first `typedef` in the file creates an alias to a native or elaborated type since it would be a compilation error to declare a `typedef` for an alias that has not been defined yet. For each `typedef` that is visited, every declaration below that `typedef` is visited to determine if the type of the declarator is the current `typedef` alias. If a match is found, the type alias of the declarator is replaced. This will also flatten `typedefs` themselves. For example, if a `typedef` defines an alias for a previously defined alias (e.g., `typedef my_char my_char2;`, where `my_char` is an alias for `char`), then it will be transformed to alias the original type (e.g., `typedef char my_char2;`). Hence, a declaration can have its type changed multiple times as the native and elaborated types are propagated down the parse tree.

A `typedef` can also be used to declare a type that is an array, as illustrated by the global type `char_buf` in Figure 10. In this case, the dimensions of the array given in the `typedef` declarator are propagated along with the type that the `typedef` aliases. The array dimensions are appended to the end of the declaration being flattened to preserve the semantics of the original source code. An example of this is shown in Figure 10.

Step 7: Unique Global Declarations. As described in Section 2.1, all preprocessor statements are removed from the file prior to transformation by TXL. However, this can often lead to source code that is no longer valid semantically. The preprocessor statements act as markers to GCC about where the original files, included via `#include` directives, start and stop. This allows GCC to track the file scopes that prevent re-declaration errors of function prototypes from occurring. For example, a file can include the GNU C standard library to have access to the function prototype for `malloc()`. If that same file also declares its own function prototype for `malloc()` (e.g., as an `extern`), then a re-declaration error can occur at compile time.

To account for this change in the file, Gemini scans over the global function and variable declarations in the file and removes duplicates. The end result is that only one function prototype for each function definition and only one `extern` declaration for each variable will remain at the end of the transformation.

4. Conclusion

The syntax of GNU C is very flexible. It gives the developer many ways to declare variables and types that are semantically equivalent, but the cost of this flexibility is ambiguity while parsing C source code.

Production GNU C code is difficult to automatically transform due to the presence of GNU C preprocessor statements. These statements give the developer a way to create macros and make other compile-time decisions. However, since these statements are not part of the GNU C syntax, they must be removed prior to transformation. Since the compiler is expecting these statements, removing them changes the semantics of the source code.

To perform automatic transformation on GNU C source code, many normalizing efforts must transpire. The normalization process ensures predictable formats of declarations and statements, and removes ambiguities that would otherwise make automated transformation very difficult.

References

[1] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source Transformation in Software Engineering Using the TXL Transformation System. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[2] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. In *Proceedings from IEEE International Conference on Computer Languages*, October 1988.

[3] C. Dahn and S. Mancoridis. Using Program Transformation to Secure C Programs Against Buffer Overflows. In *Proceedings of the Working Conference in Reverse Engineering (WCRE'03)*, 2003.

[4] Gemini, http://serg.cs.drexel.edu/gemini/.

[5] TXL homepage, http://www.txl.ca/.

A Process for Transforming Portions of Existing Software for Reuse in Modern Development Approaches

Andrew Le Gear, Jim Buckley, Seamus Galvin, Brendan Cleary
University of Limerick
Castletroy, Limerick, Ireland
{andrew.legear, jim.buckley, seamus.galvin, brendan.cleary}@ul.ie

Abstract

As development costs spiral upwards, creating ever more complex systems from scratch seems implausable. Component-based development offers the potential to tackle this issue through reuse. However, this approach must accommodate the exploitation of the embedded knowledge that already exists in non-component-based code.

This paper proposes an extension to the software reconnaissance technique [15] that identi es reusable code within existing software systems. Once identi ed, the report shows how this code can be transformed for use within a component-based development paradigm.

1. Introduction

Software Reconnaissance is a dynamic analysis, redocumentation technique, that, through the acquisition of source code coverage pro les [1] (yielded by exercising carefully selected test cases on instrumented code) creates a mapping between program features and the software elements that implement them [15]. In Norman Wilde's seminal paper on software reconnaissance he remarks on the potential worth of the source code shared across features exhibited by a system [15]. Our hypothesis is that such code could provide the basis for reuse, due to the self-evident assertion that, if code is being reused in a running system then it must be reusable, at least within that context.

However, applying software reconnaissance will only identify the behavioural aspects of potentially recoverable components. We propose applying a complementary static analysis, ltered by the ouput from software reconnaissance, to determine data accesses within the software to implement this approach. As a result, not only do we identify the behavioural portions of a system for reuse, but aspects of data also. The three dimentional hyperslice across features, code and data [11] provides us with the basis for component recovery.

After code has been extracted based on this analysis, the nal step in the transformation process adapts the extracted material for use in a modern component-based development process, by applying a component wrapper [2].

The remainder of this paper is structured as follows. Section 2 explains our transformation process in detail. Section 3 describes the application of the transformation process on a scrabble emulator program. Section 4 discusses our current work in verifying the effectiveness of our transformation process. Finally, conclusions and points of discussion are voiced in section 5.

2. The Transformation Process

2.1. Software Reconnaissance

As previously introduced, software reconnaissance is a dynamic source code analysis technique and is primarily used as an aid to software comprehension [15, 6, 16]. The link between program features and code is achieved through the use of test cases [14]. This allows us to describe two relations: EXERCISES(t,e), which is *true* when the test case, t, exercises the software element, e, and EXHIBITS(t, f), which is *true* when a test case, t, exhibits the feature, f. Instrumentation may be undertaken at le, function or branch level and this choice de nes the granularity of the set of software elements outputted.

Several sets of source code elements may be calculated from these retrieved coverage pro les. Of particular interest are:

- IIELEMS(f): The set of software elements exercised by every test case exhibiting f or $\{e{:}\text{ELEMS} \mid \forall t \in T,\ \mathit{EXHIBITS}(t, f) \Rightarrow \mathit{EXERCISES}(t, e)\}$.

- UELEMS(f): The set of software elements exercised by any test case exhibiting f except for any elements that are also exercised in testcases that do not exhibit f, or $\{e{:}\text{ELEMS} \mid \exists t \in T,\ \mathit{EXHIBITS}(t, f) \wedge \mathit{EXERCISES}(t, e)\}$ - $\{e{:}\text{ELEMS} \mid \exists t \in T,\ \neg \mathit{EXHIBITS}(t, f) \wedge \mathit{EXERCISES}(t, e\}$.

- CELEMS: The software elements that will always be executed regardless of the test case or $\{e{:}\text{ELEMS} \mid \forall t \in T,\ \mathit{EXERCISES}(t, e)\}$.

The set UELEMS(f) has been shown experimentally to provide a useful starting point to begin searching when attempting to understand a particular functionality exhibited by the system, with IIELEMS(f) providing a context of use within the system for UELEMS(f) [15]. CELEMS, on the other hand, generally represents, utility code within the system that is executed every time it is run.

2.2. Exploiting Software Recconnaissance for Reuse

Using the sets described in section 2.1 the set of shared software elements across features exhibited by the system may be calculated:

$$\text{SHARED}(f) = \text{IIELEMS}(f) - \text{UELEMS}(f) - \text{CELEMS}$$

This equation yields a set that is neither utility code nor unique to a feature, but software elements shared between two or more distinct features of the system. From a reuse perspective, the SHARED set gives a genuine snapshot of software elements being reused by the running system. If software elements are being reused by distinct functionalities exhibited by the system then they may provide a useful starting point when trying to recover reusable components.

However, in order to obtain stateful components, data analysis would seem to be a necessary pre-requisite. By supplementing SHARED sets with a static analysis, the data accesses made by the software elements of the set can be revealed. This combination of shared behaviour with its corresponding data accesses provides a basis for the recovery of stateful components with reusable behaviour.

2.3. Modernising Reusable Code

To complete the code transformation process, and therefore recover a component for reuse within a modern development paradigm, the portion of the system extracted must be modi ed. Wrapping is a modi cation technique whereby source code may be supplemented to allow the system to conform with other development paradigms [2]. Here we choose to wrap identi ed reusable code as a xADL component [7]. xADL represents the state-of-the-art in architectural description languages [7, 5]. Its design is deliberately generic, and is therefore not constrained by the problem speci c nature of other ADL's. In addition, its basis in xml schemas eases both its application and its ability to be tailored to speci c usage context.

xADL describes a system in terms of its components (*ArchTypes*) and the manner in which they are assembled (*ArchStructure*). Thus the software elements it models can be used in component compositions as shown by the prototype implementations developed by [9] and [12]. In the case study described in the next section, the process of creating an ArchType using reconnaisance and xADL wrapping is described.

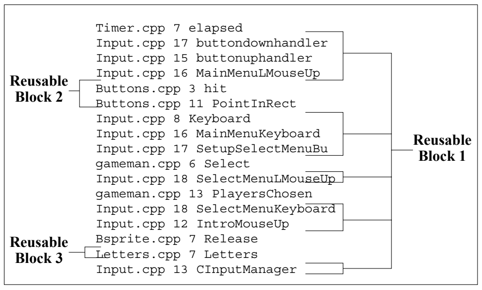
3. Case Study

We took a scrabble emulator program written in C++, approximately 8KLOC in size and distributed over 37 les, as an example. 23 features were identi ed in the program. Using this as a basis for analysis, pro les of the features were retrieved using the RECON3 tool set [13] and the software reconnaissance performed using an in-house automation of the technique.

From this output we manually examined the SHARED sets with the aid of the original developer. Figure 1 shows the shared set output for feature 8 - "Complete Player Selection." The listing contains methods shared by this feature with other features exhibited by the system. From this it was possible to identify three reusable blocks of code to form a basis for component recovery - an input manager component (reusable block 1), a button manager component (reusable block 2) and a letters component (reusable block 3). Interestingly, the rst two reusable blocks already formed part of reusable libraries intended to provide generic functionality for handling user input and button wiget management respectively. The fact that the technique identi ed code that was designed as reusable should be viewed as a step towards proof of concept.

Analysis of shared sets for other features revealed similar results. High proportions of their respective shared sets contain reusable code, sometimes in excess of 90% according to the developer. When the amount of reusable code was small, the proportion did not decrease, only the size of the set.

The third block identi ed in gure 1 provides a starting point to recover a letter-handling component by yielding the constructor for a class named *letters*. Such a component could be reused in a variety of word games beyond the domain of scrabble. We performed a manual static analysis to reveal data accesses that the letters component makes and therefore provide both behavioural and data information as



```
Timer.cpp 7 elapsed
Input.cpp 17 buttondownhandler
Input.cpp 15 buttonuphandler
Input.cpp 16 MainMenuLMouseUp
Buttons.cpp 3 hit
Buttons.cpp 11 PointInRect
Input.cpp 8 Keyboard
Input.cpp 16 MainMenuKeyboard
Input.cpp 17 SetupSelectMenuBu
gameman.cpp 6 Select
Input.cpp 18 SelectMenuLMouseUp
gameman.cpp 13 PlayersChosen
Input.cpp 18 SelectMenuKeyboard
Input.cpp 12 IntroMouseUp
Bsprite.cpp 7 Release
Letters.cpp 7 Letters
Input.cpp 13 CInputManager
```

Figure 1. The shared set for feature 8.

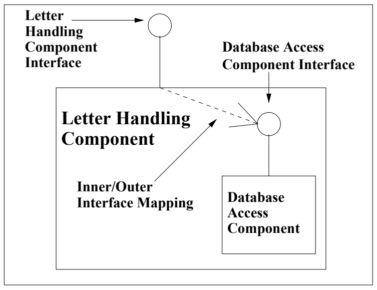

Letter Handling Component Interface
Database Access Component Interface
Letter Handling Component
Inner/Outer Interface Mapping
Database Access Component

Figure 2. Structure of the Letter Handling Component.

a basis for recovery. Due to the small size of the potential component the static analysis could be performed manually. However, we envisage employing tool automation, such as the XREF compile option [4], to garner such information in larger systems. Our analysis revealed access to the le *letters.txt*. The le is used to store the characters in use in the game as well as their value and quantity.

With behavioural and data aspects now identi ed for our potential letters component, all that remains is the application of the xADL component wrapper. For our example it is suf cient to use xADL to describe our recovered system portion as an *ArchType* which can later be used by developers in a variety of ArchStructures. First we need to gather the required information for the xADL description [7]:

- Services the code *provides*.
- Services the code *requires* to function correctly.
- *Database accesses* the code makes.
- The *implementation les* for this code.

While the SHARED set automatically provides us with a starting point for component recovery, we currently rely on the domain knowledge of the original developper to manually identify the remainder of the component interface that the SHARED set suggests. Scope for further automation exists here. After manual examination 8 *provided* services were identi ed:

- `void Initialise()`
- `void GetLetterDataFromFile()`
- `voidFillLetterArray()`
- `void ShuffleLetterArray()`
- `char GetNextLetter()`
- `voidReturnLetterToArray(char x)`
- `int ReturnValueOfLetter(char p)`
- `int NumOfTilesLeft()`

No *required* services were present as all dependencies were on the C++ framework. A single *database access* to the *letters.txt* le described earlier was present. Finally two implementation les *letters.cpp* and *letters.h* were identi ed.

Based upon this information it was decided to create two components - A *letter handling component* and a *database access component* centered around the data le *letters.txt*. The letter handling component would contain the database access component as a *sub architecture* and the services of the database access component would be made visible externally on the letter handling component's interface via an *inner-outer interface mapping* [8]. Figure 2 visually describes this structure.

After gathering this information, writing the xADL wrapper becomes trivial. This completes the component recovery and transformation process. Unfortunately due to spacial constraints it is impractical to include the wrapper in this publication, however, if the principal author is contacted directly, further information can be provided.

4. Current Work

Sections 3 demonstrates the preliminary success of our technique on an example of reasonable size. Current work, in conjunction with our industrial partner, is seeking to con rm the effectiveness of the approach on systems of a more representative, industrial scale. Initial results have been quite promising and again, seem to show similar proportions of reusable code as with the example described here. The XREF compile option is successfully being used here to generate database ACCESS and UPDATE information from the system, hence reproducing the three dimensional hyperslice [11] through the system described earlier.

However, as the size of shared sets increase so too do the problems of partitioning it into separate reusable blocks and subsequently identifying the potential component from it. We are currently investigating both re e xion modelling [10] and the chive visualisation engine [3] as a semi-automated solution to these problems. Also, we feel that the application of other reengineering and design recovery techniques such as graph clustering, textual searches and domain ontologies would further alleviate the problem.

5. Conclusion

The approach described here successfully provides the rst steps in transforming portions of legacy systems for reuse in modern software development processes. Early results have been quite promising. SHARED sets identi ed by software reconnaissance contain very high proportions of reusable code and provide excellent starting points to identify reusable code within a system. This combined with a static analysis of data creates a new and useful link from feature to source to persistent data, identifying a potential portion of the system for stateful component recovery.

The application of a component wrapper to create a xADL arch type successfully creates a new and reusable component. While writing the xADL wrapper became trivial once the prerequisite information was gathered, an easily written automatic xADL generator would seem to be a useful tool addition to the process. In the letters component example described here, the original developer was available to con rm the usefulness of the SHARED sets and the reusability of the recovered component. Also in this case study, the identi cation of known reusable libraries by the technique should also be viewed as step towards proof of concept.

Problems with partitioning the shared set into reusable blocks and subsequently identifying the entire interface from the starting point that the SHARED set provides, still exists and will be the focus of research in the near future. Furthermore, it should be noted that the SHARED set outputted by software reconnaissance only provides a starting point for identifying reusable code within a system and it not an end in itself.

Acknowledgements

We would like to thank the contribution of our funding and research partners, QAD and Enterprise Ireland, and the *Software Architecture Evolution*(SAE) group at the University of Limerick.

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 216–234, Toulouse, France, 1999.

[2] J. Bergey, L. O'Brien, and D. Smith. Mining existing assets for software product lines. Technical Note CMU/SEI-2000-TN-008, Software Engineering Institute, Carnegie-Mellon University, May 2000. Product Line Practice Initiative.

[3] B. Cleary and C. Exton. Chive - a program source visualisation tool. In *International Workshop on Software Comprehension*, June 2004.

[4] P. S. Corporation. *Openedge Development: Progress 4GL Reference*. Progress Software Corporation, http://www.progress.com, 2004.

[5] E. M. Dashofy and R. N. Taylor. An infrastructure for the rapid development of xml-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering.*, Orlando, Florida, May 2002. IEEE, IEEE Computer Society.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *IEEE International Conference on Software Maintenance (ICSM'01)*, page 602, Universität Stuttgart, Breitwiesenstr, 20-22, 70565, Stuttgart, Germany, November 7-9 2001. IEEE.

[7] S. Galvin, J. Collins, C. Exton, and F. McGurren. Enhancing the role of interfaces in software architecture description languages (adls). In *The Workshop of Architecture Description Languages (WADL '04)*, Toulouse, France, August 2004.

[8] T. Kalibera and P. Tuma. Distributed component system based on architecture description: The sofa experience. 2002.

[9] F. McGurren. Component composition and architectural re ection. Master's thesis, University of Limerick, University of Limerick, Plassy, Castletroy, Co. Limerick, Ireland, February 2004.

[10] G. C. Murphy and D. Notkin. Reengineering with re e xion models: A case study. *IEEE Computer*, 17(2):29–36, 1997.

[11] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[12] S. Sadler. Process-oriented architecture platforms. *Q-Link Technologies Executive Brie ng Series*, 2003.

[13] N. Wilde. *Recon3*. University of West Florida, Pensacola, FL 32514. http://www.cs.uwf.edu/∼recon/recon3/.

[14] N. Wilde, J. Gomez, T. Gust, and D. Strasburg. Locating user functionality in old code. In *Conference on Software Maintenance*, pages 200–205. IEEE, November 9-12 1992.

[15] N. Wilde and M. C. Scully. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[16] W. E. Wong, S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating program features using execution slices. In *Proceedings of the 1999 IEEE Symposium on Application - Speci c Systems and Software Engineering and Technology*, page 194. IEEE, IEEE Computer Society, 1999.

Session 4:
Applications of Software Transformations

Improving Variability Management in a Product Line of Embedded Systems – A Case Study from Industry

Thomas Patzke and Dirk Muthig
Fraunhofer Institute Experimental Software Engineering
Sauerwiesen 6, 67661 Kaiserslautern, Germany
{Patzke,Muthig}@iese.fraunhofer.de

Abstract

Many software development organizations have been developing corporate software components to benefit from reusing preexisting solutions instead of creating each of their products always from scratch. However, this has typically been done in an unsystematic way, that means without deliberately using a variability mechanism that supports system evolution. As the number of different products grows, it becomes more and more difficult to manage the variabilities and their interdependencies. This paper presents an industrial case study of applying PuLSE^{TM}(Product Line Software Engineering) [1], where we extracted the common and variable characteristics of a set of products from several embedded systems components, and improved their variability mechanisms incrementally.

1. Introduction

Many software development organizations today develop and maintain several products in the same or a similar application domain. In order to decrease development and maintenance effort, common or similar parts of these systems are collected in components. However, as variability among the individual products is typically managed in an ad-hoc manner, these platforms are often hard to adapt when evolving the set of products.

In this paper, we will present a process of improving variability management for subsystems of programs written in C, which form a corporate platform for embedded systems software.

This platform grew from a common set of reference systems sharing a large degree of commonality, and only differing in some features. However, as the number of products using the subsystems grew, it became necessary to introduce more and more slight variations in the common code. This was implemented on an ad-hoc basis by using the common C variability mechanism of conditional compilation. However, beyond a certain number of products, the variations could not be maintained efficiently. A more systematic way of managing variability was required.

[1]PuLSE is a registered trademark of Fraunhofer IESE

The aims of this project were to extract the product line decision model [1], capturing product line variations and their interdependencies, and to evaluate good implementations for the necessary variabilities.

In Section 2, we describe our experiences in selecting and analyzing the existing components. In Section 3, we present ways to improve them by refactoring their variability mechanisms, for example, to frame technology [2], an advanced product line implementation mechanism. Section 4 summarizes the results and lessons learned and gives an outlook of future activities.

2. Decision Model Extraction

In order to extract the decision model from the existing components, we followed this process: After selecting an appropriate subsystem and component, it was then analyzed in detail to extract facts relevant to product lines, such as groups of important variabilities belonging together.

2.1. Subsystem and Component Selection

The system platform consisted of hundreds of files mainly written in C containing several hundred thousand lines of code. In order to get the highest benefits from improving the variability mechanisms, we first identified and prioritized single components. We took the following criteria into account:

- their degree of variability; the most variable components should be selected first
- how common their variability mechanism is among all components, so that approaches used to improve one

component can also be applied in as many other parts of the system as possible
- the business value of a component, as it is not efficient to focus on improving components of low business value.

The existing system consisted of three layers: an engine, service, and application layer. We selected an engine layer component because it was rich in variability, it was well-documented, and it shared the programming language C and the conditional compilation variability mechanism with many other components.

2.2. Component Analysis

After selecting a component, we semi-automatically analyzed the variability mechanisms used in its given implementation. During this reverse variability engineering process we focused on answering the following questions: Which kinds of variability exist, and where do they exist?

We performed the following sub-process: In the first step, we made sure that variations had not been introduced by making copies of huge parts of common code and modifying these slightly. As the modified parts are hard to re-integrate, this technique known as "clone and own" [4] should be avoided. We developed a clone detection tool suite similar to DupLoc [13], which was used to find consecutive identical code lines across all source code files. The selected component did not contain a significant amount of clones (more than 6 identical non-blank, non-comment lines of code), but applied conditional compilation, the major variability mechanism offered by the C programming language. [10] gives an overview of these and other state-of-practice and state-of-the-art product line implementation techniques.

In conditional compilation, variable code parts are enclosed in #ifdef statements, and by defining or undefining the macros, the appropriate code lines are included or excluded.

Next, we detected all uses of the variability mechanism. We used the ifnames unix tool [14] to analyze which macros were used for conditional compilation in all .c and .h files. The command `ifnames *.[ch] | wc -l` detected 179 macros, but by manually inspecting their names and uses it became clear that only a portion of them was used for product-specific purposes (in an empirical analysis [5], this kind of macro usage formed the largest category of conditional compilation directives). Many other macros served as development aspects [8] such as tracing (e.g. the VERBOSE macro), debugging (CHECK_ and DEBUG_ macros), or as include guards (FILENAME_H) to prevent multiple header inclusion.

Consequently, we next extracted the macros most relevant for the management of variable product features. This was done using using expert knowledge and usage statistics collected by a Python script. In an iterative way, we identified the variabilities belonging together, considering the macro names, available documentation and expert support.

Next, we gathered variability statistics to reach the following goals according to the GQM method:

- find product line features affecting the component most
- locate subcomponents containing highest degree of variability
- detect default product line features that vary only in few products.

We again performed an analysis of the component, starting with its largest parts, the physical subcomponents [9]. We extracted the macro names contained in #ifdefs, and in which files this occurred by invoking `ifnames *.c` and counting the number of words in each line with a Python script. In contrast to the previous step, we restricted our analysis on the C implementation files here, because they contributed the majority of files and contained the most important macros. Furthermore, we only considered the macros identified as relevant for variability management in the previous step. With one exception, all the macros classified as irrelevant had an extremely localized effect: they only appeared in one or two files. The result is shown in Fig. 1.

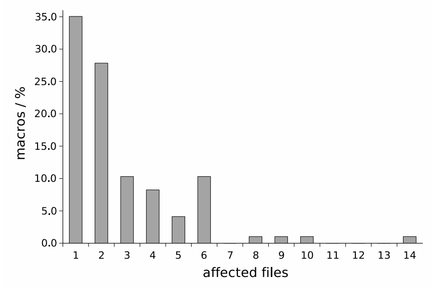


Figure 1. Files affected by product line relevant macros

The figure shows at the extremes that 35% of all macros affected only one .c file, whereas 1% of the macros affected 14 files.

In order to objectively characterize the macro usage of this component and to make this comparable to other components, we calculated the average number of affected files (3 = 13% of all files) and the standard deviation (2 = 10%). We grouped the macros above average in intervals of one standard deviation; the 1% of the macros affecting most files were about 5 standard deviations above average, and 3% were between 3 and 4 standard deviations above average.

This process achieves the first goal mentioned above, the identification of those groups of variabilities with the highest impact on physical components within the product line. Members from these groups are top candidates for variability mechanism improvement.

In order to localize the subcomponents containing the highest degree of variability, we analyzed the degree of variability across all physical components. For a first overview, we counted the number of different macros used in each file with this script:

```
for FILE in *.[ch];
  do ifnames \$FILE | wc -l > numbers;
done .
```

A closer analysis would also measure the amount of macros used in #ifdefs per file or per line.

We detected for the extreme cases that more than 7% of the files were not affected by macros at all, whereas 4% of the files were affected by 28% of all macros at the same time. For the top five groups this means that each file contains between 41 and 51 different kinds of variable parts that are possibly interdependent.

2.3. Decision Model Creation

As a result of this analysis, groups of physical subcomponents containing a higher-than-average degree of variability were obtained in a reproducible manner. Members from the top groups are candidates for splitting into smaller components, as they contain too many features.

With expert support, we prioritized the variabilities expressed by macros from these groups, and clustered variabilities belonging together. We renamed each of these features or feature groups according to their purpose in the product line, and mapped these decisions to the existing products. The resulting product map is a matrix containing the relevant features/decisions vs. the products.

With the analysis results and expert support we finally completed the decision model by documenting questions and resolutions for each decision from the product map, including constraints, defaults and decision interdependencies.

3. Variability Mechanism Improvement

The component analysis motivated an improvement of the variability mechanism. We performed the following process:

- Removal of code inconsistencies
- Refactoring within the given variability mechanism
- Refactoring to an advanced variability mechanism, where necessary

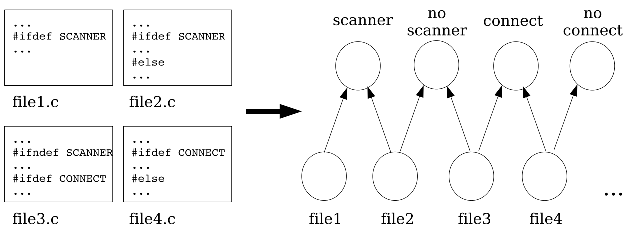
In the given component, we detected and removed four main categories of inconsistencies in conditional compilation: Commenting (0 and 1 macros), development aspect and redundant macros detected during analysis and unreachable parts or redundant nested macros. Where possible and sensible, these were automatically removed with Python scripts.

Refactoring the given variability mechanism mainly involved making macro definition and use consistent. We limited macro definitions to a single location (e.g. a config.h file), introduced a common use scheme (e.g. by automatically replacing all #ifdefs by #ifs or introducing a common macro naming convention).

As shown in [10], beyond a certain level of complexity, the variability mechanisms used in practice are inappropriate for managing variability, for example by leading to unmanageable feature explosions or tangled code. In this case it becomes necessary to refactor to an advanced variability mechanism such as frame technology.

In frame technology, the source code parts to be reused are tagged with explicit variation points, similar to marking variable C source code with #ifdef macros in conditional compilation. Frame technology, however, separates the common and the different kinds of variable code parts physically, so that each physical unit only contains the minimal amount of code that it needs to be reused. The variable code parts are stored in different files, referring to the explicit variation points in the more common files by adapting them. A frame hierarchy [2] emerges, with one specification frame (SPC) per product adapting the corresponding combination of frame subassemblies. The frame processor, an advanced kind of preprocessor, is then used to assemble a specific product according to its SPC. As a static variability management mechanism, this does not affect program size or performance, an important aspect in embedded systems software. The benefits of frame technology and our tool support [7] is discussed in more detail in [11] and [12].

For the groups of macros collected in the analysis phase, we automatically converted all source code files to frames. This involved first replacing all relevant macro uses by explicit variation points and moving the respective code parts into separate frames. The major challenge in this task was to handle macro combinations correctly without code duplication, as the macro processor allows nested conditionals, whereas the frame processor does not allow nested variation points. We solved this problem by declaring variation points in adapting frames. After that, we optimized variation points, frames and the frame hierarchy, for example by eliminating redundant variation points or merging frames or frames pieces belonging together. This process had to be performed only once. Fig. 2 illustrates this transformation: Initially, the four files contain conditional compilation blocks intermingled with common parts, whereas after the transformation the common parts residing in the lower part of the frame hierarchy are separated from the variable parts in the adapting frames.



```
...
#ifdef SCANNER
...
```

```
...
#ifdef SCANNER
...
#else
...
```

```
...
#ifndef SCANNER
...
#ifdef CONNECT
...
```

```
...
#ifdef CONNECT
...
#else
...
```

Figure 2. Example frame hierarchy

As a second step, the product-specific configurations were generated for each product: Each specific product according to the decision model was automatically converted into a specification frame by adapting the appropriate combination of frames. For example, in Fig. 2, a product with scanner and without connection support can be obtained by a wrapper frame adapting both the `scanner` and `no connect` frames. Finally, the generated result was automatically validated for all individual products by comparing the outputs of the C preprocessor with the frame processor-generated files.

The benefits of the frame approach were that the relevant features belonging together could be stored at one place, instead of scattering them across the entire product. The common and variable parts were cleanly separated, with the most reusable parts remaining in one place. Refactoring to frames could be performed incrementally, so that only variations above a certain priority level were converted, whereas other variable parts remained conditionally compiled. System extension became possible nearly without changing the common parts, and a fine-grained control over the relevant variable parts was achieved.

4. Results and Outlook

This paper presented a process performed in practice for improving variability management in an embedded systems product line. The first phase served to extract the decision model for the product line. In an incremental way, subsystems and individual components were selected, partially automated, and partially with expert support. For each component, its main variability mechanisms were determined and its uses were detected. For conditional compilation, the ifnames tool provided valuable support. Then, statistics about the relation between subcomponents and features were gathered and grouped, which helped to prioritize the most relevant items. As a result of this analysis, the decision model for the product line was built. In the second phase, the detected variabilities were improved incrementally by removing inconsistencies and normalizing the given variability mechanism. Selected parts were refactored to more advanced variability mechanisms such as frame technology to improve component reusability.

Where necessary, the given process is going to be refined or extended, for example by analyzing logical components in addition to physical components. The automatic detection of different static variability mechanisms can be refined, including clone detection, and also dynamic mechanisms can be included. According to the given goals, the statistical methods are going to be refined and further automated. Finally, the ways to improve the variability mechanisms used, and refactoring to other state-of-the-art mechanisms such as collaborations [3] or aspect-orientation [6] will have to be explored.

References

[1] C. Atkinson, D. Muthig: Enhancing Component Reusability through Product Line Technology. In Proceedings of ICSR-7, Springer-Verlag, 2002

[2] P. G. Basset: Framing Software Reuse: Lessons from the Real World. Prentice-Hall, 1996

[3] D. Batory, Y. Smaragdakis: Building Product-Lines with Mixin-Layers. ECOOP 99

[4] P. Clements, L. Northrop: Software Product Lines: Practices and Patterns. Addison-Wesley, 2002

[5] M. D. Ernst, G. J. Badros, D. Notkin: An Empirical Analysis of C Preprocessor Use. IEEE Transactions on Software Engineering, December 2002

[6] T Elrad, R. E. Filman, A. Bader (eds.). Aspect-Oriented Programming. Communications of the ACM, October 2001

[7] Frame Processor Homepage: frameprocessor.sf.net

[8] G. Kiczales, E. Hilsdale et al.: Getting Started With AspectJ. In [6]

[9] John Lakos: Large-Scale C++ Software Design. Addison-Wesley, 1996

[10] D. Muthig, T. Patzke: Generic Implementation of Product Line Components. In Proceedings of NetObjectDays 2002, October 2002

[11] T. Patzke, D. Muthig: Product Line Implementation with Frame Technology. IESE Technical report 18/03E, March 2003. Available at www.polite-project.de

[12] D. Muthig, T. Patzke: Implementing Software Product Lines. In Multikonferenz Wirtschaftsinformatik 2004 Proceedings, Vol. 1, Infix 2004

[13] M. Rieger, S. Ducasse: Visual Detection of Duplicated Code. ECOOP'98 Workshop, S. Demeyer, J. Bosch (ed.), Springer-Verlag, 1998

[14] G. V. Vaughan, B. Elliston et al.: Gnu Autoconf, Automake and Libtool. Pearson Education, 2000. Available at sources.redhat.com/autobook

Applying Software Transformation Techniques to Security Testing

Thomas Dean
Electrical and Computer Engineering
Queen's University
thomas.dean@ece.queensu.ca

Scott Knight
Electrical and Computer Engineering
Royal Military College of Canada
knight-s@rmc.ca

Abstract

Application protocols have become sophisticated enough that they have become languages in their own right. At the best of times, these protocols are difficult to implement correctly. Combining the complexity of these protocols with other development pressures such as time to market, limited processor power and/or demanding performance requirements make it even more difficult to produce implementations without security vulnerabilities. Traditional conformance testing of these implementations does not reveal many security vulnerabilities. In this paper we describe ongoing research where software transformation and program comprehension techniques are used to to assist in the security testing of network applications.

1. Introduction

The security of network applications is an increasingly important topic in both academia and industry. The cheap availability of bandwidth world wide has increased the ability of people to communicate, but has also provided convenient access to many systems for those with malicious intent. This increased access to bandwidth is not just access to the internet, but other networks such as the cellular phone networks (both voice and data). Additionally, implementations formerly on closed network protocols are moving to public protocols such as the move of telephone networks from packet switched networks to Voice over IP protocols.

Some recent incidents include vulnerabilities in libraries used to display images (BMP[13] and JPG[9]), a vulnerability in Cisco routers running OSPF[4], and a proof of concept of the first virus for cellular phones[1].

Conformance testing of these applications tends to focus on the correct implementation of the application to valid requests and obvious errors. However, sometimes the security vulnerabilities involve a data item that could not possibly occur in the normal operation of a protocol. As an example, an Xmas tree packet is a low level IP packet that has every single flag in the header enabled. Some of these flags are mutually exclusive. In the mid 1980's, several implementations of TCP/IP operating systems were unable to handle these packets, and this became an effective Denial of Service (DOS) attack.

Our position is that evolution transformation techniques can be fruitfully applied to structured data such as network protocols. The protocols used by network applications have become languages in their own right with both syntax and semantics. One approach to security testing is Syntax Testing [2]. In this approach, syntax and semantic errors are intentionally made to produce variants of the data to attempt to expose vulnerabilities.

The PROTOS project[8] at Oulu University uses a protocol grammar to generate variant packets. The grammar specifies the possible packets right down to the values of fields. The grammar is modified manually to allow the desired errors and then a walker walks the grammar tree, automatically generating the packets.

While the general technique is the same, our approach is different. We capture a valid set of data by sniffing the network and transforming it to generate alternate packets. We also are automatically generating the test plans based on the syntax and semantics of the protocol without manual intervention. This represents a novel cross-fertilization between the software transformation and the security communities.

2. System Structure

Figure 1 shows the overall structure of our system. At the bottom of the figure we have a network containing the test system and a client system that is interacting with the test system. A sniffer is used to capture a valid protocol data unit (PDU) that was sent from the client to the test system. A PDU may be a single packet, or it may be spread over multiple packets. The PDU at this point in time is a binary data file. This file is decoded into a textual representation by a decoder.

The markup and execution engine, implemented in TXL[5], are used to generate variants of the packet which are then re-encoded and injected into the network. The original, valid packet is injected between each of the mutated packets to verify that the test system is still functional and responsive.

The markup and execution approach is modeled on previous software evolution and transformation research [6]. This approach separates the planning of the testing suite from the execution of the testing suite. The markup that is generated is rather simple. It includes markup to delete a field, change the encoding of a field,

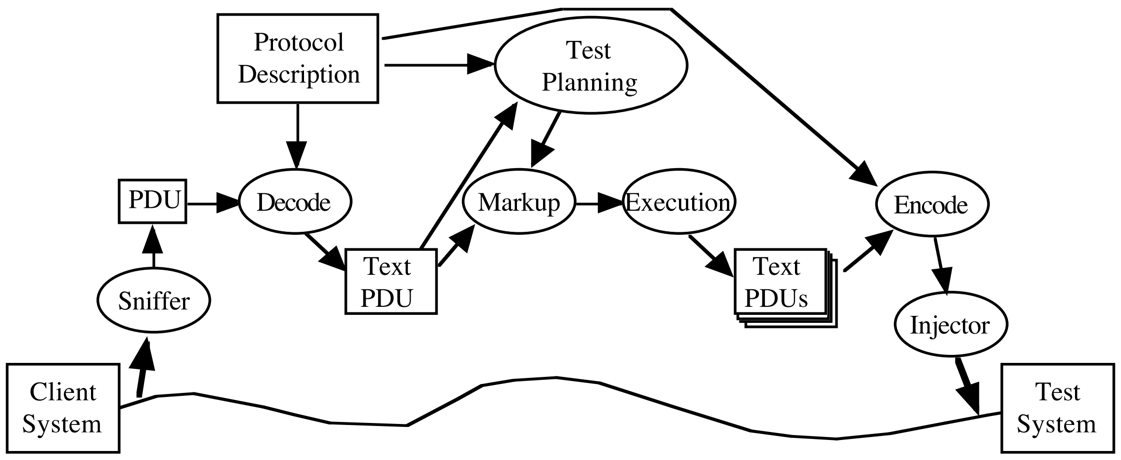


Figure 1. Protocol Tester General Structure

duplicate a field, change the value of a field, and other similar tasks. The execution engine carries out most of the markup (encoding markup is carried out by the encoder). Thus markup is always done on the original valid packet, may generate more than one packet, while execution generates the modified packets. The markup phase may generate more than one marked up packet, each of which is independent. This separation of concerns is important. Testing strategies that depend on simultaneous changes to multiple fields communicate through the markup. That is, they make markup to simultaneous fields. The execution engine, responsible for implementing the transforms, need not know about relationships between fields.

The description of the protocol contains a variety of information. It contains the syntax of the protocol, transfer encoding information and semantic information such as constraints between fields, ordering of sequences and if the sequence must be unique. Most protocols are described in a document which contains the syntax of the protocol in a standard form such as EBNF or ASN.1[7]. The semantic constraints of the protocol tend to be described in the prose of the document. Some means of describing these constraints in addition to the syntax is needed. We are interested in identifying the constraints that exist both between fields of a given PDU (current work) and between PDUs in a sequence (future work).

When investigating existing protocol description languages, we discovered that almost all of them describe the syntax of the protocol, some describe the transfer syntax, and some describe the semantics of the protocol either as finite state machines or as high level algorithms. We are looking for constraints such as the permissible values of a version field, the relationship between a length field and the data item governed by the length field, or that a sequence of items must be unique. In state and algorithm based protocol languages, extracting these relationships and constraints can be difficult. Furthermore, many of the protocols we are interested in are not currently described in these extended languages. Requiring a test engineer to translate the prose in a standard protocol description to a finite state machine in order to extract simple constraints seemed to be counter productive.

Figure 2 shows a description of our simple house description protocol in our protocol description language as a frame based protocol. Our protocol description language is an XML based extension to the existing ASN.1 standard. An XML markup is added to each non-terminal in the description and provides information about transfer encoding and constraints. In the Fig. 2, the transfer encoding for the first non-terminal (*HDP_Packet*) indicates that the *number_of_houses* field is encoded as a 4 byte integer. The constraint markup for the first non-terminal also indicates that the value of the first field (*number_of_houses*) gives the cardinality of the second field (*houses*). Other constraints include value constraints (e.g. version is 0, 1 or 2, range is 0 to 255), length constraints (cardinality is number of items, length is number of bytes). The third non-terminal(*House*) has a single markup indicating the transfer encoding for the *house_number* and *family_name* fields.

This description is designed for a human test engineer to read and write. Currently we are working on an Eclipse plugin which will allow the test engineer to author the ASN.1 directly and to enter the constraints interactively, never having to deal with the XML directly.

The information in this description is used in two ways. The first is to generate protocol syntax and transfer information for the decoder to decode the binary

```
HDP_Packet ::= SEQUENCE {
      number_of_houses  INTEGER
      houses            Houses
}
<size>
      number_of_houses is 4 bytes
</size>
<constraints>
Cardinality(houses):=number_of_houses
<constraints>
```

```
Houses ::= SEQUENCE OF House
House :: = SEQUENCE {
      house_number   INTEGER
      family_name    VisibleString
}
<size>
      house_number is 2 bytes
      family_name is 100 bytes
</size>
```

Figure 2. Protocol description of the House Description Protocol

PDU that was retrieved from the network. It is also used by the encoder to re-encode the packet for injection.

The other way the protocol description is used is by the test planner. A design recovery extractor is run over the protocol description to generate an instance of an ER model that contains the information in the protocol in an form easily used by the test planner. Protocols usually describe more than one PDU types (multiple request PDU types, various response PDU types). The ER instance contains the constraints for all of the PDU types, which includes constraints not relevant to the captured PDU. So the first task of the test planner is to filter the information in the ER instance based on the PDU to be mutated. The test planner then invokes appropriate test plans for each of the remaining constraints. These are invoked by using the markup engine to markup the appropriate fields. The approach currently in progress is table driven, with the first field in the table identifying the constraint type identifying one or more strategies which leads to a template expression that marks up the fields involved in the constraint. For example, the cardinality constraint given in Figure 2 leads to several mutant PDUs where the value of the *number_of_houses* field disagrees with the number of House entities in the *houses* field.

3. Current Status

The base system for DER based protocols (e.g. X.509[11,12] and SNMP[3]) was completed as part of Yves Turcotte's M.Sc. Thesis[15]. This comprises the sniffer, the decoder, markup and execution engines, the encoder and the injector. A scripting tool that drove the markup and encoding engines was built that allowed a user to indicate what errors should be applied to which fields. A large variety of error strategies were designed and implemented as part of this work. The work was used to independently confirm errors in SNMP implementations and to test an implementation of X.509 that was adopted by the Canadian Department of National Defense. A new potential denial of service attack was discovered in the implementation of the X.509 protocol which increased the processing time of a certificate from a fraction of a second to over two hours of CPU time. As ASN.1/DER based protocols are self describing, this system is completely protocol independent. That is, the system has no knowledge of the protocol syntax or semantics.

This infrastructure was extended by Dr. Oded Tal [14] to handle frame based protocols (e.g. OSPF). This involved adding a simple description of the protocol that was used by the decoder and encoder to translate between the binary and textual forms of the PDUs. But it also involved investigation of the types of errors that apply to frame based protocols. For example, some of the syntax based mutations appropriate to DER based protocols do not apply to frame based protocols. Deleting a field from the middle packet simply shortens the packet and the test system will interpret the following field as the contents of the deleted field. Thus, deleting a field is the same as generating random values for subsequent fields. Similarly encoding errors are limited. The possible DER vulnerability of valid over length integer values is not reasonable in a protocol which mandates a single length to fields.

However mutations based on values of fields and on relationships between values of fields are effective. For example, in OSPF there is a field within the packet describing the length of some data fields in the packet. In some implementations, this field is used, even if it says that the data in the packet is longer than the length of the packet itself. This leads to segmentation faults on Unix systems and system failures in Windows 2000 Server. To the best of our knowledge, this is a new vulnerability.

The protocol description language and the test planner are currently in the process of being implemented and integrated into the system.

4. Future Work

The system we have described is a very general infrastructure with a great deal of potential. Some of the future work we are planning on pursuing include:

State based protocols. The current protocols we have investigated are state independent protocols. That is, the protocols exist as request/response exchanges. Send a request to a server get a response. Each request is, in some sense, independent. Extending the framework to deal with stateful protocols such as Voice over IP protocols is an interesting avenue to pursue. This will involve analyzing constraints between packets and generating mutated packet sequences.

The protocols described are also currently binary protocols. Textual protocols such as HTTP, SMTP and SOAP (XML over HTTP) can also be security tested using a transformation based process. The interesting part of these protocols is that the decoder/encoder becomes redundant, and transfer encoding is textual.

The current approach is also based on a black box approach. On some occasions when we have had source code to the test system, we have tracked down the bug in the system manually. Expanding to a white box style of testing has some potential. One option is to use the erroneous PDU to isolate the error automatically. The other option is to use a light weight program comprehension/design recovery step to identify potential security failures in the system. A full comprehension approach can be expensive both in time and resources. A light weight identification could be more aggressive in identifying potential vulnerabilities which are used to provide information to the test planner.

Alternatively, we can have the developers provide some information to the test planner. The recent OSPF bug exposed a dependency in some versions of CISCO routers between certain requests and the value of the hello timer in the router. While the implementation of the hello timer is not part of the protocol, the existence and the relationship between the hello timer and certain PDUs is. So adding abstract implementation entities and the relationship to the protocol description can help test the implementations.

5. Conclusions

This paper has presented some of the ongoing research into network security at the Royal Military College of Canada and Queen's University. This research is the direct result of combining current research approaches from two very diverse communities: the software evolution and transformation community and the software security community.

References

[1] BBC, *'Game virus' bits mobile phone*s, BBC news, UK edition, Aug. 11, 2004.

[2] Bezier, B., *Software Testing Techniques*, 2nd Edition, Van Nostraad Reinhold, New York, 1990.

[3] Case, J., Fedor, M., Schoffstall, M., Davin, J., Simple Network Management Protocol, Internet RFC 1157, 1990.

[4] Cisco, *Cisco Security Advisory: Cisco IOS Malformed OSPF Packet Causes Reload*, Document ID: 61365, Cisco Systems, San Jose, California, Aug. 2004.

[5] Cordy, J., The TXL Programming Language, v. 10, http://www.txl.ca/, 2000.

[6] Dean, T.R., Cordy, J.R., Schneider, K.A., Malton, A.J., "Using Design Recovery Techniques to Transform Legacy Systems", *ICSM 2001 - The International Conference on Software Maintenance*, Florence, Italy, November 2001, pp 622 - 631.

[7] Dubuisson, O., "ASN.1 Communication between Heterogeneous Systems", Academic Press, San Diego, 2001.

[8] Kaksonen, R., Laasko, M. and Takanen, A., *Vulnerability Analysis of Software through Syntax Testing*,http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/index.html, 2001.

[9] Microsoft, *Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution*, Microsoft Security Bulletin MS04-28, Sept. 2004.

[10] Moy, J., OSPF Version 2, Internet RFC 2328, 1998.

[11] PKCS#7-Cryptographic Message Syntax Standard. http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/, RSA Data Security, Inc, 2004.

[12] Public-Key Infrastructure (X.509) (pkix). http://www.ietf.org/html.charters/pkix-charter.html, 2004.

[13] SecurityTracker.com, *Microsoft Internet Explorer Integer Overflow in Processing Bitmap Files Lets Remote Users Execute Arbitrary code*, Security Tracker ID: 1009067, Feb 2004.

[14] Tal, O., Knight, S., Dean., T., Syntax-based Vulnerability Testing of Frame-based Network Protocols, Proceedings of the Second Annual Conference on Privacy, Security and Trust, Fredericton, Canada, October 2004, 6 pp., to appear.

[15] Turcotte, Y., *Syntax Testing of the Entrust Public Key Infrastructure for security vulnerabilities in the X.509 Certificate*, M.Sc. Thesis, Department of Electrical and Computer Engineering, Royal Military College of Canada, 2003.

[16] Turcotte, Y., Oded, T., Knight, G.S., Dean, T., "Security Vulnerabilities Assessment Of the X.509 Protocol By Syntax–Based Testing", *Proceedings of MILCOM 04,* Monterey, California, October 2004, 7 pages, to appear.