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 identifies reusable code within existing software systems. Once identified, 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 profiles [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, filtered 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 final 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 file, function or branch level and this choice defines the granularity of the set of software elements outputted.

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

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

- 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:ELEMS$|\exists t \in T,\ EXHIBITS(t, f) \wedge EXERCISES(t, e)\}$ - $\{e$:ELEMS$|\exists t \in T, \neg EXHIBITS(t, f) \wedge EXERCISES(t, e\}$.

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

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 modified. Wrapping is a modification technique whereby source code may be supplemented to allow the system to conform with other development paradigms [2]. Here we choose to wrap identified 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 specific nature of other ADL's. In addition, its basis in xml schemas eases both its application and its ability to be tailored to specific 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 files, as an example. 23 features were identified in the program. Using this as a basis for analysis, profiles 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 first 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 identified 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 identified in figure 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

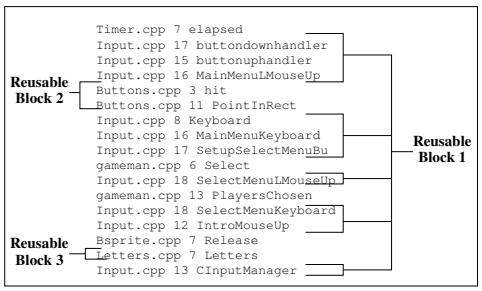

Figure 1. The shared set for feature 8.

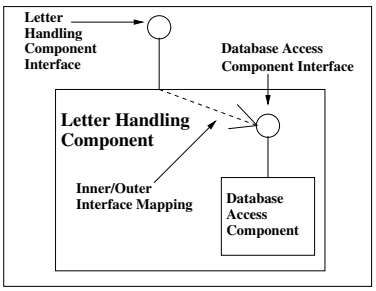

Figure 2. Structure of the Letter Handling Component.

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 file *letters.txt*. The file is used to store the characters in use in the game as well as their value and quantity.

With behavioural and data aspects now identified for our potential letters component, all that remains is the application of the xADL component wrapper. For our example it is sufficient 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 files* 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 identified:

- `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* file described earlier was present. Finally two implementation files *letters.cpp* and *letters.h* were identified.

Based upon this information it was decided to create two components - A *letter handling component* and a *database access component* centered around the data file *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 confirm 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 reflexion 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 first steps in transforming portions of legacy systems for reuse in modern software development processes. Early results have been quite promising. SHARED sets identified 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 confirm the usefulness of the SHARED sets and the reusability of the recovered component. Also in this case study, the identification 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 reflection. 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 reflexion 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 Briefing 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 - Specific Systems and Software Engineering and Technology*, page 194. IEEE, IEEE Computer Society, 1999.