Achieving a Reuse Perspective within a Component Recovery Process: An Industrial Scale Case Study

Andrew Le Gear, Jim Buckley, Brendan Cleary, J.J. Collins
University of Limerick
Castletroy, Limerick, Ireland
{andrew.legear,jim.buckley, brendan.cleary,j.j.collins}@ul.ie

Kieran O’Dea
QAD Ltd.
Plassy Technological Park, Limerick, Ireland
kpo@qad.com

Abstract

Identifying elements of existing software that are reused within a system may provide maintainers with valuable insights during system evolution. This paper evaluates an extension of software reconnaissance that can be used to analyse reuse across features in a system, as part of a component recovery process proposed in [18]. We illustrate and evaluate retrieval of reuse information in this fashion using a large, commercial ERP and warehousing application. Results suggest that the approach scales well in terms of reuse information across features in existing software, providing maintainers with a valuable new perspective on the software system in question.

1. Introduction

Software Reconnaissance is a dynamic analysis, redocumentation technique, that, through the acquisition of source code coverage profiles [2] (yielded by exercising carefully selected test cases on instrumented code) creates a mapping between program features and the software elements that implement them [32]. A feature is defined as being a realised functional requirement that produces a result of observable value to the user [7, 8, 9].

In Norman Wilde’s seminal paper on software reconnaissance he remarks on, but never explores, the potential worth of the source code shared across features exhibited by a running software system [32]. Our technique exploits this source code and its associated data accesses as a step towards component recovery. The set of shared software elements for a feature, $f$, is called $\text{SHARED}(f)$ and our hypothesis is that $\text{SHARED}(f)$, combined with its associated data accesses, only contains software elements reused across features. We further contend that $\text{SHARED}(f)$ and its associated data accesses presents the software maintainer with a valid reuse perspective of the system, which may later be considered as a useful starting point when searching for reusable and stateful elements of software as part of the process of component recovery described in [18], i.e. - if a software element is being reused, it warrants further investigation to see if it is reusable.

This contention has lead to the component recovery process described in [18]. The process follows three major steps to component recovery illustrated in figure 1:

1. Reuse in the system is identified using the $\text{SHARED}$ set derived using the software reconnaissance technique and a static data analysis.

2. Candidate component code is identified from the sets of reuse. This involves browsing the reuse perspective and partitioning the code into candidates for components.

3. A component wrapper is applied to create a component. xADL 2.0 [11, 6] is used as the wrapper in [18].

The remainder of this paper is structured as follows. The next section (section 2) explicitly outlines the contributions of this paper and alerts the reader of certain caveats to interpretation when reading. Section 3 describes software reconnaissance and how we propose exploiting the technique to produce reuse perspectives of software systems. Section 4 describes a case study using our technique on a large, commercial ERP and warehousing application. The effectiveness of our approach is evaluated by a maintainer of the
system and also by the code conventions adhered to in the company to signal reuse. Related and future work is discussed in sections 5 and 6 respectively. Finally, conclusions with respect to the technique and case study are discussed in section 7.

2. Contributions and Caveats

We have previously piloted a complete process for component recovery in [18]. The first stage of the component recovery process described in [18] is examined by this paper, i.e. - determining reuse in the system, contributing the following:

- Determines whether the approach to generating a reuse perspective can scale to large and complex systems.
- It tests the validity of the approaches application in an environmentally valid small/medium enterprise setting.
- It gains an indication from a maintainer of the system whether the reuse perspective is helpful for understanding and maintenance.

Also, a common misconception is that feature location is the focus of this work. Feature location is not the focus of this research, thus the nature of the features being examined is not at issue here. Rather the software elements that participate in the relationship between features is being examined.

The relationship between features is being examined to determine what software elements are being reused by specific features of a system at runtime. However, because the elements are being reused does not automatically mean that they are reusable. Many more factors, beyond the scope of this paper, will determine this. Thus, the only reuse context of which we can be certain is the system that the reuse perspective is derived from.

What is being suggested is that the reuse perspective provides the maintainer with a useful view of the system, of which one possible use is to use the view as a starting point for searching for code that may be reused. Other insights that the view provides are discussed in section 4.3.

3. Shared Functionality in Context

3.1. Software Reconnaissance

As previously stated, software reconnaissance is a dynamic source code analysis technique and is primarily used as an aid to software comprehension by explicitly mapping features to the code that implements them [32, 8, 33]. The relation between program features and code is achieved through the use of test cases [31]. Given a set of test cases, $T = \{t_1, t_2, \ldots, t_n\}$, a set of software elements, $E = \{e_1, e_2, \ldots, e_n\}$ ¹ and a set of features the system exhibits, $F = \{f_1, f_2, \ldots, f_n\}$ we can define two relations:

- $\text{EXERCISES} : T \times E \rightarrow \text{BOOL}$, such that $\text{EXERCISES}(t, e), t \in T, e \in E$, is true if test case, $t$, exercises the software element, $e$.

- $\text{EXHIBITS} : T \times F \rightarrow \text{BOOL}$, such that $\text{EXHIBITS}(t, f), t \in T, f \in F$, is true if test case, $t$, exhibits the feature, $f$.

Using the relations defined, several sets of source code elements may be calculated from retrieved coverage profiles [32]. Of particular interest are:

**Involved Software Elements** The set of involved software elements for a feature are those which are exercised in any test case exhibiting that feature. Figure 2 illustrates the contents of the set. Given a feature, $f$, the set of involved software elements, $\text{IELEMS}(f)$ may be calculated as:

$$\{e \in E \mid \forall t \in T, \text{EXHIBITS}(t, f) \land \text{EXERCISES}(t, e)\}$$

¹These software elements may be files, functions or branches of the decision tree in a program depending upon the level of instrumentation chosen [30].
Indispensably Involved Software Elements  The set of indispensably involved software elements for a feature are those which are exercised by every test case exhibiting that feature. Figure 3 illustrates the contents of this set. Given a feature, \( f \), the set of indispensably involved software elements, \( \text{IIELEMS}(f) \) may be calculated as:

\[
\{ e : E \mid \forall t \in T, \text{EXHIBITS}(t, f) \Rightarrow \text{EXERCISES}(t, e) \}\n\]

Common Software Elements  The set of common software elements are those elements in a system that are executed by every test case profile gathered from the system. Figure 4 illustrates this set. \( \text{CELEMS} \) generally represents utility code within the systems that is executed every time it is run [32]. Given a feature, \( f \), the set of common software elements, \( \text{CELEMS} \) is calculated as:

\[
\{ e : E \mid \forall t \in T, \text{EXERCISES}(t, e) \}\n\]

Uniquely Involved Software Elements  The set of uniquely involved software elements for a feature are those which are exercised by any test case exhibiting that feature but excluding any elements that are exercised in test cases that do not exhibit that feature. This is the same as the set of involved software elements except we exclude any software elements in that set that are exercised by test cases that do not exhibit that feature. This set is illustrated by figure 5 by the enclosed white space. \( \text{UELEMS}(f) \) has been shown experimentally to provide a useful starting point to begin searching when attempting to understand a particular functionality exhibited by a system [32]. Given a feature, \( f \), the set of unique software elements, \( \text{UELEMS}(f) \), may be defined as:

\[
\text{UELEMS}(f) = \{ e : E \mid \exists t \in T, \neg \text{EXHIBITS}(t, f) \land \text{EXERCISES}(t, e) \}\n\]

3.2. Exploiting Software Reconnaissance to Achieve a Reuse Perspective on Software

Using the sets defined in section 3.1 we can define the set of software elements shared by a feature, \( f \), with other features as:

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

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 a system. This set is illustrated in figure 5 by the enclosed dark grey area. From a reuse perspective, the \( \text{SHARED} \) set gives a view of the software elements being reused by the running system.

To gain a fuller picture of reuse elements across code and data in the system, state should also be considered. By supplementing \( \text{SHARED} \) sets with a static analysis, the data accesses made by the software elements of the set can be revealed. This provides the maintainer with a reuse perspective of the system across features, code and data as illustrated by figure 6.
Curiously, if for every feature examined in the system there is one and only one corresponding testcase assigned then $IIELEMS(f) = IELEMS(f)$. This has curious implications for the calculation of $SHARED(f)$, since $IIELEMS(f)$ is directly involved in the calculation of $SHARED(f)$ and $IELEMS(f)$ indirectly involved though the use of $UELEMS(f)$. Resultantly the following may be inferred where each feature has only a single corresponding testcase:

$$SHARED(f) = \begin{cases} 
IIELEMS(f) - UELEMS(f) - CELEMS \\
IELEMS(f) - UELEMS(f) - CELEMS \\
IELEMS(f) - \{ e : E | \exists t \in T, \\
-EXHIBITS(t, f) \land EXERCISES(t, e) \}\ \\
\{-CELEMS \\
\{( e : E | \exists t \in T, \\
-EXHIBITS(t, f) \\
\land EXERCISES(t, e) \} \} - CELEMS 
\end{cases}$$

A single test case, however, rarely provides complete coverage for a feature, since an entire feature may provide several subtle configurations that require different user executed test cases. This is the reason for defining both the sets $IIELEMS$ and $IELEMS$ in section 3.1. However, a single test case may often be enough to successfully locate a feature [12].

4. Case Study

In [18] the reuse perspective described in section 3.2, is derived from a Scrabble™ [13] emulator program of approximately 8KLOC in size [10]. While useful in demonstrating the potential of the technique the Scrabble emulator does not realistically represent the size or complexity of systems found in industry and therefore cannot provide insight into how well the technique scales to larger programs.

For this case study we choose a commercially available, enterprise resource planning (ERP) system, called MFGPRO [20], with a warehousing management module extension called AIM [19]. The system, in toto, consists of approximately 6200 procedure files, is written in Progress 4GL, a fourth generation programming language [5], and contains a database backend with several hundred tables. The AIM module and MFGPRO communicate solely using published and subscribed events (figure 7). This is done to make any extension to either system as unintrusive as possible. Furthermore, the system is implemented as a distributed application. Realistic operational scenarios would see the system deployed and accessible at many consoles in a factory installation, as illustrated by figure 8. We feel that this system is sufficiently large and complex to provide a rigorous and environmentally valid test of our proposed technique.

4.1. Feature Elicitation

A rich feature set totaling twenty one features was identified, focusing on the activities of the AIM module and related interactions with MFGPRO. The maintenance agenda of the AIM evolution team was used as a seed set [24] in guiding feature identification. These features were augmented by decomposing, where possible, the features into primitive features. Table 1 lists these features and indicates whether each feature is an aggregate or primitive feature in
Table 1. Feature set examined during case study.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start - Stop</td>
<td>Primitive</td>
</tr>
<tr>
<td>2</td>
<td>Log in (remote)</td>
<td>Primitive</td>
</tr>
<tr>
<td>3</td>
<td>Exit MFGPRO</td>
<td>Primitive</td>
</tr>
<tr>
<td>4</td>
<td>Log in (space bar)</td>
<td>Primitive</td>
</tr>
<tr>
<td>5</td>
<td>Run AIM</td>
<td>Primitive</td>
</tr>
<tr>
<td>6</td>
<td>Exit AIM</td>
<td>Primitive</td>
</tr>
<tr>
<td>7</td>
<td>Unplanned receipt (including transaction checking)</td>
<td>Aggregate</td>
</tr>
<tr>
<td>8</td>
<td>Unplanned receipt</td>
<td>Aggregate</td>
</tr>
<tr>
<td>9</td>
<td>Unplanned Receipt - Transaction checking</td>
<td>Primitive</td>
</tr>
<tr>
<td>10</td>
<td>MFGPRO communicating with AIM</td>
<td>Primitive</td>
</tr>
<tr>
<td>11</td>
<td>RF Log in</td>
<td>Primitive</td>
</tr>
<tr>
<td>12</td>
<td>Update batch picking control file</td>
<td>Primitive</td>
</tr>
<tr>
<td>13</td>
<td>Batch picking</td>
<td>Aggregate</td>
</tr>
<tr>
<td>14</td>
<td>Creating sales order and releasing goods to ship</td>
<td>Primitive</td>
</tr>
<tr>
<td>15</td>
<td>Confirmation of batch pick</td>
<td>Primitive</td>
</tr>
<tr>
<td>16</td>
<td>Change logical format</td>
<td>Primitive</td>
</tr>
<tr>
<td>17</td>
<td>Select a warehouse (logical format menu)</td>
<td>Primitive</td>
</tr>
<tr>
<td>18</td>
<td>Batch picking - work order</td>
<td>Aggregate</td>
</tr>
<tr>
<td>19</td>
<td>Create a work order</td>
<td>Aggregate</td>
</tr>
<tr>
<td>20</td>
<td>Receive a task for a work order</td>
<td>Primitive</td>
</tr>
</tbody>
</table>

relation to the feature set as a whole. A primitive feature cannot be decomposed further into other sub features. An aggregate feature may be composed of primitive features or other aggregate features [34]. For example, “batch picking” (feature 14) is an aggregate feature made of the primitive features 15 and 16. We are not saying that features 15 and 16 are definitively primitive, only that they are primitive within the the context of this feature set.

4.2. Applying the Technique

Using the profiling option in Progress4GL [5], coverage profiles for one or more test cases exhibiting each of the twenty-one features (table 1) were retrieved. Using tool automation, the SHARED sets for each of the features were calculated. A one to one mapping between features and test cases is not assumed. This is evident from table 2, where we see that many test cases exhibit several features and vice versa.

Occasionally some features cannot be distinguished using test cases. For example, features 1 and 11 in table 2 are exhibited by all test cases, therefore the software elements that implement these features will be impossible to distinguish from each other and the utility code that appears in CELEMS. No SHARED sets of use are retrievable for these features.

Conversely, the aggregate features 14 and 19 cannot be completely exhibited by a single profileable test case due to the distributed usage of the application on multiple consoles. Therefore, test cases for the primitive features that constitute features 14 and 19 are used to create a mapping from feature to software elements.

A static analysis of the system was performed using the XREF compile option in Progress4GL [5]. The output produced reports on accesses made to database fields by procedures of the system. Using in house tool automation the output of XREF was filtered by the previously calculated SHARED sets, therefore creating a reuse perspective of the system as described in section 3.2.

4.3. Evaluation of the Reuse Perspective

Once the technique was applied we presented and evaluated the results during an interview with one of the current maintainers of AIM and MFGPRO. His comments are given in the following sections as an evaluation of the reuse perspective.

4.3.1 Reuse Content

The maintainer agreed that the sets did contain reusable software elements in the system. Of the reuse perspectives evaluated by the developer, except for the reuse perspective for feature 15 he stated all almost completely contained reused, generic software elements from MFGPRO and AIM:

“...all the rest are actually generic MFGPRO files
...because of the results of what you’ve done they
[software elements in set] are generic”

4.3.2 Validation from Coding Conventions

A large number of procedures prefixed by “gp” and “px” appeared in the sets. The maintainer confirmed that these are utility procedures specifically designed and implemented in the company with the explicit intention of reuse across other products implemented in Progress4GL:

“I already know that ‘gp’ is general. Anything
with ‘gp’ has already been designed to be general
... ‘gp’ and ‘px’, I know these thing have been
designed with reuse in mind”

While on its own, their presence carries little significance, the fact that code designed specifically for reuse should appear as part of the reuse perspective of the system lends support to our evaluation in conjunction with the comments of the maintainer and may be viewed as a
step towards proof of concept. Results similar to this were achieved for the scrabble emulator example also in [18].

Furthermore, the software elements that conform to coding conventions (“gp” and “px” prefixes) constituted on average 44% of the total reuse perspectives. This suggests that 56% of the software elements identify reuse which could not have been found through traditional textual searches for coding conventions, representing a real gain for the company when locating code for reuse during development.

### 4.3.3 Software Comprehension and Reuse

As a perspective on the system he commented that it was useful to be able to go to a file and be able to immediately view reuse within the system as a whole and in the context of specifically known features:

“Here’s thirteen files that are doing a lot of generic work … and the next stage would be ‘lets go and see what going on here’ … you are getting some information on what I am working with … understanding what services are provided”

The original developer alluded to the fact that the reuse perspective may be useful when attempting to understand the system initially, since reuse in the system may represent code “hot spots” that warrant close examination:

“… you are building what are the core players”

Interestingly, the reuse elements identified were localised to approximately 13 files, representing approximately 0.2% of all procedure files in the system. However, it would be incorrect to say that the sets identified represent the only reuse in the system, mainly due to the fact that the feature set identified only focused on behaviour that includes the AIM module3. Yet, when we use a far more conservative estimate that only considers AIM procedure files, we still find that the percentage remains at approximately 1%. This is a manageable proportion of the system for the maintainer to examine and may provide a useful starting point for understanding a completely new system, the relationships between pieces of a system and even facilitate the reuse of existing code during maintenance:

“… you now know that if I am doing some new functionality that I should be using that [he said pointing to a reusable element in set] … you reuse what should be reused”

### 4.3.4 Other Reuse

Other interesting elements found in the reuse perspectives included:

3But still not the entire AIM module
• A generic program management module which included a wrapper procedure for the “run” command in Progress4GL reused across products in the company.

• A separate and highly reusable module used for global variable management.

  “there are a lot of global variables in MFG-PRO and this guy is probably managing them”

• Module interfaces appeared in the reuse set. Large amounts of procedures in the reuse perspectives were prefixed by “get” and “set.” These were confirmed as calls on reusable module interfaces in the system. In particular, calls to specifically designed APIs\(^4\) appeared in the sets. For example, the API that controls communication between the AIM module and MFG-PRO appeared in almost every shared set.

  “yeah you could say that … they’re just utility calls … this has been put in some kind of utility place by itself, so that’s interesting … AIM has an API, there’s an interface between MFGPRO and AIM”

4.3.5 Pan-System Reuse Content

Database, licencing and validation procedures that are not only intended for use in AIM and MFGPRO but across other Progress4GL applications implemented within the organisation also appeared in the sets\(^5\).

4.3.6 Possible Business Process Reuse

The reuse perspective obtained for feature 15 was an unusually large set in comparison to other features and was examined by the developer in depth. It was decided that it contained less reusable code and far more code specific to that feature. This result initially confused us considering the results from the other sets. However, upon closer examination of the original feature set (table 1) one possible explanation was observed. Feature 15, “Creating Sales Order and releasing goods to ship” is a large and high-level business process of the system, yet primitive in context of the feature set of this case study. This business process overlaps two other primitive features, 17 and 18, that exhibit much of the same functionality as, but are not subsumed by feature 15. Therefore, features 17 and 18 will share a large proportion of their software elements with feature 15. This is illustrated in figure 9. The net result produces a set with large amounts of reuse but only in the context of that particular feature or business process:

\(^4\)Application programming interfaces

\(^5\)Validation of this point is delocalised throughout the interview recording.

Figure 9. a possible explanation for the limited reuse of feature 15’s shared set.

“you’re into a specific domain there, that’s ‘sale order maintenance’ [referring to entries in the set], that’s specific to whatever you did.”

One further indication of the presence of domain specific reuse is that the reuse perspective for feature 15 contains the lowest percentage (24%) out of all the reuse perspectives of reusable code denoted by the companies coding conventions as generic (section 4.3.2). Perhaps, given further experimentation, this could be developed as a metric to measure how generically reusable the contents of reuse perspectives are.

4.3.7 Data Evaluation

Probably the most disappointing outcome of the evaluation is the maintainer’s lack of enthusiasm in the data accesses of the reuse perspective. This portion of the reuse perspective constituted an information overload for the maintainer and highlighted a serious need to provide a useful visualisation for this information. On average the information pertaining to shared data was 45 times larger than the information pertaining to shared software elements. In its current form the most probable use of the data portion of the reuse perspective is as an automated means of the recovery of state as part of the component recovery. However, while this is indeed a useful application, we feel far more can be gained using appropriate representations of the data in the perspective.

4.4. Limitations of the Case Study and the Technique

While the successes of the study have been highlighted thus far, it is also prudent that we highlight the limitations of the study and our approach in general:

• Only one maintainer is used to evaluate the reuse perspective. This is acceptable, however to provide conclusive evidence regarding the usefulness of the technique, a larger population of evaluators is needed.
However, the use of coding conventions as a means of evaluation does reduce the impact of this limitation.

- The study lacks a rigorous qualitative analysis. This could not be accomplished as part of our study due to various legal and time constraints within the organisation. We still consider this necessary work to be undertaken to validate our work further.
- Dynamic analysis, upon which the technique is founded, cannot guarantee complete coverage of the subject software.
- Features, the location of which is core to our approach, are subjective, therefore results could potentially be different depending upon who implements the technique.

5. Related Work

5.1. Aspect-orientation

Aspect-oriented programming provides the ability to program and view software with specific concerns in mind [16]. Potentially, the reuse perspective could be defined as a reuse hyperslice [23] that creates a viewpoint [14] on the system that may be used during aspect-oriented programming. This approach would be a novel take on aspect-oriented programming since it would partially take a non-aspect-oriented system through reengineering into the aspect-oriented development process. The use of reuse information to recover aspectual views has been explored in [28]. However, their approach did not consider reuse as an aspect. Instead the authors use the reuse information to recover other aspects.

5.2. Feature Identification

While the concern of the reuse perspective is not feature identification, it does rely upon Software Reconnaissance [32], a feature identification technique, for operation. Several approaches to feature identification using dynamic analysis exist [32, 31, 12, 9]. More recently the potential of concept analysis [17] (see section 5.3) has been applied to the identification of features from dynamically retrieved profiles in [9, 8, 7, 12].

5.3. Concept Analysis

Major similarities may be drawn between Concept analysis [17] and the Software Reconnaissance technique, which forms the basis for generating the reuse perspective on software. Concept analysis is a mathematical technique for describing binary relations [17]. While applied to software engineering for some time now [25, 27, 26], the technique has more recently been used as and aid to dynamic analysis [2] and even feature identification [8, 9, 7].

Central to concept analysis is the formation of a concept lattice. In terms of feature identification, this is a hierarchical, visual representation of the relationship between test cases and software elements. In [9], several sets, some identical to those derived during software reconnaissance, are identified by examining the concept lattice. In particular, the set of shared software elements that forms a basis for the reuse perspective, may be derived. However, presentation of the reuse perspective could not be easily accommodated by concept lattices.

6. Future Work

Further work in this research will seek techniques that better exploit the reuse perspective once derived. During the evaluation of our case study, the maintainer alluded to the perspective’s use when attempting to understand the system. Perhaps the reuse perspective gives insight into relationships between features that provides help when trying to understand the overall structure of that system. A rigorous qualitative analysis of the case study in this paper would also be useful in further solidifying the claims of this work.

We also see potential in the Reflexion modelling technique [1, 21, 22] as a means of partitioning the reuse perspective during a process of component recovery. Traditionally the information provided by software reconnaissance yields a starting search point in software [32]. We see Reflexion modelling as complementary to this in creating a means for domain knowledge to isolate cohesive reuse in the set. This may be beneficial as an approach to solving the partitioning problem identified in [18] during component recovery (step 2 of the process identified in figure 1), making portions of reuse more extractable.

Any components that are eventually recovered using the reuse perspective as a basis, should be carefully evaluated. We are currently investigating the use of component metrics as a means of evaluating extracted portions of the reuse perspective [3, 29].

Further research is also needed to aid the software engineer in browsing the reuse perspective for information of interest. This is especially relevant to the data portion of the reuse perspective, which the maintainer found too large to be of use. Experiments have already been undertaken using the CHIVE visualisation framework [4, 15] to present the information in the reuse perspective in a more meaningful manner.

Of all closely related techniques we see concept analysis as providing substantial benefit. Future work would see the interactive nature of the concept lattice evaluated as a means
of exploiting and exploring the reuse perspective. Reuse across features using concept lattices has already been explored for other purposes in [28]. We intend to build upon this work.

In its current state the reuse perspective focuses on reuse in the context of a selected feature with respect to the remainder of the system. However, we would also like to explore other subtly different reuse perspectives as a means of increasing the worth of the perspective to the software engineer. This would include:

- A reuse perspective for the entire system, not just from the context of a single feature.
- The correlation of reuse perspectives between two or more features. This could potentially provide information on high and low levels of reuse.
- Performance information in profiles. This may also provide information on high and low levels of reuse.

The potential to measure how reusable the contents of the reuse perspectives actually are, based upon the percentage content of known reuse, also exists. This is superficially demonstrated in 4.3.6, however, much experimentation is necessary to establish this as fact.

We have in this paper analysed the data associated with reused code, but not data reuse per se. By also examining the reuse status of data associated with reused code we may also be able to discover if associated data implements state for potential components or whether it represents passed parameters moving between components of the system.

Finally, a follow-up interview, after a selected period of time, with the maintainer of the system would be of use to assess the long-term comprehension benefits of providing the reuse perspective.

7. Conclusions

This paper attempts to create a reuse perspective of software through the use of a variation on software reconnaissance that dynamically analyses the relationships between features exhibited by the system at levels of data and behaviour (figure 6). Though the creation of the reuse perspective was implicitly demonstrated in [18], the case study used is not sufficiently realistic. For the case study in this paper we choose a large, commercial, distributed ERP and warehousing system that provides an example that is sufficiently complex to be representative of many industrial and commercial systems, therefore giving an indication regarding the scaleability of the technique.

Results from the evaluation of the case study were very positive. Interviews with a maintainer of the system would seem to indicate that the entire reuse perspective contains software elements that are reusable in the system. Even where one particular feature (feature 15) did not produce a reuse perspective in the context of the entire system it did contain reuse in the context of the business process exhibited, and, given a more careful choice of feature, this “problem” could be avoided.

Further reinforcing our view that the reuse perspective does indeed contain genuine reuse within the system in question was that appearance of large amounts of procedures prefixed by “gp” and “px.” These procedures are designed specifically with reuse in mind to be reused across several products of the company implemented in Progress4GL. The appearance of software elements designed for reuse in the reuse perspective should certainly be considered as a step towards proof of concept.

Unfortunately, using the representation we chose, the data portion of the reuse perspective proved too complex to be of use to the maintainer. We are confident, however, that this affect may be ameliorated using a suitable visualisation.

Most importantly the maintainer who participated in the evaluation found the perspective useful and wanted to use the approach during maintenance:

“it is beneficial alright...that’s very good actually, the whole concept”

Our next steps see the reuse perspective supplemented with other techniques such as concept analysis, and reflexion modelling to allow us to mine the reuse perspective for useful information during the processes of component recovery and software comprehension.

Acknowledgements

We would like to thank the contribution of our funding and research partners, QAD and Enterprise Ireland, the Software Architecture Evolution group and the B4-STEP project at the University of Limerick.

References


