This paper is to appear a the Third Working Conference on Reverse Engineering (WCRE-3) held in conjunction with ICSM-96 in Monterey, California, November 1996.

Lessons from a Domain-Based Reengineering Effort

Jean-Marc DeBaud
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
debaud@iese.fhg.de

Abstract[1]

We present in this paper the lessons and insights learned from of a domain-centered reengineering effort. Using a method we developed in a previous work, we set about to understand and transition a complete legacy system from COBOL to an executable domain model. Our work suggests that a domain-based approach is very promising but a number of issues remain to be better understood. Among these are questions about domain completeness, scoping, interleaving and evolution; concept matching at the granularity of both the programs' architecture and the details of the source-code; thoroughness and representation of the legacy programs coverage, as well as the problems inherent to the transition of a multi-programs system. We discuss these issues in details using examples. Implications on future work in the area are suggested.

Keywords: Program reengineering, domain analysis, domain interleaving, reverse engineering, program evolution, program understanding, software architecture, object-oriented frameworks.

1. Introduction

1.1. The Problem

It has been argued that a fundamental problem in software reengineering is to understand the purpose of a program, i.e., *what* a program does. After all, the very *raison d'etre* of a program is to model or approximate some aspects of the real world via structural and computational means. This is a hard task that require domain knowledge.

Application domain modeling provides key concepts to facilitate program context, or purpose, comprehension [6] [12]. We have used this principle to introduce and experiment with a domain-based reengineering technique [7]. The use of this method has been rewarding. The domain knowledge and its representation enabled an accurate, efficient and rapid understanding and recording of a program's purpose. Yet, a number of issues became apparent while we were performing this work.

The most fundamental issue one faces when using a domain model to perform reengineering activities concerns the nature of a domain: As a principle, the scope of a domain model is arbitrary. More importantly, a domain may be interleaved with other domains either within some of its constituents or within a program. At the same time, the model must also remain flexible enough to meet evolution requirements. These points are in fact symptoms of the subjectivity principle [11] [20]. This principle holds that for most types of modeling activities, no single model can truly and adequately describe the objects and relationships involved. These are bound to vary among different applications. This principle has a direct consequence when using a domain model to help reengineer systems: In essence, there is simply no hope of capturing, and therefore using, *the* domain model across systems. We must make do with a domain approximation. This entails a strong need for adaptation ease: the model must be flexible.

Another problem arises from the difficulty of matching domain concepts at the granularity of either the architecture or the details of the source code and recording the results. This involves marking the legacy code with the matching attributes of domain representation schemas (templates). The difficulty in doing this stems from the delocalization of the source-code corresponding to the domain templates. On a larger problem scale, the source-code in the programs of a system must be thoroughly

1. This work was performed while the author was at the Georgia Institute of Technology in Atlanta, GA, USA.

matched and transitioned for a reengineering effort to be successful. This compounds the difficulty.

It also became apparent, as one would expect, that reengineering a complete system gave rise to more problems than reengineering individual programs. We found there a need to evolve the domain model according to application specific criteria and to handle the flow of operations (we did not attempt to solve the later problem).

In contrast, *how* a program operationalizes its purpose is more a syntactic matter; one that concerns structure and control flow. In this realm, good progress has been made by analyzing programs according to the lexical, syntactical and semantic rules for legal source-code constructs. Tools such as Reasoning systems' Software Refinery [21] alleviate most practical problems in discovering a program structure and control flow.

1.2. Advantages of a Domain Based Approach

A domain is a problem area. Typically, many applications programs exist to solve the problem in a single domain. Arango and Prieto-Diaz [1] give the following prerequisites for the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain.

According to Neighbors [18], domain analysis "is an attempt to identify the objects, operators, and relationships between what experts perceive to be important about the domain." As such, it bears a close resemblance to traditional systems analysis, but at the level of a collection of problems rather than a single one. Domain engineering/ modeling/analysis is an emerging research area in software engineering. It is primarily concerned with understanding domains to support initial software development and reuse, but its artifacts and approaches will prove useful in support of reverse engineering as well.

In order for domain analysis to be useful for software development, reuse, or reverse engineering, the results of the analysis must be captured and expressed, preferably, in a systematic fashion, hence the need for a representation method [2]. Among the aspects that might be included in such a representation are domain objects and their definitions, including both real world objects and concepts; solution strategies/plans/architectures; and a description of the boundary and other limits to the domain. An unresolved issue, of importance both to software developers and reverse engineers, is the exact form of the representation and the extent of its formality.

What role might a domain description play in reverse engineering a program? In general, a domain description can give the reverse engineer a set of expected constructs to look for in the code. These might be computer representations of real world objects or algorithms or overall architectural schemes. Because a domain is broader than any single problem in it, there may be expectations engendered by the domain representation that are not found in a specific program. Because a program is not always accurate or up-to-date, there may be things missing or incorrectly expressed in the program, despite contraindications in the domain representation. And, because a program is often used for more than one purpose, it may include components that do not appear at all in the domain representation.

Nevertheless, a domain representation can establish expectations to be confirmed in a program. Furthermore, the objects in the domain representation are related to each other and organized in prototypical ways that may likewise be recognized in the program. Hence, a domain representation can act as a schema for controlling the reverse engineering process and a template for organizing its results.

1.3. Research Context

The software system that we analyzed for this research is the Installation Material Condition Status reporting System (IMCSRS) [3]. This standard U.S. Army management information system consists of approximately 10,000 lines of COBOL code, broken into 15 programs. IMCSRS is responsible for using input transactions to update a master file and then to produce a variety of reports describing the status of Army materiel.

The domain that we have chosen to use is Report-Writing. This is a mature, well-understood domain that has been successfully modeled by database management system vendors in the form of report-writing tools. By report writing we mean that a program's responsibility is to generate an output report whose contents represent and/ or summarize data taken from one or more input files.

1.4. Related Work

DRACO [18] is the seminal work on domain engineering applied to the context of software engineering. This work has been followed by a number of research endeavors, among these are [14], [15], [16], [23], as well as surveys of the field [1], [2].

Harris et al. [10] take an application domain independent but architecture centric approach to program under-

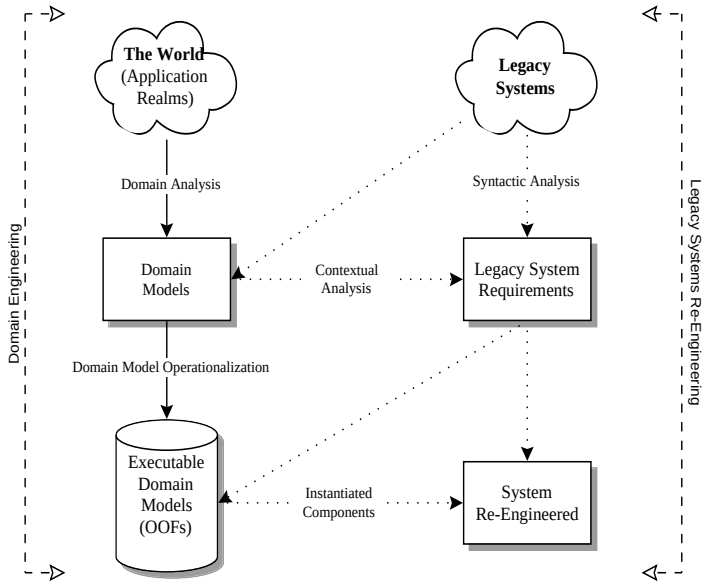

Figure 1: Using Domain Model for Software Reengineering

standing. In effect, architectural *plans* are constructed to statically match a program architecture and attempts to classify the recovered artifact with a taxonomy of architecture types. Ning et al. [19] also use a similar concept to construct and classify generic design *clichés* from abstract syntax trees (AST) that are then used as patterns to understand programs. This work is also application domain independent. Our work follows a somewhat similar idea to Ning et al. but we are using a well defined domain model to help guide the program understanding step.

Hildreth [12] proposes to use the existence of an ad-hoc domain model, i.e., non-formal, to help recover program requirements. In successfully recovering TCAS requirements from specification, Hildreth exposed the power of domain-centered reverse engineering for requirements recovery.

This paper is organized as follows. Section 2 presents a quick overview of the domain-centered reengineering method we have used in this work. Section 3 describe the application of the method with a strong emphasis on the program understanding side. Section 4 presents the analysis and issues for future work. Section 5 concludes.

2. A Domain-Based Reengineering Method

There are two phases in software artifact reengineering. In the first phase, the reengineering of a software artifact entails the comprehension of what it does and how it does it. This phase traditionally corresponds to reverse engineering. In the second phase, the artifact is evolved using both the information gained in the first phase and new requirements. Because of the complexity of each of the above phases, system reengineering is a difficult task.

The gist of our reengineering method consists of the construction of an executable, domain-specific reuse infrastructure and its use to drive, record and evolve software artifacts. The main steps of the method are shown in Figure 1. First, an application domain must be chosen. Second, a domain analysis is performed upon that realm and a domain model is created as an aggregation of domain concepts and relationships. Domain templates are created at this stage for the purpose of program understanding and are language dependent. Third, the domain model is expressed in executable form. The particular technology we use is object-oriented frameworks [13]. At that stage, what is indeed a domain-specific reuse infrastructure is in place. Fourth, the application domain model is used to guide artifact comprehension, i.e., the reverse engineering. By a process of instantiating the object-oriented framework, the results of the artifact comprehension are recorded. This process in fact amounts to specializing the reuse infrastructure according to the recovered artifact specifications. At the end of the artifact understanding process, its functionality is replicated by the framework.

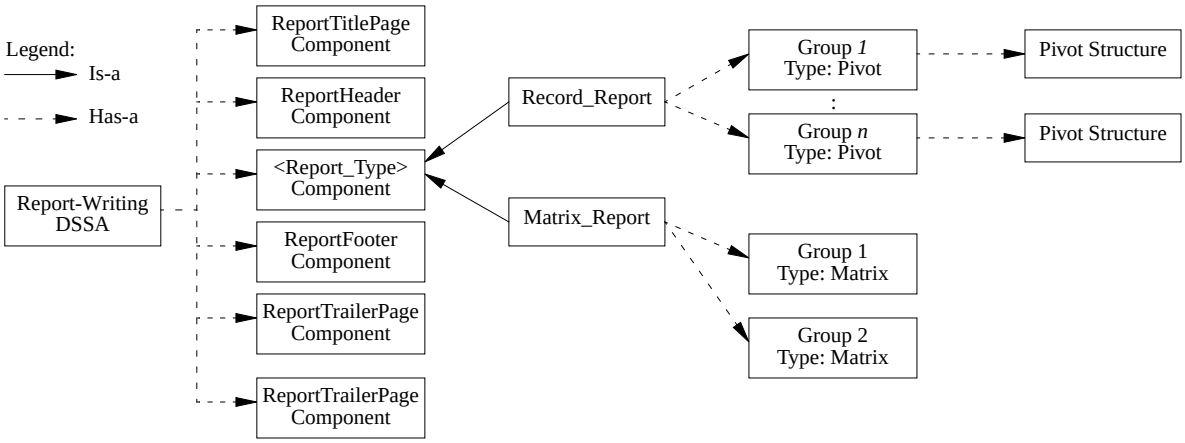

Figure 2: Structural View and Components of the Report DSSA.

Now, as the fifth and last step, the evolution of the artifact can begin. This is done by augmenting and/or modifying the previous set of instantiations.

While the process of domain analysis and framework construction is difficult and time consuming, we found our efforts rewarded in many respects. First, we experienced a substantial improvement in the time it took us to comprehend existing programs. We estimate, conservatively, to have speeded the understanding step by a factor of two. Second, recording the artifact specifications generated from the comprehension process was vastly simplified by simply having to instantiate and parametrize the framework. Third, artifact evolution was also greatly simplified because that process meant evolving the artifact specification as opposed to the source-code. Our experience shows that complex artifact evolution became a matter of minutes for someone familiar with the domain [7].

3. The Reengineering Experiment

To provide a context to the experiment, we first describe in some details the domain model that we used in our attempt to understand the system. Once this is done, we describe the experiment and its results.

3.1. The Report-Writing Domain Model Representation

Figure 2 presents a structural (surface) view of the Report domain-specific software architecture (DSSA) [24]. There are two main types of reports. A *report* is either a *Record_report* or a *Matrix_Report*. It also consists of other components such as *ReportTitlePage* whose main purpose is for general presentation and of *Report_init*, an initialization component. We concentrate here on Record_report for the sake of simplicity.

A Record_report is constructed from the sequential output of Pascal-like records or SQL-like rows taken from a master file. The selection of the records can be constrained according to different criteria. A report can also be partitioned and summaries can be accumulated over these partitions or over the entire report. In addition, text chunks can be added and parametrized to add diverse information to enhance report readability. Examples of such additions are report headers and footers. A *group* is a conceptual segmentation of a report where the unit of computation concerns one main input file over which a *pivot* is defined. A pivot is an artifact that partitions a sequence of records originating from an input file according to constrains defined upon the values of a set of data fields. Each group has a at least one pivot structure.

As we mentioned earlier, the principal benefit of a domain model for reengineering is the set of expectations it generates. These expectations can then be used to design templates for matching the domain concepts to the legacy system source-code. A partial catalogue of these templates for the Report model is showed in Table 1 below.

Templates are read sequentially from top to bottom -- though extraneous code may appear in between the template segments (sub-templates). They are algorithmic by nature. To explain what a domain template is and how one would use it, we take a closer look at the pivot template. This template represents the main mechanism to get and process the master file rows obtained from a repository. It is an architectural template.

The first statement indicates that, as a pre-condition to using the template, the master file should be sorted. In the first step, the statements associated with opening the master file must be found. So must the statements for reading the master file and the test for the end of file. In the course

Domain Concept	Description	High Level Template
Pivot	The role of the pivot is to control the segmentation of the flow of records according to potentially multiple keys and to constraints on their values.	*Pre-condition*: Stream of records in master file is sorted <open_masterfile> <read_masterfile> <EOF_?> *yes* <at_least_one_row_processed_?> yes <process_summaries> <end> <set_end_presentation> *no* <end> *no* <new_pivot_?> *no* <process_row> <process_summaries> <goto_read_masterfile> *yes* <first_time_?> *yes* <set_summaries_default> <set_begin_presentation> <goto_read_masterfile> *no* <set_presentation> <process_summary> <set_summaries_default> <goto_read_masterfile>
Summary	The role of a summary is to accumulate data information over a number of records.	*Pre-condition:* None <variable_is_declared> <variable_is_set_to_default> <variable_is_set_a_record_related_value> ;even indirectly <variable_is_send_to_output_stream>
Output_action	The role of Output_action is to signify to an artifact that a variable (or a set of these) is ready for the output stream. That concept also covers the buffer variables used for that effect.	*Pre-condition:* Master file is open and read at least once. <variable_set_definition> <variable_set_affectation> <output_action of variable set>

Table 1: Sample Domain Templates for a Record Report.

of the search, multiple candidate set of statements may be found. The correct one may only be (and is often only) certifiably found in conjunction with the identification of others. For instance, it is often the case that multiple files could serve as the master file. Yet it is only when a loop structure enumerating the records of a particular file is found that a high degree of confidence can be ascertained that this file is the master file. A certitude is acquired only when the file records are actually recurrently used for report purposes (row output or summary calculation).

The process goes on in a similar fashion until the rest of the template is found to match relevant source-code statements.

The domain concepts can be found to overlap and complement one another. This is the case in the Report domain. For example, the concept of pivot uses both the domain concepts of summary and of output_action in the <*process_row*> and <*process_summary*> sub-template. Hence, domain templates can overlap, and therefore, some source-code can be a used to match multiple templates.

One benefit of using a domain model for reengineering is the leverage given by the existence of an architectural view in the overall understanding of a program. Once the main program structure has been identified, and assuming it matches one formulated by the model, the remaining work closely relates to one of filling the blanks. The pivot template exemplify this observation.

To complete the process, the Report domain model is transitioned to an executable form using the object-oriented framework technique. This technique is highly suitable because of its customization properties and advantages; and this is especially true when one uses path expressions to document the framework dynamic structures [4], [5]. Domain templates serves as the translation mechanism to the framework.

3.2. The Experiment

The experiment was performed upon the IMCSRS system which we described succinctly in section 1.3. Each of the 15 programs that compose it were examined and their source-code matched against the domain templates.

The results are presented in table 2. As expected, the

Program Name	LOC Number	Approximation of % Recovered	General Comments
s.p01agu	1177	72%	Purpose: Creates and/or updates the UIC master file. Reeng.: Three 'reports', reporting used to create/check/validate the data.
s.p02agu	1153	79%	P.: Replaces or updates the ECC/LIN master. Re.: Same as above. Complex summary/update operations
s.p03agu	494	71%	P.: Modify principal master file to accommodate P and Q cards. Re.: Complex logic of row modifications.
s.p05agu	1000	86%	P.: Creates the 2406 valid transaction master level file. Re.: Complex logic and details.
s.p06agu	751	0%	P.: Matches the edited 2406 file to the control file. Re.: No real reporting here.
s.p07agu	1009	46%	P.: Produces a 2406 report by UIC and master files for other reports. Re.: Some reporting.
s.p09agu	226	24%	P.: Create and filter master files for further processing. Re.: One summary report produced.
s.p10agu	639	92%	P.: Proccesses IA and IB master files. Re.: Straight reporting. Little difficulty.
s.p11agu	258	84%	P.: Create a master file. Re.: Complex and obscure logic but fine if one focusses on reporting.
s.p12agu	616	87%	P.: Produce end of status report. Re.: Straight reporting.
s.p13agu	618	91%	P.: Compute operational equipment readiness. Re.: Straight reporting.
s.p14agu	609	88%	P.: Equipment availability density report. Re.: Straight reporting.
s.p15agu	437	87%	P.: Consolidated equipment avail. density report (mul. locations). Re.: Straight reporting.
s.p16agu	271	84%	P.: Equipment readiness summary. Re.: Straight reporting.
s.p20agu	502	89%	P.: Produces skeletal 'O' and 'P' card reports. Re.: Straight report, complex interdependent summaries.

Table 2: Recovered Report Concepts from the IMCSRS System

system was not only about elements of the Report domain. Hence, the system's source-code could not be entirely matched to Report templates. Nonetheless, we found the coverage rather significant because the domain model did not cover the specifics of the system's task (which was again to update and report about equipment readiness status). Using a weighted average, the sum of, the number of source-code lines * the percentage recovered; and divided by the total number of lines in the system, we arrived at the figure of 71.5 percent of the system recovered by using the Report domain model (we stress again that this is an approximation). This shows that a high percentage of the system's code was spent for the report's logic and not for equipment readiness status. We believe that an equipment readiness domain model would have enabled us to match most of the remaining system source-code with corresponding domain concepts.

For each program, the percentage of recovered source-code is an approximation that was computed as follows. The code was scanned, in the context of this experiment, by a domain expert. First, we sought to match an architecture template to the source-code. Then, using a tentative architecture match, the adjacent or related domain concepts were used for relevant source-code matching. When the source-code matched a domain template, one was attributed to the corresponding set of instructions. At the end, we had a set of domain concept matches and the source-code lines that embodied these were added together. By dividing this total by the total number of lines in the program, a good approximation of the percentage recovered was obtained. Of course, one source-code intruction used by different domain concepts would be counted only once. This situation occurs when source-code intructions interleave concepts.

In terms of time, reminding us that this effort was hand driven, we estimate that it took us about one hour for every 150 lines of source-code. This is an average. As we progressed, the common style used throughout the system as well as the experience we acquired one program after another made us more efficient in performing the program understanding.

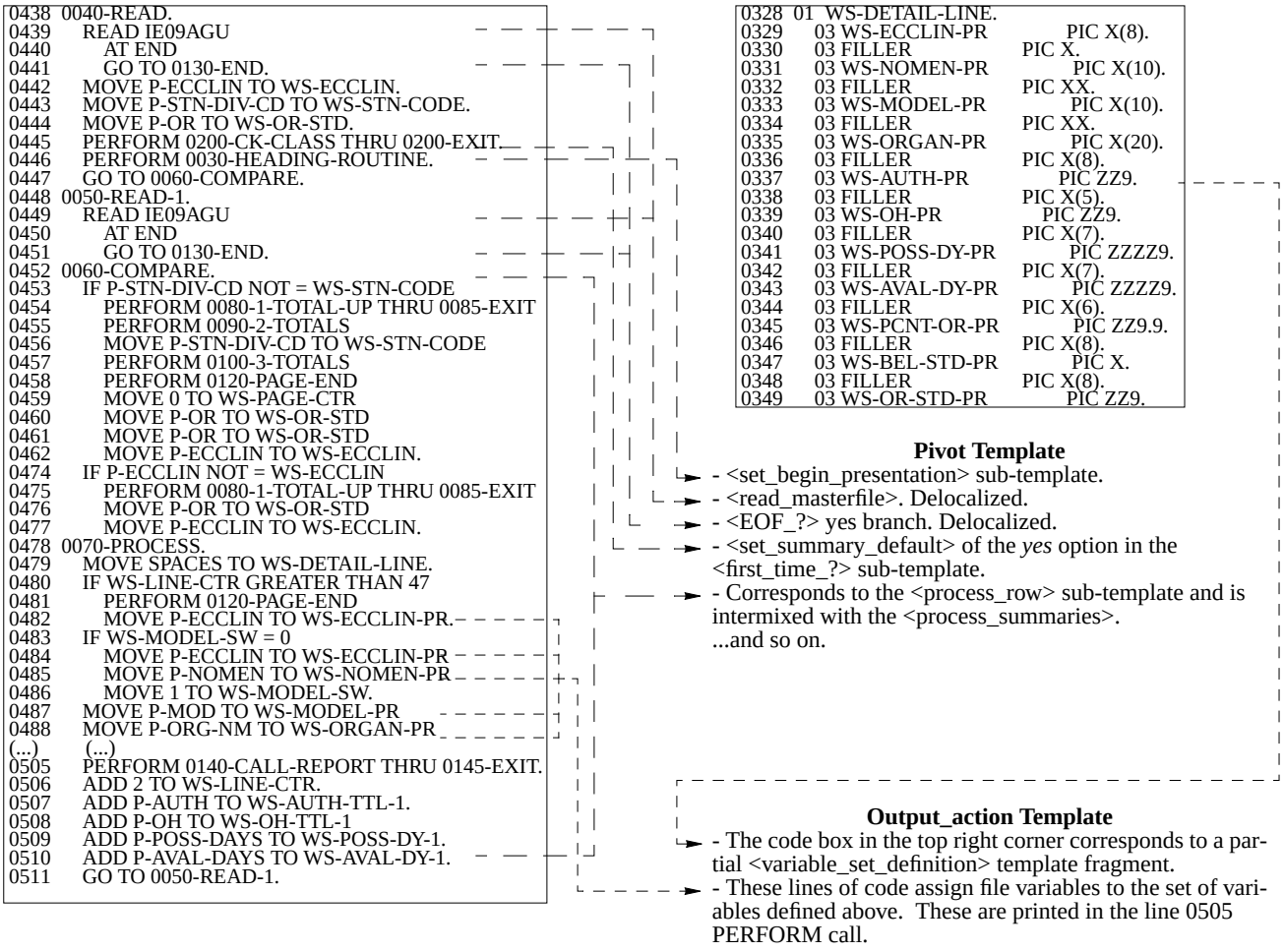

Figure 3: Code Fragments from s.p14agu with Annotated Matched Domain Concepts

4. Analysis and Recommendations.

The positive aspects of using a domain model for software reengineering have been discussed at some length in [7] and to some extend above. We choose here to concentrate on some of the difficulties that must be addressed to make using domain knowledge more effective within a reverse-engineering context. We believe there is a significant potential to a domain-based approach and that the approach should be pursued further. In summary, the experiment presented in section 3 encountered hurdles that can be grouped in four principal categories: domain subjectivity, domain concepts delocalization, insuring the completeness of program understanding coverage and what we would refer broadly to as the 'system complexity'.

4.1. Domain Subjectivity

As we mentioned in the introduction, domain models are only properly scoped, well defined or flexible up to a certain point. It is all rather subjective. The Report model is no exception to this rule as this reengineering effort has demonstrated. Subjectivity entails a need for adaptation capabilities in the process and the structure used to define, design and implement a domain. In addition, the notion of domain itself is imprecise.

A body of well encapsulated understanding is probably what leads one to think about it as a domain. But that, again, is subjective. The Report domain in fact is an aggregation of a multitude of sub-domains. The domain encompasses the page definition and management subdomain, the data repository definition and operation subdomain, the summary sub-domain and the presentation sub-domain; only to name a few. For each sub-domain the subjectivity question applies and so on. This web of subdomains, creates interleaving because of the 'contractual' specifications one may push on another.

The domain subjectivity problem is re-enforced by

domain interleaving. It is well know that designs and concepts can be interleaved [22]. As we saw in the previous paragraph, that is also the case for domains. Interleaving compounds the subjectivity problem because it reduces the clarity and localization of the definition and specification of a domain.

As an example of the fluidity of one's perspective or situation, consider the following. We were able to use the reporting domain model to understand and represent s.p05agu though this program only dealt with creating, from input files, a new master file used in subsequent reports. What needed to be changed from a common report standpoint was simply the definition of the output stream (to a file). In fact, we formed the conjecture that *reporting* could be quite similar to *filtering* and we found out this is the case. In a nutshell, the pivot structure corresponds to a filter, the selection constraints map to those of the filter and the selected fields and summaries correspond to the morphing criteria used to define the filter output. Hence, some domains share deep similarities whereas they may be quite dissimilar at first glance. Only a thourough understanding or modeling activity could reveal such similarities.

As a result of our experience, we would approach the subjectivity problem, for the purpose of constructing a domain model, with the following observations:

- Pragmatism when designing a domain model must be the rule. When a set of features appear complete, one must move on. Overdesigning does not gain one anything since there is no such thing as 'the correct' model.
- One must be ready to change one's mind about the nature of the domain as we had to do for Report. There are substantial benefits in viewing domains from different angles, not least one where a domain could be used to understand previously thought to be orthogonal programs.
- Thinking about a domain in terms of sub-domains and modeling it that way greatly helped increase the flexibility properties of the domain: clean concept interfaces meant easier customization.
- Using a notation which clearly showed the dependencies as well as the pre and post conditions among domain concepts helped tremendously in extending/ modifying the domain model.
- In the engineering of a domain, one should think hard about what is static and what can be potentially customizable, i.e., what are the situations where it is likely that one will encounter cases that could not be elicited/enumerated at the time of the analysis. Then one must plan in consequence and leave the model or the domain concept open. In the Report model, an example of such a situation is found with summaries. It is unlikely that one can enumerate every possible summaries, hence handling capabilities for a user defined summary must be provided. What is important is for the logic to treat the summaries correctly to be in place.

4.2. Delocalization of Domain Concepts

By far the hardest part in our method is to match the relevant domain concepts to the source-code. Figure 3 presents code fragments from one of the system's programs and their partial assignment to domain concepts.

Delocalization occurred in many instances in this example. One type of delocalization, or *replication* delocalization, is represented by the <read_masterfile> subtemplate. Here, there are two statements reading this particular file (lines 0439 & 0449). The first only reads the file once and at the beginning of the run, where the second actually embodies the top of the row process loop. Replication delocalization is also found to occur with the <EOF_?> sub-template (lines 0440-0441 & 0450-0451).

Another type of delocalization, or *recursive delocalization*, occurs when there is one or more indirect computations between an original piece information and the end results. Summaries can belong to this category as they may be defined using multiple computation steps.

During this reengineering effort, we found ourselves in dire need for a tool to help us recording our concept recovery results, if not to automatize it. After the first few programs, the task became rather tedious to perform. As a preliminary statement, such a tool should:

- Implement a marking and display method to help with both types of delocalization. This could be done by using point and click gestures associated with template instantiation. The logic of which and their particular coverage completeness could then be checked in an automatic fashion.
- Help adapt the template notation to account for possible or likely delocalization events. How could they be anticipated?
- Help the storage and documentation of encountered template exemplars so as to help decision making in future cases using a reference mechanism (for the matching process).
- More generally, the template notation should provide support for matching advice, partial instantiation criteria and, if possible, some form of formal reasoning. The need to show partial ´reference´ cases to help the

match is strong as we saw earlier. Not every template elements has to be matched fully to make sense in the domain, yet a minimal instantiation set should be definable. Formal reasoning used to check logical properties of the domain remains an important goal, if only an elusive one.

4.3. Completeness of Program Understanding Coverage

The previous sub-section provides a sense of complexity of matching domain concepts to source-code. In understanding and representing a program, the next arising question is how complete or thorough is the coverage? The simplest but so far most representative measure is to count the total number of lines assigned to domain concepts in a program -- with the following caveat: It is important to have matched the main architecture of the program to one concept. Otherwise this measure is a little weak.

Once the main architectural template(s) is/are matched to parts of a program source-code, then the rest of the program can be matched and attributed to domain concepts with relative ease. But there again, problems can arise during the consolidation/integration phase of the domain concepts together. For instance, domain concepts could be insufficient to represent a program's behavior or they could be only partially completed. In the first case, the domain may need to be extended and hence the relevant template(s) modified. In the second, a decision must be made whether or not the partial match is enough.

We would complement the tool specifications presented in the previous sub-section with the following broad goal: A graphical and global trace of the program understanding coverage would be very helpful. More particularly, the tool should:

- Provide the capability to follow the integration of one domain concept with another and ultimately show whether the coverage is complete or not with synoptic views of the state of the matter.
- Help manage the domain model extensibility process. As newly observed domain exemplars are encountered, they should be integrated in the tool and reused later on -- though particular care should be exercised in checking the logic of the domain when exemplars integration take place.

4.4. System Complexity

The Report domain could not cope entirely with the complexity of the IMCSRS system. As mentioned before, the reasons for this was that the system had other goals than simply generating reports. Yet, we found interesting to try to anticipate the types of changes to the model or to the reengineering mechanism that could tackle such a development within some level of generality.

We first noticed that the Report model is an *horizontal* domain, i.e., one that provides technology services[1]. This is in opposition to the *vertical* domains, i.e., those which utilize horizontal domains to model real life phenomena, or more generally, application domains. Hence, the likelihood that every aspect of the IMCSRS system could be found to fall within the Report model was slim at best. This point was reaffirmed by this effort.

Yet, there are valuable lessons that can be drawn from the types of interaction we noticed between Report Writing and Equipment Readiness Status (the vertical domain in the case of this study) -- though they have little lien with the specific nature of the later. These lessons are:

- Horizontal domains appear to be well encapsulated by nature (as service providers) and hence can be dealt with more easily, perhaps through an interface like mechanism. This property has been observed again in another study of domains we have performed [8]. Hence, they may lend themselves to be viewed and integrated as layers within other vertical models.
- The points of interaction among the different domains can usually be conceptually well separated (this may be a consequence of the point above). This is probably in part because domain designers also tend to decompose their tasks among domain features to help manage complexity. But when designed that way, domain model can be evolved and adapted with better ease.

5. Conclusions and Future Work

We have presented in this paper some of the lessons learned during a domain-based reengineering effort. The approach was very successful. Yet, a number of domain related issues must be resolved to scale the approach upward. We presented these issues and discussed potential solutions.

We feel the most important issues to approach first are, one, to get a better handle on the subjectivity problem, and second, to design tools to support the reengineering activities along the lines of what was presented in §4.

We are now moving towards the modeling of other

1. Other such services comprise domains such as distributed processing support, networking, or database among others.

domains. As mentioned earlier, the Report domain is an instance of horizontal domains. We are currently looking at modeling vertical domains. The end goal is to see how these different 'types' of domain models interact and how we can reengineer applications combining these two types as well as gaining experience in multi-domain reengineering.

Acknowledgement

The author gratefully acknowledge the original support of the Army Research Laboratory through contract DAKF 11-91-D-0004-0019 and thanks Dr. Spencer Rugaber for stimulating conversations and advices on this topic.

References

[1] Arango, Guillermo and Prieto-Diaz, Ruben. Domain Analysis Concepts and Research Directions, in *Domain Analysis and Software Systems Modeling*, ed. Ruben Prieto-Diaz and Guillermo Arango, IEEE Computer Society Press, 1991.

[2] Arango, Guillermo. Domain Analysis Methods. In *Software Reusability.* (Eds.) W. Schaeffer, R. Prieto-Diaz, and M. Matsumoto. Ellis Horwood, New York, 1993, pp. 17-49.

[3] Automated Data Systems Manual, Installation Material Condition Status Reporting System (IMCSRS), Functional User's Manual, Commander FORSCOM, AFLG-RO, Ft. McPherson, Georgia, April 1, 1984.

[4] Campbell R. H. The Specification of process synchronization by Path-Expressions. In *Lecture Notes in Computer Science*, pages 89-102, 1974.

[5] Campbell R. H. and Islam, N. A Technique for Documenting the Framework of an Object-Oriented System. Technical report UIUCDCS-1582-93, University of Illinois at Urbana-Champain.

[6] DeBaud, Jean-Marc, Moopen, Bijith, and Rugaber, Spencer. Domain Analysis and Reverse Engineering, Proceedings of the *International Conference on Software Maintenance*, Victoria, British Columbia, September 1994, pp. 326-335.

[7] DeBaud, Jean-Marc and Rugaber, Spencer. A Software Reengineering Method using Domain Models. Proceedings of the *International Conference of Software Maintenance*, Nice, France, October 1995, pp 204-213.

[8] Debaud, Jean-Marc and LeBlanc, Richard. Problem-Oriented Domain Analysis. Technical Report, Georgia Institute of Technology.

[9] Garlan, D., Allen R. and Ockerbloom, J. Architecture Mismatch or Why it's hard to build systems out of existing parts. In *17th International Conference on Software Engineering*. Seattle, Washington. April 1995. IEEE Computer Society Press, Los Alamitos, Calif., pp. 179-185.

[10] Harris, D.R., Reubenstein, H.B. and Yeh, A.S. Reverse Engineering to the Architectural Level. Proceedings of the *17th International Conference on Software Engineering*, Seattle, Washington. April 1995. IEEE Computer Society Press, Los Alamitos, Calif., pp. 186-195.

[11] Harrison, W. and Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). *OOPSLA* 1993.

[12] Hildreth, Holly. Reverse Engineering Requirements for Process-Control Software, Proceedings of the *Conference on Software Maintenance*, pp. 316-325, Victoria, British Columbia, September 1994.

[13] Johnson, Ralph E. and Foote, Brian. Designing Reusable Classes. *Journal of Object-Oriented Programming*, June/July 1988, Volume 1, Number 2, pp 22-35.

[14] Kang, K., Cohen, S., Hess, J., and Peterson, S. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA 15213, Nov. 1990

[15] Lubars, M. Domain analysis and domain engineering in IDeA. In *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 163-178.

[16] Mettala, E. and Graham, M. The domain-specific software architecture program. TR CMU/SEI-92-SR-9, Carnegie-Mellon Software Engineering Institute, June 1992.

[17] Neighbors, James. "Software Construction from Components", PhD thesis, TR-160, ICS Department, University of California at Irvine, 1980.

[18] Neighbors, James. DRACO: A Method for Engineering Reusable Software Systems. 1989 ACM, Inc. Addison-Wesley Publishing Co., Reading MA.

[19] Ning, J.Q., Engberts, A. and Kozaczynski W., Automated Support for Legacy Code understanding. *Communication of the ACM*, May 1994, Vol. 37, No. 5, pp. 50-57

[20] Osser, H. et al. Subject-Oriented Composition Rules. *OOPSLA* 1995.

[21] Reasoning Systems, Inc., Palo Alto, CA. REFINE User's Guide, 1990. For REFINE (TM) version 3.0

[22] Rugaber, S., Strirewalt, K. and Wills, Linda. Detecting Interleaving. Proceedings of the *International Conference of Software Maintenance*, Nice, France, October 1995, pp 265-274.

[23] Simos, M. The growing of an Organon: A hybrid knowledge-based technology and methodology for software reuse. In *Domain Analysis and Software System Modeling*. IEEE Computer Society Press, Los Alamitos, Calif. 1991. pp 204-221.

[24] Tracz, W. Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *ACM Software Engineering Notes*, 19(2), 1994.