Safe Clone-Based Refactoring through Stereotype Identification and Iso-Generation

Nic Volanschi
Metaware Technologies
volanschi@metaware.fr

***Abstract*—Most advanced existing tools for clone-based refactoring propose a limited number of pre-defined clone-removal transformations that can be applied automatically, typically under user control. This fixed set of refactorings usually guarantee that semantics is preserved, but is inherently limited to generally-applicable transformations (extract method, pull-up method, etc.). This tool design rules out many potential domain-specific or application-specific clone removals. Such cases are ordinarily recognized by humans as stereotypes derived from a higher-level concept and manually replaced with an appropriate abstraction. Thus, in current tools, generality is sacrificed for the safety of the transformation. This paper proposes an alternative approach, in which the spectrum of refactoring techniques is open, including manual interventions, while keeping strong safety guarantees based on the notion of iso-generation. Our method can operate on multiple languages and has been prototyped on a subset of a real-world legacy asset containing C and COBOL programs, with promising results.**

***Keywords*-clones, refactoring, maintainability, safety**

I. Introduction

Clone management technology [4] has largely achieved by now a level of maturity that makes it an inescapable assistant for refactoring large applications written in any programming language. The benefits of clone-based refactoring in terms of maintainability are widely accepted nowadays, by their ability to achieve significant reductions in code size, to avoid inconsistent changes of clone instances, but also to identify reusable programming abstractions in the source code, such as parameterized procedures or cross-cutting concerns.

Successive generations of clone-based refactoring tools gradually improved their effectiveness. Historically, most research has been performed on clone detection technologies and resulted in many effective and scalable tools, able to find many different kinds of clones: textual, token-based, syntactic, or semantic clones, identical or similar clones, etc., covering typically between 5% and 23% of an asset [4]. Scalability has definitely been proven by analyzing several assets at once to detect inter-project code reuse [7], or by comparing many different versions of a same system such as Linux [6].

However, blindly removing all the clones detected in a program can effectively reduce its size, but is not the right solution to improve maintenance, because (1) many clones do not correspond to any useful abstraction, and (2) it has been shown that some clones come from valid design patterns, and their removal could actually deteriorate the maintainability [3]. Clearly, only a fraction of the detected clones are good candidates for removal to increase the maintainability.

To help the programmer in this difficult selection, some tools further classify the clones according to different taxonomies, and sometimes even suggest several clone removal techniques that are applicable to each clone. Based on this higher-level information, the user can choose the appropriate refactoring technique and apply it manually [11], [12]. As the transformations may be quite complex, involving checking pre-conditions, renaming variables, moving and adapting the clone instances, these manual operations can be tedious and error-prone.

To address this last aspect, some more recent clone management tools, pluggable into an integrated development environment (IDE), may also perform some clone removals automatically under user interactive control. This sometimes guarantees that all the pre-conditions are met and that the transformation preserves the semantics of the program. Unfortunately, this tool design unavoidably limits the number of clone-removal transformations to a fixed set of generally-applicable techniques. For instance, if the subject language is object-oriented, transformations such as: extract method, pull-up method, and so on, are proposed [9], [10].

This highly-automated tool design has two main limitations. First, the clones identified by even the most advanced techniques may not delimit exactly the most useful underlying "concept realizations" [2]. Sometimes, close manual inspection may conclude that the identified clones have to be slightly restricted or extended, or that several neighboring clones have to be fused together so that they represent a useful and easy to understand operation, replaceable with a programming abstraction such as a new procedure. Secondly, the transformation needed to replace the clones with a programming abstraction may not be a generic technique but an application-specific one. For instance, some textual variations or repetitions can only be captured by custom code generators that cannot currently be synthesized automatically. Sometimes, the replacement involves recognizing and unifying some unjustified differences between the instances that come from incoherent evolutions.

Summarizing the above discussion, users have to choose today between manually transforming the programs based on clone information, with the risk of introducing errors, or using highly-automated tools to safely perform some fixed set of generic refactorings, usually operating exactly on the automatically-detected clones.

This paper presents an alternative method for clone-based refactoring, aimed at improving the maintainability of an asset, which is open-ended in that it integrates manual clone interpretation and manual transformations while keeping the risk of introducing errors very low.

In the first phase of our method, starting from the results of a clone detector, the user localizes *stereotypes* in the source code, that are usually indicated by clones or groups of clones, and selects those that correspond to useful abstractions, which can be generic, platform-specific, application-specific, domain-specific, etc. In a second phase, the user re-implements the selected stereotypes as code generators, using support already integrated in the development environment (e.g., macros in C), or added as a simple pre-processor. Then, an *iso-generation* test is performed, which consists in verifying that the code generated from the refactored source is identical to the original code. For more generality, a list of differences can be produced by comparing the original and regenerated versions; the user then validates this list (this operation can be semi-automated or fully automated).

The refactored code can be the final outcome of the method if it is acceptable to maintain a source code that contains code generators. Whenever using code generators is not an option, for any reason including local programming standards, incompatibility with the existing programming environment, etc., a final phase of our method consists in automatically replacing the code generators whenever possible with standard language abstractions such as procedures, and expanding the rest of the generators back in the source. Even for these last cases, an added value exists for the maintainer because the re-expanded code is delimited by comments in the code indicating the call to the corresponding generator, including its arguments if applicable. Thus, the meaningful clones are either removed from the code or thoroughly documented as higher-level abstractions.

We applied this method on a subset of a real asset of one of our customers, containing both C and COBOL programs, with encouraging results:

- the final code size reduction obtained, even in absence of code generation, is significant, of 24% on average on our sample, and up to a maximum of 40%
- we report several kinds of useful stereotypes, not detected *per se* as clones, whose refactoring can improve maintainability
- a high fraction of the code generators can be replaced with standard programming language abstractions, even in languages where abstractions are scarce and pretty low level such as in COBOL.

The context of the real project on which the method was prototyped is described in Section II. The quest for stereotypes is described in Section III. The refactoring using code generators is presented in Section IV on C programs, and in Section V on COBOL programs. The reduction of code generators to standard language mechanisms is described in Section VI. The final results of the refactoring method on a sample are given in Section VII. Related work is discussed in Section VIII, and Section IX concludes.

II. Project context

Metaware Technologies is a company specialized in modernization services for legacy assets, such as: migrations from a platform to another; software quality assessment, monitoring, and improvement; mass re-engineering projects such as extending a social security number with additional digits in a complete, multi-language asset; etc. Our company offers these services to customers in banking, finance, insurance, and retail, among others. A general characteristic of these customers is that they critically rely on their software asset, and therefore usually impose very tight availability constraints on their IT system.

The present refactoring method was developed in response to the need of one of our customers, whose name is kept undisclosed for confidentiality reasons, a leader in some specialized financial services. Our customer possesses an in-house developed software asset whose core part runs on an IBM z/OS mainframe, and must handle a high number of transactions (thousands per second) under peak load periods lasting only a few minutes, and occurring a few times per day. These transactions represent computations involving important sums of money. Therefore, a breakdown of the IT system during these peak loads can result in losses of the order of a million euros. Currently, the system performs satisfactorily, but its evolution is becoming too expensive and too slow with respect to the changing demands of the business domain. In short, the agility of the system was identified by the customer as being clearly insufficient, and they expressed the need for a significant improvement along this axis. We believed that clone-based refactoring could be an effective solution by reducing the code size and increasing the level of abstraction.

The mainframe asset consists in about 4 MLOC (millions of lines of code), among which 3.5 MLOC of COBOL and 0.5 MLOC of C. The asset is divided in 14 subsystems. The analysis effort was concentrated on one subsystem called KER, designated as critical by its central role in the business domain. The KER subsystem consists of 480 COBOL programs and 148 C programs. As a preliminary step, we analyzed KER using a simple in-house clone detector, part of our REFINE$^{\text{TM}}$ suite. This tool reports textually identical pieces of source code consisting of a minimum of 6 consecutive lines. The results indicated that 40% of the COBOL code and 10% of the C code were part of at least one clone instance. The unusually high degree of exact duplication on the COBOL part can be partly attributed to the characteristics of the language (verbosity and lack of abstraction mechanisms).

The results of our clone detector reporting only textually identical clones were complemented using CCFinderX [8], an open-source clone detector also reporting parametric near-miss clones. The default options were used for this tool.

III. Stereotypes

Starting from the exact and parametric clones detected by the tools, the first step of our method consists in finding stereotypes which can be usefully factored out as code generators. We analyzed a sample of the detected clones in order to infer from purely syntactic regularities the eventual underlying concepts. We introduce here some of the classes of stereotypes we found frequently in the code. Examples of each class will be given in the following sections.

Internal stereotypes: Internal stereotypes are repeated regular code sequences whose instances occur all in the same source file. They mostly correspond to local operations that cannot be easily generalized to cover other source files. We found many internal stereotypes both in C and COBOL programs, repeated somewhere between 2 and 20 times, either identically or with some slight variations.

Overlapped clones: We observed that a significant number of the detected clones in both C and COBOL programs were overlapped, and we saw on a few examples that the corresponding code sequences were remarkably repetitive. In fact, it is easy to see that in general, two overlapped exact clones hide a more regular structure. For instance, if the overlapped region is exactly the half of one instance, the code consists of three repetitions of a smaller clone, because the shared part a must appear at the beginning of the first instance (because it begins the second instance), and also at the end of the second instance (because it ends the first instance). The other cases also present sub-clones. Therefore, we inspected overlapped clones with special priority.

Host variables: As many COBOL programs are using embedded SQL queries, they contain host variables corresponding to tables in the database. Some host variables were written by hand, while some others were initially generated with a standard include file generator called DCLGEN, and then textually incorporated in the programs for eventual customization. In fact, very few of these copies have really been customized, so they mostly show up as clones.

Stereotyped SQL INSERT queries: Once declared, the host variables are used in various SQL queries. Among them, they are numerous SQL INSERT queries that are very regular: they consist in inserting the values of all the columns in a database table row, starting from the fields of the same names in the host variable. Given that many tables have dozens of rows, these long queries would better be generated starting from the database schema description. In fact, this is one possible generalization of the DCLGEN generator. Indeed, DCLGEN automates only the declaration of the host variable, but not its stereotyped uses.

Stereotyped SQL cursors declarations: One program, for instance, contained as much as 18 cursor declarations of about 50 lines each, that corresponded to different lookups in the same table. The table contains two keys; the different cursors look up the records matching, not matching, or ignoring each of the two keys (which gives 9 different combinations), and some cursors retrieve a few extra fields, and slightly reorder the fields (which doubles the previous 9 combinations). It is clear that factoring out this kind of stereotypes can bring important savings at maintenance time, and also during development.

Stereotyped procedure invocations: In COBOL, user procedures internal to a program are implemented as sequences of statements labeled by a paragraph name. In practice, paragraphs are not only used for local procedures, but also as small shared procedures used by many programs. For this purpose, paragraphs are sometimes declared in a shared include file (known as a “copycode”). Paragraphs cannot handle arguments, so the copying of parameters from/to the caller is handled by MOVE statements before and after the call. This tends to create stereotyped sequences for invoking paragraphs in a program, or even across programs. We found, for instance, a dozen of shared paragraphs called about 1600 times in the KER subsystem, implementing some marshaling of different data types such as date, binary, and string. The corresponding invocation sequences and checks of the result have very frequently the same form or a few similar forms, and therefore constitute useful stereotypes.

Duplicated paragraphs: Some paragraph names appear in many different programs, and frequently contain the same or similar code. These paragraphs usually correspond to technical stereotypes such as standard error handling, initialization of system services, exit protocols from a program or transaction, etc. For instance, a paragraph called START-SQL-ERR appears in 79 programs of KER, most of the time with the same contents, that corresponds to a default error handling.

Structural clones: By listing the most frequent paragraph names, we found a group of 68 programs all containing the same 9 paragraphs at the end, always with the same content. A closer inspection of some programs in this group revealed that they really constitute a program family, sharing a same skeleton. Part of this common skeleton comes from the platform: all these programs are using the CICS transaction server, so they have to declare some technical variables and implement some paragraphs for handling some events. Other parts of the skeleton come from programming standards of our customer and from the various cloning patterns [3] apparently used during development.

After the stereotypes have been identified, the next phase of our method consists in re-implementing them as code generators.

IV. Refactoring using code generators in C

The C language offers the possibility to write quite powerful code generators using its pre-processing features, including textual file inclusion, macros with arguments, and conditionals. The pre-processor, called "cpp", can be invoked alone to interpret these directives; as a result, calling cpp has the effect of executing all the code generators. The main limitation of the C pre-processor directives is that conditional compilation cannot be used within macros. In other terms, C macros cannot contain logic. This prevents implementing variant parametric stereotypes, i.e. parametric stereotypes for which certain instances contain some statements that are absent from (or different in) other instances. Nevertheless many kinds of stereotypes can be easily implemented in C.

In particular, internal stereotypes can be expressed as local macros, provided they are not simultaneously parametric and variant. We found many cases in the KER subsystem where a parametric stereotype occurs in several branches of a same switch statement. These clones, which may include more than 10 lines each, and are replicated up to 17 times in a file, do not always constitute an abstraction simple to define, and usually access many variables in scope. For these reasons, these clones have not been factored out as C local functions. However, all these clones must be maintained together, so factoring them as a local macro seems to be a good solution.

Indeed, local macros do not incur any overhead, as they have no run-time cost, nor a "cognitive" cost at maintenance: they can simply be defined in place, and used as documented clones interpreted by the pre-processor. As such, they can be used even for factoring a few instances of a few lines each. In contrast, external stereotypes implemented in shared include files should be chosen more carefully, because there is a cognitive cost in this case, as the maintainer has to know about their existence, location, meaning, and usage.

Iso-generation test: Once the code generators are written and the stereotypes are manually replaced by invocations of the generators, the verification step consists in executing the cpp pre-processor, once on the original code and once on the refactored code. The results are compared using a standard text comparison tool to produce a list of differences. Because of the fact that C macros always fit on a single line, differences in white space have to be filtered out by a simple complementary script. If there are no other differences, the transformed code is equivalent to the original code. In the more general case, the list may contain differences that can be checked manually or automatically, and which may concern: equivalent statements or even equivalent small sequences of statements; for instance, a pair of independent statements coming in opposite orders in different clones is an easily justifiable difference. These differences may of course come from certain liberties taken by the developer of the code generator to unify not only strict clones, but also stereotype instances that are syntactically distinct but semantically equivalent.

V. Refactoring using code generators in COBOL

The COBOL language also includes a simple facility that can be used for code generation: the COPY statement, which includes an external file, optionally performing some textual substitutions. The COPY facility can be used to implement simple code generators such as exact or parametric clones. However, this mechanism has some severe limitations:

- a file included by COPY REPLACING cannot include another file; therefore, code generators cannot call each other, which prevents building complex generators out of simpler ones
- there is no notion of local macro like in C; this prevents using code generators for documenting internal stereotypes
- there is no conditional compilation facility; this prevents implementing variant stereotypes.

A. Adding macros to COBOL

Fortunately, it is quite easy to add powerful code generation facilities to COBOL. At first sight, it might seem possible to use cpp as a COBOL pre-processor, when available, but that turns out quickly to be a bad idea. Indeed, cpp is designed to work well with the lexical conventions of C, and maybe other similar languages. For instance, paragraph labels in COBOL must be placed near the beginning of a new line; as a C macro must hold on a single line, this means that a macro cannot contain a paragraph and its ending label.

Rather, we developed in Perl a simple open-source pre-processor called "metapp"[1], similar in spirit with cpp, but with a more suitable syntax and less composition constraints, providing:

- local and external macros with parameters, invoked by a "#copy macro(...)" directive; the parameters are defined in the macro using the #bind directive
- conditional text inclusion using a #if directive; the conditional expressions are full Perl expressions, and are forwarded to the Perl interpreter using the "eval" function.
- invocation of any Perl statement, also passed to "eval"; this is useful for setting variables and for calling complex code generators fully implemented in Perl.

With respect to C macros, metapp unifies the invocation syntax of local and external macros, allows nesting conditional text inclusion within macros, provides default values for macro parameters, and is extensible with external code generators. It is important to notice that as opposed to C, COBOL compilers do not always offer the possibility to separately perform the pre-processing pass, so as to only expand COPY statements. Therefore, metapp not only

[1]metapp can be downloaded at http://www.metaware.fr/metapp/

enhances the code generation possibilities, but also allows to test the iso-generation according to our method.

Stereotyped SQL INSERT statements: Using this code generation support, it took us less that one man-day to implement the stereotype for SQL INSERT statements, generalizing the DCLGEN code generator: The code generation is performed by a macro written in Perl that takes the name of the database table as an argument, scans the corresponding description in the database schema, extracts the names and types of all its columns, and then generates a complete SQL INSERT statement compatible with the host variable generated by DCLGEN.

In some programs of KER, we found a number of INSERT statements that *almost* had this regular form, with a few differences: some columns in the table did not took their value from the corresponding host variable field, but rather from SQL expressions such as "CURRENT_TIMESTAMP"). To cope with these cases, our code generator accepts optional arguments to provide such exceptional values for certain fields, or to omit updating a given field:

```
# insert_into(MVCREST,
    "TSMAJ=CURRENT_TIMESTAMP", "USMAJ=")
```

This call omits updating column USMAJ, and sets column TSMAJ to the current time; all the other fields are updated from the corresponding field in the host variable. The macro generates 32 lines of code in this example. Besides the fact that it saves time during development and maintenance, the code generator also ensures some degree of coherence with the database when the schema evolves.

Stereotyped procedure invocations: Stereotyped invocations of either paragraphs or sub-programs can be easily implemented using metapp macros. Here is a typical, simple example of variant parametric stereotype, capturing calls to paragraph START-CODIF-BIN1, pervasively used throughout the asset.

```
#bind $val, $j
  MOVE 1 TO W-ZFREE02.
  MOVE $val TO W-BIN1O.
  PERFORM START-CODIF-BIN1
    THRU END-CODIF-BIN1.
  MOVE W-BIN1I TO
    W-TFREE01(W-ZFREE01:W-ZFREE02).
  ADD W-ZFREE02 TO W-ZFREE01.
#if defined($j)
  MOVE $j TO W-ZFREE02.
  MOVE $val TO W-INDIC-NUM$j.
  MOVE W-INDIC-NUM$j TO
    W-RECEP-RESULT(W-ZFREE04:W-ZFREE02).
  ADD W-ZFREE02 TO W-ZFREE04.
#fi
```

Duplicated paragraphs: The stereotyped paragraphs involving SQL error handling, for instance, can also be easily implemented as macros even when there are slight variations between the instances, expressed as #if sections.

Structural clones: A closer analysis of the program family composed of 68 programs sharing a common skeleton showed that even more sharing can be achieved by considering several specialized sub-versions of the skeleton. Using macros that call each other, it is possible to refine a common abstraction into more specialized ones, by putting the generic abstraction in a shared macro that is either called by the more specialized macros, or invokes the more specialized macros as callbacks. Thus, we put the 9 shared paragraphs and other pervasive declarations in the common skeleton (a set of generic macros), and we created a more specialized skeleton for one of the programs. The result was a code size reduction of 33% on this program, but we feel that we only scratched the surface of the potential refactoring opportunities exhibited by this program family.

VI. Reducing the code generators

After the stereotypes have been factored out as code generators and tested for iso-generation, the next, optional, step is to reduce them to native or simpler language mechanisms.

In the case of C programs, the motivation for replacing macros by function calls, whenever possible, (and perhaps declared "inline") is that for functions the compiler checks the type of each argument, whereas for macros only the number of arguments is checked.

In the case of COBOL programs, the motivation for reducing metapp macros to standard mechanisms is more related to the impact of adopting an extra pre-processor within the build chain of the company. In our experience, many companies using COBOL are already using different kinds of pre-processors, either off-the-shelf such as embedded SQL pre-processors, or in-house developed. These companies may be more open to adopt a new pre-processor. For companies where the reduction step is a requirement, it is important to know what impact this step may have on the size of the refactored source code. The impact is that only some of the metapp macros can be reduced to standard mechanisms, for example:

- macros without conditionals and not invoking other macros can be reduced to COPY REPLACING files or to paragraphs
- macros containing conditionals can always be transformed into several macro versions without conditionals, and then reduced as above; whenever the number of versions is low (typically 2), this may be perfectly acceptable.

Some other macros cannot be reduced to standard mechanisms, or their reduction would not bring the same improvement in maintainability.

For all these non reduced macros, metapp expands the stereotypes in the code, but introducing stylized comment delimiters to indicate that the code section was automatically

Program	**Original size (LOC)**	**Gain with generators (%)**	**Gain without generators (%)**
PRG041.cbl	2973	46%	40%
PRG431.cbl	2347	11%	10%
PRG101.cbl	2180	45%	42%
PRG104.cbl	2331	30%	29%
PRGPD4.cbl	701	18%	14%
PRG015.cbl	6050	24%	22%
PRG001.cbl	6470	13%	13%
PRG028.c	725	44%	-
PRG114.c	661	35%	-
PRG460.c	2865	8%	-

Table I
RESULTS OF REFACTORING ON A SAMPLE

generated. These markers are clearly an added value for the maintainer, because (1) the coherent maintenance of all instances is now greatly simplified, and (2) when the stereotype represents a technical artifact, it probably means that the maintainer can skip these sections altogether, unless there is a change in the platform. In particular, some irreducible instances can be expanded as separate include files, like the host variables generated by DCLGEN, to completely shift them out of focus; such "external code generators" can be kept and re-run de-coupled from the regular build.

The reductions from macros to standard mechanisms have not yet been automated in our tools, but these are well understood program transformations, left as future work.

VII. EXPERIMENTAL RESULTS

Because of time constraints imposed by our customers' budget for this study, we really performed the refactoring on a sample of only 10 programs from this list: 7 in COBOL and 3 in C. The sample is detailed in Table I along with the code size reductions obtained by factoring the stereotypes. For instance, the first program was reduced from 2973 lines to 1609 lines using macros and to 1813 lines without macros, which represents reductions of 46% and 40%.

The average code size reduction on COBOL programs is of 26% using macros, and this percentage is reduced to 24% when macros are reduced to standard COBOL mechanisms. The relative difference is thus very small on this sample. Even though this result is clearly insufficient to draw general conclusions, we find it particularly encouraging for the applicability of our method in companies that require the reduction step. The average code size reduction on C programs is of 29%. For C programs, we did not apply the reduction step, because replacing macros with functions should have no noticeable impact on code size.

Several factors contributed to the success of these important code reductions: the high fraction of cloned code in the initial programs, the selection of a sample covering various refactoring opportunities, the radical refactoring enabled by the safety of iso-generation, and perhaps the presence of many latent abstractions in the target business domain.

VIII. RELATED WORK

DeBaud [2] describes a refactoring method applied on an existing asset consisting of COBOL and BASIC programs, aimed at replacing some domain-specific code sections in the source code with calls to a domain library such as a report writing library. In their method, one first builds a dictionary of domain concepts, then declined as a set of code patterns; these patterns are then matched with the source code to detect the "concept realizations". While there are some striking similarities with our method, their process is not clone-based, but concept-directed. As a result, as they admit, the method cannot be performed without a domain expert. In contrast, in our method the clone detection phase obviates the need for domain expertise as concepts are inferred bottom-up from the code, rather than matched top-down starting from a domain dictionary.

Baxter et al. [1] present a clone detector that not only reports exact and near-miss clones, but also produces C macros that generates all the instances. They further suggest that such clones can indicate a higher-level concept that is being realized through a programming idiom. Our method can be seen as an elaboration of this research track, in which we detail the analysis phase going from clones to stereotypes, and the way back using code generators. As stereotypes do not directly correspond to clones, the macros cannot be generated automatically like in their work. We further explore the application of this approach on languages not providing powerful macro features such as COBOL, and add the optional step of reducing the macros to native language mechanisms.

Kapser and Godfrey [3] enumerate the development practices involving cloning, and document a complete design pattern for each, showing that each of these patterns can be valid in some situations and unjustified in others. Their main point is that the decision to remove a clone has to be carefully weighted in order to improve maintenance. In their approach, raw clone information is automatically refined to find higher-level grouped clones, covering most part of a program block. Clearly, the insights on the original driving forces for cloning can be most important during our analysis phase when concluding whether a stereotype is "useful". However, the fact that we envision refactoring using code generation mechanisms sometimes radically changes the tradeoffs considered when deciding whether a stereotype is useful or not, and extends the scope of stereotypes. For instance, if only standard mechanisms are used, the stereotyped declaration of 18 SQL cursors would clearly create a too complex API; when using a code generator, the abstraction is very easy to build, understand, and use. We also diverge on the treatment of overlapped clones, which they mostly drop.

Jarzabek and Li [13] analyze the numerous clones found in the Java Buffer library of JDK 1.5, and show that most of this code duplication can be removed by adding a code generator on top of Java, expressed in XVCL, able to regenerate the library in its original form. They obtain thus an impressive reduction in code size of 68%. Their approach is conceptually the same as ours until the iso-generation step, but an important detail differ: the XVCL language is really a meta-language embedding Java, while we do the opposite, embedding macros within the host language COBOL (or using the macros already embedded in C). This detail is very important for adoption in conservative milieus, and clearly smooths the learning curve. Besides, we show how the code generators can be reduced to fully standard programs in most cases, and assess this ratio on some concrete examples. Nevertheless, a system such as XVCL allowing inheritance between templates may be interesting when refactoring structural clones.

Cordy [5] teaches some very interesting lessons about the adoption (and resistance to adoption) of refactoring tools in the maintenance process within the industry practice, and especially in the finance domain. He points out many reasons for which brilliant solutions may fail to convince in that context, and insists, among others, on the importance of testing in such dependable systems. By radically decreasing the need to test the refactored code, iso-generation can be an effective argument for adoption in that context.

IX. Conclusions

We presented a refactoring method exploiting the information computed by clone detectors. Instead of applying this raw information by direct clone removals, higher-level regularities are manually searched in the code, and re-implemented as code generators; optionally, the code generators can be automatically reduced to standard mechanisms of the subject language. The potential risk introduced by the manual transformation is eliminated by an iso-generation step in which some differences may be tolerated. It is important to notice that the degree of confidence in the final result tends to decrease gracefully with the number of such differences as opposed to a scenario in which such a large restructuring is done with only testing as the safety net. The results on a real case tend to show that the reduction step does not seem to penalize significantly the gain ultimately achieved. The method was illustrated on C and COBOL programs, but it should be applicable to other programming languages as well, provided that code generation features are available in the language or added by an external pre-processor.

Future work will investigate semi-automating some manual steps of the method, such as the stereotypes identification. Also, once a stereotype is implemented as a macro, it would be interesting to find new instances by deriving from it a specialized matcher, perhaps more tolerant to specific variations than a standard clone detector. We also feel that there is important potential for improvement in the area of structural clone removal using a hierarchy of program skeletons.

References

[1] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. *Clone Detection Using Abstract Syntax Trees.* In Proc. Intl. Conf. on Softw. Maint. (ICSM '98). IEEE Computer Soc.

[2] J.-M. DeBaud. 1997. *DARE: Domain-Augmented ReEngineering.* In Proc. Working Conf. on Reverse Eng. (WCRE '97). IEEE Computer Soc.

[3] C. Kapser and M. Godfrey. 2006. *'Cloning Considered Harmful' Considered Harmful.* In Proc. Working Conf. on Reverse Eng. (WCRE '06). IEEE Computer Soc.

[4] R. Koschke. 2008. *Frontiers on software clone management.* In Proc. Intl. Conf. on Softw. Maintenance (ICSM '08). IEEE Computer Soc.

[5] J. Cordy. 2003. *Comprehending Reality: Practical Barriers to Industrial Adoption of Software Maintenance Automation.* In Proc. Intl. Work. on Program Comprehension (IWPC '03). IEEE Computer Soc.

[6] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue. 2007. *Analysis of the Linux Kernel Evolution Using Code Clone Coverage.* In Proc. Intl. Work. on Mining Softw. Repositories (MSR '07). IEEE Computer Soc.

[7] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. 2007. *Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder.* In Proc. Intl. Conf. on Softw. Eng. (ICSE '07). IEEE Computer Soc.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. *CCFinder: a multilinguistic token-based code clone detection system for large scale source code.* IEEE Trans. Softw. Eng. 28:7.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. 2004. *ARIES: Refactoring Support Environment Based on Code Clone Analysis.* In Proc. IASTED Intl. Conf. on Softw. Eng. and Applications.

[10] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. 2000. *Advanced Clone-Analysis to Support Object-Oriented System Refactoring.* In Proc. Working Conf. on Reverse Eng. (WCRE'00). IEEE Computer Soc.

[11] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. 1999. *Measuring Clone Based Reengineering Opportunities.* In Proc. Intl. Symp. on Softw. Metrics (METRICS '99). IEEE Computer Soc.

[12] S. Schulze, M. Kuhlemann, and M. Rosenmüller. 2008. *Towards a refactoring guideline using code clone classification.* In Proc. Work. on Refactoring Tools (WRT '08). ACM.

[13] S. Jarzabek and S. Li. 2006. *Unifying clones with a generative programming technique: a case study: Practice Articles.* J. Softw. Maint. Evol. 18:4.