# Experiences from Designing and Validating a Software Modernization Transformation

Alexandru F. Iosif-Lazăr
IT University of Copenhagen
afla@itu.dk

Ahmad Salim Al-Sibahi
IT University of Copenhagen
asal@itu.dk

Aleksandar S. Dimovski
IT University of Copenhagen
adim@itu.dk

Juha Erik Savolainen
Danfoss Power Electronics
juhaerik.savolainen@danfoss.com

Krzysztof Sierszecki
Danfoss Power Electronics
ksi@danfoss.com

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

***Abstract*—Software modernization often involves complex code transformations that convert legacy code to new architectures or platforms, while preserving the semantics of the original programs.**

**We present the lessons learnt from an industrial software modernization project of considerable size. This includes collecting requirements for a code-to-model transformation, designing and implementing the transformation algorithm, and then validating correctness of this transformation for the code-base at hand. Our transformation is implemented in the TXL rewriting language and assumes specifically structured C++ code as input, which it translates to a declarative configuration model.**

**The correctness criterion for the transformation is that the produced model admits the same configurations as the input code. The transformation converts C++ functions specifying around a thousand configuration parameters. We verify the correctness for each run individually, using translation validation and symbolic execution. The technique is formally specified and is applicable automatically for most of the code-base.**

***Keywords*—*Experience Report, Functional Equivalence, Program Transformation, Symbolic Execution.***

## I. Introduction

The panelists of the ASE 2013 conference [1] listed the growing size and complexity of legacy code as one of the key challenges of the software industry. Software modernization is an important contender among the measures to address this challenge. However, relatively little literature is available about software modernization projects, especially in the safety critical domain. We present experiences from a research partnership around an industrial software modernization project, between IT University of Copenhagen and Danfoss Power Electronics[1], a global producer of components and solutions for controlling electric motors driving various machinery. This particular modernization project involves a *configuration* tool used to adapt Danfoss frequency converters to a particular application. The configuration tool consists of thousands of C++ functions for accessing and validating the configuration parameters.

The configurator code base is undergoing a modernization process where each parameter function must be converted from imperative C++ code to a declarative specification. All the functions compute a result (either the value of the parameter or a Boolean value checking its accessibility and validity). The declarative form of a function is an expression that must compute the same result as the original imperative code. The code modernization is done automatically by applying a transformation—implemented in the rule-based transformation language TXL [2]—on each function individually. It performs a sequence of syntactic replacements, gradually eliminating C++ preprocessor directives, local variables, C++ control statements and leaving behind a pure (i.e. side-effect free) expression.

The execution of a TXL transformation can only catch trivial errors revealed by a syntactic analysis of the code (enforcing constraints through pattern matching). This does not guarantee that the semantics of the source program is preserved. In the Danfoss case, determining the semantic equivalence is a decidable problem due to certain properties of the programs—the execution always terminates, the code fragments do not contain inline assembler code and they generally employ a small subset of C++.

We assessed that symbolic execution [3] is mature enough to handle this task efficiently. We implemented a *lightweight* wrapper which compiles both the input and the output of the transformation and employs the symbolic executor KLEE [4] to assert program equivalence. KLEE produces a set of *path conditions* that represent distinct execution paths of the two programs along with their return values as symbolic formulas. If it fails to establish the equivalence of the return values of any execution path then it outputs the path condition and a counter-example.

Our contributions are:

- A synthesis of experiences from the design of a non-trivial modernization transformation for a real project.
- Designing and implementing a transformation validation technique for this case study, which produces counter-examples if two programs are not equivalent.
- Lessons learnt from using the validation technique in the case study, including an analysis of the kind of errors that have been identified.

To the best of our knowledge this is the first analysis of transformation errors extracted from an industrial project of this

[1] http://www.danfoss.com

```
1  Configuration config = selectedConfigParameter;
2  Option opt = selectedOptionParameter;
3  bool result = false;
4  switch (config) {
5    case config1:
6      if (opt == option1) result = true;
7      break;
8    default:
9      result = true;
10     break;
11 }
12 return result;
```

Fig. 1: An example input program.

```
1 (selectedConfigParameter == config1
2    && selectedOptionParameter == option1)
3  ? true
4  : false
```

Fig. 2: The output program after the transformation.

scale and complexity (4119 functions or 14502 lines of input code of which: 105 error cases were found at transformation time, 104 error cases were found at verification time and 3910 were successfully verified).

## II. Example

We illustrate our technique for modernizing a function and validating its correctness by using a simple example. The example is a simplified and anonymized version of a real function from the Danfoss code base.

The input program (given in Fig. 1) is a constraint-enforcing function. It checks that the input variables `selectedConfigParameter` and `selectedOptionParameter` have correct values with respect to each-other and to some constants `config1` and `option1`. We need to turn this program in a declarative constraint that must preserve the names of the input variables and must compute the same truth value as the imperative program.

Syntactically, the input program consists of a `switch` statement with two cases where one of them contains an inner `if`-statement. The program returns the value stored in the variable `result`. The corresponding declarative C++ expression (Fig. 2) obtained by running the TXL transformation is produced in several steps:

- The `switch`-statement is replaced with a nested `if-else` conditional statement.
- A series of simplifications are performed on the conditional statements, so we end up with a straightforward control flow.
- The conditional statement is finally reduced to a ternary expression of the form `e1 ? e2 : e3`.

Even for such a small example, checking that the transformation has preserved the semantics of the input program is non-trivial. We use KLEE to symbolically execute both the input and transformed output program to obtain all possible symbolic execution paths. Then the input and output programs

```
1 (selectedConfigParameter == config1)
2 ? (selectedOptionParameter == option1)
3 : true
```

Fig. 3: The expected output.

are compared for equivalence, by checking whether there are corresponding matches between the obtained path conditions. If there exists a path condition in one program for which no matching path condition is generated in the other, then KLEE reports the corresponding path as a counter-example, i.e. as a witness of an execution that is possible in one program but not in the other.

For example, the path condition $selectedConfigParmeter \neq config_1 \wedge selectedOptionParameter = option_1 \wedge result =$ true is generated for the input program in Fig. 1 but not for the transformed program in Fig. 2, which shows that these two programs are not semantically equivalent. By investigating the counter-example we are able to determine the expected output program shown in Fig. 3.

We were also able to trace the transformation rules that were applied to the input program and successfully diagnosed which rule produced erroneous output. The erroneous rule was an `if`-statement simplification rule which tried to simplify nested `if`-statements correctly, but forgot to take the `else`-branch into consideration. The verification technique was therefore very useful in accurately tracking the places in which bugs occur and an experienced transformation specialist would be able to fix this issue relatively swiftly.

## III. Research Methodology

We followed the general framework of action research [5]. The study can also be seen as an exploratory case study, whose objective is to establish the feasibility of solving a concrete industrial problem.

### A. Study Description

*Objective:* To establish the feasibility of transforming a large body of configuration code from an imperative implementation in C++ to a declarative model, in a manner that is *automatic*, *trustworthy* and *cost effective*.

The cost effectiveness is understood as being cheaper than reimplementing the code from scratch. The study is exploratory, as only the problem statement is known initially. The researchers have access to the input C++ code and to three Danfoss engineers/architects knowledgeable about the code, the context and the use case. The study combines engineering (transformation development) and research (designing semantically sound transformations and validating them). The case has been initiated as a pilot project for larger modernization activities in the same organization.

The first study proposition is to establish that freely available transformation and validation tools are sufficiently mature to execute this modernization process. The second study proposition—of greater interest from a research point of view—is to explain what kind of errors might appear in transformation projects involving complex code, even if implemented by

experienced model transformation developers and language specialists.

*The case:* The input code consists of 4119 C++ functions from a configuration application. These functions encode dependencies, visibility constraints, and default values of approximately one thousand configuration parameters of a frequency converter. The use of C++, as opposed to C, is modest. No object-oriented aspects are used, except for member access and limited encapsulation. Most functions have straightforward control flow, without `goto` statements, loops and recursion. The most common statements are conditional branching and `switch` statements. The rare `for` loops all have a constant number of iterations. Other constructs include variable declarations and usage of local variables in arithmetic and comparison expressions, calls to pure functions (for instance for converting physical units), and casts between different types (both C-style casts and `static_casts`). There are few references to static and singleton member variables and functions, which act similarly to other function calls.

There are 14502 source lines of code (SLOC) in total that need to be modernized in the pilot project, and more similarly-looking configurators for other products waiting for modernization afterwards. As many as 3348 of the 4119 functions are already in expression form; these do not need to be modernized, but should not be broken by the modernization. The remaining 771 functions have 14.47 SLOC of code on average.

*Goal of the modernization project:* It has been decided that this code is suboptimal from a maintenance point of view and that it needs to be replaced by an off-the-shelf verifier using a declarative configuration model [6]. The model has to be trustworthy since the configuration code contains crucial domain information; missing configuration constraints could lead to creation of unsafe configurations by customers.

The code base in question is being actively worked on, so a manual transformation that puts the normal development on hold for a longer period is not possible. The modernization has to be automatic. Automation allows to limit the time when the code base is inaccessible for developers in two ways. First, the development of the modernization is done by implementing a transformation, which can be done in parallel with the evolution of the transformed code. Secondly, the transformation itself can be executed efficiently within minutes as opposed to weeks if it had to be executed manually.

*Theory:* In principle, it is not possible to know in advance whether refactoring the code in question preserves semantics. Validation of the transformation itself is generally undecidable. However, in recent years progress has been made by recognizing that many actual analysis problems appearing in engineering can be handled using incomplete and (partly) unsound [7] methods. This is true in particular for bug finding, which is our goal here. The effectiveness of the transformation is discussed in Sect. IV, and the effectiveness of the validation is discussed in Sect. V.

Existing model transformation validation technology (see Sect. IX) is insufficient for the needs of this case, as it focuses on preservation of structural properties of the transformed artifacts. In here we need to reason about equivalence of semantics of the transformed programs.

*a) Research questions:*

RQ1 Is it feasible to design the aforementioned transformation using off-the-shelf technology in a limited time?
RQ2 What are the main obstacles and challenges in designing and implementing the transformation? Are the transformation tools sufficiently efficient for the task?
RQ3 Can high assurance methods be used at acceptable costs to validate the transformation?
RQ4 What kind of errors are found in a transformation implemented by experts?

Questions RQ1–3 are interesting for companies looking into technology transfer in modernization projects, and for research stakeholders looking into setting up industry collaborations on software modernization. The last research question is more relevant for researchers in rewriting and model transformation. We are not aware of any studies of errors in realistic software transformations. Thus this work can be used to guide further research in quality assurance and verification of model and code transformations by formulating a hypothesis on what kind of problems are worth addressing.

*Methods:* We decided to address RQ1 and RQ2 by searching for the most effective way to implement a transformation. We elicited the requirements from the industry partner and evaluated a number of approaches and supporting technologies. A similar process was executed for RQ3. We recorded experiences during the process and report them in this paper. For RQ2 we have collected statistics about the effectiveness and efficiency of the transformation. For RQ3 we have measured the ratio of false positives and explained why they appear when following our validation methods. Regarding RQ4, we collected counterexamples from the validation process, classified them, and qualitatively analyzed them to understand what kind of errors arise in the transformation.

*Study Participants:* Three teams have been involved in the project: i) The industrial partner presented the software modernization problem, requirements and artifacts. Two engineers and one architect participated in the team. ii) The transformation team consisted of a transformation expert, a language semantics expert and a project leader. The transformation expert designed and implemented the transformation in dialog with the other members of the team. iii) The validation team consisted of a junior applied verification researcher (with 2 years of experience in verification), a junior PhD student in programming languages, and the same language semantics expert and project leader that were involved in designing the transformation.

### B. Threats to Validity

*Construct Validity:* The industrial partner had selected the case and the problem they wanted to be solved, and the research team only had access to the above mentioned parts of the configurator. It may be possible that the researchers misunderstood some aspects of the software architecture, which would influence the validity of the results reported here; however, we believe that the impact of this would only be limited. First, the transformation is of substantial complexity,

so even if it was misaligned with the requirements, it still sheds a lot of light on pragmatics of such transformations. More importantly, the validation method finds actual errors that are easy to confirm manually.

*Internal Validity:* Since the validation procedure has been developed in the same study in which it is evaluated, there is certainly a risk that it had overlooked some important errors. The transformation had been designed with a focus on effectiveness, with no thought of later verification in mind. The validation project is largely independent of the transformation project, and the key developers of the two parts have not communicated in any significant manner at all; several months have passed between the end of the transformation implementation project and the beginning of the validation project. The validation team mapped between the classes of bugs and the programming errors causing them, so there is a slight risk of misinterpretation. To minimize this risk, we have cross checked the result with other team members.

*External Validity:* Limited external validity is in the very nature of an individual study. For this reason we describe the properties of the case in detail. Simple C-like code with bounded for-loops is common in safety critical software, and we thus believe that the findings can generalize to additional modernization projects.

*Reliability:* The analyzed transformation errors can be biased towards the weaknesses of the particular development team. Note that the involved developers were experts in model transformations (more than 6 years of applied research experience) and verification (more than 6 years of experience with program and model verification), thus it is unlikely that the errors they made were trivial and would not happen to other developers.

Nevertheless, we report these findings only existentially (without quantifying them), providing a single data point for the research space where very little evidence is available so far. More studies are definitely needed in order to understand the nature of design errors in transformative and generative programming.

## IV. Designing the Transformation

### A. Design Principles Established for the Project

We found that implementing an automatic transformation is superior to a manual refactoring, as it allows to minimize down-time on the main development of the code base. The code to be modernized is only locked for the time it takes to run the transformation and it can be evolved freely while the transformation is being implemented and tested. This is somewhat different from the standard use case for transformations which is automating repetitive tasks in conversions of data, code or models. This transformation was meant to be executed only once, but automation was key to minimize disruptions in the regular development process.

**Observation 1.** Automatic transformations allowed us to decouple and parallelize the regular development and the modernization activities.

Translation of imperative C++ code into a declarative form in general may be very complex. However, our objective was much more modest: we only needed to handle the code at hand (and similar). Thus we settled on saving resources whenever possible by sacrificing generality. In fact, we even gave up transforming the entire code at hand, agreeing that a small number of complex fragments (involving loops) were simpler to modernize manually rather than designing rules that would handle them correctly from first principles. The manual migrations can be hard-coded into a transformation as special cases, so that the transformation execution remains automatic.

**Observation 2.** Since modernization is a one-off transformation it was economically beneficial to sacrifice generality, and instead focus on the code at hand whenever it simplifies things.

To control the lack of generality, the implementation should follow a *fail-fast* programming style [8], [9]. It should succeed on the inputs that it was designed for, but it should fail as early as possible on inputs that violate the assumptions made when sacrificing generality (e.g. arbitrary nesting of constructs in C++, or use of unexpected language elements). This was achieved by making preconditions for rules as precise as possible (so that rules are not applied when failing) and writing explicit assertions when possible (when a rule is in principle applicable, but it does not cover all the cases, for simplicity of development). Without the fail-fast programming we would have very limited trust that the transformation works correctly on hundreds of code fragments that we were not able to inspect manually.

**Observation 3.** The fail-fast programming approach helped to avoid implementing anything that is not strictly required for the modernization project to succeed, while retaining quality on the expected inputs.

Initially, we considered using type analysis and other semantic mechanisms (such as static single assignment form) to solve the task. However, it soon became clear that this would raise the complexity of the implementation considerably, and likely also lead to polluting the output with identifiers generated in the process (unfamiliar to developers). Full understanding of static semantics might only be required if one implements a general program rewriter. For an incomplete transformation of a known code base it seems much easier to work with syntactic transformations. Even typing and simple data flow information can be captured with a finite number of patterns if we only need to work with limited code base.

**Observation 4.** We found working with syntax much more effective for an ad hoc transformation task, than when using semantic data and semantically informed rewrites.

### B. Tool Selection

Since the input language (C++) is rather complex, we understood early on that the transformation tooling should be driven not by our personal preferences, but by the availability of a C++ grammar. Thus we considered using existing open source compiler front-ends (for instance GCC[2]), or language tools (such as Eclipse CDT[3]). However, we found them challenging to use. Then we turned to transformation tools and found that both Spoofax [10] and TXL [2] have C++ grammars, though the

[2] https://gcc.gnu.org
[3] http://eclipse.org/cdt/

```
1 rule convert_simple_sel_stmt
2   replace [selection_statement]
3     'if '(EXP [expression] ')STMT [statement]
4   where not STMT [contains_selection_stmt]
5   where not STMT [is_compound_stmt]
6   construct TRUE_STMT [true_case_statement]
7     'TRUE ';
8   by '(EXP ')'? '(STMT ')': '(TRUE_STMT ')
9 end rule
```

Fig. 4: An example TXL rewrite from our project: a translation of a C++ conditional statement into a conditional expression.

grammar of the former was unmaintained and hard to migrate to modern versions of the tool. We ended up using TXL which has a good C++ grammar, handles ambiguity quite well and provides mechanisms for relaxing the grammar, making it easy to adapt to our needs. TXL is a standalone command-line tool, with few dependencies, so it took virtually no time to get it to run. In fact, simple proof of concept rewrites were working after only 4 hours of experiments with TXL.

**Observation 5.** Simplicity and the integration with the languages to be transformed influenced the selection of tools stronger than the properties of the transformation language or a rewriting paradigm.

### C. Transformation Implementation

TXL is a rewrite tool that transforms syntax trees into syntax trees. It accepts two kinds of definitions: *grammars* and *transformations*. Grammars serve for parsing and unparsing. TXL works with one grammar at a time—in other words, the input and output grammars must be the same. To overcome this we selected a subset of C++ expression language as our target language (C++ expression language is sufficiently good to express declarative constraints over finite domain variables). To handle this format we needed to relax the C++ grammar only slightly to allow top-level expressions in C++. We also wrote a simple rule that validates whether the output program is indeed an expression in the subset of interest.

Transformation definitions specify how to rewrite a particular input syntax to output syntax. Figure 4 shows an example transformation rule, `convert_simple_sel_stmt`, from our project. All caps identifiers refer to syntax trees. The rule matches a conditional statement without an else clause (line 3) and translates it into a conditional expression (line 8). Lines 4–5 specify that the rule should fail if the guarded statement is either a compound statement or another conditional statement. Due to the grammar construction and the interaction with other rules, this means that the rewrite will only apply to conditional statements that themselves already guard a simple expression. In lines 6–7 a constant `true` expression is constructed, which fills in for the missing branch in the conditional expression.

The overall algorithm applied by the transformation is:

1) The program fragment is checked for format assumptions: all branches return a value, there are no loops and goto jumps, no calls to non-pure methods, etc. The transformation does not establish the purity of functions itself, but consults a white-list of names of pure functions provided as a parameter.
2) All preprocessor `#ifdef` directives in the program are cleaned up, and converted to ordinary `if` statements.
3) Local variable assignments are inlined in the following expressions in the right order (i.e. going from the last assignment to the first). When all references to the local variables are eliminated, their declarations are also removed.
4) All `switch` statements are converted to a series of `if` statements.
5) Series of sequential `if` statements are simplified into nested `if` statements, such that the fragments are reduced to a single root statement, all additional functionality being implemented through substatements of the root.
6) `if` statements are converted to ternary expressions and `return` statements are replaced by the expression they return.

**Observation 6.** We succeeded to implement a flow-aware syntactic transformation, including constant folding and variable inlining, which enabled us to produce code that uses the same identifiers, and reminiscent structure of the input programs.

Readability of the ultimate output is important in modernization projects, as it is expected that the developers will further evolve the generated code.

### D. Basic metrics

The entire transformation (including grammar definitions, excluding white space and comments) spans 6515 lines of code. The core C++ grammar, which is provided as a resource from the TXL website, has 137 nonterminal rules (595 lines of code). In addition, 98 grammar rules are defined or redefined in the transformation to adapt the grammar to our needs.

The transformation has 468 definitions (rules or functions) in total, where 171 of the definitions are function definitions and the remaining 297 are rule definitions (as opposed to functions, rules are called repeatedly until a fixed point is reached).

It took 3 months full time work of experienced software developer to implement this transformation (including learning TXL, domain understanding, unit testing and meetings with the industrial partner). The cost is deemed acceptable, especially given that the company has several more products to modernize, that include configuration code, for which the transformation would be largely reusable.

The transformation execution lasts 30 minutes on the 4119 functions out of which 105 functions are not handled—the transformation reported errors, marking that these functions contain special cases and should be migrated manually to keep the whole process cost-effective.

## V. Validating the Transformation

### A. Verification challenges

TXL is a very powerful transformation language that can express arbitrary computation using deep pattern matching, complex side conditions, global state and rewriting. Individual rules and functions of a transformation are non-modular and might intermediately break syntactic and semantic correctness properties. This makes it hard to verify individual rules. Instead, it is important that the transformation is validated as a whole.

**Observation 7.** Our transformation was written in a non-modular fashion (as most transformations we have seen). This made it hard to verify the transformation rules individually.

Our transformation makes heavy use of the generic programming and dynamic reparsing features which allow serialization of abstract syntax trees to textual syntax and then reinterpretation as other syntactic structures. An external script controls the sequence of transformation steps, adding even more complexity to the algorithm. Furthermore, the rules are designed in an ad hoc fashion specifically to handle patterns present in the use case. For example, we use the transformation rule `if` $(E)$ `return` $true \rightarrow$ `return` $E$, which is not correct in general, but it is correct under assumption that the `if` statement occurs last in the `main` function. To combat this complexity, our verification method treats the transformation as a black-box and checks that the input and corresponding transformed output agree semantically. This works since the size of input is manageable and we do not expect the transformation to be generally applicable in other unrelated settings.

**Observation 8.** We were able to treat a complex transformation as a black-box, reducing the validation to checking whether the provided input and transformed output agree according to specific semantic properties.

### B. Approach

Ordinary TXL rules and functions are constrained to replacing well formed syntactic trees with newly built ones, effectively making it impossible for TXL programs to produce syntactically incorrect output. The correctness criterion of the transformation was to produce *semantically* equivalent C++ programs. Therefore our validation technique must be able to reason about the semantics of C++ programs.

We considered several abstract interpretation and analysis techniques and we found that symbolic execution [3] is able to build a very precise semantic model. We decided on using the precise symbolic executor KLEE [4] which handles the majority of features used in the code at hand, and integrating it in the automation process proved to be cost-effective.

One of the challenges of using KLEE is that it requires the input code be compiled to LLVM [11] intermediate representation (LLVM–IR), including all external libraries (otherwise the symbolic execution may not terminate or provide incomplete results). However, our sample code-base consisted of individual functions which called external functions with unknown implementations. Therefore, we had to close the code with suitable instrumentation:

1) We created stubs for unknown functions—ordinary, static member and singleton member functions alike—such that a set of arguments is matched to the same symbolic result on every call.
2) We created stubs for the data structures with straightforward constructors (that initialize all members) and structural equality comparison operations.

Function stubs are created in the *symbolic function* [12] style. For each stub an execution table which matches input arguments with symbolic return values is allocated. When the function is called, it looks up the arguments in the execution table: if they are found, it returns the same symbolic result as before; otherwise, a new record that stores the arguments along with a fresh symbolic result variable is created in the table.

```
1   bool defined(int p) {
2     static node<int, bool> *results;
3     static int* counter = new int(0);
4     bool* val = new bool;
5     if(!getResult(&results, &results, p, counter, val)) {
6       char symbolicname[40];
7       sprintf(symbolicname, "defined%d", *counter);
8       *val = klee_range(0, 2, symbolicname);
9     }
10    return *val;
11  }
```

Fig. 5: A stub for an unknown Boolean function.

Figure 5 shows a stub for an unknown Boolean function. The static execution table `results` and call counter are reused for all calls to the function. The function `getResults` looks up the input argument `p` in the execution table. If found, the existing symbolic variable is returned through the pointer `val`. Otherwise, the call counter is incremented and `val` points to a new memory address which is made symbolic with `klee_range` and returned.

**Observation 9.** We were able to use off-the-shelf tools to perform semantic verification of programs, with a moderate amount of effort required to pre-process the input to the tool.

### C. Research questions

In the experiment we answer the following detailed questions, refining RQ3:

- RQ3.1 How large a part of the transformed code base can be verified automatically?
- RQ3.2 How much additional effort would it require to verify the rest of the code base?
- RQ3.3 How can our verification effort be generalized to other similar modernization projects?

### D. Method

We address RQ3.1 in Section VI by running the verification procedure on the input and transformed output, and reporting and classifying the results. Section VIII discusses the challenges (and solutions to these challenges) that appeared during verification (RQ3.2) and what parts of our verification procedure is generalizable to other transformation tools and projects (RQ3.3).

The validation execution lasts 7 minutes on all the transformed functions out of which 3348 are evaluated trivially to pass verification–the output is identical to the input.

## VI. Bug Analysis

### A. Bugs in Numbers

Out of the 4119 functions in the code base there were 771 which could not be validated trivially (the output was not identical to the input). We present the statistics in Table I.

**Observation 10.** It was possible to analyze a substantial amount of the modernized code automatically, and only 20 corner cases were left to be handled manually.

We analyze the bugs found by verification (3 & 4b) below. In Section VIII we will discuss the types of spurious counter-examples (4c), the unhandled cases due to design limitations (5), and propose solutions for these issues.

### *B. Analysis of Bugs Found by Verification*

Our technique had identified seven bugs present in the transformation. While these bugs varied in nature, they had one important thing in common: they were all related to execution semantics and would have been hard to find with a syntactic check or simpler static semantics check (like a type system).

**Observation 11.** When the code base of our modernization project reached a certain complexity it became infeasible to find all bugs through expertise and unit testing. Validation of semantics was essential to ensure that the output code worked correctly.

**Bug 1: *Function call is dropped in some paths*.** Perhaps the most widespread output errors were missing function calls—absent in the output expression but present in the original code. This happened with a variety of calls to different functions, and could happen multiple places in the output expression; furthermore, calls to the same function might still be present in other branches of the output expression.

The bug was caused by an *incomplete rewrite rule*. When a rule matches a functional call in a return statement and forgets to reinsert the function call on replacement.

**Bug 2: *Structure replaced by a constant integer*.** Another simple bug is the one where the input declares a variable of a class type, calls its object initializer with multiple arguments and returns it. Here, the transformation seems to return the first argument given to the variable—which is often an integer and therefore having incompatible type—instead of the whole object.

This bug also happens due to *misuse of deep pattern search* and a *broken rule assumption*. It happens when trying to inline a variable with its initial value, but expecting the variable to be of a simple type. Since there is a use of a deep pattern search to extract an expression from the initialiser, it will simply pick the first one (ignoring the others) and the rest of the transformation would continue without noticing the bug.

TABLE I: Erroneous transformation cases caught by each step of the validation process.

| | Step | #Cases |
|---|---|---|
| 1 | Failing transformation precondition (not handled, requiring manual inspection) | 105 |
| 2 | Failing silently due to unhandled syntactic structures (caught statically during preliminary steps of verification) | 3 |
| 3 | Caught by C++ compiler | 3 |
| 4 | Checked for equivalence using KLEE | 640 |
| 4a | Validated being equivalent | 562 |
| 4b | Concrete bug cases with provided counter-examples | 50 |
| 4c | False positives with spurious counter-examples (due to over-approximation of functions, and representation mismatch) | 28 |
| 5 | Unhandled cases containing assertions (intentional, due to design limitations of the validation technique) | 20 |

**Bug 3: *Conditional branches are dropped*.** This bug caused the transformation to ignore all branches following a nested `if`-statement (also referred to in Section II). It was caused by *incomplete rewrite rules*. The rewrite rule matches a nested conditional followed by other branches, and then rewrites the conditional correctly but forgets to handle the other branches.

**Bug 4: *The unexpected exceptions*.** This bug is surprising and happens mostly in very large functions with complex nesting of conditional control flow. While the input function seems total and returns a correct result on all paths, the transformation produces an output which contains a branch that throws an exception stating that the branch should be invalid.

This bug happens due to *overconstrained pattern matching* and *broken rule assumption*. When a sequential composition of nested conditional followed by a return statement is matched by the transformation, it tries to put the final return statement inside the previous conditionals. However, in this case the pattern was overconstrained and so it did not match the form of input it was given; later, when the transformation tries to convert the statement to an expression it finds a branch with no `return` statements and replaces it with a `throw` statement because it did not expect this case to be possible (a part of the fail-fast approach).

**Bug 5: *Use of undeclared variables*.** This bug is the only one that appears during compilation. The original input contained declarations to local variables that were not inlined correctly and the transformation removed all local declarations—but retained references to their respective identifiers—leaving compiler errors.

This bug occurs due to a combination of *dynamic reparsing capabilities* and *wrong target type* in expression. To control the number of iterations of inlining substitution, the transformation replaces variable nodes with the string representation of their assigned expressions, using the textual output capabilities of TXL. This ensures that the substitution terminates, but might also be incorrect if the transformation has not finished migrating the serialised subtrees. In general, it is caused by challenges in implementing a semantic operation (i.e. inlining) syntactically.

**Bug 6: *Negation dropped in result*.** The simplest bug found by the KLEE-based verifier is where the transformation had transformed the whole input correctly except a negation operation which was missing in the output. This bug occurs due to *misuse of deep pattern search* of TXL. The rewrite rule searches for a more specific object type than necessary, making it ignore more complex objects that do not fit to the expected pattern.

**Bug 7: *Conditional with error code assignment dropped*.** One interesting bug is where the input has a function which contains a conditional statement that assigns a value to an error code pointer variable, in addition to returning a value (both in the conditional and outside). In this case, the transformation will produce output that will completely remove the conditional branch and only keep the final return value, which makes the function produce the wrong result.

This bug happens due to the *dynamic reparsing capabilities* and *eager removal of source data*. It originates in the inlining

phase where some abstract syntax is broken by wrongly inserted textual syntax, and subsequently a rule that removed empty conditional branches was applied.

**Bug 8:** ***Variable declarations without assignment not handled.*** This bug was caught statically in the cases when the transformation finished, but the output was empty. Similar to Bug 2, a combination of a *broken rule assumption* and *misuse of deep pattern search* was the cause of this bug. In ordinary circumstances, the transformation tries to inline all locally declared variables with their assigned expressions and then remove the declarations. However, in these cases the declaration and initialization of the local variables were situated in separate statements. The declaration removal rule used deep search to identify statements which contained local variables and since program consistent of one large if-statement containing the assignments, it was completely removed.

*Classification summary:* Most cases were affected by bug 1 where there were 23 cases in total, and followed by bug 2 which had 15 cases in total; both of which were simple in nature. This is perhaps unsurprising since function calls and object initialisations are common constructs in C++, and a simple mistake in the transformation of these features will therefore affect a large number of analyzed functions. The more interesting (complex) bugs 3 and 4 had 5 cases in total each. This type of bugs often appeared in larger files with a complex nesting of conditionals, and would therefore have been hard to immediately spot manually or with simpler unit tests. Finally, the remaining bugs (5, 6, 7, 8) had 3, 1, 1 and 3 cases in total, respectively. These errors represent issues that appear to be corner cases that were either not caught by the preconditions of the transformation, or occurred where an intermediate assumption of the transformation was wrong.

**Observation 12.** *Simple bugs hit wide, complex bugs hit deep.* Simple semantic errors affected a large number of functions while complex errors were found in a few but bigger functions.

## VII. Formal Justification of the Procedure

### A. Concrete execution

The transformation translates many functions individually. Each of them needs to be translated in a semantics preserving manner. We view functions (or programs in general) as input–output relations.

**Definition 1.** A *program* $P$ is a set of imperative instructions that state how to calculate the designated output variable $ret$ from a set of input variables $Var_{\text{in}} = \{i_1, \ldots, i_k\}$.

**Definition 2.** A *concrete state* (store) $\sigma$ is a *function* mapping program variables $Var$ into values $Val$, i.e. $\sigma : Var \to Val$. The values $Val$ are constants from ordinary C++ types: Boolean, bounded integer, float, etc.

A concrete execution starts with an *initial state*, $\sigma_{\text{in}}$, where all input variables are assigned some initial values. During the execution of the program, the effect of executing each statement $s$ in a state $\sigma$ produces a successor state $\sigma'$, written as $\sigma \xrightarrow{s} \sigma'$. When there are no statements left to execute, the program reaches a final state $\sigma_{\text{out}}$, in which the value of the output variable $ret$ is well-defined.

**Definition 3.** A *concrete execution path* $\pi = \sigma_{\text{in}}, \sigma_1, \ldots \sigma_{\text{out}}$ of the program $P$ is a sequence of states, such that $\sigma_{\text{in}}$ is an initial state, every next state in the sequence is obtained by sequentially executing statements from $P$ one by one, and $\sigma_{\text{out}}$ is the final state.

**Definition 4** (Concrete program path semantics)**.** The *path semantics* of program $P$—called $[\![P]\!]_{trace}$—is defined to be the set of all valid concrete execution paths $\pi$ of $P$.

**Definition 5** (Denotational program semantics)**.** The *denotational semantics* of program $P$ is a partial function $[\![P]\!] : Val^k \rightharpoonup Val$ defined by: $[\![P]\!](\sigma_{\text{in}}(i_1), \ldots, \sigma_{\text{in}}(i_k)) = \sigma_{\text{out}}(ret)$, for any concrete execution path $\pi \in [\![P]\!]_{trace} = (\sigma_{\text{in}}, \ldots, \sigma_{\text{out}})$.

**Definition 6** (Semantic equivalence)**.** Two programs $P$ and $P'$ are *semantically equivalent*, written $P \sim P'$, if for any collection of values $v_1, \ldots, v_k$ it holds: $[\![P]\!](v_1, \ldots, v_k) = [\![P']\!](v_1, \ldots, v_k)$ .

Determining the semantic equivalence of programs using concrete path semantics is infeasible due to the immense range of input values. Instead, we use symbolic execution to cluster the input values using a set of constraints called path conditions.

### B. Symbolic execution

In *symbolic execution* the program does not assign values to its variables; instead, it assigns symbolic expressions containing uninterpreted symbols abstractly representing user-assignable values in a concrete execution of the program. For ease of notation, we will use capital letters $X, Y, \ldots$ to represent uninterpreted symbols, and we use *Sym* to represent the set of all of these symbols. In the initial execution state, each possible input variable $i$ is usually assigned a corresponding unique symbol $I$.

For example, let $x$ and $y$ be input integer variables. The concrete semantics of $ret = x+y$ is the set $\{([x \mapsto 0, y \mapsto 0], [x \mapsto 0, y \mapsto 0, ret \mapsto 0]), ([x \mapsto 0, y \mapsto 1], [x \mapsto 0, y \mapsto 1, ret \mapsto 1]), \ldots\}$, where initial states are all possible assignments of integer values to $x$ and $y$. However, the symbolic path semantics of $ret = x + y$ will contain only one symbolic execution path $([x \mapsto X, y \mapsto Y], [x \mapsto X, y \mapsto Y, ret \mapsto X + Y])$, where $X$ and $Y$ are symbols.

Symbolic execution confounds a set of different concrete paths into one by following all branches whenever a branching or looping statement is encountered. In the same time, for each branch it maintains a set of constraints called the *path condition*, which must hold on the execution of that path.

**Definition 7.** A *symbolic expression* $se$ from the set $SExp$ can be built out of constant values from $Val$, symbolic values from $Sym$, and arithmetic-logic operations.

**Definition 8.** A *symbolic state* $\sigma^{\#}$ is a function mapping program variables $Var$ into symbolic expressions $SExp$, i.e. $\sigma^{\#} : Var \to SExp$. The initial symbolic state $\sigma^{\#}_{\text{in}}$ maps all input variables $i \in Var_{\text{in}}$ into a fresh symbolic value $I \in Sym$.

**Definition 9** (Constrained symbolic state)**.** A *constraint* is a Boolean symbolic expression. A *constrained symbolic state* is a pair $\langle \sigma^{\#}, sb \rangle$, which constraints the symbolic expressions in $\sigma^{\#}$ with a Boolean symbolic expression $sb$.

**Definition 10** (Symbolic execution path)**.** A *symbolic execution path* of the program $P$ is a sequence of constrained symbolic states $(\langle\sigma^{\#}_{\text{in}}, \text{true}\rangle, \langle\sigma^{\#}_{1}, sb_1\rangle, \ldots \langle\sigma^{\#}_{\text{out}}, sb_{\text{out}}\rangle)$, where the initial state $\sigma^{\#}_{\text{in}}$ is unconstrained, and the constraint produced for the final state, $sb_{\text{out}}$, represents the path condition.

It is notable that the resulting set of symbolic execution paths partitions the set of concrete execution paths. For the program that computes the absolute value of an integer variable $i$, there are two different paths returned by symbolic execution: $\big(\langle[i \mapsto I], \text{true}\rangle, \langle[i \mapsto I, ret \mapsto I], I \geq 0\rangle\big)$ and $\big(\langle[i \mapsto I], \text{true}\rangle, \langle[i \mapsto I, ret \mapsto -I], I < 0\rangle\big)$. If the initial value of $i$ is non-negative, then the return value is the symbolic expression $I$; otherwise, the return value is $-I$. Hence, the set of all concrete execution paths (determined by the input values of $i$) has been partitioned in two sets: those for which $i \geq 0$ holds and those for which $i < 0$ holds.

**Proposition 1.** For each concrete execution path $\pi = (\sigma_{\text{in}}, \sigma_1, \ldots \sigma_{\text{out}})$ of the program $P$, there exists the corresponding symbolic execution path $\pi^{\#} = (\langle\sigma^{\#}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#}_{\text{out}}, sb_{\text{out}}\rangle)$, such that $\sigma^{\#}_{\text{in}} = [i_1 \mapsto I_1, \ldots, i_k \mapsto I_k]$, $\sigma_{\text{out}}(ret) = \sigma^{\#}_{\text{out}}(ret)[I_0 \mapsto \sigma_{\text{in}}(i_0), \ldots, I_k \mapsto \sigma_{\text{in}}(i_k)]$, and $sb_{\text{out}}[I_0 \mapsto \sigma_{\text{in}}(i_0), \ldots, I_k \mapsto \sigma_{\text{in}}(i_k)]$ is true.

*Proof:* See online appendix [4]. ∎

**Theorem 1.** *Two programs $P$ and $P'$ are semantically equivalent $P \sim P'$ iff for each valuation $V \in Val$ it holds:*

$$\Big(\bigvee_{1..m}^{j} sb^{j}_{\text{out}} \wedge \sigma^{\#,j}_{\text{out}}(ret) = V\Big) \iff \Big(\bigvee_{1..m'}^{i} sb'^{i}_{\text{out}} \wedge \sigma'^{\#,i}_{\text{out}}(ret) = V\Big)$$

*where $(\langle\sigma^{\#,1}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#,1}_{\text{out}}, sb^{1}_{\text{out}}\rangle)$, …, $(\langle\sigma^{\#,m}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma^{\#,m}_{\text{out}}, sb^{m}_{\text{out}}\rangle)$ are symbolic paths of $P$, and $(\langle\sigma'^{\#,1}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma'^{\#,1}_{\text{out}}, sb'^{1}_{\text{out}}\rangle)$, …, $(\langle\sigma'^{\#,m'}_{\text{in}}, \text{true}\rangle, \ldots \langle\sigma'^{\#,m'}_{\text{out}}, sb'^{m'}_{\text{out}}\rangle)$ are symbolic paths of $P'$.*

*Proof:* It follows from Prop. 1 and Def. 6. ∎

## VIII. Discussion

A known challenge of action research in software engineering is unreliability of academic tools. Tools developed by academic partners are rarely adopted in companies due to a lack of reliable support service (unless the industrial partner can reasonably take over the tool maintenance itself). This is apparently a much smaller problem in software modernization projects, as the tools are only used for a short period. We note that modernization is a very good domain for research-based tools, where the actual adoption is likely easier than elsewhere.

The applied design and validation principles translate easily to other program and model-transformation languages. All such languages work at the level of (abstract) syntax, so designing the rewriter syntactically is achievable. Since we validate the output of the transformation against the input, but the transformation itself is treated as a black-box, the method is oblivious to the choice of the transformation language. Because of that, it could work even for manual transformations, for instance for manual refactoring. However it is unclear, whether the identified bugs are specific to this case, this input and output languages, and TXL.

[4] http://www.itu.dk/people/afla/files/ase-2015-appendix/prop1-proof.html

We have met the following technical verification challenges:

*1) Representation of Boolean expressions:* In C++ any integer valued expression can be used as a logical condition (inside `if`-statements etc.), and so any non-zero value would count as true and zero would count as false. If an integer variable `a` is used only as a logical condition both in the input and output programs it would be pragmatically fine. However, our transformation contains simplification rules which convert statements of form `if (a) return true; else return false;` to `return a;` which clearly has different semantics. In cases where we are certain that specific integer variables are only used as conditionals we instruct KLEE to assume that these variables have values lying in range $[0, 2)$.

*2) Over-approximation of Function Semantics:* Because we do not know the implementation of all external functions, we use an over-approximated function call representation with stubs that simply map equal parameters to equal unique symbolic results. However, if any of these functions actually had equivalent implementations and we used a different stub for each one, calling the two stubs with the same parameters would result in distinct return values. This led to a number of false positives where KLEE decided that the input and output programs were not equivalent. This was solved by using the same stub for functions which were known to be identical *a priori*.

*3) Assertions:* When KLEE meets a C++ assertion that has a failing condition on a possible path, it will immediately halt execution for that particular path. This concretely means that it will never check whether the input and output functions have the same results, or in this case rather both fail. Instead of using the default assertion function, one could instead use a stub that throws a recoverable error (rather than halt) on failing conditions; thereafter, one could check whether both the input and output programs failed on the same paths and if they did one could consider the paths to be equal. Our code base contains 20 cases affected by this limitation, but implementing the suggested solution was not feasible in the allocated time.

## IX. Related Work

Translation validation [13] is a verification technique for translator tools (compilers, code generators). It requires a common semantic framework for the representation of the source code and the generated target code, a formalization of the notion of *correct implementation* and a method which allows to prove that one model of the semantic framework, representing the produced target code, correctly implements another model which represents the source. Our approach is aligned with translation validation in the sense that it validates concrete translations instead of the transformation tool or algorithm. The path conditions produced by KLEE are a semantic framework of the compared C++ programs. KLEE uses an SMT solver to prove the equivalence of the path conditions and provides a counter example when they are not equivalent.

Other KLEE extensions aim to solve the same problem. UC−KLEE [14] can save and load program states so it can execute two programs in the exact same memory context. By comparing memory states at program termination UC−KLEE offers a more precise equivalence test which also covers programs that terminate unexpectedly (crash). The task would

have been easier to solve with UC−KLEE, but we used regular KLEE because there was no release of UC−KLEE available.

KLEE−FP [15] proves the equivalence of two symbolic floating point expressions by first applying a series of expression canonicalization rules, and then syntactically matching the two expressions, whereas regular KLEE silently concretizes floating point values to zero. However, when we used it to validate our transformations it failed to identify some of the bugs that regular KLEE had previously found and did not reveal any new bugs. We did not investigate this further.

Proof of program transformation correctness is often studied in conjunction with optimization. Parametrized equivalence checking (PEC) [16] uses a form of translation validation which tries to find a bisimulation between the control flow graphs of the original and optimized programs in a way that parameterizes over some program components (such as expressions, statements and declarations). Other work [17] specifies the optimisation rewrite rules as temporal logic formulas, and proves the correctness of the transformation manually. Both techniques lack tools that support them.

Currently, there exist various robust techniques for verifying preservation of properties by model transformations [18]–[21]. These techniques use (bounded) model finders and SMT solvers to verify the preservation of structural properties of model transformation rules. However, our interest lies in checking behavioral equivalence between the input and output, which is significantly more complex to check than structural properties, and thus not supported by the presented techniques.

A similar automation-based modernization effort is presented in a joint project by Semantic Designs (SD)[5] and Boeing[6] [22] which uses Semantic Designs's commercially-available transformation and analysis tool DMS [23] to convert an old component-based C++ codebase to a standardized CORBA [24] architecture. Our case study shows that it is possible to do automation-based modernization relying solely on freely available tools (TXL, etc.), and we were able to present additional useful observations. More importantly, our evaluation effort is significantly larger and includes validation in addition to rigorous testing and code reviews, which we have shown to be useful since it caught many subtle bugs and corner cases that were missed earlier in the process.

A series of papers [25]–[27] discuss a similar case study that aims to design and verify a model transformation for modernizing an existing collection of proprietary models such that they conform to the standardised AUTOSAR [28] format. The transformation [25] was initially encoded in a (non-Turing complete) subset of the ATL model transformation language [29] and then verified for structural properties [26]. The same verification effort was then repeated [27] more efficiently by by symbolically executing a version of the transformation re-encoded in DSLTrans [30]. While these verification tools and the presented case study have significant contributions to the model transformations community, they were not applicable in our study due to the difference in expressiveness between TXL and the verified non-Turing-complete subset of ATL/DSLTrans, and the complexity of the property we wanted to check (behavioral equivalence versus structural properties).

[5]https://www.semanticdesigns.com

[6]http://www.boeing.com

## X. Conclusion

We have reported experiences from an industrial software modernization project, including requirements elicitation for a code-to-model transformation, designing and implementing the transformation, and verifying the correctness of the transformation against semantic properties using symbolic execution. The project allowed us to derive observations regarding automation in software modernization as well as choices and challenges in design and validation of modernization transformations. Probably, the biggest technical challenge seen in the transformation is that it seems impossible to reason about it inductively (rule-by-rule), because the intermediate transformation results are incorrect by design. Moreover, our method is oblivious to complexity of the transformation language, but since it is property driven, it depends strongly on the properties of the *transformed* languages. Observe though that, a white-box method would be vulnerable to both kinds of complexity.

Our validator finds many semantic bugs that have been missed by unit tests of an experienced transformation developer. These errors would be very difficult to find without verification. For each bug, the tool provides a counter-example consisting of execution paths on which the input and transformed programs differ. This way, we have obtained helpful debugging information, which can be used to improve and correct the transformation. We group the identified bugs into seven classes. Our paper is, to the best our knowledge, the first ever study reporting errors from a realistic transformation project, including validation using real bugs (as opposed to planted bugs) and operational semantic properties of input and output (as opposed to syntactic and typing properties).

*Acknowledgements* We thank Karl Potratz for introducing us to the subject system, Rolf Helge-Pfeiffer for implementing the transformation and Claus Brabrand for discussions on its design. The project was supported by The Danish Council for Independent Research under a Sapere Aude project, VARIETE, and by ARTEMIS JU under grant agreement n°295397 together with Danish Agency for Science, Technology and Innovation.

## References

[1] J. Penix, "Big problems in industry (panel)," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, p. 3. [Online]. Available: http://dx.doi.org/10.1109/ASE.2013.6693060

[2] J. R. Cordy, "The TXL source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.04.002

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[4] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[5] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29044-2

[6] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, *Knowledge-based Configuration: From Research to Business Cases*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[7] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015. [Online]. Available: http://doi.acm.org/10.1145/2644805

[8] J. Gray, "Why do computers stop and what can be done about it?" in *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*. IEEE Computer Society, 1986, pp. 3–12.

[9] J. Shore, "Fail fast," *IEEE Software*, vol. 21, no. 5, pp. 21–25, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MS.2004.1331296

[10] L. C. L. Kats and E. Visser, "The spoofax language workbench: rules for declarative specification of languages and ides," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 444–463. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869497

[11] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, pp. 75–88. [Online]. Available: http://dx.doi.org/10.1109/CGO.2004.1281665

[12] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings*, ser. Lecture Notes in Computer Science, Ú. Erlingsson, R. Wieringa, and N. Zannone, Eds., vol. 6542. Springer, 2011, pp. 58–72. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19125-1_5

[13] A. Pnueli, M. Siegel, and E. Singerman, "Translation validation," in *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, ser. Lecture Notes in Computer Science, B. Steffen, Ed., vol. 1384. Springer, 1998, pp. 151–166. [Online]. Available: http://dx.doi.org/10.1007/BFb0054170

[14] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 669–685. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_55

[15] P. Collingbourne, C. Cadar, and P. H. J. Kelly, "Symbolic crosschecking of floating-point and SIMD code," in *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, C. M. Kirsch and G. Heiser, Eds. ACM, 2011, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966475

[16] S. Kundu, Z. Tatlock, and S. Lerner, "Proving optimizations correct using parameterized program equivalence," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 327–337. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542513

[17] D. Lacey, N. D. Jones, E. V. Wyk, and C. C. Frederiksen, "Compiler optimization correctness by temporal logic," *Higher-Order and Symbolic Computation*, vol. 17, no. 3, pp. 173–206, 2004. [Online]. Available: http://dx.doi.org/10.1023/B:LISP.0000029444.99264.c0

[18] F. Büttner, M. Egea, and J. Cabot, "On verifying ATL transformations using 'off-the-shelf' SMT solvers," in *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. B. France, J. Kazmeier, R. Breu, and C. Atkinson, Eds., vol. 7590. Springer, 2012, pp. 432–448. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33666-9_28

[19] F. Büttner, M. Egea, J. Cabot, and M. Gogolla, "Verification of ATL transformations using transformation models and model finders," in *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, ser. Lecture Notes in Computer Science, T. Aoki and K. Taguchi, Eds., vol. 7635. Springer, 2012, pp. 198–213. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34281-3_16

[20] F. Büttner, M. Egea, E. Guerra, and J. de Lara, "Checking model transformation refinement," in *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. Duddy and G. Kappel, Eds., vol. 7909. Springer, 2013, pp. 158–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38883-5_15

[21] X. Wang, F. Büttner, and Y. Lamo, "Verification of graph-based model transformations using alloy," *ECEASST*, vol. 67, 2014. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/view/943

[22] R. L. Akers, I. D. Baxter, M. Mehlich, B. J. Ellis, and K. R. Luecke, "Case study: Re-engineering C++ component models via automatic program transformation," *Information & Software Technology*, vol. 49, no. 3, pp. 275–291, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2006.10.012

[23] I. Baxter, C. Pidgeon, and M. Mehlich, "DMS ®: program transformations for practical scalable software evolution," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, May 2004, pp. 625–634.

[24] J. Siegel, Ed., *CORBA 3 fundamentals and programming*, 2nd ed. New York: OMG Press, John Wiley & Sons, 2000.

[25] G. M. K. Selim, S. Wang, J. R. Cordy, and J. Dingel, "Model transformations for migrating legacy deployment models in the automotive industry," *Software and System Modeling*, vol. 14, no. 1, pp. 365–381, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0365-1

[26] G. M. K. Selim, F. Büttner, J. R. Cordy, J. Dingel, and S. Wang, "Automated verification of model transformations in the automotive industry," in *Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, ser. Lecture Notes in Computer Science, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. J. Clarke, Eds., vol. 8107. Springer, 2013, pp. 690–706. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-41533-3_42

[27] G. M. K. Selim, L. Lucio, J. R. Cordy, J. Dingel, and B. J. Oakes, "Specification and verification of graph-based model transformation properties," in *Graph Transformation - 7th International Conference, ICGT 2014, Held as Part of STAF 2014, York, UK, July 22-24, 2014. Proceedings*, ser. Lecture Notes in Computer Science, H. Giese and B. König, Eds., vol. 8571. Springer, 2014, pp. 113–129. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09108-2_8

[28] S. Bunzel, "AUTOSAR - the standardized software architecture," *Informatik Spektrum*, vol. 34, no. 1, pp. 79–83, 2011. [Online]. Available: http://dx.doi.org/10.1007/s00287-010-0506-7

[29] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.08.002

[30] B. Barroca, L. Lucio, V. Amaral, R. Félix, and V. Sousa, "Dsltrans: A turing incomplete transformation language," in *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, B. A. Malloy, S. Staab, and M. van den Brand, Eds., vol. 6563. Springer, 2010, pp. 296–305. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19440-5_19