# The Application Of Correctness Preserving Transformations To Software Maintenance

J. Paul Gibson, Thomas F. Dowling
Department of Computer Science
National University of Ireland, Maynooth
Kildare, Ireland
{pgibson,tdowling}@cs.may.ie

Brian A. Malloy
Department of Computer Science
Clemson University
Clemson, SC 29634
malloy@cs.clemson.edu

*Abstract*— **The size and complexity of hardware and software systems continues to grow, making the introduction of subtle errors a more likely possibility. A major goal of software engineering is to enable developers to construct systems that operate reliably despite increased size and complexity. One approach to achieving this goal is through formal methods: mathematically based languages, techniques and tools for specifying and verifying complex software systems. In this paper, we apply a theoretical tool that is supported by many formal methods, the *correctness preserving transformation* (CPT), to a real software engineering problem: the need for optimization during the maintenance of code. We present four program transformations and a model that forms a framework for proof of correctness. We prove the transformations correct and then apply them to a cryptography application implemented in *C++*. Our experience shows that CPTs can facilitate generation of more efficient code while guaranteeing the preservation of original behavior.**

*Keywords*— **Reverse engineering, formal methods, public key cryptography, correctness preserving transformation, code optimization.**

## I. Introduction

The size and complexity of hardware and software systems continues to grow, making the introduction of subtle errors a more likely possibility. Some of these errors may cause inconvenience or loss of money, while some errors may even cause loss of life. A major goal of software engineering is to enable developers to construct systems that operate reliably despite increased size and complexity. One approach to achieving this goal is through formal methods: mathematically based languages, techniques and tools for specifying and verifying complex software systems[8]

Although formal methods have increasingly been applied to the specification and verification of software models and systems, they have rarely been applied to software maintenance[24]. Perhaps the rare application of formal methods to maintenance is due to the difficulties involved: the program developer has complete control over the structure and organization of the development process, whereas the reverse engineer must maintain a completed system, possibly poorly documented and poorly constructed. Due to this difficulty, many attempts at applying formal methods to software maintenance have targeted toy programs rather than real applications[24].

In this paper, we apply a theoretical tool that is supported by many formal methods, the *correctness preserving transformation* (CPT), to a real software engineering problem: the need for optimization during the maintenance of code. We present four program transformations and a model that forms a framework for proof of correctness. We prove the transformations correct and then apply them to a cryptography application[11] implemented in the *C++* programming language[1]. Our experience shows that CPTs can facilitate generation of more efficient code while guaranteeing the preservation of original behavior. Our ongoing work includes the application of the CPTs to other sections of the cryptography application, and reusing the CPTs in other applications[21], [22].

The remainder of this paper is organized as follows. In the next section we provide background about cryptography, the cryptography application that we use as a case study, and the formal techniques that we employ in this paper. In Section III we present our methodology for proving correctness. Section IV contains the four CPTs and the proofs of correctness. In Section V we present the results of our case study where we apply the CPTs to a cryptography application[22], showing a dramatic increase in efficiency with only a single application of the CPTs. Finally, in Section VI, we draw conclusions.

## II. Background

In the next section we provide background about a cryptography application that we use to demonstrate the ef-

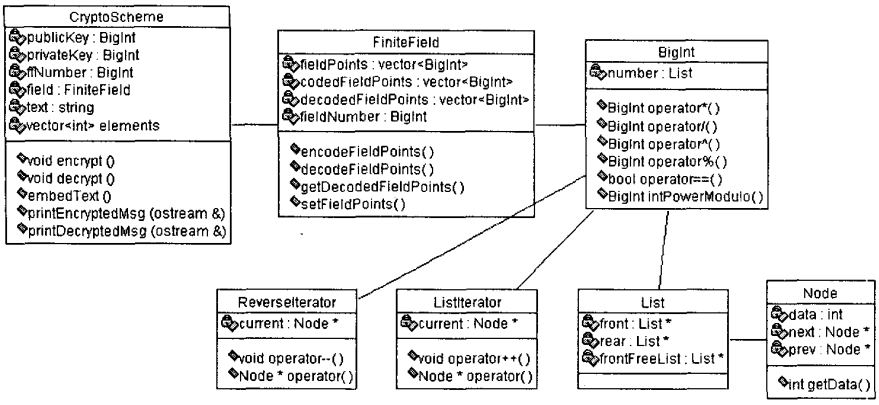


Fig. 1. *Class diagram for cryptography application.* This figure illustrates the important classes in our application for encryption and decryption. All four of our CPTs were applied to the BigInt and List classes. However, the other classes contain opportunity to apply some of the CPTs that we present.

ficiency of our CPTs. Section II-B contains background about the formal techniques that we use in this paper.

### A. The Cryptography Application

Our cryptography application uses *public key* encryption where the encryption and decryption keys are distinct. It's possible to determine the decryption key from the encryption key but, by using sufficiently large keys, this determination is computationally infeasible. This protection is achieved by the use of a *trapdoor process.* A trapdoor process is computationally trivial in one direction but computationally infeasible in the opposite direction without some additional information. There are many examples of trapdoor processes in mathematics but our application uses finite fields. To make decryption computationally infeasible we will use sufficiently large Finite Fields. By sufficiently large we mean the number of elements in the field to be of the order of one hundred digits.

The implementation that we use is based on the El Gamal encryption scheme[12]. The finite field version works as follows: Let the integer equivalent of the message to be transmitted be denoted by $P$. Users A and B start by deciding upon a very large finite field $\mathbf{F}_p$ and a generator $g$ of that field. User A randomly chooses an integer $a$ in the range $0 < a < p - 1$. This is the secret deciphering key. A then computes and publishes $g^a$. This is the public key. User B does the same thing. To send a message to user A, an integer $k$ is chosen at random and A is sent the following pair of elements of $\mathbf{F}_p$ ,

$$(g^k, Pg^{ak})$$

Recall that $g^a$ is publicly known but $a$ is known only to the user A. With this knowledge the user A can strip off the $g^{ak}$ and retrieve $P$, but without the knowledge of $a$ no one else can retrieve $P$.

Figure 1 illustrates the important classes for implementing our cryptography application. Users of the system need only instantiate CryptoScheme, the class shown in the upper left corner of Figure 1, with the text to be encrypted or decrypted. CryptoScheme contains methods to encrypt or decrypt the text, and uses a finite field to afford public key protection. The FiniteField class uses extended precision numbers; thus the association with BigInt, shown in the upper right corner of the figure. To implement extended precision numbers, the digits of the number are stored in a list, with iterators to traverse the number in either direction. The List class and corresponding iterator classes are shown at the bottom of Figure 1. All of our CPTs were applied to the BigInt and List classes.

### B. Formal Methods and CPTs

Software development has reached the point where the complexity of the systems being modeled cannot be handled without a thorough understanding of underlying fundamental principles. Such understanding forms the basis of scientific theory as a rationale for software development techniques that are successful in practice. This scientific theory, as expressed in rigorous mathematical formalisms, must be transferred to the software development environment. In this way, we can more accurately refer to the development of software systems as *software engineering*: the application of techniques, based on mathematical theory,

towards the construction of abstract machines as a means of solving well defined problems. This paper reports on such a *technology transfer*.

### B.1 Introducing Formal Methods

Formal methods are techniques whose principle goal is the construction of theories (and tools based on these theoretical models) for the development of *correct* software[10], [2]. A theoretical tool that is supported by many formal methods is the *correctness preserving transformation* (CPT). In this paper we are concerned with the practical application of this theory in a real software engineering problem: the need for optimization during the evolution and maintenance of code, without compromising an already *well behaved* system.

### B.2 Targeting Formal Methods

Formal methods are not widely used in real software development (the best-known exceptions are in telecommunications, safety-critical systems and embedded systems). In many cases, this is because they are not suitably supported with development tools. Furthermore, engineers are justifiably wary of some of the over-inflated claims coming from the formal methods community. This, in turn, has led to a certain mythology about the use and abuse of formal methods [16], [3]. Finally, these methods, for a variety of reasons, are viewed as being *costly*. A partial solution to this problem, supported by this paper, is the application of a very specific formal technique, in a specific problem domain, in order to provide much needed focus, so that software engineers can make up their own minds about the utility of formality.

We acknowledge that it is impossible for a software system to be *perfect*. When developing software we must decide upon the primary characteristics that will drive the process. Correctness is only one such characteristic among many others such as: time-to-market, user-friendliness, completeness, performance, fault tolerance, scale-ability, extensibility, portability, re-usability and cost. In this paper we report on our choice to employ formal techniques in the verification of our code optimizations. Our experience shows that *correctness preserving transformations* can facilitate the generation of more efficient code while guaranteeing the preservation of original behavior.

### B.3 Transformational Design in formal development

Design is the process that transforms an initially abstract (implementation independent) specification of system requirements into a final, more constructive (implementation oriented) specification. A fundamental notion in this work is design trajectory: a sequence of steps, where each step changes the previous specification in some way. The important thing is that something must also be preserved along this trajectory: the *correctness of the design*[23].

At each step, a transformation can be applied that reflects some architectural choice, without altering the external (observable) behavior of the system. In theory, it is possible to verify the correctness of any given design (or programming) step by mathematical means[10], [26]. In practice, the complete formal verification of most design steps is not possible because of combinatorial problems. In these cases, specifications are partly verified by simulation and testing.

In this paper we take the view that a code optimization corresponds precisely to the formal methods notion of a design transformation. We treat the non-optimum code as if it were a functional specification of the required behavior, and we consider the optimized code to be the result of applying some sort of design modification to the original code. A formal framework provides the means of proving that the transformation (optimization) is *correct*, by proving that the functional behavior is maintained. Of course, such proofs can be long and tedious and prone to error. However, if we can re-use the formal analysis or proof every time we perform the same transformation, then this re-use would go a long way toward overcoming the expense of the proof. This re-use is one of the contributions of this paper.

### B.4 Re-usable Proofs — CPTs

In CPT-driven design, we get verification *for free* because we apply only transformations (design changes) whose correctness has already been proven. For a classic CPT-driven approach to verification we recommend the paper by Brown[5]. We proceed by introducing formal terminology about CPTs.

A specification can be said to be *correct* if it fulfills some property. Assume a specification $S$, a transformation $T$ and define $S' = T(S)$, i.e. $S'$ is the result of applying $T$ to $S$. $T$ can be said to be correctness preserving with respect to the property $P$ if $P(S) \Rightarrow P(S')$. In other words, the property $P$ is preserved across the transformation $T$.

We chose to distinguish between external and internal

properties. *External* properties are those that can be observed through interaction with a system at its external interface. They are said to be purely functional as they are concerned with *what* the system does rather than *how* (well) it does it. *Internal* properties are those that can be derived through examination of the text that specifies the system in question. They cannot be 'extracted' through interaction with the system interface alone. Formulation of these properties requires the definition of a non-standard interpretation of the specification. This interpretation is said to provide a *view* on the system. In this paper we are concerned with CPTs that maintain external properties. Such CPTs are said to be *structural.*

By differentiating between what should stay the same and what should be different, as the result of a design change, an elegant and formal statement of the requirements of a design step can be given as follows. Given:
A specification $S_1$
An implementation relation $R$
A view function $V$, which has $S_1$ in its domain
A view property $P$ that is fulfilled by $V(S_1)$, i.e. $P(V(S_1))$ is true.
A view property $P'$ and a second view $V'$ such that not$(P'(V'(S_1)))$

A *structural* design change corresponds to the specification of $S_2$, the next design, such that:
$R(S_1, S_2)$, and $R$ is a strong bisimulation equivalence[1].
$P(V(S_2))$ and $P'(V'(S_2))$

In other words, $S_2$ maintains the external behavior of $S_1$, maintains the view property $P$ and adheres to a new view property $P'$, which was not fulfilled by $S_1$. One could say that the reason for defining $S_2$ was the fulfillment of this new property.

### B.5 Code Optimizations as CPTs

The formulation of a code optimization as a CPT is straightforward:
$S_1$ corresponds to the original piece of code to be optimized
$S_2$ corresponds to the new code
$V$ is some view formalized by the algorithmic complexity
$P$ is some (complexity) property exhibited by $S_1$
$P'$ is some (complexity) property exhibited by $S_2$ but not $S_1$
$V'$ is usually taken to be the same as $V$, but it could be formulated, for example, as a more precise statement of complexity that could address such issues as best/worst-case scenarios and/or memory usage.

## III. Methodology: Reverse Engineering for Correctness

In our formulation of a code optimization as a CPT, we have hidden the crux of the problem, namely: proving that the final behavior of our 2 pieces of code are functionally equivalent. The semantics of our chosen programming language ($C$++) make such a proof practically impossible to construct, given the current state-of-the-art, in all but the most trivial cases. However, this does not imply that we should completely abandon our theoretical goal of proving the optimization to be correct. In fact, we can, using expert knowledge of $C$++ and the implementation of its semantics (as specified by the compiler), argue for equivalence in a semi-formal (rigorous) manner. Our first goal should be to verify the soundness of such rigorous arguments. This can best be achieved by expressing the argument in a formal framework. There are two possibilities:

- The $C$++ code was developed from a formal specification and we can re-use this specification framework in order to formulate and verify our reasoning. (In the current context of software engineering, this is very unlikely.)
- No formal specification of the code exists and we have to reverse engineer the functional requirements through a process of abstract interpretation, in order to verify our reasoning.

In this paper, we report on our experiences with the second case. The advantage over the first case is that we can choose to formulate our reasoning in whatever framework best suits our needs. The disadvantage is that we have more work to do in order to build an abstract model.

### A. The problem of multi-semantic models

We must acknowledge that a weakness inherent in our approach arises from our use of at least two potentially very different semantic frameworks. The proof of consistency between models in different semantic domains is a very important area of on-going research (see [13], [15] for some of our previous work in this area). Without consistency, our reasoning in one domain may not be valid in the other. It is beyond the scope of this paper to directly address this issue: we do return to it in later sections and argue that our experiments provide further motivation for continuing the research into constructing unified mixed-semantic models.

### B. Abstract interpretation for reverse engineering

When we reverse engineer the code to a more formal mathematical model, it is important that we abstract away from irrelevant implementation details[20]. The goal is to develop an abstract interpretation that captures precisely the functional requirements, the internal view and the non-functional properties (and no more). In many cases, this

[1]Strong bisimulation equivalence states that the behavior trees offered by $S_1$ and $S_2$ are the same (even if the way in which they are specified is different). In programming languages, this would correspond to being functionally equivalent.

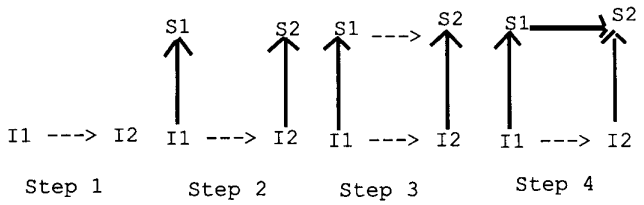


Fig. 2. Implementation Oriented Optimization

will require a close structural mapping between implementation code and the reverse engineered specification. This could prove problematic if the structures supported in the implementation language are not directly supported in the formal framework. However, as we shall see later in this section, the use of object oriented structuring mechanisms makes it easier for us to find correspondences between structures at all levels of abstraction (see [14] for a more comprehensive treatment of this argument).

### C. Implementation oriented optimization

Consider the diagram in Figure 2.

The four steps, of the implementation oriented optimization, are as follows:

**Step 1** —
Transform the initial code, $I_1$, to optimize it as $I_2$. Sketch an argument that the transformation is correct.
**Step 2** —
Reverse engineer $I_1$ and $I_2$ to $S_1$ and $S_2$, respectively.
**Step 3** —
Formulate a proof, in the formal framework, that $S_2$ is correct with respect to the functional requirements of $S_1$. This proof must follow the argument sketched in step 1.
**Step 4** —
Attempt to verify the proof (using tools available in the formal domain). If the proof is correct then we have verified our informal reasoning of the correctness of the optimization. Otherwise, we can change $I_2$, in order to *fill in the holes in the proof*, or we can change our reasoning process, in order to re-formulate the proof.

In this approach we say that the optimization is implementation-oriented because the transformation is first formulated at the code level. This is the approach followed in this paper.

### D. Specification oriented optimization: re-usable correctness

Consider the diagram in Figure 3.

The three steps, of the specification oriented optimization, are as follows:

**Step 1** —
Reverse engineer the code, mapping $I_1$ to $S_1$.
**Step 2** —
Apply an already proven CPT to transform $S_1$ into $S_2$, where $S_2$ fulfills the additional properties required in our optimization.
**Step 3** —
Use $S_2$ to develop $I_2$. This development could possibly be supported by some sort of automated code generator.
**Ideal step** —
If the reverse engineer mapping and the code generation mapping can be proven to be correct then we have a fully formalized proof that $I_2$ is correct with respect to the external functionality of $I_1$.

### E. Correctness of the mappings

Proving the correctness of the mappings across semantic frameworks is a difficult task. However, progress has been made and this is continuing research[19]. It is clear that it is much easier to analyze the mappings when the semantic domain is small. Consequently, proving the correctness of our reverse engineering of the $C++$ code is a much more difficult task than proving the correctness of our code generation from the formal specification. In fact, the mapping from specification to implementation can be relatively straightforward to formulate: in the next section we see how the $C++$ class can be mapped onto an abstract data type (ADT[2]) specification of a type. (It is well accepted that an ADT can be conceptualized as an abstract class specification[4], [9].) This is the basis upon which we build the mappings (in both directions).

## IV. Proof of Correctness

In this section, we apply our methodology for reverse engineering the functional requirements through a process of abstract interpretation, in order to verify our transformations. Each transformation represents an ad-hoc, expert-driven optimization; currently, application of the transformations is not automated. We begin each section by describing the transformation under consideration, and then we proceed with a proof of correctness.

[2]The text by Cardelli[6] provides a good introduction to such algebraic specification languages.

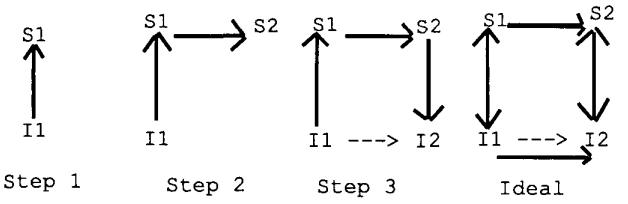


Fig. 3. Specification Oriented Optimization

```
BigInt BigInt::power(const BigInt & exponent) const {
    if (exponent == 0) return 1;
    else if (exponent == 1) return number;
    else {
        BigInt temp = power(exponent/2);
        if (exponent % 2 == 0) {
            return temp * temp;
        }
        else {
            return ((*this) * temp * temp);
        }
    }
}
```

Fig. 4. Tree Pruning (TP). This figure illustrates the C++ version of our optimization to improve the efficiency of raising an extended precision number to a power.

### *A. The TP transformation*

The goal of the tree pruning transformation, tree pruning (TP), is to reduce the computation, from O(n) to O(log n), through the use of a symmetrical argument whereby both branches of a tree can be shown to evaluate to the same value. As an example of this transformation, consider raising an extended precision number (BigInt) to an exponent. Half of the computation can be obviated by computing the result of raising the number to half of the exponent and then using multiplication to square the result. Then, in the evaluation of one branch of the tree, the same argument can be re-applied recursively. The C++ version of our code for TP is illustrated in Figure 4.

We formalize this transformation by modeling the two functions as ADT operations, on a BigInt type, and proving equivalence (through consistency analysis) of the two operations. Consider, below, the specification, written in ACT ONE (an ADT used in an LOTOS[18], an internationally recognized formal method).

**TYPE** BigInt **SORTS** BigInt
**OPNS** 0,1:-> BigInt
+,*: BigInt, BigInt -> BigInt
*equals*: BigInt, BigInt -> Bool
power1: BigInt, BigInt -> BigInt
power1local: BigInt, BigInt, BigInt, BigInt -> BigInt
**EQNS FORALL** this,exponent,answer,count: BigInt
power1(this, exponent) = power1local(this,exponent,1,0);
[equals(count,exponent)] =>
power1local(this,exponent,answer,count) = answer;
[not(equals(count,exponent))] =>
power1local(this,exponent,answer,count) =
power1local(this,exponent,answer*this,count+1);
**ENDTYPE** (*BigInt*)

This is the formal specification that arises when we reverse engineer the original `power` method of the BigInt *C++* class. There are a number of things to note, in general, about this reverse engineering process:

- The methods of the class correspond to operations of the ADT type. This maps the purely syntactic notion of interface.
- The method bodies of the class correspond to the equations of the type. This maps the semantics (meaning) of the code to be reverse engineered.
- We explicitly represent the *C++* `this` operator as the first parameter in each of the ADT operation definitions.
- We have abstracted away from the operation overloading of the *C++* power operator. It is not relevant to our proof.
- We have not given the semantics for the operators `+,-,*` and `equals`: for the moment we assume that they work as required ... we re-examine this assumption when we return to our proof of correctness.
- We explicitly provide the literal constructors (for the values 0 and 1) of `BigInt`.

Now let us consider the mapping of the *C++* `while` loop. There is a standard technique for this type of reverse engineering, which we summarize below:

- The `while` condition is mapped directly onto an expression precondition (in the square brackets).
- The use of temporary variables in the `while` loop (the `count` and the `answer`) requires us to define a local function, called `power1local`, in which these values are stored as additional parameters.
- The initialization of the local parameters, before the `while` loop, corresponds to our call to the local function

where the parameters are given the required values.

- The body of the loop maps to the re-initialization of the parameter values, in the recursive calls, as required.

The next step is to reverse engineer the optimized code, as specified by `power2` in $C$++. The resulting ADT specification is:

**TYPE** BigInt **SORTS** BigInt
**OPNS** 0,1:-> BigInt
+,*,mod,/: BigInt, BigInt -> BigInt
equals: BigInt, BigInt -> Bool
power2: BigInt, BigInt -> BigInt
**EQNS FORALL** this,exponent,temp: BigInt
square(temp) = temp*temp;
[equals(exponent,0)] => power2(this,exponent) = 1;
[equals(exponent,1)] => power2(this,exponent) = this;
[(exponent>= 1) and equals((exponent mod 2),0)] =>
power2(this,exponent) = square(power2(this, exponent/2);
[(exponent>= 1) and not(equals((exponent mod 2),0))] =>
power2(this,exponent) = this* square(power2(this, exponent/2);
**ENDTYPE** (*BigInt*)

The mapping procedure is similar to the first function, but we make the additional notes:

- We have additional assumptions to make about the operators `mod` and `/`.
- The recursive structure of the second $C$++ function maps directly onto the same recursive structure in the ADT specification
- We introduce a `square` operation in order to store the the value of our temporary variable `temp`, defined as `power2(this, exponent/2)`.
- The sequence of $C$++ `if-else` statements *requires a bit more work* in the ADT specification because there is no syntactic sugar for representing the `else` case. (However, the generation of the equivalent sequence of preconditions is easy to automate.)

To prove that power1 and power2 are equivalent in our formal framework, we follow the following strategy:

- Formulate the assumptions that we make about the operators of the BigInt class as preconditions of correctness. It is straightforward to prove these preconditions, in our formal ADT framework, but we do not report on this as part of this paper.
- State the equivalence property as: `power1(x,y) = power2(x,y)`, forall x and y of type BigInt.
- Prove the property by using the ADT tools to prove consistency of the BigInt specification containing `power1`, `power2` and the equivalence property; or prove the property directly using a proof by structural induction on the y variable.

It is beyond the scope of the paper to report on the use of the ADT tool to prove consistency of specifications. However, we do *sketch* the proof by structural induction, below:

The **base case (when y =0)** —
By definition of `power1` and `power2`,
`power1(x,0) = power1local(x,0,1,0) = 1` and `power2(x,0) = 1`
Thus, `power1(x,y) = power2(x,y)` when `y = 0`.

The **inductive case** —
We assume that:
`power1(x,y) = power2(x,y)` for some y, and we are required to prove that:
power1(x,y+1) = power2(x, y+1)

By definition of `power1` and `power1local`,
`power1(x,y+1)=power1local(x,y+1,1,0)=power1local(x,y+1,x,1)`.
Now, for any two `BigInts` (a and b, say) then `a equals b` iff `a+1 equals b+1`, thus:
`power1local(x,y+1,x,1) equals power1local(x,y,x,0)`
By definition of `*`, `power1` and `power1local`,
`power1local(x,y,x,0) equals (x*power1local(x,y,1,0)) equals (x * power1(x,y))`
Now, by the induction hypothesis, this equals `x*power2(x,y)`.

It remains only to show that: `power2(x, y+1) equals x*power2(x,y)`. This is done by considering the case where y is odd and the case where y is even. Each follows from the definition of `power2(x,y)` and the assumption that the `*` operator is *correctly defined.*

(A complete copy of the proof can be obtained from the authors, on request.)

### *B. The AR transformation*

The goal of the array reference transformation (AR), is to obviate an O(n) array lookup. To do this, data values, that are to be searched in the array, are mapped to an integer that will be used as an index into the array. Then, rather than searching the array for the value and returning the corresponding index in O(n) time, the value can be found with a direct lookup in O(1) time.

The proof of the correctness of this transformation is based upon proving that the alphabet array corresponds to the identity function. In the reverse engineering of the original non-optimized code, we map the notion of an array into a function whose domain is the set of valid array indices. For this, we chose a pure functional language (like SML[25]) as our formal framework (whose semantics corresponds to the lambda-calculus of Alonzo Church [7]). We *sketch* the proof, below:

Define: `alphabet = [ (0,0); (1,1); (2,2); ...(255,255)`
`arrayindex((x,y)::z, i)= if i=x then y else arrayindex(z,i)`
Now, we prove that:
`arrayindex(alphabet, x) = x`, provided `0<= x <= 255`
Thus, the non-optimum access function defined as:
`access1 (alphabet, i) = arrayindex(alphabet, expression(i))`
Can be re-written as an equivalent function `access2`, defined as:
`access2 (alphabet, expression(i)) = expression(i)`

Although this proof is straightforward, we have gained some insight into why it is superior to reasoning directly in the $C$++ framework: the formulation of the abstract interpretation improves our understanding of the reasoning

process and identifies assumptions that need to be verified for the transformation to be correct. In this example, there is an important aspect of this abstract interpretation that needs explanation: the `expression(i)` in our formal model is evaluated twice without any side-effects. In fact, our reasoning would break down if there were any side-effects in the expression when used in the *C++* code. Through simple inspection, we argue that instantiating the expression as `(int)text[i]` is valid (since its evaluation is side-effect free in *C++*).

### C. The MM transformation

The goal of the memory management transformation (MM), is to minimize calls to new and delete to manage dynamic memory. In MM, the delete operator is overloaded to place the deleted object into a free list. Also, the new operator is overloaded and the actions of new are to first check the free list: if the free list is not empty, then the requested storage is allocated from the free list; otherwise, the system new is used to allocate memory from the heap.

The proof of correctness of this transformation cannot be treated satisfactorily in this paper. It is more complex than the other three: we are preparing a separate paper on the approach taken to prove this complicated transformation to be correct. (For those interested, the crux of the problem is in finding a suitable abstract interpretation for the memory in the system and choosing the best formal representation. Unlike the other transformation, these choices are not straightforward.)

### D. The BA transformation

The goal of the boundary analysis transformation (BA), is to avoid a computation by exploiting the shortcut provided by a boundary computation. For example, when computing the modulus two of a number, the result is determined by the digit in the unit position: if this digit is even, then the result is zero, if this digit is odd, then the result is one.

The modeling of this transformation is trivial, in almost any formal framework. The proof of correctness is also trivial: all we need is an abstraction (defined as a function, `AI`, say) of the data values that specifies an equivalence between different numbers provided they return the same result with respect to the specified modulus computation. Then, in the optimized function, the computation can be defined directly on the abstraction rather than on the number itself. The proof can then be formulated as: `Modulus (number) = Modulus2 (AI(number))`, for all valid numbers

## V. Applying the CPTs to an Application: A Case Study

In this section, we report on the efficiency improvement that we achieved by applying the four correctness preserving transformations (CPTs) to a cryptography application[22]. All executions that we report were executed on a 500 MHz Dell Optiplex, running the Linux Red Hat 6.1 operating system. We implemented the cryptography application with the egcs C++ compiler, release 1.1.2. Each of the executions that we report represent the results of twelve executions, the lowest and highest value was discarded and the reported result is the average of the ten remaining values.

In the next section we report our results after applying each of the four CPTs in turn, so that we can compare the effect of each optimization. We used profile information, acquired from the gprof profiling tool, to guide placement of each of the CPTs.

In Section V-B, we report our results after applying all four CPTs to the application; we only applied each CPT a single time to facilitate clarity of analysis. We applied the CPT at the same point in the application that it was applied for the executions described in Section V-A.

### A. Results for each individual CPT

Figure 5 illustrates the results of applying each of the CPTs, in turn, to the cryptography application. The table at the top of the figure shows the execution times after applying each of the CPTs, the bar graph at the bottom of the figure further illustrates these timings. For the table, the first column illustrates the applied optimization; the remaining columns illustrate reported results. For example, the second column illustrates the number of seconds required to both encrypt and decrypt a file containing 1000 randomly generated characters. The first row of data illustrates timings for the original program, without any optimizations applied, the second row illustrates timings for the array reference optimization (AR), the third row for the memory management optimization (MM), the fourth row for the boundary analysis optimization (BA) and the last row illustrates timings for the tree pruning optimization (TP). For example, in the original program using none

| Program Optimization | seconds for 1000 bytes | seconds for 2000 bytes | seconds for 3000 bytes | seconds for 4000 bytes | seconds for 5000 bytes |
|---|---|---|---|---|---|
| original | 9.13 | 16.98 | 25.18 | 34.70 | 43.14 |
| AR | 8.96 | 16.39 | 24.60 | 33.78 | 42.09 |
| MM | 5.67 | 10.50 | 15.75 | 21.58 | 26.74 |
| BA | 6.21 | 11.52 | 17.28 | 23.66 | 29.54 |
| TP | 6.35 | 11.95 | 17.97 | 24.69 | 30.64 |

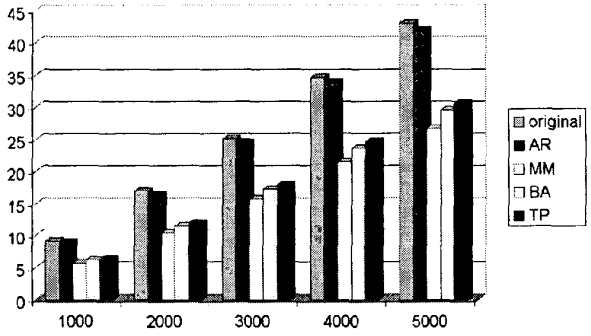


Fig. 5. *The effect of each optimization.* The results depicted in this table show the outcome of applying each of the optimizations once in the program.

of the transformations, the average time to encrypt and decrypt a file of 1000 characters was 9.13 seconds, as shown in the first row, second column of the table.

The bar graph in Figure 5 better illustrates our results. Time, in seconds, is plotted on the y-axis of the graph and file size, in 1000 byte increments, is plotted on the x-axis of the graph. The first set of five bars represents timings for the original program and each of the four optimizations. For each set of experiments, the highest bar is always the original program and the lowest bar is always the program containing the MM optimization. The array reference transformation provided the smallest improvement in execution time, but consistently provided improvement. This transformation was not placed in an area of code that was frequently executed.

### *B. Results for all four CPTs*

Figure 6 illustrates the results of applying all four of the CPTs, a single time, to the cryptography application. The table at the top of the figure illustrates timings for both the original program and for the program containing all four transformations. The columns are similar to those in Figure 5. The bar graph at the bottom of the figure further illustrates our timings, where the four transformations consistently provided sixty percent increase in efficiency. We found many more opportunity for exploiting some of the transformations; for example, we found seven places in the cryptography application where we could apply the BA transformation but only two places where we could apply the AR transformation. By applying each transformation only once, we can better see the cumulative effect of the four transformations.

## VI. Concluding Remarks

In this paper, we applied *correctness preserving transformations*, CPTs, to a real software engineering problem: the need for optimization during the maintenance of code. We have presented a framework for proving CPTs correct and four CPTs together with proofs of their correctness. We have applied the CPTs to an existing cryptography application, implemented in *C++*, and have shown dramatic improvement in efficiency with only a single use of each CPT. Our ongoing work on this project includes efforts to apply the CPTs to other sections of the cryptography application as well as the development of additional CPTs. We are also applying the CPTs to other applications to further demonstrate their re-use. Our future work includes an examination of the effects of the transformations on test set adequacy[17].

## VII. Acknowledgement

We would like to thank Paul Redkoles for his help acquiring profile information and for his tireless experiments with gprof.

## References

[1] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. American National Standards Institute, first edition, September 1998.

| Program Optimization | seconds for 1000 bytes | seconds for 2000 bytes | seconds for 3000 bytes | seconds for 4000 bytes | seconds for 5000 bytes |
|---|---|---|---|---|---|
| original | 9.13 | 16.98 | 25.18 | 34.70 | 43.14 |
| All four | 3.75 | 7.01 | 10.50 | 14.49 | 17.83 |

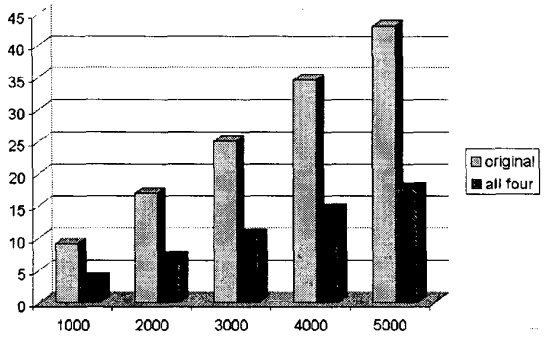


Fig. 6. *The effect of applying all four optimizations.* The results depicted in this table show the outcome of applying all four of the optimizations once in the program.


[2] R.L. Baber. *The Spine of Software — Designing Provably Correct Software: Theory and Practice, or: A Mathematical Introduction To The Semantics Of Computer Programs.* John Wiley and Sons, 1987.

[3] J. Bowen and M. Hinchley. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[4] R. Breu. *Algebraic Specification Techniques in Object Oriented Programming Environments.* Springer-Verlag, 1991. Lecture Notes in Computing Science, number 562.

[5] N. Brown. Correctness-preserving transformations for the design of parallel programs. In *ECOOP '94 Workshops : Models and Languages for Coordination and Parallelism and Distribution.* Springer Verlag, 1995.

[6] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[7] Alonzo Church. The calculi of lambda conversion. *Annals of Mathematics Studies*, 6, 1941.

[8] E. M. Clarke, J. M. Wing, and et al. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[9] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[10] J.W. de Bakker. *Mathematical Theory of Programming Correctness.* Prentice-Hall, 1980.

[11] T. Dowling and B. A. Malloy. The design of a component-based encryption scheme. *Proceedings of the Fifth International Conference on Computer Science and Informatics*, February 2000. (to appear).

[12] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on information Theory IT-31*, pages 469–472, 1985.

[13] J.-P. Gibson and D. Méry. A Unifying Model for Multi-Semantic Software Development. Rapport Interne CRIN-96-R-110, CRIN, Linz (Austria), July 1996.

[14] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS.* Tech. report csm-114, Stirling University, August 1993.

[15] Mermet Gibson and Méry. Feature interactions: A mixed semantic model approach. In *Irish Workshop on Formal Methods*, Dublin, Ireland, July 1997.

[16] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[17] M. J. Harrold. The effects of optimizing transformations on dataflow-adequate test sets. *Proceedings of the Symposium on Testing, Analysis and Verification*, pages 130–138, 1991.

[18] ISO. LOTOS — a formal description technique based on the temporal ordering of observed behaviour. Technical report, International Organisation for Standardisation IS 8807, 1988.

[19] Andy Galloway Keijiro Araki and Kenji Taguchi (editors). *Integrated Formal Methods conference (IFM99).* Springer, 1999.

[20] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[21] B. A. Malloy, D E. Bushey, and S. Yang. Using jet routes to model path re-routing in the national airspace system. *Proceedings of the 13th European Simulation Multiconference (ESM99)*, pages 543–550, June 1994.

[22] B. A. Malloy, J. D. McGregor, and S. Hughes. Integrating a gui into a command driven application. *International Journal of Computer and Applications*, 2000. (to appear).

[23] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach To Software Development.* Springer-Verlag, 1990.

[24] M. P. Ward. Reverse engineering through formal transformation. *Technical Report for Computer Science Labs*, pages 1–19, August 1994. http://www.dur.ac.uk/ dcs0mpw/martin/papers/.

[25] A. Wikstrom. *Functional Programming Using Standard ML.* Prentice-Hall, 1987.

[26] N. Wirth. Program development by step-wise refinement. *Comm. ACM*, 14:221–227, 1971.