A Language and Framework for Invariant-Driven Transformations

Yanhong A. Liu Michael Gorbovitski Scott D. Stoller
Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794
{liu,mickg,stoller}@cs.sunysb.edu

Abstract

This paper describes a language and framework that allow coordinated transformations driven by invariants to be specified declaratively, as invariant rules, and applied automatically. The framework supports incremental maintenance of invariants for program design and optimization, as well as general transformations for instrumentation, refactoring, and other purposes. This paper also describes our implementations for transforming Python and C programs and experiments with successful applications of the systems in generating efficient implementations from clear and modular specifications, in instrumenting programs for runtime verification, profiling, and debugging, and in code refactoring.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—invariants

General Terms Design, Languages, Performance

1. Introduction

Transformation systems are important for program manipulations such as optimization, instrumentation, and refactoring. Even though not always stated explicitly, these transformations are always driven by invariants, such as maintaining them for optimization, checking them for verification, and so on. Generally, we use invariants to refer to properties that hold during program executions.

For example, for optimization, to quickly return the size of a collection of data, at all program points where elements are added or removed, we must add code that updates the variable that holds the size of the collection; the invariant is that the value of the variable equals the size of the collection. For another example, for instrumentation, to check that memory is managed correctly, at any program point where a reference is added to or removed from an object, we can insert code that checks whether the variable that holds the reference count of the object is incremented or decremented appropriately, and if not, prints an error message and stops the program; the invariant is that either the variable equals the number of references or the error message is printed and the program is stopped. For yet another example, for refactoring, for any code fragment that is the same as the body of a given method modulo a substitution for the parameters of the method, we can replace the code fragment with a call to the given method with arguments obtained from the substitution; the invariant is that each call to the method is equivalent to the corresponding replaced code fragment.

This paper describes a language and framework that allow coordinated transformations driven by invariants to be specified declaratively, as invariant rules, and applied automatically. This allows important program design and development knowledge to be captured explicitly and reused from application to application. The language also allows explicit specification of cost considerations. The framework supports incremental maintenance of invariants for program design and optimization, as well as general transformations for instrumentation, refactoring, and other purposes. The declarative nature also allows alternative implementations to be used, such as static vs. dynamic checks, based on efficiency trade-offs.

We have developed two implementations, InvTS/py and InvTS/c, for transforming Python and C programs, respectively. The systems have been used successfully in many applications, including generating efficient implementations from clear and modular specifications [Liu et al. 2005], instrumentation for profiling, runtime invariant checking [Gorbovitski et al. 2008a], and debugging [Gorbovitski et al. 2008b], as well as code refactoring during the implementation of InvTS/py. We describe experiments showing the efficiency and effectiveness of InvTS/py and InvTS/c for these applications.

There is a large amount of work on program transformation languages and systems, including more than a decade of work on aspect-oriented programming, as discussed in Section 7. Coordinated transformations for maintaining invariants were implemented as early as 20 or 30 years ago and recently for incrementalization, runtime invariant checking, and query-based debugging. This paper is the first complete and precise description of such a powerful language, its different usages and the key ideas that connect them, the main choices in implementations, and extensive experiments with applications.

2. Invariant-driven transformations

Transformations for optimization and verification, as well as refactoring, instrumentation, and debugging, are all driven by invariants. We motivate *invariant rules* as a concrete form for capturing program design knowledge as invariant-driven transformations.

Maintaining invariants for design and optimization. What programs do on data can be classified as, or decomposed into, two kinds of operations: queries and updates, where queries compute results using data, and updates change data. For a simple example, consider the `LinkedList` class in Java 1.5. It has a query method `size` that returns the number of elements in the list, 12 other query methods that return elements, element indices, etc., and 15 update methods that add or remove elements.

How to implement the queries and updates can vary dramatically. In a straightforward implementation, each method performs

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.
Copyright © 2009 ACM 978-1-60558-494-2/09/10. . . $10.00

its respective query or update. In the `LinkedList` example, `size` can iterate over the list, and each update method can simply do its addition or removal. This is clear and modular, but can have poor performance when such queries are performed frequently. A sophisticated implementation can maintain the results of these queries—i.e., maintain the invariants that the values retrieved from certain variables equal the results of these queries—incrementally with respect to updates to the query parameters—i.e., variables or fields on which the queries depend. In the `LinkedList` example, the result of `size` may be maintained in a field and simply be returned when queried. This is efficient, but no longer clear and modular, because each of the 15 update methods must also update this field appropriately.

This conflict between clarity and efficiency is much worse for complex systems with many queries and updates, where queries may involve objects from different classes, and updates may be spread in many classes. A query can be affected by many updates, and an update can affect many queries. It poses a serious challenge to consider all the complex dependencies and trade-offs and decide where and how to maintain what invariants. The resulting code can be significantly more difficult to understand.

To resolve this conflict, it is desirable to automatically transform straightforward yet inefficient implementations into efficient yet sophisticated implementations, and further to express these transformations with explicit invariants and cost considerations. We express these transformations declaratively using *invariant rules*. An invariant rule expresses how to maintain an invariant under a set of possible updates, together with the costs of the query, updates, and maintenance.

For example, the rule in Figure 1 expresses that, to maintain the invariant that r equals the size of set s when every update that may affect the size of s is assigning s a new empty set, adding an element x to s, or removing an element x from s, the respective maintenance is assigning r the value 0, incrementing r by 1 if x is not in s before the addition, or decrementing r by 1 if x is in s before the removal; the cost of the original query is linear in the size of s, and the cost of each update and maintenance is asymptotically the same as the cost of evaluating x, denoted $\mathrm{cost}(x)$, assuming that the set operations used in the rule take constant time. Thus, the

```
inv  r = s.size()                 O(|s|)

 at  s = new set()                O(1)
 do  r = 0                        O(1)

 at  s.add(x)                     O(cost(x))
 do  before
       if not s.contains(x):      O(cost(x))
         r = r+1

 at  s.del(x)                     O(cost(x))
 do  before
       if s.contains(x):          O(cost(x))
         r = r-1
```

Figure 1. An invariant rule for set size.

linear-time `size` query can be replaced by a constant-time retrieval from r at no extra asymptotic cost in maintenance, regardless of the frequencies of queries and updates. The precise language constructs and cost models are described in Section 3.

Expressing coordinated incremental maintenance of invariants using invariant rules is high-level and declarative, making the transformations easier to understand, use, extend, and verify. The semantics of the rules encapsulates many low-level, procedural details. For example, all updates to the parameters of a query must be detected, one way or another, even in the presence of object aliasing, and maintenance must be performed at all updates. This contrasts with traditional use of individual rewrite rules with programmed strategies for tree walking, program analysis, and rule applications.

Invariant rules can be put in a library and reused from application to application, as opposed to being re-discovered and manually embedded in scattered places in each application program. While it may be extremely difficult to manually maintain multiple scattered invariants under many scattered updates correctly, doing so by automatically applying a library of invariant rules is easy.

General program transformations. While invariant rules are designed to express coordinated transformations that together preserve invariants, they can also express general program transformations that do not require such strong coordination. Nevertheless, it is important to note that general program transformations also preserve various kinds of invariants, albeit generally done implicitly. Invariant rules can help make the invariants more explicit, and help express these transformations more easily and declaratively. We discuss examples in instrumentation for profiling, monitoring, and debugging, and in refactoring.

Program instrumentation transforms a program to do additional logging, checking, etc. It is important for addressing performance, security, and general correctness issues, by profiling frequencies of operations, monitoring accesses to data, etc. The invariants are that the behavior of the involved program fragments is preserved and the additional logging, checking, etc. are done when certain conditions hold. For example, to profile the frequencies of queries and updates, an invariant rule can match the queries and updates and increment a corresponding counter when a query or update is executed; to check a complex invariant efficiently at runtime, an invariant rule can incrementally maintain the results of expensive computations in the invariant. These rules can be generated automatically from the invariant rules for incremental maintenance.

Instrumentation to help debugging, e.g., to log certain kinds of events, can easily be inserted with invariant rules, similar as with aspect-oriented programming, for which debugging is a show-case application. For example, to track where the value of a variable was last changed to a bad value, an invariant rule can match all assignments to that variable and appropriately record the program point when the variable is last assigned the bad value.

Program refactoring generally refers to transformations that improve code quality, e.g., readability, extensibility, or modularity. Typical examples are renaming variables and turning blocks of code into subroutines. It is not hard to observe the invariants. For example, for renaming variables, the invariant is that the value of the old variable in a desired context always equals that of the new variable. For introducing subroutines, the invariant is that the original block of code is equivalent to the introduced subroutine call. Preservation of semantics is a nontrivial issue in refactoring. For example, when one wants to rename variable `i` to `interest` in a certain scope, occurrences of `i` in other scopes should not be changed.

3. Invariant rule language

An invariant rule declaratively specifies that an invariant holds if all updates to the values that the invariant depends on are certain kinds of updates, and the corresponding maintenance work is performed at each update. It can also specify additional conditions on the query and updates, and additional declarations needed for the maintenance.

3.1 Core form of invariant rules

The core form of an invariant rule is:

$$\begin{array}{l} \mathbf{inv}\ \ r = \mathit{query} \\ (\mathbf{at}\ \ \mathit{update} \\ \ \mathbf{do}\ \ \mathit{maint})+ \end{array} \qquad (1)$$

where $query$, $update$, and $maint$ are patterns for matching queries, updates, and maintenance operations, respectively. The "+" indicates that there may be one or more instances of the clause.

The semantics of an invariant rule is: if a query in a program matches the $query$ pattern, and every update to the parameters of the query in the program matches at least one of the $update$ patterns, then a fresh variable instantiating r is declared in the program, occurrences of the query are replaced with uses of that variable, and at every update to the parameters of the query, the maintenance corresponding to the matching $update$ patterns is inserted. Note that if a rule does not handle some updates to the parameters of a query in a program, then the rule does not apply to the query and its updates. We say that a rule *preserves* the invariant $r = query$, if (1) $r = query$ holds after initialization of r and (2) for each pair of $update$ and $maint$, if $r = query$ holds, then it still holds immediately after execution of $update$ and $maint$, for all instances of $query$, $update$, and $maint$. (We do not consider concurrency here.) It is easy to see that preserving an invariant is a property that can be checked individually for each rule.

In the core form above, the maintenance work corresponding to an update can be done either before or after the update; this is correct if the maintenance code does not use the values of the variables assigned to by the update. To accommodate maintenance code that uses the values of those variables, the do-clause may have the form:

$$\begin{array}{ll}\textbf{do} & maint?\\ & (\textbf{before}\ maint_1)?\\ & (\textbf{after}\ maint_2)?\end{array} \tag{2}$$

where $maint$ can be done either before or after the update, $maint_1$ must be done before, and $maint_2$ must be done after. A "?" after a clause indicates that the clause may be omitted. We allow a do-clause to be omitted if no maintenance needs to be done at an update.

To facilitate cost consideration, an invariant rule may specify the costs of the query, updates, and maintenance, by including a cost-clause of the following form after each of them:

$$\textbf{cost}\ \ cost \tag{3}$$

In this paper, we use asymptotic running time as the cost model, and we assume that standard hashing is used for set and map operations. Other cost models that consider running time with constant factors, space usage, etc. could also be used. For ease of reading, we omit the keyword **cost** and align the costs to the right.

For example, the invariant rule in Figure 1 has the core form.

Meta variables and meta functions. Variables in the rules in italic font are *meta variables*.

A meta variable in a query or update pattern may match any program syntax element, except for restrictions imposed by the specific contexts of the variable in the pattern. For example, in the rule for set size in Figure 1, s and x are meta variables in the query and update patterns; s `= new set()` restricts s to match an lvalue, and s`.add(`x`)` restricts x to match an expression. Standard substitution is used to replace meta variables in patterns with matched program text. Other parts of patterns that are displayed in teletype font match program text exactly.

The scope of a meta variable in the query pattern is the entire rule. The scope of a meta variable that appears in an update pattern but not in the query pattern is the update clause and the corresponding maintenance clause. When matching occurrences of a name in the program, the scoping rules of the program being transformed are followed.

Meta variables not in the query and update patterns, including r, denote distinct names not used for other purposes in the program being transformed in the scopes of these names. Such a name can be introduced in any scope that contains all uses of the name in maintenance, but for program clarity and modularity, by default, it is introduced in the smallest of these scopes. In particular, if the query and all updates and maintenance are in the same method, then r is instantiated with a new local variable of that method; otherwise, r is instantiated with a new field of the class that contains the query.

Finally, functions may be used in rules to help specify application conditions and form new program text, as discussed in the following subsections. They are called *meta functions* and are displayed in normal font.

Aliasing. A meta variable can match different expressions that are aliases for the same object. For example, if `s1` and `s2` are aliases at an update `s2.add(x2)`, then this update affects the invariant `r1` = `s1.size()`, just like `s1.add(x1)` does.

3.2 Conditions on query and updates

Conditions that must be satisfied by a query or an update matched by the inv-clause or an at-clause of a rule can be specified by an if-clause of the following form immediately after the inv-clause or at-clause, respectively:

$$\textbf{if}\ \ condition+ \tag{4}$$

where $condition$ is a Boolean expression in the invariant rule language.

For example, a rule may maintain the size of a set only if elements of the set are of a certain type. This condition involves only the matched query, and may be specified in an if-clause immediately after the inv-clause. For another example, a rule may maintain the size of a set only if all updates to the set appear in the same class as the query. This condition involves also matched updates, and may be specified in an if-clause immediately after each at-clause.

Conditions may use meta variables in the query and update patterns. For convenience, the special meta variable $query$ refers to the matched query, and the special meta variable $update$ under an at-clause refers to the matched update.

Conditions may also use meta functions that provide syntactic and semantic information from program analysis; this paper does not restrict the kinds of analysis that can be used. In particular, meta function $\mathrm{alias}(x, y)$, which returns whether x and y may alias each other, is used in detecting all updates that may affect a query result. It may also be used explicitly in conditions. It can be computed using the analysis in [Gorbovitski et al. 2009].

Conditions may impose strong static requirements. For example, Figure 2 shows another invariant rule for set size, where meta function $\mathrm{isin}(x, s)$ returns whether x is a member of set s if this can be determined statically, and unknown otherwise. This rule applies if the membership conditions are statically known to hold at all updates to the query, so the maintenance does not need to test membership at runtime, and the maintenance can be done either before or after the update.

$$\begin{array}{lll}
\textbf{inv} & r = s\texttt{.size()} & O(|s|)\\[1ex]
\textbf{at} & s\ \texttt{= new set()} & O(1)\\
\textbf{do} & r\ \texttt{= 0} & O(1)\\[1ex]
\textbf{at} & s\texttt{.add(}x\texttt{)} & O(\mathrm{cost}(x))\\
\textbf{if} & \mathrm{isin}(x,s) = \mathrm{false} & \\
\textbf{do} & r\ \texttt{=}\ r\texttt{+1} & O(1)\\[1ex]
\textbf{at} & s\texttt{.del(}x\texttt{)} & O(\mathrm{cost}(x))\\
\textbf{if} & \mathrm{isin}(x,s) = \mathrm{true} & \\
\textbf{do} & r\ \texttt{=}\ r\texttt{-1} & O(1)
\end{array}$$

Figure 2. Another invariant rule for set size.

3.3 Declarations

An invariant rule may specify declarations needed for maintenance. Declarations used by maintenance under multiple at-clauses or a single at-clause may be specified by a de-clause of the following form after the inv-clause or the corresponding at-clause, respectively:

$$\textbf{de}\ \ ((\textbf{in}\ \textit{scope}\ :)?\ \textit{declaration}+)+ \qquad (5)$$

where *scope* is a scope expression, defined below, in the invariant rule language that evaluates to a scope in the program being transformed, and each *declaration* is a declaration in the language of the program being transformed and may contain meta variables and meta functions.

For example, a rule for maintaining set size may declare r to be a field in the class that contains the set size query in a de-clause after the inv-clause. For another example, a rule for maintaining the minimum of a set under element addition and deletion may declare a heap data structure in a de-clause after the inv-clause, and may use a de-clause after an at-clause to declare local variables used only within the maintenance code under that at-clause.

A scope expression has the form $(\textit{kind}\ \textit{name})+$, or **global**, where *kind* is **method**, **class**, **package**, or **file**, and *name* evaluates to the name of a method, class, package, or file. For transforming programs in a given language, only the kinds allowed in that language may be used. An omitted kind uses as the default value the scope of the query or update pattern in the inv- or at-clause preceding the de-clause. For example, rules for transforming Java or Python programs may use the scope expression

class `myset` **method** `add`

to indicate that local variables should be declared in method `add` of class `myset` of the default package. Specification of the scope for a list of declarations is optional. Recall from Section 3.1 that, by default, the smallest suitable scope is used.

If the variable, field, method, class, or package name in a de-clause is a meta variable not used in the query and update patterns, it denotes a distinct name not used for other purposes in the given program. Variables declared with global scope may be read and written from everywhere in the program; the implementation depends on the language of the program being transformed. Note that multiple maintenance clauses, and even multiple rules, may refer to the same declarations simply by using program text without meta variables.

In examples, we assume the language being transformed uses declarations of the form $\textit{name} : \textit{type}$. For example, to declare r of type `int` to be global, one may specify

de in global : r`: int`

and to declare r of type `int` in the class that contains the set size query, one may specify

de in class class(*query*) : r`: int`

where meta function class(p) returns the enclosing class of the program syntax element p.

3.4 General form of invariant rules

In general, work can also be done at the query to help with incremental maintenance. Such work can be specified as a do-clause below the inv-clause. For example, to incrementally maintain the average of a set of numbers, one may incrementally maintain the sum and the count, and do a division right before the query, instead of doing the division immediately after the maintenance of sum and count.

We also allow the inv-clause to specify an equality between any two program syntax elements, not just a variable and a query expression. This is convenient, for example, if a query result is stored in a part of a data structure instead of a variable: the invariant may equate an expression that retrieves the query result with the query.

Finally, in the do-clause after an at-clause, the keyword **instead** can be used to indicate that an update matched by the update pattern should be replaced with the maintenance. This is useful when the update needs to be transformed. For example, the rule in Figure 1 has a problem: x can match any expression, not only a variable, and that expression will be evaluated both in the original call to `add` or `del` and in the maintenance; this is incorrect if x has side-effects. We can fix this problem, and reduce the maintenance cost to $O(1)$, by either adding a condition restricting x to match only variables, or replacing the do-clause under `add` with the following and changing the do-clause under `del` similarly:

```
do instead
    v = x
    if not s.contains(v):
      r = r+1
    s.add(v)
```

In summary, the general form of an invariant rule is:

```
inv  result = computation
(if  condition+)?
(de  ((in scope :)? declaration+)+)?
(do  maint? (before maint)? (after maint)?)?
( at  update                                          (6)
  (if  condition+)?
  (de  ((in scope :)? declaration+)+)?
  (do  maint? (before maint)? (after maint)?
        (instead maint)?)? )+
```

where *computation*, *result*, *update*, *declaration*, and *maint* are program text in the language of the program being transformed, except that they may contain meta variables and meta functions; and *condition* and *scope* are a Boolean expression and a scope expression, respectively, in the invariant rule language. Cost may be specified for *computation*, *result*, and each *update* and *maint*. In this paper, we indicate meta variables with italic font, indicate meta functions with normal font, and indicate program text with teletype font. In our implementation, we indicate meta variables with a preceding "`$`", indicate meta functions with a preceding "`$$`", and indicate program text, possibly containing meta variables and meta functions, with a pair of curly braces following a language indicator, for example `py{$s.size()}` for program text in Python containing meta variable `$s`.

4. Additional invariant rule examples

We give additional examples that show different usages of invariant rules and discuss developing and verifying invariant rules.

Incrementally maintaining join queries. The rule in Figure 3 maintains the result of the query

```
{r: r in ROLES | (s,r) in SR, ((op,o),r) in PR}
```

under initialization and element addition and deletion for sets `ROLES`, `SR`, and `PR`. Given these sets and the values of `s`, `op`, and `o`, the query includes a role `r` from `ROLES` in the result set if the session-role pair `(s,r)` is in `SR`, and the permission-role pair `((op,o),r)`, where an operation-object pair is called a permission, is in `PR`. The query is used for the `CheckAccess(s,op,o)` operation in RBAC [ANSI INCITS 2004]. Its incremental maintenance was presented in pieces previously [Liu et al. 2006] without an expressive invariant rule language. `CheckAccess` is the most frequently used and most time critical operation in RBAC.

The incremental maintenance uses a map `MapSP2R` that maps any given values of `s`, `op`, and `o` to the desired set of roles. The inv-clause says to retrieve the query result from the map using `MapSP2R[(s,op,o])`, and it takes $O(1)$ time. Two additional maps are maintained: `SRMapR2S` is the inverse map of `SR`, and `PRMapR2P` is the inverse map of `PR`.

In the cost-clauses, `SR21` denotes the maximum number of elements in the first component of `SR` for any element in the second component of `SR`, and similarly for `PR21`. Applying this rule allows the query to be done in minimum time, at the expense of more expensive updates.

```
inv MapSP2R[(s,op,o)] =                              O(1)
    {r: r in ROLES | (s,r) in SR, ((op,o),r) in PR}
                                                     O(|ROLES|)
 at ROLES = new set()                                O(1)
 do MapSP2R = new map()                              O(1)

 at SR = new set()                                   O(1)
 do MapSP2R = new map()                              O(1)
    SRMapR2S = new map()

 at PR = new set()                                   O(1)
 do MapSP2R = new map()                              O(1)
    PRMapR2P = new map()

 at ROLES.add(r)                                     O(1)
 do for s in SRMapR2S[r]:                            O(SR21*PR21)
      for (op,o) in PRMapR2P[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)

 at SR.add((s,r))                                    O(1)
 do if ROLES.contains(r):                            O(PR21)
      for (op,o) in PRMapR2P[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)
    SRMapR2S[r].add(s)

 at PR.add(((op,o),r))                               O(1)
 do if ROLES.contains(r):                            O(SR21)
      for s in SRMapR2S[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)
    PRMapR2P[r].add((op,o))

 ...//deletion is the same as addition, except
    //without not in conditions and with add replaced by del
```

Figure 3. An invariant rule for a join query.

Profiling for frequency analysis. We describe how to automatically extend any invariant rule to generate instrumentation for profiling the frequencies of queries and updates, which helps justify incremental maintenance of the invariant. The extension has three steps: (1) under the inv-clause, declare a method `inccount`, in a package `invtslog`, that takes two parameters—the location of the query and `null` when a query is matched, and the locations of the corresponding query and the update when an update is matched—and counts the number of executions of each query and of each update for each query, (2) under the inv-clause, insert into the do-clause (creating the do-clause first if it does not exist) a call `invtslog.inccount(`loc(*query*)`, null)`, where meta function loc(p) returns the unique location of the program syntax element p, and (3) under each at-clause, insert into the do-clause (creating the do-clause first if it does not exist) a call `invtslog.inccount(`loc(*query*)`, `loc(*update*)`)`.

For example, the invariant rule in Figure 1 is transformed into the rule in Figure 4.

```
inv  r = s.size()                                    O(|s|)
 de  in package invtslog:
       inccount(query,update):
         ... //increment count of query-update pair
 do  invtslog.inccount(loc(query), null)

 at  s = new set()                                   O(1)
 do  ...//as before                                  O(1)
     invtslog.inccount(loc(query), loc(update))

 at  s.add(x)                                        O(cost(x))
 do  ...//as before                                  O(cost(x))
     invtslog.inccount(loc(query), loc(update))

 at  s.del(x)                                        O(cost(x))
 do  ...//as before                                  O(cost(x))
     invtslog.inccount(loc(query), loc(update))
```

Figure 4. An invariant rule for profiling set size and updates.

Runtime invariant checking and debugging. We describe how to check invariants of the form `myr` = `myquery` at given program points, where `myr` is a program variable, and `myquery` is an instance of a *query* that can be incrementally maintained by an invariant rule. We simply insert `x = myquery` at the given program points, where `x` is a fresh dummy variable, and apply a variant of the rule for incrementally maintaining r = *query*. The variant can be generated automatically: it takes all the clauses for incrementally maintaining r = *query* and adds under the inv-clause an if-clause that equates the query pattern with the query `myquery` and a do-clause that checks whether `myr` equals r at the *query*, and does error handling if the check fails. For example, to check `myr` = `mys.size()`, we can generate the rule in Figure 5, which simply inserts the if-, de-, and do-clauses under the inv-clause, starting with the invariant rule for set size in Figure 1 or Figure 2. Applying such a rule transforms the program to incrementally maintain the result of `myquery` in an instantiated `r` and check that `myr` equals the instantiated `r`. This avoids computing `myquery` from scratch every time the program checks the invariant. It is a significant saving if the query in the invariant is expensive, and the program points to be checked are in a loop, as when checking loop invariants.

```
inv  r = s.size()
 if  s = mys
 de  error(): print 'size computed incorrectly'
 do  if myr != r: error()
 ... //the rest is the same as in the rules for set size
```

Figure 5. An invariant rule for runtime verification of set size.

If assertions are supported in programs, then one can simply insert the assertion `myr = myquery` at the given program points and keep only the if-clause, not the de- and do-clauses, under the inv-clause.

The method for runtime invariant checking can be extended to facilitate debugging, by extending the do-clauses in the rules to insert bookkeeping code that helps determine the sources of invariant violations or other bugs.

Refactoring. As a small example of refactoring, the invariant rule in Figure 6 renames a variable from `old` to `new` if the declaration of `old` is at a specified location, where meta function decl(x) returns the program syntax element that declares x. The renaming respects scoping rules automatically. Conceptually, the rule matches all updates using *update* and does nothing at all of them, since no update affects the invariant. An efficient implementation simply omits matching of updates.

```
inv new = old
 if  loc(decl(old)) = ...//some specific location
 at  update
```

Figure 6. An invariant rule for variable renaming.

Developing and verifying invariant rules. Some rules are easy to write, such as local rewrite rules for various commonly used transformations, but rules for maintaining invariants involving more complicated queries are nontrivial to develop. Even though invariant rules make it easier to express invariant-driven transformations, without systematic methods for deriving invariant rules that are guaranteed to correctly maintain invariants, unverified manually written rules might not preserve invariants.

There are methods to automatically derive large classes of invariant rules [Liu et al. 2006, Rothamel and Liu 2008], including rules for join queries, which are well known to be difficult, and queries over objects, which are even harder because of aliasing between object references. Still, some invariant rules will be developed manually, for example, to capture new data structures.

It is important to verify the correctness of invariant rules, especially ones developed manually. We believe that three features make invariant rules much easier to verify than invariants in programs, even though the exact methods for verification are open for study. First, an invariant rule specifies an invariant with all updates of certain kinds that may affect the invariant and the corresponding maintenance together. Second, an invariant rule may explicitly specify applicability conditions. Third, an invariant rule is usually much smaller than the programs to which it is applied.

We have developed and used invariant rules for a variety of applications. Figure 7 in Section 6 gives examples for which we have used invariant rules for optimization, runtime verification, debugging, refactoring, etc. The rules for optimization by incrementally maintaining queries over objects and sets were developed manually, following a systematic method [Liu et al. 2006, Rothamel and Liu 2008]; such rules are difficult to develop without the systematic method. The method is still being extended but has partly been automated for runtime invariant checking [Gorbovitski et al. 2008a] and query-based debugging [Gorbovitski et al. 2008b]. Other rules were easy to develop manually.

5. Implementation methods

We describe how to apply an individual invariant rule and ensure applicability conditions before giving the overall algorithm.

Application of an invariant rule. An invariant rule applies if (1) a computation in the program matches the $query$ pattern (and more generally, the $computation$ pattern), and the conditions after the inv-clause hold, (2) every update to the parameters of the query matches at least one $update$ pattern, and the conditions after that at-clause hold, and (3) for optimization, the following cost condition holds for each matched update u, where $cost_q$ and $freq_q$, $cost_u$ and $freq_u$, and $mcost_u$ are the cost and frequency for the matched query q, the cost and frequency for update u, and the cost of the maintenance associated with update u, respectively:

$$mcost_u \leq cost_u \text{ or}$$
$$\textstyle\sum_{u \text{ where } mcost_u > cost_u} mcost_u \times freq_u < cost_q \times freq_q$$

If frequency information is not available from analysis or profiling, the second disjunct can safely be ignored.

Transformations for applying the rule are as follows, where all meta variables in the $query$ and $update$ patterns are instantiated according to the matches above.

1. Add declarations associated with the $query$ and each $update$, with each $declaration$ in its respective $scope$ if specified, or in the smallest suitable scope that contains all uses of the declared name otherwise. A *declaration* has no effect if the declaration it specifies already appears in the program.
2. Insert maintenance operations at the $query$ and each $update$, with each $maint$ before, after, or in place of the $query$ or $update$ as specified, or after the $update$ if the position is not specified.
3. Replace each occurrence of the $query$ with $result$.

A rule applies only if it matches a query and all updates in the program that may affect the query; transformations for the matched query and updates are applied together or not at all. It can be proved by induction that application of an invariant-preserving rule preserves the invariant in the program.

Static analyses and dynamic checks. Application of an invariant rule requires nontrivial program analyses, including alias analysis and type analysis that help identify updates to query parameters, and for optimization, analysis of frequencies and costs of the query, updates, and maintenance. We refine the analyses described in [Liu et al. 2005]. Specifically, we use the alias analysis and type analysis in [Gorbovitski et al. 2009], and we use both heuristic complexity analysis and profiling to help determine costs and frequencies. The alias analysis is based on a flow-sensitive analysis [Choi et al. 1993] improved to an optimal running time algorithm [Goyal 2005], and then extended to analyze object-oriented and dynamic features precisely with trace sensitivity, a powerful form of context sensitivity.

The declarative nature of invariant rules allows their applicability conditions to be checked statically whenever possible and dynamically otherwise. In particular, due to aliasing and dynamic features, it may be difficult to statically determine precisely which updates affect the parameters of a query. Our static analysis conservatively identifies and matches all possible updates to the query parameters and, for updates that are possible but not definite, our transformation guards the inserted maintenance code with a runtime check of the statically uncertain conditions. For example, if the query is `s1.size()`, and `s2` may be aliased to `s1` at a call `s2.add(...)`, then maintenance code guarded with `if s2==s1` is inserted at that update.

Reversely, when conditions in the maintenance code in a rule can be evaluated statically, they can be eliminated from the inserted maintenance code. For example, for using the invariant rule in Figure 1, when the conditions about set membership can be checked statically, we eliminate them from the inserted maintenance code. In the best case, all the membership tests can be eliminated, yielding the same effect as using the invariant rule in Figure 2. This optimization method allows our framework to obtain the most efficient implementation possible with any given static analyses.

Thus, implementations of invariant rules can make trade-offs between efficiency of the analysis and transformation and efficiency of the transformed program.

Overall transformation algorithm and complexity. The overall algorithm repeatedly applies rules to queries in the given program until no rule applies. The given program is first analyzed for queries, updates, and other information, and then re-analyzed after each rule application. Our system caches analysis results to reduce the cost of repeated analysis.

For efficiency, a rule r_1 is considered before a rule r_2 if applying r_1 can make r_2 applicable, i.e., the maintenance pattern in r_1 contains parts that match the query and update patterns in r_2, and not vice versa. Other than this heuristic, our implementation applies rules in the order they are encountered.

Incremental re-analysis after applying each rule is implemented by logging each piece of analysis results in a custom-written database as the analysis proceeds, and reusing valid analysis results after the program is changed. Valid analysis results are those obtained before the first changed node in the program and those at program nodes not reachable from the changed nodes. The database supports efficient lookups and insertions. Each insertion has a timestamp. A log-structured merge tree is used to support efficient lookups using keys and time ranges. This data structure is implemented on two storage tiers: memory and disk. Eviction of entries from memory to disk uses an LRU algorithm. Additionally, the last analysis results for a procedure or method at each call node are cached. Every time a procedure or method is to be analyzed at a call node, the current analysis results for all parameters (including global variables) of the call are compared with the last analysis results for these parameters, and if they are all the same, then the last analysis results for the procedure or method at the call node are reused.

Altogether, at most $O(M)$ rule applications occur, where M is the total number of matched queries and subqueries in the program being transformed. Each application requires pattern matching, analysis or re-analysis, and transformation of the program. The most expensive step in our implementation is the alias analysis. Caching and reuse of analysis results is critical and yields up to a 100-fold speedup in rule applications.

6. Applications and experiments

We have developed InvTS/py and InvTS/c, two implementations of InvTL, the invariant rule language described above, for applying invariant-driven transformations for Python and GCC C, respectively. Both systems are built on a common base, called InvTS.

We chose Python and GCC C for several reasons. Python is particularly well suited for expressing complex queries over objects and sets, which are commonly used in higher-level, clear and modular specifications. GCC C is primarily used for efficient implementations of system software, nearly the opposite of Python, and for which there is significant need for program monitoring, debugging, etc. We believe that an implementation for Java would require about the same effort as for Python and much less than for GCC C.

InvTS base consists of about 30000 lines of Python, with about 2500 lines for the rule application engine and 27500 lines for the parser generator and other libraries. InvTS/py consists of another about 16500 lines of Python, with about 3000 lines for the Python frontend and 13000 lines for Python program analysis. InvTS/c consists of about 4000 lines of Python and 9000 lines of C. Even though InvTS/c has fewer lines of code than InvTS/py, it is significantly harder to implement, because C is lower-level. InvTS/py also includes an analysis visualizer of about 6000 lines.

InvTS/py uses and extends PyPy, a Python implementation in Python, primarily for type analysis and for part of visualization. InvTS/py uses a precise may-alias analysis and does incremental re-analysis, as described above.

InvTS/c uses a plugin architecture for GCC 4.2 [Callanan et al. 2007] that provides access to all information available to GCC during its GIMPLE optimization phase. Powerful type analysis and an optimal-time interprocedural flow-sensitive may-alias analysis [Gorbovitski et al. 2009] are implemented, and are made incremental during the transformations to improve efficiency. The alias analysis results for C are still very imprecise, so some examples require many runtime aliasing checks to determine whether updates are to query parameters.

6.1 Applications

We used InvTL and InvTS for a wide range of applications. Figure 7 summarizes 24 examples grouped by whether the purpose is optimization, runtime verification, debugging, other instrumentation, refactoring, or other transformations.

Use	Application	Program	Lang
O	Core RBAC	core RBAC spec	py
	Constrained RBAC	constrained RBAC spec	py
	Graph Reachability	test program	py
	Join Query	test program	py
	Wireless Protocol	test program	py
	Set Size Demo	test program	py
V	SMB Valid Ticket	pysmb	py
	SMB Repeated Auth	pysmb	py
	BitTorrent Peer No Dup	BitTorrent Peer	py
	BitTorrent Peer No Mod	BitTorrent Peer	py
	BitTorrent No Mismatch	BitTorrent Mainline	py
	InvTS No Shared Child	InvTS	py
	InvTS Own Child	InvTS	py
D	DOM Valid Parent	lxml benchmarks	py
	DOM No Shared Child	lxml benchmarks	py
	DOM Exception Cause	lxml benchmarks	py
	FTP Client	nftp	py
I	File Access Profiling	test program	py
	Reference Counting	test program	py
	Memory Coverage	ViM 7.0	c
R	InvTS Refactoring	file Rule.py in InvTS	py
	Variable Renaming	ViM 7.0	c
T	InvTS/py Test Suite	test program suite	py
	InvTS/c Test Suite	test program suite	c

O: optimization. V: runtime verification. D: debugging.
I: instrumentation. R: refactoring. T: other transformation.

Figure 7. Example applications.

Optimization. Our main applications for optimization are generating efficient implementations from clear specifications for the ANSI standard for role-based access control (RBAC) [ANSI INCITS 2004]. Core RBAC defines core functionalities on permissions, users, sessions, roles, and relations among these sets; its efficient implementation was studied previously [Liu et al. 2006]. Constrained RBAC extends Core RBAC with static and dynamic separation of duty constraints. Core RBAC contains only flat queries; Constrained RBAC adds nested queries. Graph Reachability, Join Query, and Wireless Protocol are small but nontrivial examples for generating efficient implementations from clear specifications; these examples are from [Liu et al. 2005]. Set Size Demo uses the example rule in Figure 1. The largest and most complex application, Constrained RBAC, is new.

We also generated rules and did runtime invariant checking and frequency analysis for Core RBAC and other examples; the results are not reported separately because they are similar to those reported.

Runtime verification. pysmb is an SMB client in Python. SMB Valid Ticket checks that all packets sent are authenticated. SMB Repeated Auth checks that authentication does not occur more often than necessary. BitTorrent is a peer-to-peer file distribution protocol. BitTorrent Peer is the core functionality of BitTorrent Mainline, containing only the code for running an instance of a BitTorrent peer, without various interfaces, internationalization, DHT (the distributed hash table feature), and tracker. BitTorrent Peer No Dup works on BitTorrent Peer and checks that the same data is not received from multiple sources. BitTorrent Peer No Mod checks that packets are not modified in transit. BitTorrent No Mismatch works on BitTorrent Mainline and checks that all packets sent are received and all packets received are sent. InvTS No Shared Child checks that no two parents refer to the same child in an AST in InvTS. InvTS No Own Child checks that no node is a child of itself. The smaller applications are from [Gorbovitski et al. 2008a]. The largest application, on BitTorrent Mainline, is new.

Example Application	#inv	#at	size	max	min	#query	#update	before	after	time
Core RBAC	14	78	613	97	9	24	248	201	798	23
Constrained RBAC	21	114	1137	124	9	33	752	381	2183	34
Graph Reachability	1	2	14	14	14	1	1	60	83	14
Join Query	1	2	23	23	23	1	3	69	113	13
Wireless Protocol	1	3	19	19	19	1	3	66	148	14
Set Size Demo	1	3	15	15	15	1	3	5	10	12
SMB Valid Ticket	1	5	134	134	134	1	13	1100	1251	33
SMB Repeated Auth	1	8	250	250	250	1	7	1100	1203	38
BitTorrent Peer No Dup	1	4	132	132	132	1	31	9871	11835	98
BitTorrent Peer No Mod	1	10	332	332	332	1	90	9871	13468	109
BitTorrent No Mismatch	1	8	174	174	174	2	19	29450	29707	391
InvTS No Shared Child	1	2	46	46	46	1	413	12510	21651	312
InvTS No Own Child	1	2	53	53	53	1	412	12510	22513	387
DOM Valid Parent	1	4	96	96	96	1	37	1193	1344	43
DOM No Shared Child	1	5	141	141	141	1	43	1193	1472	38
DOM Exception Cause	1	8	271	271	271	1	81	1193	2135	53
FTP Client	1	12	303	303	303	1	27	891	1184	31
File Access Profiling	1	3	43	43	43	2	5	163	209	18
Reference Counting	1	3	28	28	28	105	850	1379	2349	32
InvTS Refactoring	6	0	132	39	3	19	0	3931	6118	21
InvTS/py Test Suite	12	13	160	17	3	27	117	630	1674	49
Memory Coverage	2	2	59	30	29	4590	1900	305969	319617	1168
Variable Renaming	4	0	12	3	3	13	0	305969	305969	1071
InvTS/c Test Suite	12	7	81	13	3	12	19	190	351	183

#inv: number of rules used, i.e., number of inv-clauses	#query: number of queries matched in program
#at: total number of update patterns, i.e., number of at-clauses	#update: number of updates handled in program
size: size of rules, in lines of code	before: program size before transformation (lines of code)
max: maximum size of a rule (lines of code)	after: program size after transformation (lines of code)
min: minimum size of a rule (lines of code)	time: running time of InvTS on transforming the program (seconds)

Figure 8. Results of applications of invariant rules.

Debugging. lxml is a Python XML library; for its benchmark programs, Dom Valid Parent checks that each element in the XML DOM tree has a parent whose child field refers back to the element, and Dom No Shared Child checks that no two XML elements have a common child element. Dom Exception Cause detects sources of an index-out-of-bound exception. FTP Client finds the location where an `ls` command is executed when a `cwd` command is pending; this detected a real bug in an FTP client program, nftp. These examples are from [Gorbovitski et al. 2008b].

Other instrumentation. File Access Profiling looks for a specific file access pattern. It demonstrates that one can easily express access patterns of interest in InvTL, much like in an aspect-oriented programming system. Reference Counting emulates a reference-counting garbage collector, by incrementally maintaining reference counts. It uses static analysis in InvTS to avoid keeping reference counts for values of primitive types, and thus avoid severe performance degradation usually associated with explicit reference counting. Memory Coverage instruments ViM, a text editor, to intercept all calls to `malloc` in order to monitor memory access patterns. It is challenging because `malloc` may be called indirectly through function pointers aliased to `malloc`.

Refactoring. InvTS Refactoring was a real experience in the implementation of InvTS. The goal was to rewrite InvTS, factoring out the Python language module from InvTS core to form InvTS/py. The rules mainly did variable and class renaming, and method extraction, and turned a tedious manual task into an easy one. Variable Renaming is for C and involves small rules that rename only local, only global, only static, or only global static variables. These rules are easy to write because InvTL takes scope of variables into account. Implementing such renaming using, say, Perl scripts, would be significantly harder and very error-prone.

Other transformations. InvTS/py and InvTS/c Test Suites consist of miscellaneous transformations where transformed programs are checked automatically against known results for testing purposes. The transformations insert code that prints certain strings when certain code patterns are matched. These strings are then counted for simple tests.

6.2 Experiments

We ran InvTS/py and InvTS/c on a large sample of rules and programs. Extensive blackbox testing was performed to confirm correctness of the transformed programs. For optimization applications, we obtained drastic performance improvements for all expensive operations in the new, larger example of Constrained RBAC, similar as for all the previous examples [Liu et al. 2005, 2006]. For runtime verification, debugging, and other instrumentation applications, we observed low runtime overhead (14-92%) for most examples [Gorbovitski et al. 2008a,b], except for a factor of 23 for Memory Coverage for ViM, because the imprecision of alias analysis caused extensive insertion of runtime checks that disabled otherwise possible optimizations. For all refactoring and testing applications, the transformed programs were exactly as desired.

We discuss measurements that help show the effectiveness and efficiency of InvTL and InvTS and the results of the two new, largest applications so far. The first set of measurements is on the size of invariant rules and size increase in the transformed Python and C programs, as well as the running time of InvTS. The other measurements are on the results of using InvTS for optimization of Constrained RBAC and for improvements of BitTorrent when there is a slight increase in communication errors. The transformations were run under Windows Vista 64-bit Edition on a quad-core Core 2 Duo Q6600 3.2 GHz with 8 GB of memory, of which around 6.3GB was free when running our programs, and were run under Python 2.5.1.

Figure 8 reports information about the rules and the transformed programs, as well as running times of the transformations. For most examples, the size of the rules is much smaller than the increase in size in the transformed program. Core RBAC and a few others are exceptions because each rule matched relatively few queries and updates, only code in de- and do-clauses, not inv-, if-, and at-clauses, are inserted in the transformed program, and some generated rules contain at-clauses that do not occur in the program. Note, however, that invariant rules can be reused; in fact, all of the rules for Core RBAC are reused for Constrained RBAC. More importantly, each rule specifies in one place how to maintain an invariant under all possible updates, rather than having the maintenance code scattered throughout the program.

For optimization applications, we obtained drastic performance improvements for all the examples. Measurements for several of the examples were reported in [Liu et al. 2005], and more detailed description and measurements for Core RBAC were reported in [Liu et al. 2006]. Similar speedups are obtained for Constrained RBAC operations. Figure 9 compares the performance of a fully incrementalized implementation of Constrained RBAC with an implementation partially incrementalized on only expensive queries in Core RBAC, the latter as done previously [Liu et al. 2006]. The measurements were taken under the same set up as for running the transformations. The graph shows the running times of `CreateSession` and `AddActiveRole`, two important operations that involve checking of constraints. Clearly, the curves do not increase for fully incrementalized implementations but increase linearly with the number of sessions for partially incrementalized implementations; fully incrementalized `CreateSession` even decreases in running time because its complexity has the number of roles per session as a factor.

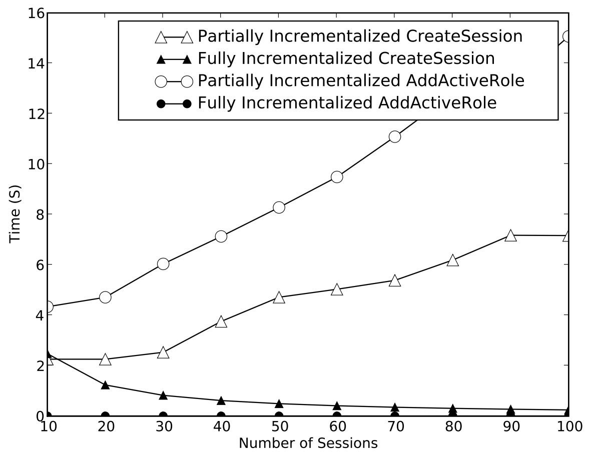

Figure 9. Running times, in seconds, of RBAC operations as number of sessions increases, with 100 roles, 100,000 repetitions.

For runtime invariant checking, we instrumented a BitTorrent implementation and checked it for potential errors. One of the invariants we checked, for BitTorrent Peer No Mod, is that a packet sent from one peer is received by another peer without a change in the payload. We first experimented with centralized checking [Gorbovitski et al. 2008a], by creating a server to which peers send summaries (containing hashes of the payloads) of the packets they send and receive. These summaries are collected on the server. A query is used to compute the set of violations, i.e., the set of packets received that have a different hash of the payload than the corresponding packet sent. The invariant we check is that this set is empty, and all violations are reported. Naive checking of this invariant has a formidable cost because it uses a join query over all the summaries sent and received. We use an invariant rule to specify how to incrementally maintain the query result as each summary is received, reporting violations incrementally too. The incremental maintenance in the invariant rule is complex and is automatically generated. Our new experiments, for BitTorrent No Mismatch, use distributed checking, by having each peer send a hash of each packet it sent or received to every peer in the peer horizon (the set of peers known to a given peer, usually much smaller than the peer swarm, the set of all peers involved in sending a file). This is much more scalable and more fault-tolerant. Again, we need invariant rules to make the checking efficient. Furthermore, we used additional invariant rules to instrument BitTorrent to discredit senders and receivers that cause violations. This has some dramatic results. For example, we experimented with transferring a 1GB file from a server to 29 peers, over 10MBit links.

1. Ideally, if the server is not saturated, it should take about 1100 seconds (TCP overhead is about 10%), with a total bandwidth consumption of about 29GB. However, without BitTorrent, usually the server is saturated; in our experiment, it can take 50-60 times as long.
2. Using BitTorrent with no error, it takes 2393 seconds with a total bandwidth of 31.9GB; that is, just over 2 times as long, and a small percentage increase in total bandwidth.
3. Using BitTorrent with 10% error rate on 3 of the 29 peers, it takes 4856 seconds with a total bandwidth of 91.6GB; that is, 4+ times as long, and a factor of 3+ in total bandwidth.
4. Using BitTorrent with the same error rate as in case 3, but using our discrediting scheme, it takes 2975 seconds with a total bandwidth of 34.2GB; that is, 2-3 times as long, and another small percentage increase in total bandwidth.

BitTorrent did so badly in case 3 because its error handling is done at the chunk level, much coarser-grain than the packet level at which we check invariants and trigger discrediting. BitTorrent usually works well because the error rate is usually extremely small, but in the case of slightly more errors (in our examples, 10% error in 10% of the peers, so 1% error total), from malicious attack or otherwise, its performance becomes significantly worse.

7. Related work and conclusion

There is a vast amount of research on program transformation languages and systems, as described in a number of surveys, e.g., [Partsch and Steinbrüggen 1983, Visser 2005], and collected on the web [ProgramTransformationOrg]. Eminent systems include, e.g., APTS [Paige 1994], KIDS [Smith 1990], CIP [Bauer et al. 1989], and Stratego [Bravenboer et al. 2008].

Compared to previous frameworks and systems, the most important and unique characteristics of invariant rules are the declarative specification of coordinated transformations explicitly driven by invariants and the generality of the language. With the exception of APTS [Paige 1994], previous frameworks and systems use rewrite rules together with rewriting procedures or strategies [Visser 2005], where invariants, updates, and cost considerations are programmed more implicitly. These systems do give programmers more control over the transformation process, which is similar to the advantage of procedural languages compared to declarative languages. Our InvTS implementation overcomes efficiency problems that are typical for declarative languages by using efficient analyses and caching and reusing analysis results.

APTS [Paige 1994] supports specification of transformations explicitly around invariants, for finite differencing [Paige and Koenig 1982], but it applies only to a simple language with set expressions and statements in straight-line code. InvTL is much more general and powerful in specifying transformations of full-fledged programming languages, including object-oriented languages. InvTS has been applied to many applications of different sizes, while APTS was applied only to a few small SETL programs.

Many program query languages and tools have been developed, e.g., ASTLOG [Crew 1997], Partiqle [Goldsmith et al. 2005], and PQL [Martin et al. 2005]. Some of these languages, such as PQL, allow matches, analyses, and replacements throughout the program, much like in program transformation systems. However, these systems do not provide automatic detection of updates to invariants.

Aspect-oriented programming (AOP) [Kiczales et al. 1997, 2001] also allows code for cross-cutting concerns, such as debugging, to be specified separately and inserted automatically at a set of matched program points. Connections between AOP and invariants are studied specially [Smith 2008, 2007]. Compared with existing AOP languages, InvTL has an explicit definition of invariant-preserving rules, to facilitate formal verification; it allows explicit specification of costs, to assist effective optimization; and it provides powerful static analysis, especially for automatically detecting updates, to coordinate transformations at queries and updates and minimize runtime overhead.

A limited version of InvTL has been used for incrementalization [Liu et al. 2005], and variants of it have been used for runtime invariant checking [Gorbovitski et al. 2008a] and query-based debugging [Gorbovitski et al. 2008b]. Besides their different applications, contributions of those works to the general framework are analysis of updates to query parameters [Liu et al. 2005], automatic generation of incremental maintenance code for a class of queries [Gorbovitski et al. 2008a], and efficient analyses [Gorbovitski et al. 2008b]. The current paper gives the first complete and precise definition of the rule language, with more powerful clauses for invariants, conditions, declarations, and maintenance than before. It also studies, for the first time, connections to invariants among a wider range of applications, and it explains the overall implementation that puts all the analyses and transformations together. Furthermore, it evaluates the effectiveness and efficiency of the implementations with applications and experiments on a set of 24 programs, including the new largest example for optimization and the new largest example for runtime invariant checking.

We believe that invariant rules capture the design knowledge underlying a more general framework for program development. Consider the three core components of an invariant rule: the invariant, updates, and maintenance. Invariant maintenance for program design and optimization corresponds to being given the invariant and updates and wanting the maintenance. Invariant discovery for program understanding and reverse engineering corresponds to being given the updates and maintenance and wanting the invariant. Invariant checking for validation and verification corresponds to being given all three of the invariant, updates, and maintenance and wanting to check.

Future work includes methods and tools for verification of invariant rules, possible enrichment of the invariant rule language, and additional usages of invariant rules.

References

ANSI INCITS. Role-Based Access Control. ANSI INCITS 359-2004, American National Standards Institute, International Committee for Information Technology Standards, Feb. 2004.

Friedrich Ludwig Bauer, Bernhard Möller, Helmut Partsch, and Peter Pepper. Formal program construction by transformations—Computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, Feb. 1989.

Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

Sean Callanan, Daniel J. Dean, and Erez Zadok. Extending GCC with modular GIMPLE optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, July 2007.

Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, 1993.

Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the Conference on Domain-Specific Languages*, pages 229–242, Oct. 1997.

Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 385–402, 2005.

Michael Gorbovitski, Tom Rothamel, Yanhong A. Liu, and Scott D. Stoller. Efficient runtime invariant checking: A framework and case study. In *Proceedings of the 6th Sixth International Workshop on Dynamic Analysis*, pages 43–49, July 2008a.

Michael Gorbovitski, K. Tuncay Tekle, Tom Rothamel, Scott D. Stoller, and Yanhong A. Liu. Analysis and transformations for efficient query-based debugging. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 174–183, Sept. 2008b.

Michael Gorbovitski, Tuncay Tekle, and Yanhong A. Liu. Assessing alias analysis for object-oriented and dynamic languages. Technical Report DAR 09-44, Computer Science Department, SUNY Stony Brook, 2009.

Deepak Goyal. Transformational derivation of an improved alias analysis algorithm. *Higher-Order and Symbolic Computation*, 18(1-2):15–49, June 2005.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, June 2001.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11th Europeen Conference on Object-Oriented Programming*, pages 220–242, 1997.

Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni E. Liu. Incrementalization across object abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–486, Oct. 2005.

Yanhong A. Liu, Chen Wang, Michael Gorbovitski, Tom Rothamel, Yongxi Cheng, Yingchao Zhao, and Jing Zhang. Core role-based access control: Efficient implementations by transformations. In *Proceedings of the ACM SIGPLAN 2006 Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 112–120, Jan. 2006.

Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, Oct. 2005.

Robert Paige. Viewing a program transformation system at work. In *Proceedings of Joint 6th International Conference on Programming Languages: Implementations, Logics and Programs and 4th International Conference on Algebraic and Logic Programming*, 1994.

Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4 (3):402–454, July 1982.

Helmut Partsch and Ralf Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, Sept. 1983.

ProgramTransformationOrg. The Program Transformation Wiki. http://www.program-transformation.org.

Tom Rothamel and Yanhong A. Liu. Generating incremental implementations of object-set queries. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 55–66, Oct. 2008.

Douglas R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

Douglas R. Smith. Requirement enforcement by transformation automata. In *Proceedings of the 6th Workshop on Foundations of Aspect-Oriented Languages*, pages 5–14, 2007.

Douglas R. Smith. Aspects as invariants. In O. Danvy, H. Mairson, F Henglein, and A. Pettorossi, editors, *Automatic Program Development: A Tribute to Robert Paige*, pages 270–286. Springer, 2008.

Eelco Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.