ISSN 1342-2812

Research Reports on Mathematical and Computing Sciences

Generating Java Compiler Optimizers Using Bidirectional CTL

Ling Fang and Masataka Sassa

Nov 2006, C–230

Department of
Mathematical and
Computing Sciences
Tokyo Institute of Technology

SERIES C: Computer Science

Generating Java Compiler Optimizers Using Bidirectional CTL

Ling Fang and Masataka Sassa
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
{fang3, sassa}@is.titech.ac.jp

There have been several research works that analyze and optimize programs using temporal logic. However, no evaluation of optimization time or execution time of these implementations has been done for any real programming language.

In this paper, we present a system that generates a Java optimizer from specifications in temporal logic. The specification is simpler, and the generated optimizers run more efficiently than previously reported work.

We implemented a new model checker for a bidirectional CTL (computation tree logic), a branching temporal logic. The model checker can check future and past temporal CTL operators symmetrically without any conversion. We also present a new specification language based on the bidirectional CTL that can express typical optimization rules very naturally. By adding rewriting conditions and handling of temporary variables, the system can perform optimization of Java programs.

So far, a compiler optimizer using temporal logic was assumed to be impractical, because it consumes too much time. However, with our method, the generated Java compiler optimizer can compile all seven of the SPECjvm98 benchmarks with a compile time from 15 seconds to 5 minutes.

We also gained insights into improving existing techniques for decreasing the compilation time and expertise in specifying compiler optimizations.

1 INTRODUCTION

In compiler design, code optimization is one of the most important passes, improving execution speed and spatial efficiency of the target code [1] [3].

Current optimizers are almost always implemented by some kind of programming language. However, the approach of implementing optimizers by CTL (computational tree logic) [8], a branching temporal logic, has attracted interest in recent years. This approach has two main advantages.

- The transformations are easier to write and prototype. They can be achieved by writing several lines of specification language rather than many hundreds of lines of code. Also, they can be written by developers as well as compiler experts.

- The transformations can be formally analyzed because they are more simply expressed.

Compiler optimization by CTL can concisely express a lot of classic optimization by using the following specification language (conditional rewrite rule), which is denoted as follows.

$$I \Longrightarrow I' \text{ if } \phi$$

For example, the rule describing the dead code elimination will be as follows.

$$x := e \Longrightarrow skip$$
$$\text{if } AX((AG\neg use(v)) \vee A\neg use(v)\ U\ def(v))$$

In our work, we adopt the bidirectional CTL [8] where past temporal operators can be used symmetrically with future temporal operators. We implemented a new model checker that can check future and past temporal operators symmetrically. We also propose a new specification language for writing compiler optimizers that can express optimization rules very naturally. Compiler optimization using temporal logic was assumed not to be useful for real-world programming languages because it takes hours for compilation to be done using the model checker. However, we have improved previous techniques and have implemented a new Java compiler optimizer that can act very efficiently, given the specification of optimization.

Compiler optimization can be described naturally and concisely by using bidirectional CTL [8].

CTL-FV [10] is a kind of bidirectional CTL proposed by Lacey in which free variables are introduced. Many traditional program optimization conditions can be described by CTL-FV. We adopted CTL-FV as a base. However, as described later, Lacey's implementation is not realistic because it cannot handle all classical optimizations, and binding of free variables is time consuming.

Ban's methods [2] can also handle past temporal operators by using NCTL [12]. This method uses 12 conversion rules to convert temporal expressions including past temporal operators into expressions with only future temporal operators. However, the conversion process consumes a lot of time, and the model checking is very time consuming because expressions become longer by conversion. Moreover, there is a limitation with NCTL in that past temporal operators cannot be written freely.

We implemented a model checker that directly handles bidirectional CTL. Past temporal operators are checked symmetrically with future temporal operators. Because the process time used to convert to NCTL [12] or μ-calculation is not incurred and the formulas never become more complex because of the elimination of past temporal operators, our model checker is very efficient.

As for the description ability of optimization, Ban's [2] and Yamaoka's research [21] can rewrite only one place corresponding to one condition at a time. Therefore, their methods cannot handle complex optimizations such as partial redundancy elimination that rewrite several places at the same time using several conditions. Their work cannot process edges, either.

Before model checking, free variables must be bound. Therefore when there are a lot of free variables in the conditional expression, processing time becomes unrealistic.

Moreover, using the node number of the Kripke structure as in Lacey's specification is a common drawback of all previous work. This results in a computation time explosion in most cases. Another drawback of previous work is that the optimization condition cannot be described naturally when the condition refers to several nodes.

The description in our research does not write the node number of the Kripke structure. What the model checker calculates is not the instruction of a specific number but sets of instructions that satisfy the same condition. Therefore, it becomes very easy to describe a complex rewrite rule that rewrites many instructions. Moreover, it can improve efficiency because the free variables corresponding to the node number of the Kripke structure are omitted.

By adding some processing for real-world language features, such as extensions to handle rewrite rules and temporary variables, we achieved a typical optimizer for a Java language compiler, the performance of which is now close to the optimizers using traditional algorithms.

So far, optimizers with temporal logic have been assumed to be impractical because of the processing time. In our research, the seven SPECjvm98 benchmarks were able to be optimized in a time ranging from 15 seconds to 5 minutes using the aforementioned improvement.

Additionally, our optimizer can perform some optimizations that cannot be done with previous work.

We think that our implementation is the first realistic Java compiler optimizer with temporal logic. Insights into existing problems, and techniques for shortening the optimization time and the recommended style of its specification were acquired.

2 Bidirectional CTL

Bidirectional CTL [8] is one of the temporal logics. In this section, we introduce bidirectional CTL and how it is used in program analysis and transformation.

Bidirectional CTL Bidirectional CTL was first proposed in [8]. Past temporal operators are introduced symmetrically with future temporal operators.

Each point of the temporal structure has two trees, one for the future and the other for the past. Bidirectional CTL has past temporal operators $\overleftarrow{A}$ and $\overleftarrow{E}$ as well as the usual quantifiers A and E, made by reversing A and E.

Because past temporal operators can be written and verified symmetrically with future temporal operators without any limitation, the conciseness and power of the conditional part, and the efficiency of its implementation using bidirectional CTL, are excellent.

Syntax The syntax of bidirectional CTL is:

$$\begin{aligned} \textit{bi-directional CTL formulas} \ni \phi \quad ::= & \ \alpha \mid \neg\phi \mid \phi_1 \wedge \phi_2 \\ & \mid EX\phi \mid E\phi_1 U \phi_2 \\ & \mid \overleftarrow{E}X\phi \mid \overleftarrow{E}\phi_1 U \phi_2 \end{aligned}$$

Other combinations like $EF\phi$, $EG\phi$, $AF\phi$, $\overleftarrow{A}X\phi$, ... can be converted into a formula that uses only the combination of the syntax rules above.

Semantics The semantics of bidirectional CTL is given on the Kripke structure.

A Kripke structure K is a triple (S, R, L). S is a set of states, $R \subseteq S \times S$ is the transition relation, and $L : S \to 2^{Pred}$ is a function that maps each state to a set of predicates that are true for that state.

A path from s_0 in K is the (finite or infinite) sequence of states $\pi = (s_0, s_1, \ldots)$ such that $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. A backward path from s_0 is a sequence such that $\forall i \geq 0 : (s_{i+1}, s_i) \in R$.

$K, s \models \phi$ denotes that the value of logical formula ϕ is true in state s in a Kripke structure K. K can be omitted if it is obvious. Relation $\models$ is defined as follows.

$s \models \alpha$ iff $\alpha \in L(s)$

$s \models \neg\phi$ iff not $s \models \phi$

$s \models \phi_1 \wedge \phi_2$ iff $s \models \phi_1$ and $s \models \phi_2$

$s \models EX\phi$ iff $\exists s'$; sRs' and $s' \models \phi$

$s \models \overleftarrow{E}X\phi$ iff $\exists s'$; $s'Rs$ and $s' \models \phi$

$s \models E\phi_1 U\phi_2$ iff a path $s_0 s_1 \ldots (s_0 = s)$ starting from s exists, and $\exists i \geq 0$; $s_i \models \phi_2$ and $0 \leq \forall j < i$; $s_j \models \phi_1$

$s \models \overleftarrow{E}\phi_1 U\phi_2$ iff a backward path $s_0 s_{-1} \ldots (s_0 = s)$ starting from s exists, and $\exists i \leq 0$; $s_i \models \phi_2$ and $i < \forall j \leq 0$; $s_j \models \phi_1$

A bidirectional CTL has two tree structures. One is a CTL tree that is expanded from the forward transition relation of the Kripke structure, and the other one is a $\overleftarrow{CTL}$ tree that is expanded from the backward transition relation of the Kripke structure.

The left-hand side of Figure 1 shows a finite Kripke structure. The bidirectional CTL (infinite) tree expanded from it is shown on the right-hand side of the figure. For starting state S_0, the CTL tree is shown in solid lines, and the $\overleftarrow{CTL}$ tree is shown in dotted lines.

How to use Bidirectional CTL for the purposes of program analysis and transformation The task of program optimization can be divided into "where" and "what" stages of the transformation are to be performed. As for the judgment for "where" the transformations should be performed, temporal logic is very natural because different states of the Kripke model made from the program can hold properties that are satisfied for each state. Model checking can be done to check these properties for the Kripke structure. Once the optimizer has decided where to apply a particular transformation, rewrite rules that specify what to do with the instructions can be applied.

3 Control Flow Model

Syntax of program Here we assume a simple language with no procedure.

$\pi = read\ X; I_1; I_2; \ldots I_{m-1}; write\ Y$

Here, $I_1; I_2; \ldots$ are instructions. Including the first read instruction and the last write instruction, the instructions are labeled by $n \in Node_\pi = \{0, 1, 2, \ldots, m\}$.

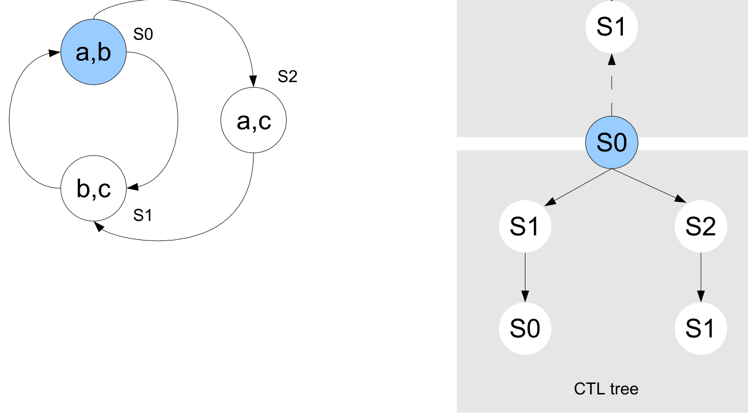

Figure 1: Kripke structure (left) and its bidirectional CTL (infinite) trees (right) for S0

The BNF of the instructions is as follows.
$I ::= skip \mid X := E \mid if\ X\ goto\ n\ else\ n\ \mid\ goto\ n$
$E ::= X \mid E\ O\ E$
$O ::= + \mid - \mid * \mid / \mid \dots$
$X ::= variable$
$n ::= 1 \mid 2 \mid 3 \mid \dots \mid m$

We define the control flow model as the Kripke structure of the program.

Control Flow Model The control flow model of code π is defined as a triple $M(\pi) = (Nodes_\pi, \rightarrow_\pi, L_\pi)$. $Nodes_\pi$ is the set of labels of instructions. The relation $\rightarrow$ is defined as follows.

$n_1 \rightarrow_\pi n_2$ iff
$(I_{n1} \in X := E, skip, read\ X) \wedge n_2 = n_1 + 1$
$\vee(I_{n_1} = goto\ n \wedge n_2 = n)$
$\vee(I_{n_1} = if\ X\ goto\ n\ else\ n' \wedge (n_2 = n \vee n_2 = n'))$
$\vee(I_{n_1} = write\ Y \wedge n_2 = n_1)$

$L_\pi(n)$ is defined as follows for $n \in Nodes_\pi$.

$$
\begin{aligned}
L_\pi(n) = \{stmt(I_n)\} \\
\bigcup \{def(X) \mid I_n\ is\ of\ the\ form\ X := E\ or\ read\ X\} \\
\bigcup \{use(X) \mid I_n\ is\ of\ the\ form\ Y := E\ with\ X\ in\ E, \\
I_n = if\ X\ goto\ n\ else\ n', or \\
I_n = write\ X\} \\
\bigcup \{trans(E) \mid E\ is\ an\ expression\ in\ \pi \\
and\ for\ all\ X\ in\ vars(E), \\
I_n\ is\ not\ of\ the\ form : X := E'\ or\ read\ X\}
\end{aligned}
$$

$\bigcup \{entry(n) \mid n\ is\ an\ entry\ of\ a\ program\}$
$\bigcup \{exit(n) \mid n\ is\ an\ exit\ of\ a\ program\}$

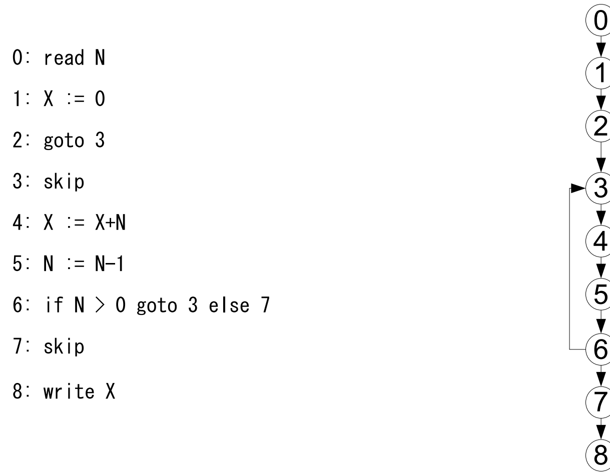

Figure 2: Example of code and its control flow model (L(n) is omitted)

The $L_{\pi(n)}$ corresponding to the code and control flow model shown in figure 2 is as follows.

$$
\begin{array}{rcl}
L(0) & = & \{stmt(readN), def(N), trans(N-1)\ldots\} \\
L(1) & = & \{stmt(X := 0), def(X), trans(N)\ldots\} \\
 & \ldots & \\
L(8) & = & \{stmt(writeX), use(X)\ldots\}
\end{array}
$$

4 Bidirectional Model Checker

This section describes the bidirectional model checker implemented in our work.

The bidirectional model checker is an extension of model checkers used in previous work. Checking the future temporal logic operations is calculated from the CTL tree by the algorithm described in Ban's work [2]; however, checking past temporal logic operations is calculated from a $\overleftarrow{CTL}$ tree in a symmetric fashion to that of the future operators by simply reversing the direction.

Analysis of Bidirectional CTL Formula A bidirectional CTL formula can be expressed by a tree structure, which we call the CTL syntax tree (note that it is different from the CTL tree). The leaves of the tree are atomic predicates.

Figure 3 (right) shows the syntax tree of the CTL formula of figure 3 (left). Table 1 shows the partial formulas corresponding to each node of the bidirectional CTL syntax tree.

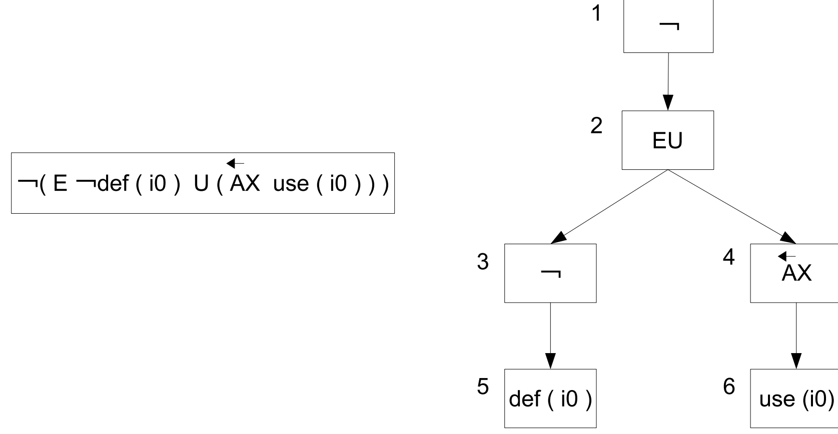

Figure 3: Bidirectional CTL syntax tree

node no.	node op	child node	corresponding partial formula
1	$\neg$	2	$\neg(E\neg def(i_0)U\overleftarrow{A}X(use(i_0)))$
2	EU	3 4	$E\neg def(i_0)U\overleftarrow{A}X(use(i_0))$
3	$\neg$	5	$\neg def(i_0)$
4	$\overleftarrow{A}X$	6	$\overleftarrow{A}X(use(i_0))$
5	ap	$def(i_0)$	$def(i_0)$
6	ap	$use(i_0)$	$use(i_0)$

Table 1: CTL syntax tree nodes and partial formulas of Fig. 3

Model Checking of Bidirectional CTL Each partial formula ϕ_n should be calculated to know whether it is satisfied at each state s. Namely, if we denote the truth value of partial formula ϕ_n at state s_i by $label(\phi_n, s_i)$, we calculate the truth value of these labels as follows.

$$label(\phi_n, s_i) = true \text{ iff } s_i \models \phi_n$$

For that purpose, for each state s of the Kripke structure and for each ϕ_n, a bottom-up calculation is done from the leaves to the root of the CTL syntax tree.

Because the results of the model checking will be used in the following rewriting process, the results needed in the rewriting process are stored in a data structure during the model checking. This data structure is a set corresponding to the nodes of the bidirectional CTL syntax tree.

Computational Complexity of Model Checking Let $n1$ be the number of instructions , which represents the size of the program, and let $n2$ be the number of nodes of a bidirectional CTL syntax tree, which represents the size of the CTL formula. Because model checking calculates truth values at every state of every node of the CTL syntax tree, the computational complexity of model checking is $O(n1 \times n2)$. It is proportional to the size of the program and the bidirectional CTL syntax tree.

5 The Specification Language for Optimization

This section describes a language that can be used to specify optimizing transformations. The language is very simple, but it can express many standard compiler optimizations naturally.

5.1 Composition of the Specification for Optimization

The specification of optimization consists of three parts: **MATCH**, **CONDITION**, and **PROCESS**. **MATCH** denotes the strategy of restricting variables in conditional formulas subject to model checking. It specifies the pattern of instructions in the program where the condition is to be checked. Variables and expressions appearing in it must be bound to specific program variables or expressions before the condition is checked. **CONDITION** is the conditions of optimization written in bidirectional CTL, which should be satisfied if the instructions are to be optimized. **PROCESS** specifies what kind of transformation is to be performed on the instructions when they satisfy the conditional formulas in the CONDITION. The optimization description is as follows. Detailed explanations can be found in [5].

MATCH
<variable> := *<expression>*
CONDITION
point_<string> : *<CTL formula>*
edge_<string> : *point_<string>* → *point_<string>*
PROCESS
point_<string> : *<Comand>* *<instruction>*
point_<string> : *Replace* *<expression>* → *<expression>*
edge_<string> : *EdgeSplit* *<instruction>*

The MATCH part specifies the form of instructions that are the target of optimization. It binds meta variables written in the CONDITION part (free variables) to variables or expressions in the program. For example, when MATCH contains $v := b$, instructions in the program will be the target of optimization if its right-hand side is an expression with binary operator. A variable is denoted as v, and an expression with binary operator is denoted as b. As a result, $z := x + y$ is the target and $x := y$ is not the target. When $z := x + y$ is the target of optimization, free variables {v, b} in the rule are bound to the target instruction as follows.

$$\{v \mapsto z, b \mapsto x + y\}$$

Thus, binding all combinations of variables and expressions is avoided.

In the CONDITION part, $conditional\ formulas$ and $partial\ formulas$ can be written. *point_<string>* and *edge_<string>* are called $(patial) fomulanames$. The conditional formulas and the partial formulas can be given names expressing their meaning and cannot be written recursively. Conditional formulas are the conditions that must hold when an optimizing transformation is to be performed. Conditional formulas are named so that they can be referred to in the PROCESS part.

When a conditional formula is long and difficult to understand, it can be written by subdividing it into several partial formulas, which are also given names. When the names of partial formulas appear in a conditional expression, they are substituted iteratively into the partial formulas that they represent before model checking.

Conditions about edges can also be written. They are called *edge conditions*. The edge condition is not related to the CTL tree. It indicates the edge from a node to another node, both satisfying the conditions written. It is calculated as a result of model checking of nodes.

The PROCESS part states how to process the instructions or edges that satisfy the conditional formulas in the CONDITION part. The names written before the ":" correspond to the names before the ":" in the CONDITION part.

A line beginning with " *point_<string>*: " transforms an instruction, and a line beginning with " *edge_<string>*: " transforms an edge. Commands that transform instructions are: *InsertBefore*, *InsertAfter*, *Delete*, and *Replace*, and the only command processing edges is *EdgeSplit*. Every command acts literally as is written; for example, *InsertBefore* inserts an instruction before another instruction. The *Replace* command acts to

$$\textit{Replace expression} \rightarrow \textit{expression}$$

and replaces part of the expression appearing before the "$\rightarrow$" symbol by the expression appearing after "$\rightarrow$" in an instruction.

Instructions or expressions can contain temporary variables when they are written in the PROCESS part, but they cannot contain temporary variables in the CONDITION part.

The (patial) fomula name (i.e. the name before ":") of a formula in the condition or process part is different from the number associated with an instruction, i.e. the node number of the Kripke structure, in previous work. In our system, the (patial) fomula name is not a free variable [1] or the node number of the Kripke structure and need not be bound before use. The (patial) fomula name represents the set of instructions satisfying the same conditional formula. Similarly, what the model checker calculates is not a specific instruction but a set of instructions satisfying the conditional formula. As a result, it becomes very easy to describe the process of rewriting many instructions that satisfy several conditional formulas at the same time. Moreover, efficiency can be improved because free variables for instruction number are now omitted. This is one of the main differences from previous research.

The following is the example of our specification language for dead code elimination.

MATCH
$v := e$
CONDITION
$point_delete: \ AX((AG\neg use(v)) \lor A\neg use(v)\ U\ def(v))$
PROCESS
$point_delete: \ Delete\ v := e$

[1] In this paper, free variables denote free variables of logical formulas. They should not be confused with variables in programs.

There are two free variables $\{v, e\}$ in the above specification.

However, there would be three free variables $\{n, v, e\}$ if we used the specification found in previous work such as Lacey's, as follows:

$$n : v := e \Longrightarrow skip$$
$$\text{if } n \models AX((AG\neg use(v))) \vee A\neg use(v)\ U\ def(v))$$

Processing time can be greatly improved by decreasing the number of free variables because free variables greatly influence the efficiency of the system (see section 6).

5.2 Formalization of Optimization with Our Specification Language in Bidirectional CTL

This section describes the formalization of compiler optimization using our specification language.

In our research, we can write a specification language in two ways. One is based on the meaning of optimization written in bidirectional CTL similar to previous work, the other is based on the dataflow equations.

5.2.1 Formalization by Describing CTL Formulas Based on the Condition of Optimization

This is the same as the formalization in previous work, but it is necessary to take features and efficiency into consideration when dealing with a real-world language. The specification for dead code elimination mentioned above can be referred to as an example.

5.2.2 Formalization by Describing CTL Formulas Based on the Dataflow Equation

The crucial connection between model checking and dataflow analysis was made by Steffen [17] [18].

Complex optimization like partial redundancy elimination is needed for real optimizing compilers. Many conditional formulas are necessary to specify it. The system must rewrite a set of instructions that satisfy the same conditional formula at the same time. Writing specifications in CTL from scratch considering the meaning and condition of optimization is difficult.

For such complex optimizations, specifications based on the dataflow equation are much easier than writing them from scratch. Partial redundancy elimination has been investigated for many years, and many algorithms exist. We adopted the method of Paleri [14] and formalized it very easily. Optimality of computation after transformation is proved in his paper. The specification for partial redundancy elimination is given in the Appendix. In general, dataflow equations containing mutually recursive equations can be solved using a general algorithm by iterating until dataflow values such as $AVIN_0$, $AVOUT_0$, $AVIN_1$, $AVOUT_1$, $AVIN_2 \ldots$, etc. converge. However model checking of CTL formulas cannot repeat until the values converge. Therefore in our specification in CTL formulas, we have slightly modified the original dataflow equations to avoid repeating calculations, carefully preserving their semantics [5].

6 Free Variables

In this section, we will discuss the binding of free variables and its computational complexity.

A *free variable* is a variable that appears in a conditional formula that is not yet bound to a specific variable or expression of program. The use of free variables was first introduced into CTL in Lacey's thesis [10] and is called CTL-FV. It is used to prove full semantics preservation of some classical compiler optimizations. However, the proof is done by hand.

Free variables must be bound to actual variables or expressions in the program before handling the program. As an example, consider a specification as follows (note that it is not a formula for any optimization).

$$point_process\text{: } AX((AG\neg use(e)) \vee A\neg use(v)\ U\ def(v))$$

The free variables in this formula are $\{v, e\}$. If the variables and expressions of the program are $\{x, y, z, a+b, x+y, -z\}$, the domains of free variables will be as follows.

$v \mapsto \{x, y, z \ldots\}$
$e \mapsto \{a+b, x+y, -z \ldots\}$

As a result the binding will be as follows.

$\{v \to x,\ e \to a+b\}$
$\{v \to x,\ e \to x+y\}$
$\{v \to x,\ e \to -z\}$
$\{v \to y,\ e \to a+b\}$
$\ldots$

Because model checking is done on each of these combinations, the number of free variables greatly influences processing time, i.e. the optimization time of the system.

Computational Complexity of Free Variable Binding

Let

n : number of free variables in conditional formulas

m : number of objects that can be the target of binding of free variables in conditional formulas, such as variables or expressions in the program

Then,

the *computational complexity* is $O(m^n)$.

CTL-FV has been adopted by most previous work, as well as ours, and it seems very convenient and expressible. However, as mentioned above, binding of free variables will cause exponential computational complexity. Therefore, introducing free variables needs to be avoided as much as possible in practical compiler optimizers.

In our research, we try to formalize optimizations so that all the free variables can be bound at the MATCH stage. However, the question of how to make a compiler optimizer in CTL without free variables is the subject of future research that we are planning.

7 Compiler Optimizer with Bidirectional CTL

Our compiler optimizer with bidirectional CTL is composed of three parts: the preprocessing part, the model checker, and the rewrite part. The preprocessing part binds free variables in the conditional formulas to the variables or expressions in the program after reading the source program and transforming it into a kind of 3-address intermediate code. The model checker calculates the points and edges of the program that satisfy the conditional formulas. The rewrite part applies optimizing rewrite rules to the result of the model checking part, and outputs the optimized program. Figure 4 is the outline of our optimization system.

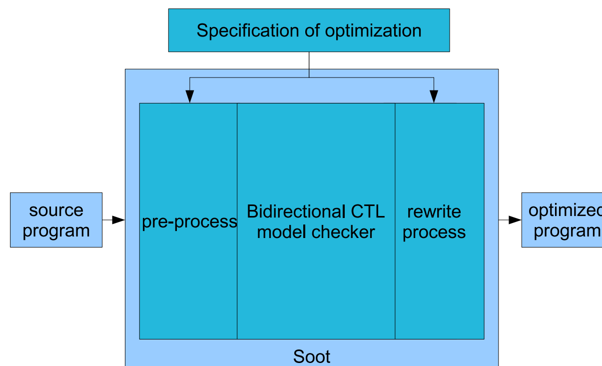

Figure 4: Outline of the optimization system

The system works within Soot [19], a Java optimization framework used as an environment for development and testing of new optimization processing. The new optimization process is added as a new phase to the existing optimizers of Soot.

Figure 5 is an example of model checking. The source program (left) is optimized using the CTL syntax tree (right), which is made from the CTL formula (center). The nodes of the CTL syntax tree can hold names of formulas or partial formulas if necessary. The result of model checking will be put into sets corresponding to such (patial) fomula names e.g. $point_a$ in the right figure.

Figure 6 is an example of rewriting for partial redundancy elimination using the result of model checking. The program before optimization is shown on the left-hand side, and the program after optimization is shown on the right-hand side.

8 Experiments

We built a system by extending some parts of the implementation of Ban [2].

Our experimental data were acquired by using the seven benchmarks of SPECjvm98 [16] and Okumura's Java code [13].

Experimental Environment The experimental environment is as follows.
CPU: Celeron 2 GHz
Memory: 512 MB
Soot: version 2.2.0

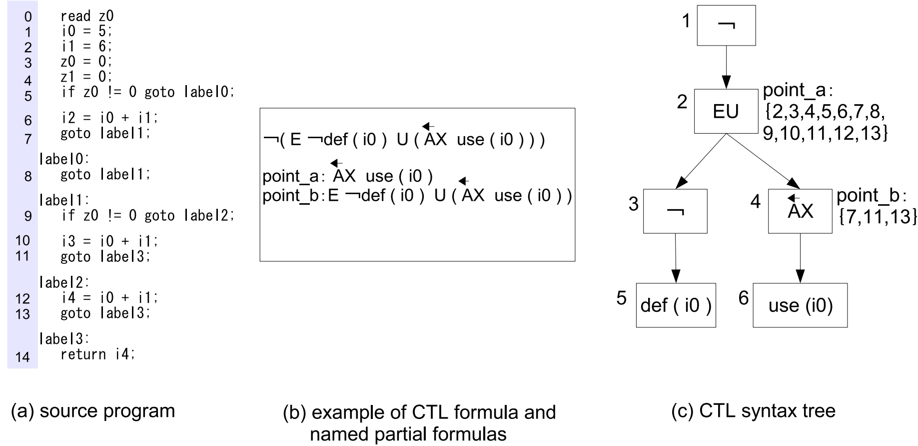

Figure 5: Model checking

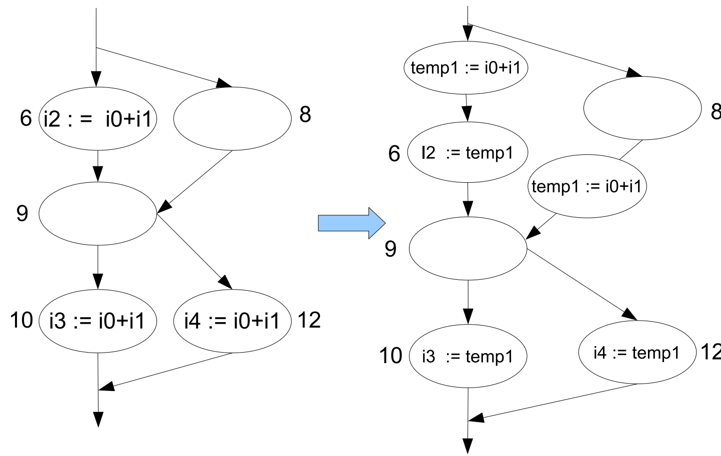

Figure 6: Example of rewriting

JDK version: 1.5.0_06-b05
JVM options: -Xint -Xms128m -Xmx128m (to exclude the influence of JIT and the memory)
Optimization applied by our system:
Partial redundancy elimination (includes common subexpression elimination and loop invariant code motion), copy propagation (includes constant propagation), dead code elimination.
Optimization applied by Soot (for comparison):
common subexpression elimination, partial redundancy elimination, copy propagation, constant propagation and folding, conditional branch folding, dead assignment elimination, unreachable code elimination, unconditional branch folding, and unused local elimination.

Processing Time of Optimization The optimization time for the SPECjvm98 benchmark by our optimizer is shown in Table 2, and the optimization time for Okumura's Java

code is indicated in Table 3.

test code	compile time
200_check	20
201_compress	29
202_jess	123
209_db	15
213_javac	219
227_mtrt	160
228_jack	316

Table 2: The optimization time of the SPECjvm98 benchmark (unit: second)

We see that the optimization time of our system is from several seconds to several minutes. It is slow compared to common compilers that take from milliseconds to several seconds. However, optimizations using temporal logic are generally slow because of traversal and searches on all instructions, and there seems to be no other choice for the moment.

test code	compile time
PiByMachin	0.8
CubeRoot	1.5
Cardano	7.1
CountingSort	2.5
NQueens	3.7
Jacobi	48.6
LogE	20.4
Fibonacci	1.4
Exp	12.1

Table 3: The optimization time of Okumura's Java code (unit: second)

Comparison of Execution Time Figure 7 and Figure 8 show the comparison of execution times of the object code before and after optimization (execution time without optimization is normalized to 1). Optimization by our system has a modest effect, although our optimization implements only a part of the optimization applied in Soot, and there is a few program where our technique beat Soot. What influences the effect of optimization will be discussed in section 9.1.

In 202_*jess* of the SPECjvm98 benchmarks, more than a 10% improvement is achieved by our system.

Comparison with Lacey's work Lacey did not give the data of the optimization time and the execution time, so we cannot make comparisons.

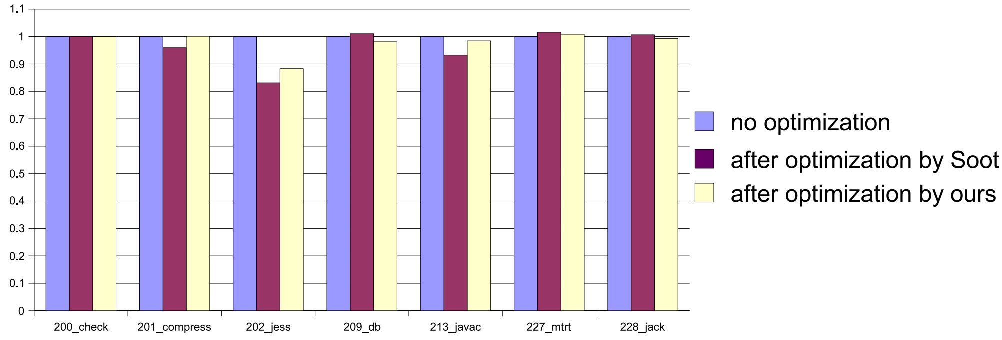

Figure 7: Effect of optimization for SPECjvm98 benchmark

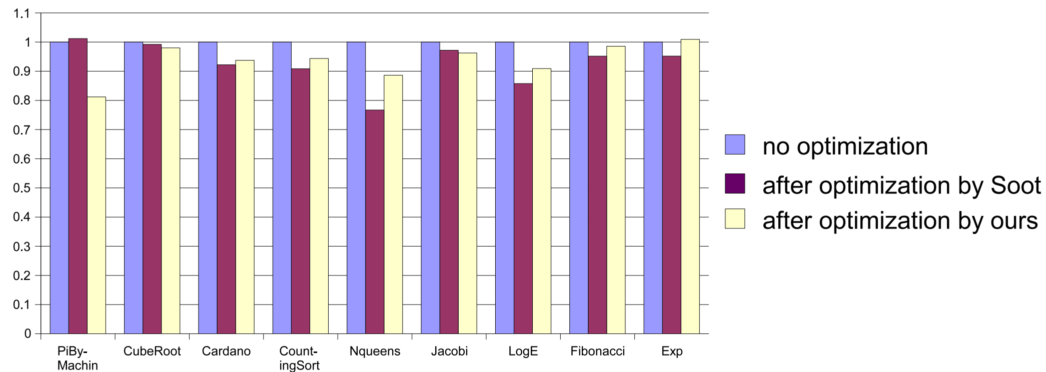

Figure 8: Effect of optimization for Okumura's Java code

Comparison with Ban's work The characteristic feature of Ban's work is to enable the handling of the limited past temporal operations in NCTL [12] by rewriting the formulas to contain only future temporal operators. This is needed to handle, e.g. copy propagation. Therefore, we compared the optimization time of copy propagation.

Figure 9 shows the result of comparison (Ban's optimization time is normalized to 1). The optimization time of our optimizer is about 15% to 30% of Ban's and is very short.

Example of Optimization Time Explosion Caused by Free Variables Figure 10 shows an example of optimization time explosion caused by free variables. We described the specification of copy propagation as an example using different CTL formulas, one with 2 free variables (left bar) and the other with 4 free variables (right bar). Optimization time increased from 20 seconds to 59 minutes, showing the dramatic increase in computational cost when the number of free variables is increased by two.

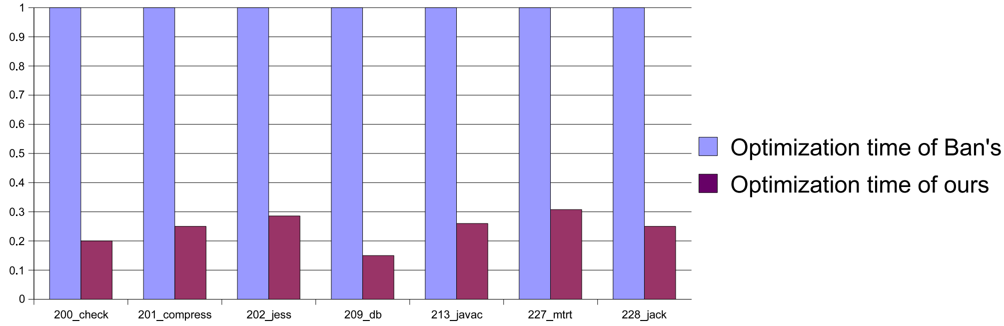

Figure 9: Optimization time of our system compared with Ban's work

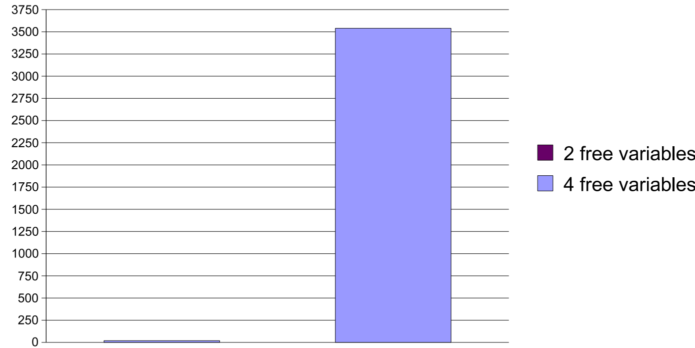

Figure 10: Example of optimization time explosion (Vertical axis: second)

9 Discussion and Future Work

To our knowledge, our system is the first system that can make optimizers for real Java programs from specifications in bidirectional CTL by using a model checker. Our main purpose is to clarify the possibility and problems of this approach and to consider how to overcome the problems. Consequently, there are many items for consideration about the current system in this section.

9.1 Consideration and Discussion

This section gives several considerations and discussions about our system and related work.

9.1.1 Expressive Power of CTL

Optimization can be specified easily and concisely in several lines by CTL, but the expressive power of CTL formula is inferior to common optimization algorithms in some cases, e.g. when we need to describe detailed processing or when the algorithm cannot be represented by first-order logic.

When we want to specify detailed processing in CTL, formulas will become long and tedious, and the proof of the correctness of the formulas will become very difficult. For

instance, “Partial redundancy elimination should not be done when moving a computation from a path that is not executed very often to a path that is executed often, because it will have a negative effect.”, or “Copy propagation shall be done only when the original instruction can become dead code.” ... are difficult to describe in CTL formulas.

Moreover, optimization that uses complex algorithms cannot be described, for instance, conditional constant propagation [20] is a complex algorithm that uses a table instead of a dataflow equation. We have to describe it as “The result of the model checking is not directly applied. The result should be remembered until the system arrives at a certain state.” However, this is difficult to express in a first-order logic.

9.1.2 Efficiency of The Compiler Optimizer

Binding of free variables greatly influences the efficiency of the system as mentioned in section 6. Moreover, model checking is an exhaustive algorithm. Therefore, its efficiency is inferior to that of common algorithm-based compiler optimizers.

9.1.3 Effectiveness of Optimization

Because our target is Java programs, moving instructions cannot cross an exception. Instructions that may cause run-time exceptions like array indexing, division and the mod (remainder) operation etc. must all be excluded from optimization too.

In Soot, there is a graph called Brief-Graph that shows the control flow graph of the program directly. On the other hand, there is a graph called Complete-Graph where in addition to edges in the control flow graph, edges are drawn from all instructions enclosed by the “try” instruction to the “catch” instruction (bold lines).

An example of optimization of these graphs is shown in Figure 11. When partial redundancy elimination (PRE) is applied to the program shown in the left-hand figure using Brief-Graph, the result is as shown in the central graph. However, when it is done using Complete-Graph, redundancy cannot be eliminated because of the obstruction caused by the exception as shown in the graph at the right. This is because the movement of $x+y$ will be given up in the algorithm of a conservative partial redundancy elimination because it influences the paths indicated by bold lines.

The problem of exception obstruction such as code moves that cannot cross an exception exists not only in our research but also in common Java optimizers [19].

9.2 Future Work

This section presents some possible future directions of research. The possibilities can be divided into two categories: improving optimization time and improving the efficiency of optimized programs.

9.2.1 Reducing Optimization Time

It will be possible to introduce binary decision diagrams (BDD), partial evaluation or some other technology to make model checking faster.

Free variables need to be reduced or eliminated as much as possible because binding of free variables greatly affects processing (optimization) time. The following program is an example illustrating the reduction of free variable binding and model checking.

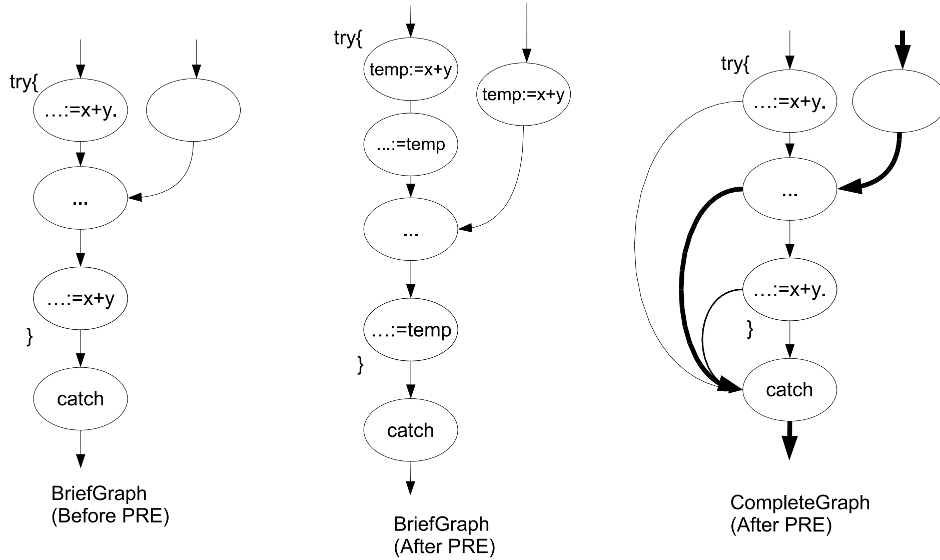

Figure 11: Obstruction of optimization by exceptions

1: $x := 100;$
2: $y := 1;$
3: $z := 2;$
4: $w := 3;$
5: $x := w + 1;$
6: $z := x + y;$
...

- Model checking can be omitted if it is not necessary. For example, if the target of copy propagation is x in "6: $z := x + y$", instructions before statement 5, such as statement 1, need not be checked because x is assigned in "5: $x := w + 1$".
- Binding can be omitted if the target is not on the path related to the temporal formula. For example, checking instructions on the past paths can be omitted if the bidirectional CTL formula includes only future temporal operators. Instructions far away from the next instruction can be omitted if the temporal operator is AX or EX. In our experiment, when this technique is applied to the dead code elimination (which only includes future temporal operators), processing time is reduced to about 1/3.

9.2.2 Improving the Efficiency of the Optimized Program

To improve the effectiveness of optimization, overcoming the obstruction of optimization caused by exceptions and detailed analysis of loops, for statements, goto statements, etc., will be necessary. The analysis of exceptions by CTL may be considered. Specifying complex optimization such as conditional constant propagation [20] with CTL is also our future work.

Furthermore, implementing a model checker with bidirectional CTL* [7], which is more expressive than bidirectional CTL, can also be considered. We are planning to carry out

the above in the future.

10 Related Work

The crucial connection between model checking and program analysis was made by Cousot [4].

Lacey et al. [10] introduced a temporal logic named CTL-FV that can use free variables in predicates and past temporal operators in addition to the original CTL. Many traditional optimizations can be done by specifying the condition in CTL-FV and specifying the rewriting of instructions using the result. The papers [9] [11] describe the proposal and prove the correctness of some traditional optimization formulas by hand. The thesis [11] describes the detail of the technique. However, as for implementation, it just explains that they solved the problem by finding the fixed point after converting it into the μ-calculus. Moreover, the formulas that have been proven in the thesis are only a part of the optimization that can be done by real-world optimizers. No experimental data such as optimization time are given, so we think that implementation with a real-world program has not been done previously.

Yamaoka et al. [21] have implemented an optimizer with CTL using the existing model checker SMV [15], but it can only deal with dead code elimination because only future temporal operators are allowed.

Ban's research [2] is able to treat the NCTL [12] including the past temporal operator in a limited form, by using 12 conversion equations, but it consumes time to remove past temporal operators, and the formulas become very long after conversion. As a result, the model checking time is considerable. Moreover, only the dead code elimination and copy propagation can be done because it can rewrite only one instruction corresponding to one condition in the optimization specification.

Note that in almost all the past research, the node number of the Kripke structure corresponding to an instruction should be written in the specification. This will contribute to the binding explosion problem mentioned before.

Experimental data on optimization time and execution time of optimized code using the benchmarks are not presented in any previous work.

11 Conclusion

The main contribution of our research is as follows.

- We implemented a very efficient model checker that directly handle the bidirectional CTL. It does not convert into μ-calculus and does not perform any other conversion.
- We proposed a transformation language that is very expressible. As a result, even complex optimization formulas that can deal with real-life optimization can be written very naturally and easily.
- We developed a practical Java optimizer using bidirectional CTL that has a modest effect.

- It is the first time that experimental data on optimization time and the effect of optimization has been measured using the benchmarks and the test programs in our research. Most optimizations are approaching practicality.
- Moreover, some problems of optimization by CTL were clarified through this experience.

We think that these data and experiences are significant and valuable in this field and can contribute to future research work.

We plan to solve the existing problems and improve the performance toward a more realistic optimizer.

References

[1] Aho, A. V., Lam, Monica S., Sethi Ravi, and Ullman, Jeffrey D.: Compilers Principles, Techniques, and Tools, Second ed., Addison Wesley, 2006.

[2] Ban. N, Hu Zhengjiang, and Takeichi Masato: Declarative description of Java program optimization and its efficient implementation (in Japanese), *Proc. 6th Programming and Programming Language Workshop (PPL2004)*, pp. 65–75, 2004.

[3] Cooper Keith D. and Torczon Linda: Engineering A Compiler, Elsevier Science Publishers, 2004.

[4] Cousot, P. and Cousot, R.: Automatic synthesis of optimal invariant assertions: mathematical foundations, *PLDI*, Vol. 12, No. 8, pp. 1–12, 1977.

[5] Fang Ling: Java optimizer with bidirectional CTL (in Japanese), Master Thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2006.

[6] Gupta, M., Choi, J.-D., and Hind, M.: Optimizing Java programs in the presence of exceptions, *Proc. 14th European Conference on Object-Oriented Programming*, pp. 422–446, Springer-Verlag (2000).

[7] Jan Van Leeuwen (Ed.): Handbook of Theoretical Computer Science Vol. B, Formal Models and Semantics, Elseveir Science Publishers B. V., 1990.

[8] Kupferman, O. and Pnueli, A.: Once and for all, *In Proceedings of the 10th IEEE Symposium on Logic in Computer Science (LICS 1995)*, pp. 25–35, 1995.

[9] Lacey David, Jones, Neil D., Van Wyk Eric, and Frederiksen Carl Christian: Compiler optimization correctness by temporal logic, *Higher-Order and Symbolic Computation*, Vol. 17, No. 3, pp. 173–206, 2004.

[10] Lacey David, Jones, Neil D., Van Wyk Eric, and Frederiksen Carl Christian: Proving correctness of compiler optimizations by temporal logic, *In Proceedings of Symposium on Principles of Programming Languages*, pp. 283–294, 2002.

[11] Lacey David: Program transformation using temporal logic specifications, Dissertation submitted for PhD examination at the University of Oxford, 2003.

[12] Laroussinie François and Philippe Schnoebelen: Specification in CTL+Past for verification in CTL, *Information and Computation*, Vol. 156, No. 1/2, pp. 236–263, 2000.

[13] Okumura Haruhiko: Algorithm dictionary by Java, http://oku.edu.mie-u.ac.jp/˜okumura/Java-algo/

[14] Paleri Vineeth Kumar, Srikant, Y. N., and Shankar Priti: A Simple algorithm for partial redundancy elimination, *ACM SIGPLAN Not.*, Vol. 33, No. 12, pp. 35–43, 1998.

[15] SMV Model Checker, http://www.cs.cmu.edu/modelcheck/smv.html

[16] SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98

[17] Steffen, B.: Data flow analysis as model checking, *In Proceedings of Theoretical Aspects of Computer Science*, pp. 346–364, 1991.

[18] Steffen, B.: Generating data flow analysis algorithms from modal specifications, *Science of Computer Programming*, Vol. 21, No. 2, pp. 115–139, 1993.

[19] Vallée-Rai Raja, Co Phong, Gagnon Etienne, Hendren Laurie, Lam Patrick, and Sundaresan Vijay: Soot — a Java optimization framework, *In Proceedings of CASCON 1999*, pp. 125–135, 1999, http://www.sable.mcgill.ca/soot/

[20] Wegman, M. N. and Zadeck, F. K.: Constant propagation with conditional branches, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2, pp. 181–210, 1991.

[21] Yamaoka Yuji, Hu Zhengjiang, Takeichi Masato, and Okawa Mizuhito: Program analysis with model check (in Japanese), *IPSJ Transactions on Programming*, Vol. 44, No. SIG13 (PRO18), pp. 25–37, 2003.

Appendix Specification of Partial Redundancy Elimination

MATCH
$v := e$
CONDITION
$point_comp : use(e) \wedge trans(e)$
$point_avin : \overleftarrow{A} X(\overleftarrow{A}\ trans(e)\ U\ use(e))$
$point_avout : point_comp \vee (point_avin \wedge trans(e))$
$point_antout : AX(A\ trans(e)\ U\ use(e))$
$point_antin : point_comp \vee (point_antout \wedge trans(e))$
$point_safein : point_avin \vee point_antin$
$point_safeout : point_avout \vee point_antout$
$point_spavin : point_safein \wedge \overleftarrow{E} X(\overleftarrow{E}\ (trans(e) \wedge (point_safeout))\ U\ use(e))$

$point_spavout : point_safeout \wedge (point_comp \vee (point_spavin \wedge trans(e)))$
$point_spantout : point_safeout \wedge EX(E\ (trans(e) \wedge (point_safein))\ U\ use(e))$
$point_spantin : point_safein \wedge (point_comp \vee (point_spantout \wedge trans(e)))$
$point_insert : point_comp \wedge \neg point_spavin \wedge point_spantout$
$point_replace : (point_comp \wedge point_spavin) \vee (point_comp \wedge point_spantout)$
$point_edge1 : point_spavin \wedge point_spantin$
$point_edge2 : \neg point_spavout$
$edge_split : point_edge1 \rightarrow point_edge2$
PROCESS
$point_insert : InsertBefore\ temp := e$
$edge_split : EdgeSplit\ temp := e$
$point_replace : replace\ e \rightarrow temp$