Translating Flowcharts to Non-Deterministic Languages

Surinder Kumar Jain
School of Information Technologies, University of Sydney, NSW, Australia
sjai8466@uni.sydney.edu.au

Chenyi Zhang
School of ITEE, University of Queensland, Brisbane, QLD, Australia
chenyi@uq.edu.au

Bernhard Scholz
School of Information Technologies, University of Sydney, NSW, Australia
scholz@it.usyd.edu.au

Abstract

Modeling languages are used to verify software and can be classified into deterministic modeling languages and non-deterministic modeling languages. Deterministic modeling languages have a single thread of control whereas non-deterministic ones have a multitude of threads of control and are more amenable for program transformations and analyses. However, deterministic languages such as control-flow graphs are pre-dominantly used in programming language tools.

In this work, we translate programs in a deterministic flowchart language to a non-deterministic algebraic modelling language. For the translation, we employ the technique of converting a finite state automata to a regular expression. The states of the finite state automata represent states in the control-flow graph, and the edges represent the edges in the control-flowgraph. We construct a homomorphism to show that the translation is sound, i.e., we prove that the semantics of the program in the deterministic flowchart language is preserved in the translation. Experiments on our implemented algorithm are conducted on the SPEC benchmark suite.

Categories and Subject Descriptors F3.2 [*Logic and Meanings of Programs*]: Program Analysis

General Terms Languages, Theory, Verification

Keywords Non-deterministic language, Program transformations, Modelling

1. Introduction

Tools to assist the software development process are crucial to reduce the amount of time needed to design, implement and debug software. A lot of attention has been given to techniques such as model checking [2] to check properties of software [3]. Gulwani et al. [10] introduced a non-deterministic language that enables loop-refinement transformations for improving the detection of runtime complexity classes automatically. The introduced loop-refinement technique in [10] employs techniques such as *lookahead widening* [9], and improves the precision on estimating the termination and bounds of multi-path loops, achieving better results than many other existing techniques. The technique for detecting complexity classes by Gulwani et al. [10] may be extended to other program analysis problems such as invariant detection and safety checking.

PEPM'12, January 23–24, 2012, Philadelphia, PA, USA.


```
1:  y := 1;                    9:  ...
2:  if x <= φ;                 10: if x > φ;
3:      ...                    11:     ...
4:      y := −y;               12: else
5:      ...                    13:     ...
6:  else                       14:     y := −y;
7:      ...                    15:     ...
8:  /* cf. next column */      16:
```

Figure 1. A sketching example, where ϕ is an expression

Consider the example in Figure 1. For this example, we assume there are no other side effects on the value of y than the three existing assignments given in the figure. Under this assumption, variable y can only have value 1. However, classical data flow analysis (i.e., MOP) will conservatively assert that y's value is in $\{-1, 1\}$. A better precision can be achieved by defining a more expressive abstract domain (eg. trace partitionings which interpret a system as a set of traces guarded by a sequence of branching conditions that are chosen in history [11, 19]). Such approaches elaborate on the level of abstractions of deterministic programs. The technique of [10] flattens the above code into a nondeterministic program of the following form.

$y := 1;$ **choose**(
  assume$(x <= \phi) \ldots y := -y \ldots$ **assume**$(x > \phi) \ldots$,
  assume$(x <= \phi) \ldots y := -y \ldots$ **assume**$(x <= \phi) \ldots$
    $\ldots y := -y \ldots$,
  assume$(x > \phi) \ldots$ **assume**$(x > \phi) \ldots$,
  assume$(x > \phi) \ldots$ **assume**$(x <= \phi) \ldots y := -y \ldots$,
)

In the resulting program, after the first assignment $y := 1$, the execution nondeterministically proceeds on four distinct paths, and the control dependency is made explicit inside each path. A straightforward analysis is able to identify that the first path and the last path are infeasible in the above representation, revealing y as a constant. In general, this solution first translates a deterministic program into a semantically equivalent nondeterministic program, which proceeds on several branches at the same time and may have a multitude of program states at a program point, and if a point is not reachable, the set of program states at this point will be empty. Such an approach derives modelling templates from programs, from which various program analyses such as exploring feasible paths, computing loop complexity and invariant detection may be applied, via symbolic execution, constraints solving, model checking or other related techniques. Over all, this technique presents a straightforward structure for program analysis, and moreover, allows the algebraic reasoning of translated arbitrary (i.e., even irreducible) flow graphs.

Deterministic flowchart languages are commonly found in programming language tools and compilers as intermediate representations [1, 6, 14]. Flowchart programs have an execution control-flow and a unique program state that changes as control-flow traverses from one program node to the next. Deterministic semantics of a flowchart language either produces a single program state as a final result of the computation or does not terminate.

The translation of a flowchart language to a structured imperative program may be performed by a naïve translation template that encodes the current state in a program variable. A simple loop transits from one state in the flowchart to the next until the end state is reached and the loop terminates. This translation template may be converted to a non-deterministic program using translation rules as outlined in [10]. However, this translation template and the subsequent translation to a non-deterministic language converts control dependencies to data dependencies, which will hamper the loop-refinement techniques [10] and other non-deterministic transformations that rely on control-flow.

The contributions of our work is in providing a translation technique that translates a flowchart program to Gulwani et al.'s non-deterministic language by preserving control-flow dependencies. The technique adopts the notion of Tarjan's path homomorphism [21], i.e., the flowchart is interpreted as a deterministic finite state automaton with the edges as an alphabet. The regular expression of the finite state automaton is constructed and mapped to Gulwani et al.'s non-deterministic language. In this paper we illustrate the technique by a motivating example and provide the correctness proofs for the translation. There is a notably rich literature on graph transformation techniques on flowcharts, as discussed in [5, 16]. Some earlier literature also discusses ways of removing *goto* statements to tidy up legacy programs [4, 17]. Our work aims to provide a generic approach, which first assigns an irreducible flowchart with a well-organised structure, and then further reduce it into a nondeterministic program. The translated nondeterministic program can be proved to preserve the semantics of the original flowchart.

Due to the inherent complexity bound in the translation from finite automata to regular expressions, the generated nondeterministic programs may have the size exponential to the input flowchart programs, if we are to put them down explicitly. However, a compact representation can be used to circumvent this problem, so that in practice, storing the resulting nondeterministic program only requires comparable space usage to the input flowchart. Moreover, such a representation handles function calls in an inter-procedure analysis well. We refer to Section 5 for more details.

The paper is structured as follows. In Section 2 we introduce a deterministic flowchart language called DFL and a nondeterministic language called NDL. In Section 3 we motivate why a more complex translation from DFL to NDL is necessary rather than using a naïve translation. In Section 4 we introduce the formal framework of the translation and justify its correctness. Experimental results are presented in Section 5, and in Section 6 we draw our conclusion.

2. Background

This section introduces basic notions that are used in the technical part. We present syntax and semantics of the *Deterministic Flow Language* (DFL), and the (simplified) *Non-Deterministic Language* (NDL).

Let *Var* be a domain of variables of a program in either DFL or NDL, and $\mathbb{Z}$ a set of integer values that a variable $x \in \mathit{Var}$ may have. To manipulate values of variables, we assume a set *BINOP* of binary operators of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, and a set *UNOP* of unary operators of type $\mathbb{Z} \to \mathbb{Z}$. For both languages, we define two types of building blocks to construct the languages, i.e., expressions and statements. Let *EXPR* and *SSTMT* be the smallest sets that are defined in Figure 2 as the set of expressions and the set of *simple* statements. An expression is a composition of unary operators, binary operators, integer numbers, and variables. A simple statement is either "**skip**" denoting that no variable changes its value or "x**:=***Expr*" denoting that variable $x \in \mathit{Var}$ changes its values to an element of $\mathbb{Z}$ computed by *Expr*.

$$
\begin{array}{lcl}
\mathit{SStmt} & \longrightarrow & \textbf{skip} \\
\mathit{SStmt} & \longrightarrow & x\textbf{:=}\mathit{Expr} \\
\\
\mathit{Expr} & \longrightarrow & \mathit{Expr}_1 \oplus \mathit{Expr}_2 \\
\mathit{Expr} & \longrightarrow & \ominus \mathit{Expr}_1 \\
\mathit{Expr} & \longrightarrow & x \\
\mathit{Expr} & \longrightarrow & a
\end{array}
$$

where $\ominus \in \mathit{UNOP}$, $\oplus \in \mathit{BINOP}$, $x \in \mathit{Var}$ and $a \in \mathbb{Z}$.

Figure 2. Building Blocks of DFL and NDL

$$
\begin{array}{l}
\mathcal{E} : \mathit{EXPR} \to \Sigma \to \mathbb{Z} \\
\\
\mathcal{E}[\![\mathit{Expr}_1 \oplus \mathit{Expr}_2]\!](\sigma) = \mathcal{E}[\![\mathit{Expr}_1]\!](\sigma) \oplus \mathcal{E}[\![\mathit{Expr}_2]\!](\sigma) \\
\mathcal{E}[\![\ominus \mathit{Expr}]\!](\sigma) \quad\quad = \ominus \mathcal{E}[\![\mathit{Expr}]\!](\sigma) \\
\mathcal{E}[\![x]\!](\sigma) \quad\quad = \sigma(x) \\
\mathcal{E}[\![a]\!](\sigma) \quad\quad = a
\end{array}
$$

where $\ominus \in \mathit{UNOP}$, $\oplus \in \mathit{BINOP}$, $x \in \mathit{Var}$ and $a \in \mathbb{Z}$.

Figure 3. Semantics of Expressions

Deterministic Flow Language: We define a flowchart language, which we call *deterministic flow language* (DFL), and its operational semantics. DFL is a dynamic discrete system [8, 18] that consists of states and a transition relation on states. The states and the transition relation is represented as a flowgraph $G = (\mathit{Nodes}, \mathit{Edges}, n_0, n_x)$ where Nodes is a set of nodes, $\mathit{Edges} \subseteq \mathit{Nodes} \times \mathit{Nodes}$ a set of edges, $n_0 \in \mathit{Nodes}$ a distinguished start node, and $n_x \in \mathit{Nodes}$ a distinguished exit node. For an edge $e = (n, n') \in \mathit{Edges}$ we say n is the *source* of e, written as $\mathrm{src}(e)$, and n' is the *destination* of e, written as $\mathrm{dst}(e)$. A DFL program P is a tuple $P = (G, \mathit{Var}, \mathit{bp}, \mathit{eff})$, where G is the flowgraph of program P, *Var* is the set of variables of program P, $\mathit{bp} : \mathit{Edges} \to \mathit{EXPR}$ the *branch predicates* of Edges, and $\mathit{eff} : \mathit{Edges} \to \mathit{SSTMT}$ the side effects of Edges.

An edge $(n, n') \in \mathit{Edges}$ indicates a possible transfer of control from n to n'. Associated to an edge is a branch predicate $bp(n, n')$ that governs the traversal of the edge, i.e., (n, n') is enabled only if $bp(n, n')$ is evaluated true. After a transition, an edge may change the values of variables by the side effect $\mathit{eff}(n, n')$. The function *eff* either maps an edge to a **skip** statement denoting that there are no effects or an assignment x**:=***Expr* which changes the value of variable x. Note that a control flowgraph (CFG) [1] has the side effects defined on nodes rather than edges. A straightforward flowgraph translation converts a CFG to a DFL program by pushing statements in nodes onto their incoming edges as side effects and controlling structure onto their outgoing edges as branch predicates.

We define an operational semantics for DFL by assigning each edge a transition rule. Let Σ be the set of configurations of type $\mathit{Var} \to \mathbb{Z}$, and we use symbols such as σ, σ' for configurations in Σ. Informally, a configuration is a snapshot during an execution of a program P that maps each variable to a unique value in $\mathbb{Z}$.

The transition rules rely on an interpretation $[\![.]\!] : \mathit{SSTMT} \rightarrow \Sigma \rightarrow \Sigma$ for simple statements, such that $[\![\textbf{skip}]\!](\sigma) = \sigma$ and $[\![x\textbf{:=}\mathit{Expr}]\!](\sigma) = \sigma_{x \leftarrow \mathcal{E}[\![\mathit{Expr}]\!](\sigma)}$ where $\sigma_{x \leftarrow v}$ is a configuration identical to σ except that the valuation on x is now v. We interpret *Expr* by means of denotational semantics as shown in Figure 3 so that $\mathcal{E}[\![\mathit{Expr}]\!]$ can be evaluated as a unique value by configuration σ.

A branch predicate acts as a guard for a transition to be taken at an edge $e \in \mathit{Edges}$, i.e.,

$$\frac{\sigma(bp(e)) \neq 0}{\sigma \xrightarrow{e} [\![\mathit{eff}(e)]\!](\sigma)}. \tag{1}$$

where boolean values of predicates are 0 if false and not equal to zero otherwise. We use $\sigma \xrightarrow{e} \sigma'$ to represent that configuration σ is transformed into σ' when the flow goes from $\mathrm{src}(e)$ to $\mathrm{dst}(e)$.

A program P in DFL has to be *well-formed* such that for a given node $n \in \mathit{Nodes} \setminus \{n_x\}$ and a configuration $\sigma \in \Sigma$, *exactly one* transition in the set of possible transitions for n can be taken. This condition naturally corresponds to the *if-then-else* (or *if-goto*) structure which is commonly used in the programming languages, i.e., given an arbitrary program state, either the *if* branch or the *else* branch is to be taken. Without loss of generality, we assume that n_x is the only node without outgoing edges, since if there exists another $n \in \mathit{Nodes}$ that has no outgoing edges, we add an additional edge $e = (n, n_x) \in \mathit{Edges}$ with $bp(e)$ evaluated to constant 1 and $\mathit{eff}(e) = \textbf{skip}$. A program terminates with configuration σ if it reaches n_x with σ. Programs do not necessarily terminate, in which case it passes through infinitely many edges (possibly by repeating cycles) after starting with an initial configuration.

A path π is a sequence of consecutive edges $e_1 e_2 \ldots$ satisfying $\mathrm{dst}(e_i) = \mathrm{src}(e_{i+1})$ for all i, and the set of all paths in G is denoted by Π. If π is a finite sequence of edges with the last edge e_k, we say it is a path from $\mathrm{src}(e_1)$ to $\mathrm{dst}(e_k)$. The set of all paths from node n to node n' is denoted by $Path(n, n') = \{e_1 e_2 \ldots e_k \in \Pi \mid \mathrm{src}(e_1) = n, \mathrm{dst}(e_k) = n'\}$. W.l.o.g, we assume every node in a flowgraph is reachable from n_0, i.e., there exists a path $e_0 e_1 \ldots e_k$ such that $\mathrm{src}(e_0) = n_0$ and $\mathrm{dst}(e_k) = n$ for all $n \in \mathit{Nodes}$. An execution of P with an initial configuration σ_0 is a sequence of transitions $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots$ where $e_1 e_2 \ldots$ is a path in G, and $\sigma_i \xrightarrow{e_{i+1}} \sigma_{i+1}$ is semantically derivable (i.e., by rule (1)) for all $i \geq 0$. Note that an execution can have either an infinite or a finite sequence of edges. If a execution $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots \xrightarrow{e_k} \sigma_k$ is finite, then naturally we have $dst(e_k) = n_x$, and σ_k can be seen as the output of program P. In this case we also write $\sigma_0 \xrightarrow{e_1 \ldots e_k} \sigma_k$.

Non-Deterministic Language: In this work, we use a variation of NDL introduced in [10] as shown in Figure 4. Programs in NDL use simple statements and more complex program constructs including **choose**, **repeat**, **assume**, and an sequential composition operator ";".

Non-determinism in NDL is modelled as a set of configurations $C \in \mathcal{P}(\Sigma)$, which we refer to as a *context*. The denotational semantic function $\mathcal{S}[\![\mathit{Stmt}]\!](C)$ translates context $C \in \mathcal{P}(\Sigma)$ to a new context. In abuse of notation we write $\mathcal{S}[\![\mathit{Stmt}]\!](\sigma)$ for $\mathcal{S}[\![\mathit{Stmt}]\!](\{\sigma\})$ in case of a singleton context $\{\sigma\}$. Note that C may contain possibly infinite system configurations in the presence of **repeat** statements.

Semantic function $\mathcal{S}[\![.]\!]$ is of type $\mathit{STMT} \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ as shown in Figure 5. The semantics of the statements **choose**, **repeat** and **assume** are typically nondeterministic, which is in contrast to *while* and *if-then-else* statements of imperative languages [15, Chapter 11]. Intuitively, statements **choose** and **repeat** increase nondeterminism, by potentially adding more configurations into the result of their translations, whilst statement **assume** prunes them.

Stmt	$\longrightarrow$	**choose**(*Stmt*,*Stmt*)
Stmt	$\longrightarrow$	**repeat**(*Stmt*)
Stmt	$\longrightarrow$	**assume**(*Expr*)
Stmt	$\longrightarrow$	*Stmt*;*Stmt*
Stmt	$\longrightarrow$	*SStmt*

Figure 4. NDL Syntax

$$\mathcal{S} : \mathit{STMT} \cup \mathit{SSTMT} \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$$

$$\begin{aligned}
\mathcal{S}[\![\textbf{choose}(\mathit{Stmt}_1,\mathit{Stmt}_2)]\!](C) &= \mathcal{S}[\![\mathit{Stmt}_1]\!](C) \cup \mathcal{S}[\![\mathit{Stmt}_2]\!](C) \\
\mathcal{S}[\![\textbf{repeat}(\mathit{Stmt})]\!](C) &= \textstyle\bigcup_{i \geq 0} \mathcal{S}[\![\mathit{Stmt}^i]\!](C) \\
\mathcal{S}[\![\textbf{assume}(\mathit{Expr})]\!](C) &= \{\sigma \in C \mid \mathcal{E}[\![\mathit{Expr}]\!](\sigma) \neq 0\} \\
\mathcal{S}[\![\mathit{Stmt}_1;\mathit{Stmt}_2]\!](C) &= \mathcal{S}[\![\mathit{Stmt}_2]\!](\mathcal{S}[\![\mathit{Stmt}_1]\!](C)) \\
\mathcal{S}[\![\textbf{skip}]\!](C) &= C \\
\mathcal{S}[\![x\textbf{:=}\mathit{Expr}]\!](C) &= \textstyle\bigcup_{\sigma \in C} \{\sigma_{x \leftarrow \mathcal{E}[\![\mathit{Expr}]\!](\sigma)}\}
\end{aligned}$$

where $x \in \mathit{Var}$ and $\mathit{Stmt}^i \equiv \underbrace{\mathit{Stmt};\ldots;\mathit{Stmt}}_{i \text{ times}}$ for $i > 0$; $\mathit{Stmt}^0 \equiv \textbf{skip}$.

Figure 5. Semantics of NDL

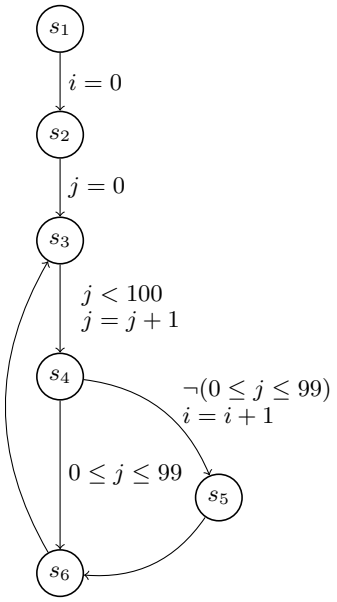
```
state:=s1;
while state<>sx do
  switch state do
    ⋮
    case si:
      if bp(si,sj1) do
        eff(si,sj1);
        state:=sj1
      elif bp(si,sj2) do
        eff(si,sj2);
        state:=sj2
      ⋮
      elif bp(si,sjk) do
        eff(si,sjk)
        state:=sjk
      end
    ⋮
  end
end
```
(a) State Automata

```
state:=s1;
repeat(
  choose(
    ⋮
    assume(state=si);
    choose(
      assume(bp(si,sj1));
      eff(si,sj1);
      state:=sj1,
      assume(bp(si,sj2));
      eff(si,sj2);
      state:=sj2,
      ⋮
      assume(bp(si,sjk));
      eff(si,sjk);
      state:=sjk),
    ⋮
));
assume(state=sx)
```
(b) State Automata in NDL

Figure 6. Naïve Translation Template.

3. Motivation

Most programming language tools such as compilers and parser frameworks use flowchart languages as intermediate representations, e.g., LLVM [14], whereas techniques such as Gulwani [10] employ a non-deterministic language that consists of structural elements such as **repeat**, **choose**, **sequence**, **assume**, assignment and **skip**. To translate a flowchart program to an NDL program, a naïve translation could be applied. The translation technique follows the coding templates that implements finite state automata [22] in an imperative programming language. In such a code template, a variable keeps track of the current state of the finite state automaton. A program is constructed in the way that sets the state variable to the



(a) A DFL Program

```
state:=1;
repeat(
  choose(
   assume(state=1);i:=0;state:=2,
   assume(state=2);j:=0;state:=3,
   assume(state=3);assume(j<100);
    j=j+1;state:=4,
   assume(state=4);choose(
    assume(0≤j≤99);state:=6,
    assume(¬(0≤j≤99));i:=i+1;
     state:=5),
   assume(state=5);state:=6,
   assume(state=6);state:=3
  )
)
```

(b) Naïve Translation

```
i:=0;
j:=0;
repeat(
  assume(j < 100);
  j:=j+1;
  choose(
   assume(¬ 0≤ j ≤ 99),
   assume(0≤ j ≤ 99);i:=i+1
  )
)
```

(c) Our Translation Technique

Figure 7. A Motivating Example

initial state of the automaton and continues to execute a loop. Inside the loop, the state variable chooses a code-block that is associated to the current state. This code-block transits to a new state by assigning the state variable a new state. The transition to a new state involves a sequence of checking on the branch predicates, and in case that a predicate evaluates $true$, the program updates the value of the corresponding program variable as well as the state variable. The loop is executed until the end state is reached or no predicates are satisfied in a certain state.

With the naïve translation technique control-flow dependencies are transformed to data-dependencies and techniques such as loop-refinement of Gulwani [10] will have limited success. To overcome the issues of the naïve translation from DFL to NDL, we introduce a new translation that simulates the techniques of dataflow analysis [13, 21]. However, instead of using dataflow analysis as a framework for gathering information about a flowchart program, we use it as a technique to translate DFL programs to NDL programs.

The naïve template for translating DFL to NDL is given in Figure 6(a) and (b). Figure 6(a) gives the implementation of the automaton in a simple imperative language without **goto** statements. This template can be translated to an NDL program following the transformation rules as outlined in [10]. The translation to NDL is given in Figure 6(b). The NDL program consists of an assignment that assigns the start state to the state variable and is followed by a **repeat** statement. Inside the repeat statement we have a **choose** statement that non-deterministically executes the statements inside. For each state we have an **assume** statement that filters all configurations whose value of the state variable equals the corresponding state. The **assume** statement is followed by a **choose** statement that non-deterministically branches to a new state. The configurations are then filtered by the branch-predicate and the side-effect is performed.

The example in Figure 7 illustrates the translation from DFL to NDL based on our technique. The example has node s_1 that is the start node and has node s_3 as an end node. Instead of using the code template of Figure 6 (which flattens the input program to a single loop), we develop a new translation technique which does not eliminate the control-flow of the DFL program, as shown in Figure 7(c).

The new translation technique simulates Kildall's monotone data-flow analysis framework [13] and Tarjan's path homomorphism [21] in the concrete semantic domain. By interpreting path expressions [21] as regular expressions over the set of edges in a flowgraph, we generate path expressions for DFL programs, which are later translated into NDL programs. Kleene stars * are introduced to simulate loops, and such an operator has a natural correspondence to the **repeat** structure of the NDL language.

4. Translating DFL to NDL

We introduce a transformation that maps programs in DFL to NDL. Starting from the graph-based operational semantics as presented in Section 2, we further derive a denotational interpretation for each node in a flowgraph for DFL. We start with its concrete semantics of DFL, and then on the syntactic level we treat the set of edges in a flowgraph as a finite alphabet, upon which a regular language is defined by the flowgraph in the form of finite automata. We then apply a standard algorithm to produce a path expression from the DFL program. Such a path expression is then naturally interpreted as an NDL program. We show that the semantics of the original DFL programs are preserved throughout the translation. In theory the resulting path expressions can be exponential in size to the DFL programs, however they can always be stored in space polynomial to DFL programs by reusing subexpressions, and thus concisely represented. As the real world programs usually have sparse branching structures, we are able to justify that our translation does not usually blow up the space usage in practice, by the experimental results shown in Section 5.

4.1 A Flow Semantics for DFL

Taking an initial configuration σ_0 in a DFL G_D, an execution can be uniquely determined. Recall that such an execution can be either

an infinite sequence $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots$, or a finite sequence $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots \xrightarrow{e_k} \sigma_k$ with the final configuration σ_k stopping at node $\mathrm{dst}(e_k) = n_x$. Prefixes of such an execution can be used to define intermediate values at each node in a flowgraph. We let $[\![G_D]\!](n)$ be an interpretation of G_D between n_0 and an arbitrary node $n \in Nodes$ as follows, given arbitrary initial configurations.

Definition 1. *Given $n \in Nodes$, $[\![G_D]\!](n)$ is an interpretation on node n of type $\Sigma \to \mathcal{P}(\Sigma)$, defined by $[\![G_D]\!](n)(\sigma_0) = \{\sigma \mid$ there exists $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots \xrightarrow{e_i} \sigma_i$ and $\sigma = \sigma_i$ and $\mathrm{dst}(e_i) = n\}$ for all initial configurations $\sigma_0 \in \Sigma$.*

Intuitively, $[\![G_D]\!](n)(\sigma_0)$ is the set of configurations that are encountered at node n when a program G_D is on the execution starting at σ_0. Such an execution is *deterministic*, but it may pass through node n multiple times, each time with a distinct configuration. Plainly, a node in a cyclic path can appear more than once in an execution.

4.2 From DFL to Path Expressions

As previously shown, a DFL program defines possible (finite) paths as a subset of $(EXPR \times SSTMT)^*$, following the regular pattern of its flow graph. Given a flowgraph G_D with a finite set of edges, we define a set of alphabet for the Path Expressions (PE) associated with G_D as $\Omega = \{(\textbf{assume}(bp(e)), \mathit{eff}(e)) \mid e \in Edges\}$. Similar to the classical paradigm on finite automaton and regular expression, we define path expressions (PE) as an equivalent representation to DFL.

Definition 2. *The syntax of path expressions are given as*

$$R := \emptyset \mid \epsilon \mid \omega \mid R_1 + R_2 \mid R_1 \cdot R_2 \mid R^*$$

where $\omega \in \Omega$.

Let $PE(\Omega)$ be the set of path expressions given finite Ω. Plainly, a path expression is a regular expression that takes the pairs of branch predicate and side effect over edges as its alphabet.

Given a flowgraph $(Nodes, Edges, n_0, n_x)$ from a DFL program G_D, and starting with an initial configuration σ_0, a sequence of edges $\pi = e_1 e_2 \ldots e_k$ in $Path(n_0, n_x)$ describes a possible path that an execution may go through. If σ_0 does take transitions via π (as inferred by rule 1), then such a path π is unique for σ_0, by the *well-formedness* of DFL programs, since otherwise there would exist a node with two outgoing transitions both enabled at some configuration. However, it is also possible that no path in $Path(n_0, n_x)$ derives a concrete execution from σ_0, which can be interpreted as *non-termination* of the program given input σ_0.

On the syntactical level, we define the regular language induced by G_D as generated by treating n_0 as the initial state and n_x as the accept state, which gives a regular language $L(G_D) = Path(n_0, n_x)$. We can choose from various classical algorithms that translate finite automata to regular expressions (e.g., in the textbook of Hopcroft et al. [22], and Tarjan [20]), such that a path expression precisely represents the language $L(G_D)$ given by G_D.

The semantics of path expressions can be defined as a mapping from configurations to sets of configurations. Let R be a Path expression representing language L, then $[\![R]\!]^{PE} : \Sigma \to \mathcal{P}(\Sigma)$ is defined by $[\![R]\!]^{PE}(\sigma) = \{\sigma' \in \Sigma \mid \text{there exists } w \in L : \sigma \xrightarrow{w} \sigma'\}$. As the translations from finite automata to regular expressions are standard, we acquire the following semantic equivalence over the translations.

Lemma 1. *Let R_0 represent the language $L(G_D)$ of a DFL program G_D. For all $\sigma_0, \sigma \in \Sigma$, $\sigma \in [\![R_0]\!]^{PE}(\sigma_0)$ iff there exists a finite execution in G_D that starts at σ_0 and terminates at σ.*

Proof. The transformation guarantees $L(G_D) = Path(n_0, n_x)$, and R_0 represents $Path(n_0, n_x)$. Thus, there exists a path π in $Path(n_0, n_x)$ that maps σ_0 to σ iff $\sigma_0 \xrightarrow{\pi} \sigma$, the latter equivalent to $\sigma \in [\![R_0]\!]^{PE}(\sigma_0)$ by definition. □

Note that σ_0 does not yield a terminating run if we have $[\![R_0]\!]^{PE}(\sigma_0) = \emptyset$ and, there exists no path $\pi \in Path(n_0, n_x)$ such that $\sigma_0 \xrightarrow{\pi} \sigma$ for all $\sigma \in \Sigma$, or equivalently, we will have an execution $\sigma_0 \xrightarrow{e_1} \sigma_1 \xrightarrow{e_2} \sigma_2 \ldots$ with $e_1 e_2 \ldots$ an infinite path in G_D which does not pass by n_x. Moreover, given G_D well-formed, $[\![R_0]\!]^{PE}(\sigma_0)$ is either empty (nonterminating) or a singleton set (terminating with a unique output). Combining with the semantics of DFL, we obtain the following.

Lemma 2. *If R_0 represents the language $L(G_D)$ of a DFL program G_D, then $[\![G_D]\!](n_x) = [\![R_0]\!]^{PE}$.*

4.3 From Path Expressions to NDL

We provide a scheme for converting a path expression into a semantically equivalent NDL program. As in the previous section, the semantics of path expressions are nondeterministic, as a function mapping configurations to sets of configurations, or contexts. This can be further lifted to a function from contexts to contexts, in the same style as NDL semantics. Here we define $[\![\overline{\cdot}]\!]$ as the lifted semantics by $[\![\overline{R}]\!]^{PE}(C) = \bigcup_{\sigma \in C} [\![R]\!]^{PE}(\sigma)$.

Definition 3. *Fixing a finite set Ω, a PE to NDL translation function K is a mapping*

$$K : PE(\Omega) \to STMT$$

where STMT is the set of NDL statements.

ϵ	$\mapsto$	**skip**
(bp, eff)	$\mapsto$	$bp; \mathit{eff}$
$R_1 + R_2$	$\mapsto$	**choose**$(K(R_1), K(R_2))$
$R_1 \cdot R_2$	$\mapsto$	$K(R_1); K(R_2)$
R^*	$\mapsto$	**repeat**$(K(R))$

Figure 8. Translation to NDL

The translation rules for function K are given in Figure 8. Function K maps a path expression of a DFL program to an NDL statement. In this conversion process, a basic component (bp, eff) of an expression is mapped to an NDL statement consisting of the assertion bp and the simple statement eff. Remember the **assume** statements are already embedded in the structure of bp in path expressions. The sum of two regular expressions is mapped to a **choose** statement. The dot operator "·" is translated into sequential composition. The Kleene star "$*$" represents a cycle and is mapped to a **repeat** statement.

Lemma 3. *(PE to NDL equivalence) Let R be a path expression. Then,*

$$[\![\overline{R}]\!]^{PE} = \mathcal{S}[\![K(R)]\!]$$

Proof. By induction on the structure of R.
Base Case: The case when $R = \epsilon$ is trivial. If R is of the form $(\textbf{assume}(\mathit{Expr}), \mathit{SStmt})$. Let C be a context then

$$\begin{aligned}&[\![\overline{(\textbf{assume}(\mathit{Expr}), \mathit{SStmt})}]\!]^{PE}(C)\\ &\quad= \{\mathcal{S}[\![\mathit{SStmt}]\!](\sigma) \mid \sigma \in C \wedge \mathcal{E}[\![\mathit{Expr}]\!](\sigma) \neq 0\}\\ &\quad= \mathcal{S}[\![\textbf{assume}(\mathit{Expr}); \mathit{SStmt}]\!](C)\\ &\quad= \mathcal{S}[\![K(R)]\!](C).\end{aligned}$$

Induction Step:

- Suppose R is of the form $R_1 + R_2$ and let C be a context, then

$$\begin{aligned}\llbracket\overline{R_1 + R_2}\rrbracket^{PE}(C) &= \llbracket\overline{R_1}\rrbracket^{PE}(C) \cup \llbracket\overline{R_2}\rrbracket^{PE}(C)\\ &= \mathcal{S}\llbracket K(R_1)\rrbracket(C) \cup \mathcal{S}\llbracket K(R_2)\rrbracket(C)\\ &= \mathcal{S}\llbracket\textbf{choose}(K(R_1), K(R_2))\rrbracket(C).\end{aligned}$$

- Suppose R is of the form $R_1 \cdot R_2$ and let C be a context, then

$$\begin{aligned}\llbracket\overline{R_1 \cdot R_2}\rrbracket^{PE}(C) &= \llbracket\overline{R_2}\rrbracket^{PE}(\llbracket\overline{R_1}\rrbracket^{PE}(C))\\ &= \mathcal{S}\llbracket K(R_2)\rrbracket(\mathcal{S}\llbracket K(R_1)\rrbracket(C))\\ &= \mathcal{S}\llbracket K(R_1); K(R_2)\rrbracket(C).\end{aligned}$$

- Suppose R is of the form R^* and let C be a context, then

$$\begin{aligned}\llbracket\overline{R^*}\rrbracket^{PE}(C) &= \llbracket\overline{R^0 + R^1 + R^2 + \ldots}\rrbracket^{PE}(C)\\ &= \bigcup_{i\in\mathbb{N}} \llbracket\overline{R^i}\rrbracket^{PE}(C)\\ &= \bigcup_{i\in\mathbb{N}} \mathcal{S}\llbracket K(R)^i\rrbracket(C)\\ &= \mathcal{S}\llbracket\textbf{repeat}(K(R))\rrbracket(C),\end{aligned}$$

  where $R^0 = \epsilon$ and $R^{i+1} = R^i \cdot R$ for a path expression R, and $K(R)^0 = \textbf{skip}$, and $K(R)^{i+1} = K(R)^i; K(R)$ for an NDL statement $K(R)$.

□

Now the equivalence between DFL and NDL is established, i.e., if a DFL program evaluates an input σ_0 to a configuration while stopping at a node in the flowgraph, then the translated NDL contains the same value in its final context given σ_0, and vice versa. We present the straightforward result in the following theorem.

Theorem 1. *Let R_0 represent the language $L(G_D)$, we have*

$$\llbracket G_D\rrbracket(n_x) = \mathcal{S}\llbracket K(R_0)\rrbracket$$

Proof. By combining Lemma 2 and Lemma 3, we have that for all $\sigma_0 \in \Sigma$, we have $\llbracket G_D\rrbracket(n_x)(\sigma_0) = \llbracket R_0\rrbracket^{PE}(\sigma_0) = \llbracket\overline{R_0}\rrbracket^{PE}(\{\sigma_0\}) = \mathcal{S}\llbracket K(R_0)\rrbracket(\sigma_0)$. □

4.4 Compact Representations of PE and NDL

We implement the classical algorithm of Tarjan [20], which analyzes the connectivity of a graph, and build regular expressions of the set of edges based on the structure imposed by strongly connected components and the efficient re-use of regular expressions using union-find data structures.

$$\begin{aligned}&\emptyset + R = R\\ &R \cdot \emptyset = \emptyset = \emptyset \cdot R\\ &R \cdot \epsilon = R = \epsilon \cdot R\\ &\epsilon^* = \epsilon\\ &\emptyset^* = \epsilon\\ &R + R = R\\ &R^* \cdot R^* = R^*\\ &R \cdot R^* = R^* \cdot R\\ &(R^*)^* = R^*\end{aligned}$$

Figure 9. Identities for Regular Expressions used by Tarjan

In order to reduce the overhead in the generation of regular expression we apply algebraic rules to remove redundant structures, e.g. those by Tarjan [21], as shown in Figure 9.

$$\begin{aligned}&\textit{Prog} \longrightarrow \textit{Decl} \quad \textit{Stmt}\\ &\textit{Stmt} \longrightarrow \textbf{call } P\\ &\textit{Decl} \longrightarrow \epsilon \mid \textbf{proc } P\ \{\textit{Stmt}\}; \textit{Decl}\end{aligned}$$

Figure 10. The extended syntax for NDL

Benchmark	cfgs	nds	edgs
400.perlbench	1113	50788	83155
401.bzip2	21	1642	2647
403.gcc	2626	133797	247279
429.mcf	2	280	459
445.gobmk	2103	25881	39867
456.hmmer	92	5420	8327
458.sjeng	54	3362	5562
462.libquantum	8	1084	1735
464.h264ref	212	11349	17544
471.omnetpp	1597	34213	52119
473.astar	34	770	1153
483.xalancbmk	8708	105322	151921
all	16570	373908	611768

Table 1. Problem size of Spec2006 benchmark suite

We then extend the syntax of NDL to make the nondeterministic programs more concise, as shown in Figure 10. An NDL program is now started with a declaration consisting of zero or more procedures that are to be used in the main program. The translation K from RE to NDL is also extended, in the way that for each substructure R, we create a new declaration **proc** R' $\{K(R)\}$ at the beginning of the program, and interpret the site where R is launched by **call** R'. Note the **call** relation between path expressions (or structures) is acyclic. As we are to show in the following section, such a representation of a PE can be exponentially more concise than its actual size, for the redundant usage of a sub-PE is now replaced by a link to the site of the sub-PE.

5. Experiments

We have conducted preliminary experiments to translate large benchmarks from a deterministic flow language to non-deterministic structured languages. As a vehicle for our tests we used the LLVM compiler framework [14] that resembles a framework for compiling various high-level languages including C/C++, Objective-C, Ada, Fortran, Haskell, and Java bytecode to an intermediate low-level representation represented as a control-flow graph. In this experiment we ask the question whether the translation to non-deterministic programs is viable for real-world code. In particular, we ask the following two questions: (1) is the translation time from DFL to NDL viable, and (2) is the size of the generated NDL program in acceptable bounds.

We ran our experiments on a 64bit Linux computer with 16GB of RAM and two E5345 Intel CPUs with 2.3GHz. Although we conduct the experiments on a multi-core platform, our translation program exploits a single thread only. As a benchmark suite for our experiments, we have used the Spec CPU2006 Benchmark suite [7]. Using the LLVM 2.9 framework with a GCC front end, we obtain the problem sizes as given in Table 1, where the column *Benchmark* denotes the benchmarks taken from the Benchmark suite, *cfgs* the number of control flow graphs, *nds* the total number of nodes, and *edgs* the total number of edges.

We have used Tarjan's algorithm [20] to convert the CFG of the functions to an NDL representation in C/C++. Note that for the translation, sequences in basic blocks were compressed to a single super-statement that can have potential side-effects to several variables and may have a single **assume** statement. For the conversion from DFL to NDL, we have used one of the fastest known algorithms to transform a finite automata to a regular expression, which exhibits a worst-case complexity of $\mathcal{O}(m \cdot \alpha(m, n))$ where n is the number of nodes in the control-flow graph, m is the number of edges in the graph, and α is a very slow growing function (i.e.,

	Compact			Expanded		
Benchmark	rept	chse	seq	rept	chse	seq
400.perlbench	2180	31300	103968	2.1164e+19	5.56314e+19	7.22382e+19
401.bzip2	332	694	4068	137360426033304288	137360699285724336	961524095795415168
403.gcc	7024	109084	299428	1.08564e+20	1.68148e+20	2.15809e+20
429.mcf	58	123	730	278233	1533019	4250010
445.gobmk	1483	14606	49594	4.47128e+19	3.20687e+19	2.09956e+19
456.hmmer	456	2543	12147	124237627150955200	248894473563970816	704639121478513536
458.sjeng	264	1990	7347	4650914279140	26684661306023992	178078270770559712
462.libquantum	155	504	2472	29039508	115573277	347943562
464.h264ref	1595	4812	24836	4.70143e+19	5.35788e+19	5.63575e+19
471.omnetpp	794	18709	77346	664118694274	5270837914187761664	2636500292015777280
473.astar	110	307	1603	2170314	6466341	27204896
483.xalancbmk	6795	48512	185439	3.09579e+19	7.58877e+19	7.04882e+19
all	21246	233184	768978	2.52675e+20	3.90999e+20	4.4037e+20

Table 2. Sizes of NDL

the inverse Ackermann function). Tarjan's algorithm for converting a graph to path-expressions relies on the re-use of previously generated path-expressions. If the path-expressions are written out, the writing procedure may have exponential size (i.e., $\mathcal{O}(4^n)$) as outlined in [22]).

Since Tarjan's low worst-case runtime complexity, the first question of our experiment can be answered positively. The runtime of constructing NDL for the whole SPEC CPU2006 benchmark suite is conducted in less than 1 second on our test machine. The number of generated **repeat**, **choose**, and sequence statements are shown in the Table 2 under the column *Compact*. The labels *rept* denote the number of **repeat** statements, *chse* the number of **choose** statements, and *seq* the number of sequence statements. The *Compact* column represents the number of statements using abstraction, i.e., NDL compositions that are re-occurring are abstracted by a name and re-used. In contrast, no abstraction is used, the generation of NDL is infeasible as shown in the column *Expanded*.

6. Conclusion

We have presented a regular expression based technique for transforming a deterministic flowchart language DFL into a nondeterministic modelling language NDL. The translation procedure is purely syntactic, based on which we have shown that the semantics of DFL programs are preserved during the translation to NDL programs. Moreover our method preserves control-flow structure of the original programs and uses simple transformation rules which are independent of the original program size and structure. We conducted experiments on the Spec Benchmark Suite, which confirms that our approach is viable in practice.

In the future we are going to further explore the advantage of this translation, e.g., by applying the method as a preprocessor for verifying interesting properties in programs by model checking. Such a translation may also be used to study the semantic connections between deterministic and nondeterministic programs. As NDL has a richer semantics than deterministic flowcharts, the transformation proposed in the paper also works on flowcharts with nondeterministic semantics, e.g., by relaxing the *well-formedness* of DFL to allow nondeterministic transitions. Such a well-structured nondeterministic language has a natural correspondence to modelling languages of existing tools, such as PROMELA, the input language of the SPIN model checker [12]. Moreover, the level of nondeterminism in an NDL program may be increased via abstraction, provided that the input modelling program may be too large to be handled by existing automatic program verifiers.

Acknowledgments

This research was supported under Australian Research Council's Discovery Projects funding scheme (DP1096445) and Australian Research Council's Linkage Projects funding scheme (LP0989643).

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[2] C. Baier, J.-P. Katoen, and K. G. Larsen. *Principles of Model Checking*. The MIT Press, 2008.

[3] T. Ball, E. Bounimova, R. Kumar, and V. Levin. Slam2: static driver verification with under 4% false alarms. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 35–42, Austin, TX, 2010. FMCAD Inc.

[4] C. Böhm and G. Jacopini. *Flow diagrams, Turing machines and languages with only two formation rules*, pages 11–25. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[5] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[6] K. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publisher, 2004.

[7] S. P. E. Corporation. Spec cpu 2006. `http://www.spec.org`, 2006.

[8] P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall, 1981.

[9] D. Gopan and T. W. Reps. Lookahead widening. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466. Springer, 2006.

[10] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 375–385, New York, NY, USA, 2009. ACM.

[11] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In G. Levi, editor, *SAS*, volume 1503 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 1998.

[12] G. Holzmann. *The Spin Model Checker: primer and reference manual*. Addison-Wesley Professional, 2003.

[13] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[14] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-*

directed and runtime optimization, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] J. v. Leeuwen. *Handbook of Theoretical Computer Science Volume B: Formal Models and Semantics*. Elsevier Science Publishers B. V., 1994.

[16] J. Miecznikowski and L. J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *Compiler Construction, 11th International Conference (CC)*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127, Grenoble, France, 2002. Springer.

[17] P. H. Morris, R. A. Gray, and R. E. Filman. Goto removal based on regular expressions. *Journal of Software Maintenance*, 9:47–66, February 1997.

[18] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[19] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29, August 2007.

[20] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28: 594–614, July 1981.

[21] R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28:577–593, 1981.

[22] J. Ullman, J. Hopcroft, and R. Motwani. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison-Wesley, 2003.