Resolving Constrained Existential Queries over Context-Sensitive Analyses *

James Ezick
Department of Computer Science
Cornell University
Ithaca, New York 14853
ezick@cs.cornell.edu

4 June 2003

Abstract

A context-sensitive analysis is an analysis in which program elements are interpreted with respect to the context in which they occur. For analyses on imperative languages, this often refers to considering the behavior of called procedures with respect to the calling-stack contexts that precede them. Algorithms for performing or approximating these types of analyses make up the core of interprocedural program analysis and are pervasive; having applications in program optimization, checkpointing, and model checking. This paper presents an abstraction of a popular form of context-sensitive analysis based on iteratively encapsulating the cumulative effect of a recurring piece of code. Given an analysis fitting this abstraction, a technique is presented for resolving queries of the form: *Is there an occurring context, subject to a given stack-context constraint, in which a particular set of facts holds at a particular location?* This practical technique, based on manipulating regular languages, is capable not only of answering queries of this form, but also of generating a compact mechanism for dynamically applying the output of the analysis to contexts as they occur. A comprehensive example is presented along with performance data on a case study code. Finally, a selection of potential applications is discussed.

1 Introduction

Context-sensitive analyses are analyses in which the elements of a program abstraction are interpreted with respect to the context in which they occur. For imperative programs, the most common instances of these analyses are ones

*This work was supported by DARPA contract DAAH01-02-C-R120 and NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-0121401

in which the effect of procedure calls are considered in the context of their calling stack. Numerous program analyses rely on these techniques to aggregate information over the precise set of paths that the execution of a program may take. One popular approach to forward-flow analyses of this type is the so-called *second-order* approach. In this approach, first formalized by Sharir and Pnueli [9], the effect of a procedure call from its entry to each point in the procedure is encapsulated by a data-transforming *ϕ-function.* A *context* is a set of data facts that is assumed to hold at the entry of a procedure in an interprocedural control-flow graph. The function associated with each vertex in the graph then maps each possible context to the set of facts corresponding to the meet over all paths solution from the entry of the procedure to the program element associated with the vertex.

This analysis technique was developed as a means of generating a transform function for each procedure that completely encapsulates the data-flow effect of the procedure. Given these functions it is straightforward to determine the set of facts that hold at the the beginning of the program and, with a little extra work, at any other point. This analysis technique was an improvement over previous techniques since it allowed a meet over all paths solution to a data-flow analysis problem to be found in such a way that only those paths that respect the natural call and return structure of the program are considered. In this paper, a further refinement of this problem is considered. Here, given an analysis, a location in the program is provided along with a constraint on the set of stack-contexts leading to that location. Given this information, the question is asked whether or not a path to the location exists that both fulfills the stack-constraint and in which a specified set of data-flow facts holds. The ability to answer queries of this form has applications to program comprehension, model-checking, and application-level checkpointing.

This paper makes three contributions. First, an abstraction of context-sensitive analyses is presented that separates the data-flow facts from the calling contexts. This abstraction wholly subsumes the work of Sharir and Pnueli. Second, a practical technique based on intersecting regular languages is presented for resolving constrained existential queries. Given constraints cast as regular expressions, this technique does not require the computation of any additional fixpoints. Finally, this paper demonstrates how the solution to a such a query can be encapsulated as the accepting automata of a regular language so that it can be dynamically applied to occurring contexts.

The remainder of this paper is organized as follows: Section 2 introduces the abstraction of context-sensitive analyses. The technique for resolving and applying the output of constrained existential queries is presented in Section 3. A comprehensive example is provided in Section 4 and a case study examining a popular physics code for n-body simulation is presented with performance data in Section 5. Finally, Section 6 discusses a selection of potential applications as well as related work.

2 Context-Sensitive Analyses

The analysis abstraction defined in this section assumes a standard single-entry, single-exit, unrestricted hierarchical state machine [3] as the model. This model is a generalization of the interprocedural control flow graph over which interprocedural analyses are usually presented. It is sufficient to encode the call and return structure of imperative programs, including recursive programs. Specifically, this presentation takes advantage of the terminology from those graphs; there are designated *call* and *return*-vertices in each *procedure* graph. Further, each procedure begins with an *entry*-vertex and ends with an *exit*-vertex. A *call-edge* connects a call-vertex to an entry-vertex and likewise a *return-edge* connects an exit vertex with a return-vertex. The set of call-edges is designated Σ. Finally, there is a designated *main* procedure whose entry and exit-vertices represent the beginning and end of the model, respectively.

Definition 1 *A* **context-sensitive analysis** *is an ordered quintuple* $(\mathcal{C}, \mathcal{X}, \Gamma, \Phi, \kappa)$*, where* $\mathcal{C}$ *is a finite set of* **contexts**, $\mathcal{X}$ *is a set of* **properties**, $\Gamma : \Sigma \mapsto (\gamma : \mathcal{C} \mapsto \mathcal{C})$ *is a collection of* **context-transformers** *that associates to each element of the set of call-edges,* Σ*, a function,* γ*, mapping a context to a context,* $\Phi : \mathcal{V} \mapsto (\phi : \mathcal{C} \mapsto 2^{\mathcal{X}})$ *is a collection of* **property-transformers** *that associates to each element of the vertex set of the model,* $\mathcal{V}$*, a function,* ϕ*, mapping a context to a subset of the set of properties, and* $\kappa \in \mathcal{C}$ *is the* **initial context**.

Definition 2 *A* **stack-context** *is a finite (possibly empty) sequence of the elements of* Σ*. A stack-context,* $\bar{\sigma} = \sigma_0 \ldots \sigma_n$*, is a* **valid stack-context** *for* $v \in \mathcal{V}$ *if the source of* σ_0 *is a call-vertex in main, for each* $i : 0 \leq i < n$ *the target of* σ_i *is the entry-vertex of the procedure containing the source of* σ_{i+1}*, and finally, the target of* σ_n *is the entry of the procedure containing the vertex* v*.*

Note that for a vertex, v, in a model exhibiting recursion, the set of valid stack-contexts for v can be infinite.

Definition 3 *Given a context-sensitive analysis, the* **stack-context transformer**, $\Gamma^* : \Sigma^* \mapsto \mathcal{C}$*, induced by the analysis maps a stack-context to a context in the analysis,*

$$\Gamma^*(\sigma_0 \ldots \sigma_n) = [\Gamma(\sigma_n) \circ \ldots \circ \Gamma(\sigma_0)](\kappa).$$

Definition 4 *Given a context-sensitive analysis,* $v \in \mathcal{V}$*, and* $\bar{\sigma}$*, a valid stack context for* v*, the* **solution** *for vertex* v *in stack-context* $\bar{\sigma}$ *is the subset of properties*

$$\rho(v, \bar{\sigma}) = [\Phi(v)](\Gamma^*(\bar{\sigma})).$$

This abstraction subsumes the Sharir and Pnueli framework. In that framework, the set of contexts is equivalent to the power set of the the set of properties. In their formulation, the context-transformer associated with each call-edge is equivalent to the property-transformer associated with the call-vertex that is the source of that call-edge. Also, instances of their framework require that κ be defined as the context equivalent to the empty set of properties.

While it is necessary that both Γ and Φ be total functions, it is not strictly necessary that the context-sensitive analysis provide a total function for each element in their respective ranges. Specifically, given $\sigma \in \Sigma$ and $c \in \mathcal{C}$, the analysis need only provide a result for $[\Gamma(\sigma)](c)$ if there is some valid stack-context, $\bar{\sigma} \cdot \sigma$, for the target of σ such that $\Gamma^*(\bar{\sigma} \cdot \sigma) = c$. Likewise, given $v \in \mathcal{V}$, $\Phi(v)$ need only be provided for $[\Phi(v)](c)$ if there is some valid stack-context, $\bar{\sigma}$ for v such that $\Gamma^*(\bar{\sigma}) = c$. This relaxation of the totality requirement allows this abstraction to include demand analyses in which transformers are only defined for contexts for which there is a corresponding valid stack-context. However, the constructions of Section 3 assume that the functions are total, with the assumptions that $\gamma(c) = \kappa$ and $\phi(c) = \emptyset$ if γ or ϕ is not defined on c.

This abstraction works with both forward- and backward-flow analyses. The terminology associates the context-transformers with the call-edges making it more natural for discussing forward-flow analyses but since there is a natural one-to-one correspondence between the call and return-edges in a single-entry, single-exit unrestricted hierarchical state machine, there is no need to modify the definition to describe backward-flow analyses. Intuitively, for analyses in which contexts and sets of properties are used interchangeably, κ refers to the set of properties that holds before the entry of main in a forward-flow analysis and to the set of properties that hold after the exit of main in an backward-flow analysis.

3 Resolving Constrained Existential Queries

Given a context-sensitive analysis as described in Section 2, this section demonstrates how to resolve a constrained existential query over that analysis. A constrained existential query is a query of the form: *Given a vertex, a stack-context constraint, and a solution constraint, does there exist a valid stack-context for the vertex, satisfying the stack-context constraint, such that the corresponding solution is an element of the solution constraint?* Queries of this form have applications in many domains including program comprehension, model-checking, and application-level checkpointing. In addition to resolving such queries, this section describes how the the answer to the query can be dynamically applied to specifically occurring stack-contexts. The technique for resolving and applying queries of this form revolves around manipulating regular languages.

3.1 Valid Stack-Contexts as Regular Languages

The set of valid stack-contexts for a vertex, v, can be described as a regular language over the set of call-edges, Σ. This fact is trivially demonstrated by constructing a finite automaton that accepts precisely the set of stack-contexts that are valid for v [7].

Theorem 1 *Given v, a vertex in a single-entry, single-exit, unrestricted hierarchical state machine, U, the set of valid stack-contexts for v is precisely accepted by a deterministic finite automaton, $M = (\mathcal{P} \cup \epsilon, \Sigma, main, \delta, A)$, where, $\mathcal{P} \cup \epsilon$ is the set of procedures of U, $\mathcal{P}$, plus a dead state, ϵ, Σ is the set of call-edges in U, the procedure main in U is the initial state, $\delta(q_s, \sigma) = q_t$ where q_t is the procedure in U containing the target of σ if and only if $q_s \neq \epsilon$ and q_s is the procedure containing the source of σ, otherwise $q_s = \epsilon$, and A is the singleton set of states consisting only of the procedure containing v in U.*

Proof 1 *If $\bar{\sigma} = (\sigma_0 \ldots \sigma_n)$ is a valid stack-context for v then the source of σ_0 is main, the initial state of M. For each σ_i in $\bar{\sigma}$, $\delta(q_s, \sigma_i) = q_t$ in M, where q_s is the procedure containing the source of σ_i and q_t is the procedure containing the target of σ_i since σ_i is an element of a valid stack-context for v. Finally, the target of σ_n must reside in the procedure containing v and this is an accepting state of M. Thus, M accepts $\bar{\sigma}$.*

Likewise, if M accepts $\bar{\sigma}$, then there is a sequence of states, $q_0 \ldots q_n$, that accepts $\bar{\sigma}$. Since no accepting sequence contains ϵ, each state in the sequence is a procedure in U. By definition of M, q_0 is the procedure main in U. For each subsequence $q_i q_{i+1}$ in the accepting sequence $\delta(q_i, \sigma_i) = q_{i+1}$ thus, σ_i is a call-edge whose source is a call-vertex in the procedure q_i and whose target is the entry-vertex of the procedure q_{i+1}. Finally, since q_n is the single accepting state, it must be the procedure containing v. Hence, $\bar{\sigma}$ is a valid stack-context for v. $\Box$

Weihl [10] has shown that constructing the call graph (which is required to build the DFA of Theorem 1) is PSPACE-hard for recursive programs with function pointers. However, if the precise call graph is not available, a graph that conservatively over-estimates the set of possible calls can be used. M accepts stack-contexts corresponding to paths from *main* to the procedure containing v in the supplied call graph.

3.2 Stack-Context Constraints as Regular Languages

For posing an existential query over a context-sensitive analysis, a stack-context constraint is a regular expression over the set of call edges, Σ, in a single-entry, single-exit unrestricted hierarchical state machine, U. For convenience, the following notation is introduced as extensions to the stack constraint expression alphabet.

Notation 1 *Given P, a procedure in U, Π_P refers to the regular expression $\tau_0 + \ldots + \tau_n$ where the set $T = \{\tau_0, \ldots, \tau_n\}$ is the subset of Σ such that $\sigma \in T$ if and only if the target of call-edge σ is the entry-vertex of P in U.*

This notation captures the notion of any call to to a specified function.

Notation 2 *The symbol Ω refers to the regular expression $(\sigma_0 + \ldots + \sigma_n)^*$, where $\Sigma = \{\sigma_0, \ldots, \sigma_n\}$.*

Taken alone, Ω refers to the universal language of stack-contexts. However, in expressing a stack-context constraint, Ω is useful as a wildcard literal standing for an arbitrary sequence of call-edges. For example, if one wanted to express the stack-context constraint "contains a call to foo followed sometime later by a call to either bar_1 or bar_2" that could be represented using the notation introduced above by the regular expression $\Omega\Pi_{foo}\Omega(\Pi_{bar_1} + \Pi_{bar_2})\Omega$.

Although other constraint languages exist, regular expressions are powerful enough to express most of the types of constraints that are of interest for this type of query. They can encode fixed sequences of calls as well as any criteria based on a finite branching decision tree of calls. The Kleene-star operator allows constraints to include the notion or zero or more calls to a recursive function. Finally, regular expressions are closed under intersection, union, and complement so they are powerful enough to express the conjunction, disjunction and negation of constraints.

3.3 Solution Constraints as Regular Languages

A solution constraint for an existential query is a decidable subset of the power set of the properties of a context-sensitive analysis. Given such a constraint, there exists a regular language, again over the alphabet of call-edges, Σ, that precisely includes the set of stack-contexts generating an element of the constraint as a solution for a vertex v. This is again demonstrated by constructing a state machine precisely accepting such a language.

Theorem 2 *Given v, a vertex in a single-entry, single-exit, unrestricted hierarchical state machine, U, a context-sensitive analysis, $(\mathcal{C}, \mathcal{X}, \Gamma, \Phi, \kappa)$ over U, and a solution constraint Δ, a decidable subset of $2^{\mathcal{X}}$, the set of stack-contexts, $\bar{\sigma}$, for v such that $\rho(v, \bar{\sigma}) \in \Delta$, is precisely accepted by a deterministic finite automaton, $M = (\mathcal{C}, \Sigma, \kappa, \delta, A)$, where $\mathcal{C}$ is a set of contexts in the analysis, Σ is the set of call-edges in U, κ is the initial context of the analysis, $\delta(c_i, \sigma) = [\Gamma(\sigma)](c_i)$, and A is the set of contexts, c, such that $[\Phi(v)](c) \in \Delta$.*

Proof 2 *Given a stack-context, $\bar{\sigma} = \sigma_0 \ldots \sigma_n$, by definition of M, $\delta(\delta(\ldots\delta(\kappa, \sigma_0)\ldots, \sigma_{n-1}), \sigma_n) = \Gamma(\sigma_n) \circ \ldots \circ \Gamma(\sigma_0) = \Gamma^*(\bar{\sigma})$. Thus, if $\rho(v, \bar{\sigma}) = [\Phi(v)](\Gamma^*(\bar{\sigma})) \in \Delta$ then M terminates in a state $c = \Gamma^*(\bar{\sigma})$. This is an accepting state.*

Likewise, if M accepts $\bar{\sigma}$ then M terminates in a context, c, such that $[\Phi(v)](c) = [\Phi(v)](\Gamma^*(\bar{\sigma})) = \rho(v, \bar{\sigma}) \in \Delta$. $\Box$

While the algorithm requires that a solution constraint be a decidable subset of the power set of the set of properties, this is not a practical limitation. Any such subset can be made decidable by intersecting it with the finite set of property sets that are mapped to by the the finite number of (vertex, context) pairs. Also, it is not generally necessary to explicitly generate Δ. For instance, it is adequate to define a constraint as "the set of solutions containing property $p \in \mathcal{X}$".

3.4 Resolving a Constrained Existential Query

Given a context-sensitive analysis with call-edge alphabet Σ, let L_v be the regular language corresponding to the valid stack-context requirement, let L_c be the regular language corresponding to the stack-context constraint, and let L_Δ be the regular language corresponding to the solution constraint. Each language is contained in Σ^*. Then, a solution exists to the constrained existential query if and only if

$$\mathcal{L} = L_c \cap L_v \cap L_\Delta \neq \emptyset.$$

Recall that testing the emptiness of a regular language reduces to testing the reachability of an accepting state from the initial state in the corresponding deterministic finite automaton. The number of states in the intersection automaton, $\mathcal{L}$, is bounded above by the the product of the number of states in the automata accepting L_v, L_c, and L_Δ. This automaton, which can be minimized [6], exactly encapsulates the set of examples to the existential query. Any stack-context that it accepts is valid and meets both the stack-context constraint as well as the solution constraint.

3.5 Applications to Dynamic Data-Flow Analysis

Traditionally, data-flow analyses are static analyses. They are performed prior to any knowledge about the actual execution sequence in the model. However, given $\mathcal{L}$, the language encapsulating the solution to an existential query, a finite automaton, $M_\mathcal{L}$, accepting $\mathcal{L}$ can be associated with the vertex, v, from the stack-context constraint in the model. The execution of the model can then be monitored, and when v occurs, $M_\mathcal{L}$ can be applied to the specific stack-context, $\bar{\sigma}$, that led to v. If $M_\mathcal{L}$ accepts $\bar{\sigma}$ then the set of properties that holds at v in *the specific context in which v has occurred* is an element of Δ, the solution constraint. In this way, a static context-sensitive analysis can be encapsulated so as to provide a dynamically precise solution. This capability has numerous applications to program optimization, as it creates a mechanism for choosing among multiple dynamic behaviors based upon the specific calling context. Further, these behaviors can be distinguished by properties that range arbitrarily far up the the stack. Previously, this effect could only be partially emulated by creating specialized clones of procedures and modifying specific call-sites to redirect execution to those clones.

3.6 Constrained Universal Queries

This same technique can be applied to resolving constrained universal queries. A universal query asks whether, given a vertex v, every valid stack-context for v that meets a given stack constraint corresponds to an element in a supplied solution constraint. Using the same set of regular languages as for existential queries, a universal query is valid if and only if, for all $\bar{\sigma} \in \Sigma^*$, $\bar{\sigma} \in (L_v \cap L_c) \rightarrow \bar{\sigma} \in L_\Delta$. More concisely, the query is valid if and only if

$$(L_v \cap L_c) \cap L'_\Delta = \emptyset$$

```
     Global Integer G

v0:  Procedure A() {              v10: Procedure C() {
v1:    if (read_choice()) {       v11:   print(G)
s2:      C()                      v12:   G := 1
s3:      B()                      v13: }
       }
v4:  }                            v14: Procedure main() {
                                  v15:   G := 0
v5:  Procedure B() {              v16:   if (read_choice()) {
v6:    if (read_choice()) {       s17:     A()
s7:      A()                             } else {
s8:      C()                      s18:     B()
       }                                 }
v9:  }                            v19: }
```

Figure 1: A Sample Imperative Program with Mutual Recursion

As in the previous case, this intersection language exactly captures the set of counter-examples to the universal query and can be applied in the same way to a dynamic data-flow analysis.

3.7 Multi-Entry, Multi-Exit Models

The abstraction of context-sensitive analyses presented in Section 2 and the proofs presented in this section are framed for single-entry, single-exit models since these are by far the most common models used in program analysis. However, the technique presented in these sections can be directly extended to multi-entry, multi-exit models. In these models, the procedure graphs that make up the hierarchical state machine can have multiple entry and exit points. For forward-flow analyses over these models, a context-sensitive analysis must return a ϕ-function for each vertex for each entry of the procedure containing that vertex; an (entry, vertex) pair. Likewise, valid stack-contexts are specific to (entry, vertex) pairs. The set Σ still consists of the set of call-edges. For backward-flow analyses, Σ is defined as the set of return-edges. This distinction is necessary since call- and return-edges are no longer guaranteed to be in one-to-one correspondence. Valid stack-contexts and ϕ-functions are then associated with (exit, vertex) pairs.

4 A Comprehensive Example

Figure 1 introduces an imperative program that exhibits mutual recursion. The labels correspond to vertices in the hierarchical state machine induced by the the program's control-flow, with the s-subscripted labels also referring to the call-edges by their source. The s-subscripted labels identify the elements of Σ. A

v	$[\Gamma(v)](\alpha)$	$[\Gamma(v)](\beta)$	$[\Phi(v)](\alpha)$	$[\Phi(v)](\beta)$
v_0			{p}	{p}
v_1			{p}	{p}
s_2	β	β	{p}	{p}
s_3	α	β	{p}	{p}
v_4			$\emptyset$	{p}
v_5			{p}	{p}
v_6			{p}	{p}
s_7	β	β	{p}	{p}
s_8	α	β	{p}	{p}
v_9			$\emptyset$	{p}
v_{10}			{p}	{p}
v_{11}			{p}	{p}
v_{12}			$\emptyset$	{p}
v_{13}			$\emptyset$	{p}
v_{14}			{p}	{p}
v_{15}			{p}	{p}
v_{16}			{p}	{p}
s_{17}	α	β	{p}	{p}
s_{18}	α	β	{p}	{p}
v_{19}			$\emptyset$	{p}

The s-subscripted vertices also identify the elements of Σ by call-edge source.

Figure 2: Context-Sensitive Analysis $(\{\alpha, \beta\}, \{p\}, \Gamma, \Phi, \alpha)$

context-sensitive analysis over the program is provided in Figure 2. Intuitively, the property p denotes that the print statement in procedure C, labeled v_{11}, may occur in the future. Contexts α and β distinguish between assumptions about whether p holds at the exit of a procedure, with β being the assumption that it does hold.

Given this program and analysis, a constrained existential query is posed:

Is their a path in the program through the assignment at the vertex labeled v_{12} to the print statement at the vertex labeled v_{11} where the call stack at v_{12} begins with a call to procedure $A()$ followed directly by a call to procedure $B()$?

This query focuses on constraining stack-contexts for occurrences of the statement at the vertex labeled v_{12}. Figure 3 shows the deterministic finite automata described by Theorem 1 as accepting the language L_v; the language of valid stack-contexts for v_{12}. The query requires that contexts begin with consecutive calls to procedures $A()$ and $B()$.

This requirement is formalized as a stack-context constraint, $\Pi_A \Pi_B \Omega$. Recalling the notation of Section 3.2, this expands to the regular expression $(s_7 + s_{17})(s_3 + s_{18})(s_2 + s_3 + s_7 + s_8 + s_{17} + s_{18})^*$ over Σ. Finally, that the print statement can occur following an occurrence of vertex v_{12} corresponds to the property p holding at v_{12} in some context. This defines the solution constraint as the set of subsets of properties that include p. Specifically, $\Delta = \{\{p\}\}$. Figure 4 shows the DFA described by Theorem 2 as accepting the solution constraint language, L_Δ.

Having formalized the constraints, the solution to the existential query is captured by the intersection language $\mathcal{L} = L_c \cap L_v \cap L_\Delta$. The minimal DFA accepting this language is provided in Figure 5. From inspection of the DFA it

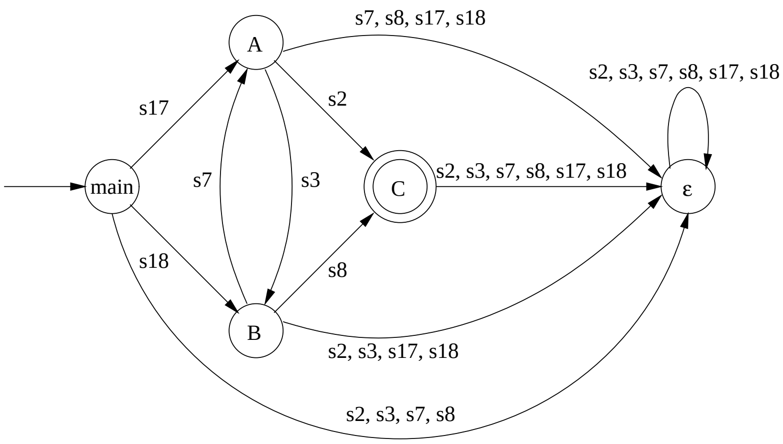

Figure 3: DFA Accepting Valid Stack-Contexts for v_{12}

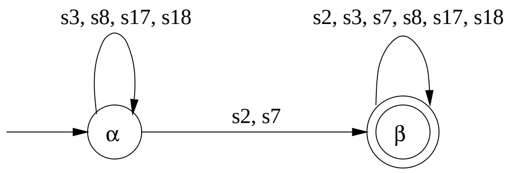

Figure 4: DFA Accepting Stack-Contexts $\bar{\sigma}$, such that $[\Phi(v_{12})](\Gamma^*(\bar{\sigma})) = \{p\}$

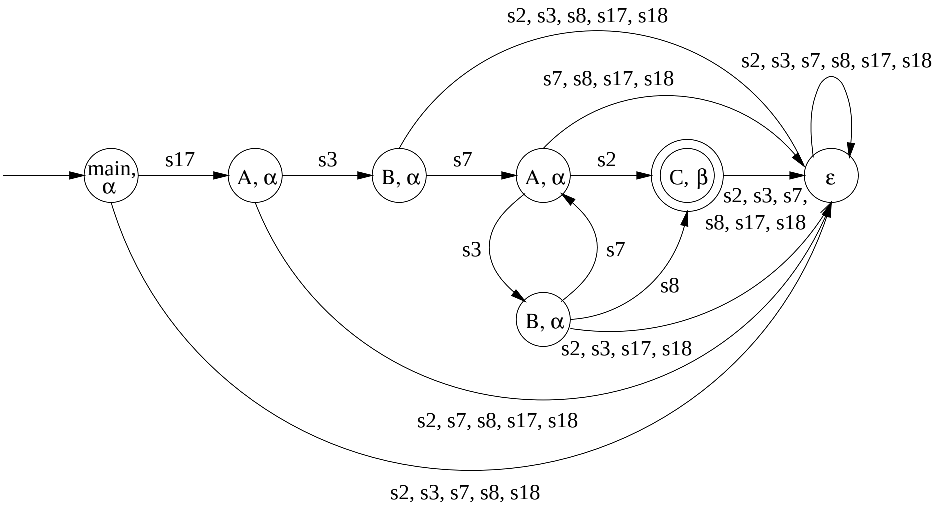

Figure 5: Minimal DFA Accepting Solutions to the Existential Query

Query Constraints			Automaton States				CPU
Location	Context	Solution	M_{L_c}	M_{L_v}	M_{L_Δ}	$M_{\mathcal{L}}$	Time (ms)
maketree:45	Ω	$\{S \mid root \notin S\}$	1	6	3	6	350
makecell:109	Ω	$\{S \mid root \in S\}$	1	8	5	8	425
savestate:328	$\Pi_{output}\Pi_{savestate}$	$\{S \mid Saved \subset S\}$	3	6	2	5	600
cputime:40	$\Omega\Pi_{stepsystem}\Omega$	$\{S \mid \|S\| > 35\}$	2	9	9	7	800
gravsum:212	$\Omega\Pi_{stepsystem}\Omega$	$\{S \mid size(S) > 640\, bytes\}$	2	10	5	12	975
allocate:27	$\Omega\Pi_{makecell}\Omega$	$\{S \mid root \notin S \vee bodytab \notin S\}$	2	15	20	8	1350

Treecode: $|\mathcal{V}| = 2496$, $|\mathcal{P}| = 68$, $|\Sigma| = 226$, $|\mathcal{C}| = 300$, and $|\mathcal{X}| = 743$

Figure 6: Examples of Constrained Existential Queries

is clear that a solution to the existential query exists. If it did not, the minimal DFA would have degenerated to a single non-accepting state. The call-sequence $\bar{s} = s_{17}s_3s_7s_2$ is a valid stack-context for vertex v_{12} that meets both the stack-context constraint, $\Pi_A\Pi_B\Omega$, as well as the solution constraint. That $\bar{s}$ meets the solution constraint can also be independently verified by checking that

$$\rho(v_{12}, \bar{s}) = [\Phi(v_{12}) \circ \Gamma(s_2) \circ \Gamma(s_7) \circ \Gamma(s_3) \circ \Gamma(s_{17})](\alpha) = [\Phi(v_{12})](\beta) = \{p\} \in \Delta.$$

Further, the universal query subject to the same constraints could have been solved by construction the intersection language $(L_c \cap L_v) \cap L'_\Delta$. This language is not empty since $\bar{s}' = s_{17}s_3s_8$ is a valid stack-context for v_{12} that meets the stack-context constraint but does not meet the solution constraint since

$$\rho(v_{12}, \bar{s}') = [\Phi(v_{12}) \circ \Gamma(s_8) \circ \Gamma(s_3) \circ \Gamma(s_{17})](\alpha) = [\Phi(v_{12})](\alpha) = \emptyset \notin \Delta.$$

Finally, if the DFA of Figure 5 was bound to executions of the statement associated with vertex v_{12}, the existential query could be answered as a specific query about the current stack-context. Precisely, the DFA would accept if and only if the current stack-context began with a call to $A()$ followed by a call to $B()$ and there existed a possible future path in which the statement associated with vertex v_{11} would be executed.

5 A Case Study

Treecode [1, 2] is an implementation of a popular algorithm for performing n-body simulation based on constructing oct-trees. A context-sensitive liveness analysis is supplied for this code. This analysis, given a statement and a context, returns the set of program variables that may be used from that point before they are redefined or the program terminates. Properties in this analysis correspond to the liveness of the individual program variables, and the contexts correspond to sets of variables that are possibly live at the end of one or more procedures.

Figure 6 shows performance information on a selection of constrained existential queries made over this analysis. The queries are meant to answer meaningful, non-trivial, questions about the liveness sets associated with different points in the code. Variables *root* and *bodytab* are the entry points for the data structures containing the oct-tree and the body position and velocity vectors, respectively. The code, as provided, contains a routine, *savestate*, for outputting its state to a file and *Saved* refers to the set of variables that comprise that recorded state. Locations are provided as a procedure name followed by a source code line number. *Automaton States* refers to the number of states in the minimized automaton accepting each of languages presented in Section 3. The CPU performance data was generated from a Scheme implementation of the various regular language routines executing on a 2.0 GHz Pentium IV with 1 GB of RAM running Windows XP Professional. Times are presented in milliseconds. All of the queries returned true.

Notice that the solution-constraints take advantage of the fact that Δ can be any decidable subset of the property power set, including sets that rely on properties beyond the scope of the original analysis. For the stack-context constraint, the number of states in the accepting automaton is dependent only on the constraint regular expression. For the valid-context automaton, the number of states is bounded above by one plus the number of procedures that occur on paths in the call-graph from *main* to the procedure containing the prescribed location. The minimal number of states required for the solution constraint automaton is bounded above by the number of distinct contexts that are Γ^* for some stack context. To increase efficiency, the states for the automaton accepting L_Δ can be restricted to the contexts that are the output of the stack-context transformer for stack-contexts comprised only of the procedures necessary for the construction of the automaton accepting L_v. The results presented in this section reflect this optimization. Finally, the number of states required for the automaton accepting $\mathcal{L}$ is bounded above by the product of the number of states of the automata accepting the constraint languages. As the performance data in Figure 6 illustrates, this is a practical technique for resolving actual queries over context-sensitive analyses for real programs.

6 Applications and Related Work

This paper has demonstrated how the problem of resolving existential queries over a context-sensitive analysis can be reduced to the tractable problem of intersecting regular languages. The ability to resolve such queries has direct application to program comprehension, model checking, and interprocedural program analysis for checkpointing. The application to program comprehension was demonstrated in Section 4. There, a non-trivial constraint was applied to a small program with a convoluted path structure. Using the technique presented in this paper the query was resolved and an example was generated that exhibited a solution. For model-checking, the abstraction of a context-sensitive analysis is sufficiently robust to encode the notion of *property-transformers* [4, 5] used to resolve modal mu-calculus queries. Given this encoding, it is straightforward to place stack-context constraints on temporal logic formulas checked against a hierarchical model. Finally, in application-level checkpointing, data is saved at each occurrence of a checkpoint command. The saved data can be restricted to the set of data that is necessary to continue the computation from the checkpoint. For each checkpoint, this set of data can be determined via the same liveness analysis that was used for the case study in Section 5. From this analysis, automata can be associated with each checkpoint statement that use the current stack to precisely decide the live data set for that invocation of the checkpoint. Future work will more fully expand on this idea.

Previous work has reduced the problem of resolving unconstrained, single-property queries on a context-sensitive analysis to the problem of graph reachability [8]. In this framework, a context-sensitive analysis is provided as an *exploded super-graph*. Given this graph, which is an outer-product of the normal state machine model with the set of properties, an unconstrained query asking whether a particular property holds at a particular point is reduced to finding a path in this super-graph from a property that holds at the entry of *main* to the desired property at the desired vertex. The super-graph has the advantage that it is an immutable structure that can be used to resolve an unlimited number of queries, whereas the approach taken in this paper requires a new accepting automaton to be generated for each query. However, unlike automata, the super-graph cannot be minimized. The super-graph cannot be used to answer queries with stack-context constraints, nor can it find paths to vertices where a particular set of data-flow facts holds. The approach presented in this paper is capable of handling the complete class of distributive functions to which super-graphs can be applied.

References

[1] `http://www.ifa.hawaii.edu/~barnes/treecode/treeguide.html`.

[2] J. E. Barnes. A modified tree code: Don't laugh, it runs. *Journal of Computational Physics 87, 161*, 1990.

[3] M. Benedikt, P. Godefroid, and T. Reps. Model checking of unrestricted hierarchical state machines. In *Proc. of ICALP 2001, Twenty-Eighth Int. Colloq. on Automata, Languages, and Programming*, Crete, Greece, July 2001.

[4] O. Burkart and B. Steffen. Model checking for context-free processes. In *International Conference on Concurrency Theory*, pages 123–137, 1992.

[5] J. Ezick, D. W. Richardson, and T. Teitelbaum. Practical model checking and example generation for context-free processes. Technical Report TR2002-1851, Cornell University, August 2001.

[6] D. Huffman. The synthesis of sequential switching circuits. *Journal of the Franklin Institute*, 257:161–190, 275–303, 1954.

[7] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, volume 34, pages 3–42. Princeton University Press, 1956.

[8] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.

[9] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[10] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of POPL '80: 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–94, Las Vegas, Nevada, January 1980.