Resource Usage Verification

Kim Marriott[1], Peter J. Stuckey[2], and Martin Sulzmann[3]

[1] School of Computer Science and Software Engineering
Monash University, Vic. 3800, Australia
marriott@mail.csse.monash.edu.au

[2] Department of Computer Science and Software Engineering
The University of Melbourne, Vic. 3010, Australia
pjs@cs.mu.oz.au

[3] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
sulzmann@comp.nus.edu.sg

Abstract. We investigate how to automatically verify that resources such as files are not used improperly or unsafely by a program. We employ a mixture of compile-time analysis and run-time testing to verify that a program conforms to a resource usage policy specified by a deterministic finite state automata (DFA) which details allowed sequences of operations on resources. Our approach has four main phases. The first is to generate a context-free grammar which safely approximates the resource usage behaviour of the program. This, rather than the original program, is analysed in subsequent phases. The second phase checks whether the grammar satisfies the resource usage policy and, if not, where the problems arise. The third phase determines where to place a minimal set of run-time tests and the fourth determines how to instrument the program to compute the state information required for the tests.

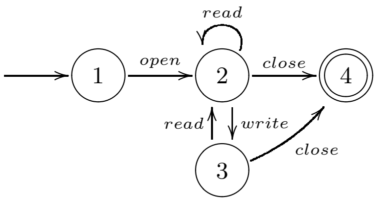
1 Introduction

The difficulty of developing and then maintaining large, complex but still reliable software systems is well known, and in spite of many efforts to improve reliability over the last few decades, the problems have not been solved. This issue has been compounded by the rise of the world-wide web, applets and e-commerce, since people are now using more software from sources they have no good reason to trust. It is thus vital that we develop better techniques for building reliable, trustworthy software systems. One promising approach is to develop better tools for automatically analyzing the integrity (reliability, security, etc) of software systems.

Here we focus on automatically verifying that "resources" such as files or global variables are not used improperly or unsafely by a program. For instance checking that all files are opened before reading or writing and finally closed. More exactly, we wish to verify that a program conforms to a resource usage policy which details allowed sequences of operations on resources such as reading, writing, or closing. This is assumed to be a regular language specified by a

```
main(n) = open(f) ; a(f,n) ; close(f)
a(f,n)  = write(f);
          if (n mod 2 == 0) then read(f);
          if (n > 0) then a(f,n-1)
```

(a)



(b)

Fig. 1. (a) A program fragment and (b) a DFA for a simple file resource policy.

deterministic finite state automata (DFA). We verify that a program does not violate such a policy by employing a mixture of compile-time analysis and run-time testing.

Consider the program fragment shown in Figure 1(a). Our running example will be to verify that this program satisfies the resource policy specified by the DFA given in Figure 1(b). Note that in order to simplify the figure the DFA is incomplete: there is an implicit error state e which is reached whenever the DFA sees an invalid token.

Our conceptual starting point is the *instrumented program* which threads a (mutable) variable s through the program to track the current state of the DFA during execution and has run-time checks on s to ensure that the resource policy is not violated:

Example 1. The instrumented version of the program of Figure 1(a) is

$$
\begin{aligned}
\mathsf{main(n)} &= \mathsf{s := 1};\ test_{open}(\mathsf{s});\ \mathsf{open(f)};\ track_{open}(\mathsf{s});\\
&\quad\ \mathsf{a(f,n)};\ test_{close}(\mathsf{s});\ \mathsf{close(f)};\ track_{close}(\mathsf{s});\ test_{final}(\mathsf{s})\\
\mathsf{a(f,n)} &= test_{write}(\mathsf{s});\ \mathsf{write(f)};\ track_{write}(\mathsf{s});\\
&\quad\ \mathsf{if\ (n\ mod2 == 0)\ then}\ (test_{read}(\mathsf{s});\ \mathsf{read(f)};\ track_{read}(\mathsf{s}))\\
&\quad\ \mathsf{if\ (n > 0)\ then\ a(f,n\text{-}1)};
\end{aligned}
$$

where $test_{usage}$ checks that the current state allows that usage of the resource or else aborts, e.g. $test_{read}(s)$ checks that s is either 2 or 3 while $test_{final}$ checks that the current state is a final state, and $track_{usage}$, updates the state given that this usage has just been performed, i.e. $track_{read}$ maps states 2 and 3 to 2.

However, naively instrumenting programs in this way may introduce substantial run-time overhead. Ideally we wish to verify conformance at compile-time, but if this is not possible, then we wish to insert a minimal number of run-time tests into the program and to reduce the overhead of state tracking.

Our approach has four main phases, all of which rely on sophisticated compile-time analysis. The first phase is to generate a context-free grammar whose terminals symbols correspond to resource usages and which safely approximates the resource usage behaviour of the program. For brevity we omit this step. The basic idea is to employ a type and effect system [1] to abstract the resource

$M \rightarrow open\ A\ close$
$A \rightarrow write\ B\ C$
$B \rightarrow \epsilon$
$B \rightarrow read$
$C \rightarrow \epsilon$
$C \rightarrow A$

(a)

```
main(n) = open(f); a(f,n); close(f)
a(f,n)  = write(f);
          if (n mod2 == 0) then (read(f); s := 2)
          else s := 3;
          if (n > 0) then (test_{2}(s); a(f,n-1))
```

(b)

Fig. 2. Corresponding grammar and modified code for the program of Figure 1(a)

usage behaviour of a simply-typed ML-like language. Details can be found in an accompanying technical report [2]. For the purposes of this paper, we restrict ourselves to a first-order language.

Example 2. The grammar corresponding to the program in Figure 1(a) is shown in Figure 2(a). Note how each if-then construct is converted to non-deterministic grammar rules, one for each case.

This grammar, rather than the original program, is analysed in subsequent phases. Importantly, the analyses employed in the subsequent three phases are exact, i.e. they give the perfect result. Thus it is only in this first phase that true approximation takes place.

Every sequence of resource operations that can occur when executing the original program is required to be a sentence in the language, $L(G)$, of the approximating grammar G. Thus the program satisfies the resource policy specified by the DFA, M, if $L(G) \subseteq L(M)$ where $L(M)$ is the language of M. The second phase of our approach (detailed in Section 2) is to check whether this is true or not, and if not, to determine at which points in the grammar strings not in $L(M)$ may be generated. For our running example our analysis shows that the program does not necessarily respect the resource usage policy, since the error state can be reached. If instead the line if (n mod 2 == 0) then read(f); was simply read(f) then our analysis would show that the program will never violate the protocol.

The next two phases are used if the grammar cannot be proven to satisfy the resource policy. In the third phase (described in Section 3) we determine where to place the run-time tests. The final phase (described in Section 4) determines how to add state computation to the code. The resulting program is guaranteed to respect the resource usage policy or abort before it actually violates the policy. It also performs substantially fewer tests and less state computation than the first naive instrumented version of the program. For instance, the code for the program of Figure 1(a) generated by our approach is shown in Figure 2(b). Note that $test_{\{2\}}(s)$ checks that state s is in the set $\{2\}$, i.e. it is 2, aborting execution if it is not.

Our main technical contributions are:

– A generic four phase methodology based on approximating the resource usage behaviour of a program by a grammar G for verifying that a program meets a resource usage policy.

– A polynomial time analysis and algorithm to annotate G, and hence P, with a minimal set of run-time tests to ensure that it satisfies the resource usage policy specified by M.
– A polynomial time analysis and algorithm to annotate G, and hence P, with minimal instrumentation needed to track state needed for run-time tests to ensure that it satisfies the resource usage policy specified by M.

2 Checking a Resource Usage Policy

We assume some fixed set of resource usage primitives such as open, read, write and close which are applied to a particular resource r. A resource is specified by its type and abstract location. For simplicity we ignore the aliasing problem between resources. That is, resources are assumed to be explicitely annotated with must-alias information. The resource usage behaviour of a program P is captured by its *resource usage traces*. These are sequences of the calls to the resource usage primitives that can be made in some execution of the program. We distinguish between *complete* traces which follow the behaviour until the program terminates and *partial* traces. For instance, for our example program from Figure 1 (a),

$$open(f), write(f), read(f), write(f), close(f)$$

is a complete trace while $open(f), write(f), read(f)$ is a partial trace.

The first step in our approach is to approximate the resource usage behaviour of a program P by a *context-free grammar* G. This consists of a set Σ of terminal symbols where each symbol corresponds to the application of a particular resource usage primitive to a particular resource (this is called a primitive resource effect), a set N_G of non-terminal symbols, a start symbol $S_G \in N_G$ and a set of production rules R_G. We require:
(a) for each complete resource usage trace w for P, $w \in L(G)$, and
(b) for each partial resource usage trace w for P, $ww' \in L(G)$, for some w'.
For instance, the grammar in Figure 2 (a) approximates Figure 1 (a). The details of the approximation process can be found in [2]. For our simply-typed ML-like language the approximation grammar is linear in the size of the input program.

We assume that a *resource usage policy* is a regular language specified by an DFA, M. For example, the policy "we must open the resource before reading and writing an arbitrary number of times, with no two consecutive writes, before closing" is specified by the DFA in Figure 1. A resource usage policy can refer to more than one resource, thus one might have a policy stating that a program has no more than two files open. Of course more than one resource usage policy may apply to a program: For simplicity we check and instrument for each policy independently.

One complication is that a particular resource usage policy M does not necessarily refer to all of the primitive resources in G. Therefore, we need to project away all of the uses of "irrelevant" resources. For each terminal symbol X in G which corresponds to the use of an irrelevant resource, we introduce a new production rule $X \to \epsilon$ where X is now considered a non-terminal symbol. We usually leave the above transformation implicit.

We assume that M, the DFA detailing the resource usage policy, is a quadruple (Σ, Q, T, q_0) where Σ is the alphabet of symbols, i.e. the set of primitive resource usage effects, for the resources in the policy, Q is a set of states containing a single distinguished error state e, $T : \Sigma \times Q \to Q$ is a deterministic complete transition function with $T(x, e) = e$ for all $x \in \Sigma$ and $q_0 \in Q$ is the start state. We define $T^\star$ the extended transition function to be

$$
\begin{aligned}
T^\star(\epsilon, q) &= q \\
T^\star(x\alpha, q) &= T^\star(\alpha, T(x, q))
\end{aligned}
$$

For simplicity we assume that all states are final states except for e. This is allowable since acceptance by final state in machine M' can be simulated by building a new machine M for M' such that $\alpha \in L(M')$ iff $\alpha\$ \in L(M)$. M is constructed by adding an extra symbol \$ into Σ and a new state f, and setting $T(\$, q) = f$ when q is a final state, and otherwise e the error state. We add an action \$ to the end of the main routine of the code we are checking. (Note *close* plays a similar role to that of \$ in the running example).

The form of the DFA we use (acceptance by non-error) is important because it allows us to restrict attention to resource traces that are prefixes of correct resource behaviour. Any prefix of a correct resource behaviour avoids the error state.

It is straightforward to use our approximating grammar G to check whether the original program satisfies a resource policy specified by the DFA M. Since, any possible resource behaviour of the program will be described by G, we know that the program satisfies the resource policy specified by M if $L(G) \subseteq L(M)$.

The problem of determining whether a regular language $L(M)$ contains a context-free language $L(G)$ is well-known to be decidable (e.g. [3]). We simply construct a push down automata for the language $L(G) \cap \overline{L(M)}$ (the intersection of a context-free grammar and a finite automata) and check whether it is empty.

However this indirect approach is not well-suited to our purposes since it does not indicate which parts of the grammar violate the policy (which is important for providing feedback to the programmer and for introducing run-time tests). For these reasons we now give a more constructive process for determining whether a regular language contains a context-free language.

Our decision procedure is based on a simple abstract interpretation of the grammar G which details which states in the DFA M can be reached when generating sentences in the language of G.

The main step of the analysis is to compute for each symbol X in G and state q in M the set of states that can be reached from q when accepting all sentences that can be generated from X. I.e. we compute $\{T^\star(w, q) \mid X \Rightarrow_G^\star w \text{ and } w \in \Sigma^\star\}$. This is the least fixpoint of the equations

$$
\begin{aligned}
&reach(X, q) = \{T(X, q)\} \text{ if } X \in \Sigma \\
&reach(X, q) = \bigcup\{reach_{seq}(\gamma, q) | (X \to \gamma) \in G\} \text{ if } X \in N_G
\end{aligned}
$$

$$reach_{seq}(\epsilon, q) = \{q\}$$
$$reach_{seq}(X\gamma, q) = \bigcup\{reach_{seq}(\gamma, q')|q' \in reach(X, q)\}.$$

It is straightforward to show by induction that

Theorem 1. $reach(X, q) = \{T^\star(w, q) \mid X \Rightarrow_G^\star w$ *and* $w \in \Sigma^\star\}$

Thus we have a decision procedure for the problem of interest

Corollary 1. $L(G) \subseteq L(M)$ *iff* $e \notin reach(S_G, q_0)$.

Example 3. The interesting calculations of reachability for the grammar of Example 2 are: $reach(M, 1) = \{4, e\}$, $reach(B, 3) = \{2, 3\}$, $reach(A, 2) = \{2, 3, e\}$, $reach(A, 3) = \{e\}$, $reach(C, 2) = \{2, 3, e\}$, and $reach(C, 3) = \{3, e\}$. We use e to indicate the implicit error state of Figure 1(b). Note that by definition $reach(X, e) = e$ for all $X \in N_G$.

Importantly, $reach(S_G, q_0)$ can be computed in polynomial time. This is because we can compute $reach(S_G, q_0)$ by computing the Kleene-sequence for *reach*. This can be done in polynomial time since the Kleene sequence is at most $|Q| \cdot |N_G| \cdot |Q|$ long and each element in the sequence can be computed in $|Q| \cdot |N_G| \cdot |R_G|$, where $| \cdot |$ denotes set cardinality and $|R_G|$ is the total number of symbols in the productions.

We can also use this information to annotate the grammar and so indicate where the policy is violated. The idea is that we annotate the grammar G with states which the DFA M could be in if M was recognising the string in parallel with G.

The first step is to compute the *call patterns*, $cp(V)$, for each $V \in N_G$, i.e. the set $\{T^\star(w, q_0)|S_G \Rightarrow_G^\star wV\gamma$ and $w \in \Sigma^\star\}$. This is the least fixpoint of

$$cp(S_G) \supseteq \{q_0\}$$
$$cp(V) \supseteq Reach_{seq}(\gamma_1, cp(X)) \text{ for each } X \to \gamma_1 V \gamma_2 \in R_G$$

where $Reach_{seq}(\gamma, Q) = \bigcup\{reach_{seq}(\gamma, q)|q \in Q\}$.

Theorem 2. *For all* $V \in N_G$, $cp(V) = \{T^\star(w, q_0)|S_G \Rightarrow_G^\star wV\gamma$ *and* $w \in \Sigma^\star\}$

We can now annotate each production $X \to X_1 \cdots X_n$ in the grammar with *program points* $X \to {}^{pp_1}X_1{}^{pp_2} \ldots {}^{pp_n}X_n{}^{pp_{n+1}}$. The program point pp_k is the set of possible states M is in before the sentence generated from X_k is accepted by M. The annotation is simply given by

$$pp_1 = cp(X)$$
$$pp_{k+1} = Reach_{seq}(X_k, pp_k), \text{ for } k = 1, ..n$$

Example 4. The calling patterns calculated for the grammar of Example 2 are $cp(M) = \{1\}$, $cp(B) = \{3, e\}$, $cp(A) = \{2, 3, e\}$ and $cp(C) = \{2, 3, e\}$. The annotations are as follows:

$$
\begin{aligned}
M &\to {}^{\{1\}}open^{\{2\}}A^{\{2,3,e\}}close^{\{4,e\}}\\
A &\to {}^{\{2,3,e\}}write^{\{3,e\}}B^{\{2,3,e\}}C^{\{2,3,e\}}\\
B &\to {}^{\{3,e\}}\\
B &\to {}^{\{3,e\}}read^{\{2,e\}}\\
C &\to {}^{\{2,3,e\}}\\
C &\to {}^{\{2,3,e\}}A^{\{2,3,e\}}
\end{aligned}
$$

Consider a production $R \equiv X \to X_1 \cdots X_n$ and let $S_G \Rightarrow_G^\star \gamma_1 X \gamma_2 \Rightarrow_R \gamma_1 X_1' \cdots X_n' \gamma_2$. We say that X_k' is an *instance* of X_k since it was generated by the rule R and corresponds to X_k.

Theorem 3. *For each annotated production $X \to {}^{pp_1}X_1{}^{pp_2} \ldots {}^{pp_n}X_n{}^{pp_{n+1}}$ and $k \in \{1, \ldots, n\}$,*

$$pp_k = \{T^\star(w, q_0) | w \in \Sigma^\star \text{ s.t. } S_G \Rightarrow_G^\star wV\gamma \text{ and } V \text{ is an instance of } X_k\}$$

Computing the call patterns and annotations can again be done in polynomial time. The important point is that for an annotated production $X \to {}^{pp_1}X_1{}^{pp_2} \ldots {}^{pp_n}X_n{}^{pp_{n+1}}$ we know that a terminal symbol X_k can cause an error if for some $q \in pp_k \setminus \{e\}$, $T(X_k, q) = e$. This indicates that the grammar (and hence maybe the program) can be in non-error state q just before resource usage X_k is performed, and that performing X_k in this state will violate the policy. This can be used to provide feedback to the programmer about where the original program may violate the protocol.

Example 5. The only terminal symbol which can cause an error in Example 4 is the *write* symbol in the rule for A, since $T(write, 3) = e$.

As we shall now see this information can also be used to add run-time tests to the program.

3 Adding Run-Time Tests

Of course we cannot always prove that a program satisfies a resource usage policy: this may be because of inaccuracy introduced by approximating the program by a grammar, or because the program when executed with some input does not satisfy the policy. In such cases we can add run-time checks to ensure that the program aborts before the policy is violated.

As we saw in Example 1 the simplest approach is to instrument the program so as to explicitly keep track of which state in M it is in and whenever a resource usage primitive is applied to the resource first checking that this usage is valid for that state, aborting if not, and then appropriately updating the state. This is not a new idea, see for instance [4]. However, naively instrumenting programs in this way may introduce substantial run-time overhead. In this and the following section we show how simple analyses of the grammar approximating the program allow us to instrument the program with substantially less run-time overhead. In particular we use analyses to determine which tests are really needed and where they can be best placed. We also use analyses to reduce the amount of instrumentation required.

In order to bridge the gap between an instrumented program and the grammar G approximating the program we introduce the *instrumented grammar*, G_{inst}, which is an attribute grammar with essentially the same structure as G but which has attributes to model the instrumentation of the instrumented program.

The instrumented grammar G_{inst} corresponding to G has a non-terminal symbol X' for each $X \in N_G \cup \Sigma$. Each such non-terminal symbol has two attributes, *in* and *out*, ranging over Q where *in* is inherited and *out* is synthesized. For simplicity and without loss of generality we assume that G only has a single instance of its start symbol S_G. The start symbol for G_{inst} is S'_G and we set $S'_G.in$ to q_0. For each production $V \to X_1 \cdots X_n \in R_G$, there is a corresponding production $V' \to X'_1 \cdots X'_n$ in the instrumented grammar with attribute rules[4]

$$\$1.in := \$\$.in \qquad \$(k{+}1).in := \$k.out \text{ for } k = 1, \ldots, n-1 \qquad \$\$.out := \$n.out$$

For each production $V \to \epsilon$ there is a corresponding production $V' \to \epsilon$ with attribute rule $\$\$.out := \$\$.in$. And for each terminal $X \in \Sigma$ there is a production $X' \to X$ with attribute rules $\$\$.out := T(X, \$\$.in)$. It is straightforward to prove the following results.

Lemma 1. *$S_G \Rightarrow^\star_G X_1 \ldots X_m$ iff $S'_G \Rightarrow^\star_{G_{inst}} X'_1 \ldots X'_m$*

Lemma 2. *$L(G) = L(G_{inst})$*

Lemma 3. *If $S'_G \Rightarrow^\star_{G_{inst}} wX'\gamma$ for $w \in \Sigma^\star$ then $X'.in = T^\star(w, q_0)$.*

Example 6. The instrumented grammar corresponding to the grammar G of Example 2 is

$$
\begin{array}{rcll}
M' & \to & open'\ A'\ close' & \{\$1.in := \$\$.in; \$2.in := \$1.out; \$3.in := \$2.out; \$\$.out := \$3.out\} \\
A' & \to & write'\ B'\ C' & \{\$1.in := \$\$.in; \$2.in := \$1.out; \$3.in := \$2.out; \$\$.out := \$3.out\} \\
B' & \to & \epsilon & \{\$\$.out := \$\$.in\} \\
B' & \to & read' & \{\$1.in := \$\$.in; \$\$.out := \$1.out\} \\
C' & \to & \epsilon & \{\$\$.out := \$\$.in\} \\
C' & \to & A' & \{\$1.in := \$\$.in; \$\$.out := \$1.out\} \\
open' & \to & open & \{\$\$.out := T(open, \$\$.in)\} \\
close' & \to & close & \{\$\$.out := T(close, \$\$.in)\} \\
read' & \to & read & \{\$\$.out := T(read, \$\$.in)\} \\
write' & \to & write & \{\$\$.out := T(write, \$\$.in)\}
\end{array}
$$

We will analyse and annotate instrumented grammars in the obvious way: by simply ignoring the attributes and treating them as a context free grammar. Thus we can actually analyse G and use this to annotate G_{inst}. Interestingly, we can understand the analyses as simple abstract interpretations [5] of the

[4] We use YACC-like notation: $\$\$.a$ refers to attribute a of the LHS symbol and $\$k.a$ refers to attribute a of the kth RHS symbol.

instrumented grammar. For instance, the information at program point pp_k is the set of values that $X'_k.in$ can take.

Of course the whole point of this exercise is to allow us to add run-time tests to the right-hand side of the instrumented grammar. We introduce as needed non-terminal symbols $test_S$ for $S \subseteq Q$ with a single inherited attribute in and the defining production $test_S \to \epsilon$ with the attribute test $\$\$.in \in S$. Note that we assume that if an attribute test fails for some sentence then that sentence is not in the language of the attribute grammar.

We extend our analyses to handle these symbols in the obvious way by defining $reach(test_S, q) = \{q\} \cap S$.

At first glance it might seem that the only place to consider adding tests is immediately before those terminal symbols which can cause an error. However, it is often possible to place tests earlier than this. Our basic idea for adding run-time tests is to take an annotated instrumented grammar and use an analysis to determine at which program points it is possible to identify states which must inevitably lead to failure. This identifies all of the places one could usefully consider adding tests.

The first thing we need to do is to determine for each $V \in N_G$, the *definite failure states* $fs(V)$. This is the set of states q such that $T^\star(w, q) = e$ for all $w \in \Sigma^\star$ s.t. $S_G \Rightarrow^\star_G \gamma V w$. This is essentially the dual problem to working out the calling patterns for each symbol. It is the greatest fixpoint of

$$\begin{aligned} fs(S_G) &= \{e\} \\ fs(X) &\subseteq \bigcap_{V \to \gamma_1 X \gamma_2 \in R_G} \{q \in Q | reach_{seq}(\gamma_2, q) \subseteq fs(V)\} \end{aligned}$$

Note that for all V, $e \in fs(V)$.

Theorem 4. $fs(V) = \{q \in Q | T^\star(w, q) = \{e\}\ \forall\ w \in \Sigma^\star$ *s.t.* $S_G \Rightarrow^\star_G \gamma V w\}$.

We can use this analysis to add information to an annotated grammar. At each program point we record the subset of states at that point that definitely lead to failure. Consider the annotated production $X \to {}^{pp_1} X_1{}^{pp_2} \ldots {}^{pp_n} X_n{}^{pp_{n+1}}$. We add for each program point pp_k, the set fpp_k which is the subset of pp_k which will definitely lead to an error. This additional annotation is simply given by $fpp_k = \{q \in pp_k | reach_{seq}(X_k \ldots X_n, q) \subseteq fs(X)\}$, for $k = 1, \ldots, n+1$.

Example 7. For grammar G of Example 2 we have $fs(M) = \{e\}$, $fs(B) = \{1, e\}$, $fs(A) = \{1, e\}$, and $fs(C) = \{1, e\}$. The annotations where pp_k is split into $pp_k \setminus fpp_k$ and fpp_k are as follows:

$$\begin{aligned} M &\to {}^{\{1\},\{\}} open^{\{2\},\{\}} A^{\{2,3\},\{e\}} close^{\{4\},\{e\}} \\ A &\to {}^{\{2\},\{3,e\}} write^{\{3\},\{e\}} B^{\{2,3\},\{e\}} C^{\{2,3\},\{e\}} \\ B &\to {}^{\{3\},\{e\}} \\ B &\to {}^{\{3\},\{e\}} read^{\{2\},\{e\}} \\ C &\to {}^{\{2,3\},\{e\}} \\ C &\to {}^{\{2\},\{3,e\}} A^{\{2,3\},\{e\}} \end{aligned}$$

For instance the annotation ${}^{\{2\},\{3,e\}}$ before A in the last rule indicates that being in state 3 at this point definitely leads to error while being in state 2 may not.

Computing the definite failure states and this additional annotation can again be done in polynomial time.

Theorem 5. *For each annotated production* $X \rightarrow {}^{fpp_1}X_1{}^{fpp_2} \ldots {}^{fpp_n}X_n{}^{fpp_{n+1}}$ *we have*

$$fpp_1 = \{q \in Q | T^\star(w, q) = e \ \forall w \in \Sigma^\star \ s.t. \ S_G \Rightarrow_G^\star \gamma X w\}$$

and for $k \in \{1, \ldots, n\}$,

$$fpp_{k+1} = \{q \in Q| \ T^\star(w, q) = e \ \forall w \in \Sigma^\star \ s.t. \ S_G \Rightarrow_G^\star \gamma V w \\ \text{and } V \text{ is an instance of } X_k\}$$

The motivation for adding this extra information is that it tells us where it is useful to add tests to the instrumented grammar and so to the original program. It is sensible to consider adding a test at any program point for which $fpp_k \setminus \{e\} \neq \emptyset$ since a test at this point can distinguish between the states in fpp_k which can never lead to success and those in $pp_k \setminus fpp_k$ which may not lead to failure.

Thus the basic step in adding run-time tests to the instrumented grammar G_{inst} is the following. Let $X' \rightarrow X'_1 \ldots X'_n$ be a production in G_{inst}, such that $fpp_k \setminus \{e\} \neq \emptyset$ for some $k \in \{1, \ldots, n+1\}$. Let $S = pp_k \setminus fpp_k$. The *refinement* of G_{inst} for this production and program point is the grammar G'_{inst} obtained by replacing this production by the production $X' \rightarrow X'_1 \ldots X'_{k-1} \ test_S \ X'_k \ldots X'_n$ with essentially the same attribute rules (more precisely, each $\$j$ must be replaced by $\$(j+1)$ for $j \geq k$) and an attribute rule to copy the state after X'_{k-1} to the inherited attribute for $test_S$, $\$k.in := \$(k-1).out$. If $k = 1$ then this action is $\$1.in := \$\$.in$.

Example 8. Consider the instrumented production rule $C \rightarrow A$ before and after the addition of the $test_{\{2\}}$ non-terminal before A to obtain the rule $C \rightarrow test_{\{2\}} \ A$. The instrumented production rules are

$$C \rightarrow A \qquad \{\$1.in := \$\$.in; \$\$.out := \$1.out\}$$

$$test_{\{2\}} \rightarrow \epsilon \qquad \{\$\$.in \in \{2\}\}$$
$$C \rightarrow test_{\{2\}} \ A \qquad \{\$1.in := \$\$.in; \$2.in := \$\$.in; \$\$.out := \$2.out\}$$

Lemma 4. *Let* G'_{inst} *be a refinement of* G_{inst}. *Then*
(a) $L(G_{inst}) \cap L(M) = L(G'_{inst}) \cap L(M)$, *and*
(b) $L(G'_{inst}) \subset L(G_{inst})$.

The algorithm for adding run-time tests to an instrumented grammar G_{inst} to ensure conformance with the protocol given by an DFA M is therefore

1. Annotate G_{inst} with program point and definite failure state information.
2. Stop if for all program points $fpp_k = \emptyset$ and return G_{inst}.
3. Choose some production R in G_{inst} and program point k in R, such that $fpp_k \setminus \{e\} \neq \emptyset$.
4. Refine G_{inst} using R and program point k.
5. Goto Step 1.

It follows from Lemma 4 that

Theorem 6. *Let G_{inst} be the instrumented grammar input to the above algorithm and G'_{inst} the output grammar. Then $L(G'_{inst}) = L(G_{inst}) \cap L(M)$.*

Furthermore, the above algorithm always terminates and has time complexity which is polynomial in the size of G_{inst} and M. This is because we can add at most one test for each program point in the original G_{inst}. Thus the main iteration loop can only occur G_{inst} times and the final and all intermediate programs are $O(G_{inst})$. The result follows since annotation and refinement of each intermediate program takes polynomial time in the size of the intermediate program and M.

Of course the algorithm for adding run-time tests is non-deterministic in the choice of which program points and production are used for each refinement step. One possible strategy is to perform tests as early as possible so as to detect failure as early as possible. Thus in this case we always choose the first program point and production found in a depth first traversal of the productions in G_{inst} from the start symbol.

Example 9. For our running example, we can add a test at the program point in C before A since the state 3 definitely leads to a resource usage error. The revised grammar, with the run-time test $test_S$ indicating that the program should abort if the DFA is not in a state in set S, and with updated program point information is as follows:

$$\begin{array}{rl}
M \to & {}^{\{1\},\{\}}open^{\{2\},\{\}}A^{\{2,3\}\{\}}close^{\{4\},\{\}} \\
A \to & {}^{\{2\},\{\}}write^{\{3\},\{\}}B^{\{2,3\},\{\}}C^{\{2,3\},\{\}} \\
B \to & {}^{\{3\},\{\}} \\
B \to & {}^{\{3\},\{\}}read^{\{2\},\{\}} \\
C \to & {}^{\{2,3\},\{\}} \\
C \to & {}^{\{2,3\},\{\}}test_{\{2\}}{}^{\{2\},\{\}}A^{\{2,3\},\{\}}
\end{array}$$

Note how the error state no longer occurs at any program point indicating that with this revised grammar $L(G) \subseteq L(M)$.

Another strategy would be to annotate each program point with a cost of putting a test there which is proportional to the number of times we expect the corresponding point in the underlying program to be reached. In this case we might choose the program point with least cost for refinement.

Regardless, the important point is that our analysis tells us precisely at which program points in the instrumented grammar and hence the original program it is useful to consider adding a run-time test.

4 Tracking State

Once we have determined where to add run-time tests to our instrumented grammar and hence to the original program we need to determine where we need to track state. At an extreme position, if we have not needed to add any run-time tests then clearly there is little point in instrumenting the grammar or program. And, in general we do not need to instrument the program beyond the point tests will be performed. However there is actually considerably more scope for reducing instrumentation. The idea is that we first identify at each program point which states behave equivalently in the sense that they will succeed or fail in exactly the same way.

We let $equiv(Q)$ denote the set of all equivalence relations of the set of states Q. $equiv(Q)$ forms a complete lattice ordered by logical implication, i.e. for $\equiv_1, \equiv_2 \in equiv(Q)$, $\equiv_1 \leq \equiv_2$ iff $\equiv_2 \rightarrow \equiv_1$, where the least element is the equivalence relation in which all elements are equivalent. For convenience we treat an equivalence relation as both a binary infix predicate and as the set of pairs (q, q') for which equivalence holds.

Our first analysis is $eq(X, \equiv)$. This computes the new equivalence $\equiv'$ such that $q \equiv' q'$ iff $T^\star(w, q) \equiv T^\star(w, q')$ for all $w \in \Sigma^\star$ such that $X \Rightarrow_G^\star w$.

This is the least fixpoint of the equations

$$\begin{aligned}
eq(test_S, \equiv) &= \{(q, q') \in \equiv \mid \{q, q'\} \subseteq S \vee \{q, q'\} \subseteq Q \setminus S\} \\
eq(X, \equiv) &= \{(q, q') | q, q' \in Q, T(X, q) \equiv T(X, q')\} \text{ if } X \in \Sigma \\
eq(X, \equiv) &= \bigcap \{eq_{seq}(\gamma, \equiv) | X \rightarrow \gamma \in R_{G_{inst}}\} \text{ if } X \in N_G \\
eq_{seq}(\epsilon, \equiv) &= \equiv \\
eq_{seq}(X\gamma, \equiv) &= eq(X, eq_{seq}(\gamma, \equiv)).
\end{aligned}$$

The effect of the first equation for eq is to split the partitions in $\equiv$ so that equivalent states behave equivalently in the test. The remaining equations state that for a terminal symbol X two states are equivalent if T maps them to equivalent states, while for a non-terminal symbol X, two states are equivalent if they are equivalent for all rules defining X.

Theorem 7. $eq(X, \equiv) = \{(q, q') | T^\star(w, q) \equiv T^\star(w, q') \ \forall w \in \Sigma^\star s.t.\ X \Rightarrow_G^\star w\}$

We can use this as the basis for further annotating an annotated instrumented grammar G (which may contain tests). We first compute the *post state equivalence*, $pse(V)$, for each non-terminal symbol V, that is the states that for all possible future sequence of resource usages w act equivalently.

This is the least fixpoint of

$$\begin{aligned}
pse(S_G) &= \{(q, q') | q, q' \in Q \setminus \{e\}\} \cup \{(e, e)\} \\
pse(X) &= \bigcap \{eq_{seq}(\gamma_2, pse(V)) | (V \rightarrow \gamma_1 X \gamma_2) \in R_{G_{inst}}\}
\end{aligned}$$

Theorem 8. *For all non-terminal symbols V,*

$$pse(V) = \{(q, q') \mid (T^\star(w, q) = e) \leftrightarrow (T^\star(w, q') = e) \forall w \in \Sigma^\star \ s.t.\ S_G \Rightarrow_G^\star \gamma V w\}$$

We can use this analysis to add information to an annotated grammar. At each program point we can also record the equivalence relation of states at that point.

Consider the annotated production $X \rightarrow {}^{pp_1}X_1{}^{pp_2} \ldots {}^{pp_n}X_n{}^{pp_{n+1}}$. We add for each program point pp_k, the equivalence relationship epp_k between the states in pp_k. This additional annotation is simply given by

$$epp_k = \{(q, q') | q, q' \in pp_k, q \equiv q'\}$$

where $\equiv$ is $eq_{seq}(X_k \ldots X_n, pse(X))$ for $k = 1, \ldots, n+1$.

This also allows us to compute the *call pattern equivalence*, $cpe(V)$, for each non-terminal symbol V, which is $\bigcap\{epp_1 | V \rightarrow {}^{epp_1}\alpha^{epp_{n+1}} \in R_{G_{inst}}\}$. Computing the equivalence relation and annotation again takes polynomial time because the height of $equiv(Q)$ is $|Q|$.

Example 10. The equivalence annotations, post state equivalences and call pattern equivalences for the instrumented grammar of Example 9 are as follows:

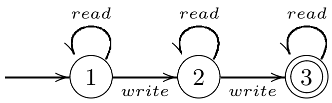
$$\begin{array}{rcl}
{}^{\{1\}}M^{\{4\}} & \rightarrow & {}^{\{1\}}open^{\{2\}}A^{\{2,3\}}close^{\{4\}} \\
{}^{\{2\}}A^{\{2,3\}} & \rightarrow & {}^{\{2\}}write^{\{3\}}B^{\{2\},\{3\}}C^{\{2,3\}} \\
{}^{\{3\}}B^{\{2\},\{3\}} & \rightarrow & {}^{\{3\}} \\
{}^{\{3\}}B^{\{2\},\{3\}} & \rightarrow & {}^{\{3\}}read^{\{2\}} \\
{}^{\{2\},\{3\}}C^{\{2,3\}} & \rightarrow & {}^{\{2,3\}} \\
{}^{\{2\},\{3\}}C^{\{2,3\}} & \rightarrow & {}^{\{2\},\{3\}}test_{\{2\}}{}^{\{2\}}A^{\{2,3\}}
\end{array}$$

If all states are equivalent at a particular point then we do not need to track state at that point. Hence, in our example we only need to track the state in B before it is tested in C. Thus, the final instrumented program is that shown in Figure 2 (b).

The first step in determining the necessary instrumentation is to choose a name r for each equivalence class at each program point pp_k. We define function $can_k(q)$ to return the name of the equivalence class of $q \in pp_k$ and $rep_k(r)$ to return some representative state $q \in pp_k$ in the equivalence class called r. Similarly, we choose names for the equivalence classes in $cpe(V)$ and $pse(V)$ for each non-terminal V and define $can_{cpe(V)}$, $can_{pse(V)}$, $rep_{cpe(V)}$, and $rep_{pse(V)}$ in the analogous way. The choice of names is important to reduce the amount of computation required in the instrumentation. We will return to this issue.

We now try and minimize the amount of state computation performed. One important case is when an equivalence relation at a program point or for a symbol's call patterns is *uniform* in the sense that all states belong to the same equivalence class. Clearly, at such points we do not need to track state.

Another case when we can reduce instrumentation is when a non-terminal symbol X is deterministic in the sense that for all relevant q, $|reach(X, q)| = 1$. In this case we do not need to track state when executing the code corresponding to X since we know that the state reached after X will be q' where $\{q'\} = reach(X, q)$.



$$
\begin{aligned}
{}^{\{1\}}M^{\{3\}} &\rightarrow {}^{\{1\}}A^{\{1\},\{2\}}B^{\{2\},\{3\}}test_{\{2\}}{}^{\{2\}}write^{\{3\}}\\
{}^{\{1\}}A^{\{1,2\}} &\rightarrow {}^{\{1\}}\\
{}^{\{1\}}A^{\{1,2\}} &\rightarrow {}^{\{1\}}write^{\{2\}}\\
{}^{\{1\},\{2\}}B^{\{2\},\{3\}} &\rightarrow {}^{\{1\},\{2\}}write^{\{2\},\{3\}}C^{\{2\},\{3\}}\\
{}^{\{2\},\{3\}}C^{\{2\},\{3\}} &\rightarrow {}^{\{2\},\{3\}}\\
{}^{\{2\},\{3\}}C^{\{2\},\{3\}} &\rightarrow {}^{\{2\},\{3\}}read^{\{2\},\{3\}}C^{\{2\},\{3\}}
\end{aligned}
$$

(a) (b)

Fig. 3. A DFA and grammar illustrating determinism

Example 11. To illustrate why determinism is useful consider a policy defined by the DFA in Figure 3(a) and the annotated and partitioned grammar shown in Figure 3(b) Even though the state is important throughout B and C, B is deterministic since $reach(B, 1) = \{2\}$ and $reach(B, 2) = \{3\}$. Hence we do not need to track state inside B or C: we can simply determine the state after B from the state before.

We can weaken this notion of determinism to take into account state equivalence. We say symbol V is *deterministic (modulo equivalence)* if it is either a terminal symbol, corresponds to a terminal symbol, is a test or if for all $q \in cp(V)$, $q_1, q_2 \in reach(V, q)$ implies $q_1 \equiv_{ps} q_2$ where $\equiv_{ps}$ is $pse(V)$.

We can now determine for each symbol V whether it needs an input state or needs to compute an output state. More exactly, V *requires an input state* if $cpe(V)$ is not uniform and either V is not deterministic or the definition of V contains a test (in particular if V is a test itself). V *requires an output state* if $pse(V)$ is not uniform and V is not deterministic.

Example 12. Given the annotated program in Example 10 we find C requires an input state and B requires an output state.

Given an instrumented grammar with tests G_{inst} we create a new minimally instrumented grammar G_{min} as follows. For each non-terminal symbol not corresponding to a terminal symbol V', we have a non-terminal symbol V'' which has an inherited attribute $in : Q$ if V' requires an input state and a synthesized attribute $out : Q$ if V' requires an output state. S''_G is the start state of G_{min}. Now consider a (non-test) annotated production in G_{inst} , $X' \rightarrow {}^{pp_1}X_1'^{pp_2} \ldots {}^{pp_n}X_n'^{pp_{n+1}}$. We first identify at which program points state is required:

- pp_{n+1} *needs state* if X' requires an output state,
- pp_k *needs state* if X'_k requires an input state or pp_{k+1} needs state and epp_k is not uniform, for $k = n, .., 1$.

We construct the corresponding production in G_{min} as follows. We replace all symbols V' in the production by V''.

Now we insert non-terminal symbols to perform the appropriate state equivalence class tracking and creation. We make use of the non-terminal symbols

$create\_state_r$ with a single synthesized attribute out defined by $create\_state_r \rightarrow \epsilon$ with attribute rule $\$\$.out = r$ where r is the name of an equivalence class of states, and the non-terminal symbols $track\_state_F$ with inherited attribute in and synthesized attribute out defined by $track\_state_F \rightarrow \epsilon$ with attribute rule $\$\$.out = F(\$\$.in)$ where F is a function from equivalence classes of states to equivalence classes of states.

We do this as follows. If pp_k needs state, but epp_k is uniform, then insert a $create\_state_r$ at the program point where $r = can_k(q)$ for some $q \in pp_k$. If X''_k requires an input state, then add a $track\_state_F$ immediately before X''_k where $F = can_{cpe(X''_k)} \circ rep_k$ to convert from the names for the equivalence states at that program point to those used for the calling patterns for X''_k. If pp_k needs state and X''_{k-1} requires an output state, then add a $track\_state_F$ immediately after X''_{k-1} where $F = can_k \circ rep_{pse(X''_{k-1})}$ to convert from the names for the equivalence states returned from X''_{k-1} to those used at that program point. Add similar conversion symbols at the start of the production if X' requires an input state, and at the end if X' requires an output state. If pp_k needs state and X''_{k-1} does not require an output state, then add a $track\_state_F$ immediately after X''_{k-1} where F is $\lambda r.can_k(q)$ where $q \in reach(X''_{k-1}, can_{k-1}(r))$ to track the state change in the deterministic (modulo equivalence) symbol X''_{k-1}. We then perform further optimisation by removing calls to $track\_state_F$ when F is the identity and composing adjacent calls to $track\_state_F$ if the intermediate state is not used. Finally we add appropriate attribute rules to link the out attribute of the preceding symbol with the in attribute of the following.

When choosing the names for equivalence states we try to ensure that the state transformer function associated with a deterministic function is the identity function and that those on call entry and exit are identity. A similar idea was previously suggested in [6].

Theorem 9. *Let G_{inst} be the instrumented grammar input to the above algorithm and G_{min} the output grammar. Then $L(G_{inst}) = L(G_{min})$.*

5 Related Work

Approaches to verification of resource usage can be categorised as either static, dynamic or mixed. Static approaches attempt to determine at compile time that a program satisfies the resource usage protocol, dynamic approaches instrument the program so as to abort at runtime before a violation occurs, while mixed approaches like ours use static analysis to minimize the runtime instrumentation.

Most attention has been paid to static approaches. Vault [7] and Cyclone [8] are both safe variants of C which employ type-checking methods to statically enforce correct resource management behaviour of programs. However, in both languages, the programmer is required to provide a significant number of declarations since the system does not perform inference.

Resource usage verification in the context of an object-oriented language has been studied by Tan, Ou and Walker [9]. They extend Java with "scoped methods". Conditions on the usage of such methods can be imposed via explicit program annotations which can be specified by regular expressions.

Igarashi and Kobayashi [10] also propose a powerful static approach for reasoning about resource usages. However, they do not fix how to specify the valid traces of resource usages. Hence, decidability and complexity of resource verification depends on the underlying policy specification language. Another difference is that resource usages are recorded in the environment whereas we use a type and effect system. This makes it difficult in their approach to determine precisely when a certain resource is used. They solve the problem by introducing boxing and unboxing operators which can however easily lead to imprecise analysis results.

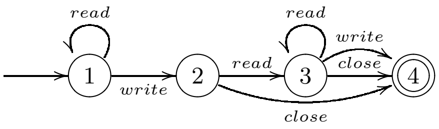
Other recent static approaches are described in Foster, Terauchi and Aiken [11], Hallem, Chelf, Xie and Dengler [12], Das, Lerner and Seigle [13], and Chen and Wagner [14]. The approach by Foster et. al. enforces correct resource usage behaviour via flow sensitive qualifiers which are attached to resources. Qualifiers are taken from a lattice which seem to be less powerful than the DFAs we consider (we believe they cannot specify the policy in Figure 1). The system by Hallem et. al. provides a specification language based on state machines to enforce correct resource usage behaviour. Their main motivation is to find as many serious bugs as possible. The system has been implemented and seems to scale well to large programs. However, soundness has been traded in for efficiency, i. e. some incorrect resource usages might remain undetected. The work by Das et. al. appears to be very similiar to [12]. One of the main differences seems that [13] provides sound analyses. The work of Chen and Wagner uses a DFA M to describe security properties and tests whether CFG G approximating the program is such that $L(G) \subseteq L(M)$ similar to the work herein. Since all of the above works are static approaches, they do not consider how to insert run-time tests into a program when the program cannot be proven to satisfy a particular resource usage policy.

Schneider [4] considers security properties specifiable by DFA's, and uses a dynamic approach that threads the security computation through the code. In [15], Erlingsson and Schneider consider how to specialize a security automaton to a program. Similiar ideas are discussed by Walker [16] and Thiemann [18] in a type-theoretic setting. However, since none of these approaches uses sophisticated static analysis the resulting instrumented program may perform unnecessary tests and state tracking when compared with our mixed approach.

The approach most similar to ours is that of Colcombet and Fradet [6] who also use a mixed approach. They propose instrumenting a program with a sufficient (minimal) number of run-time checks to enforce a certain resource usage policy and use an analysis to determine state equivalences to reduce tracking. Our approach is inherently more accurate and so leads to less instrumentation and run-time checks. This is for two main reasons. First, their approach maps

```
main(n) = if (n mod 2 ==0) write(f);
             a(f,n); write(f); a(f,n); close
a(f,n)  = if (n > 0) (read(f); a(f,n-1))
```

(a)



(b)

Fig. 4. (a) A program fragment and (b) a DFA for a file resource policy.

the program to a finite automata rather than a context-free grammar.[5] This makes the analyses much simpler, in particular determining equivalence classes of states, and extraction of the finite automata from the original program but means they lose accuracy and may introduce unnecessary tests and instrumentation. The second reason is that they do not consider moving tests earlier in the computation and perform the placement of tests and state tracking in the same phase. This may lead to more complicated state tracking since they sometimes track error states and the need to track state for a longer period. On the other hand they focus much more on how to maximize the number of state transformations which are the identity (an NP-complete problem) in the instrumented program.

To illustrate the relative benefits of our approach consider the program and the policy specified in Figure 4. The approach of Colcombet and Fradet gives

```
main(n) = s := 1; if (n mod 2 == 0) (write(f); s := 2);
             a(f,n); if (s == 1) then s := 2 else if (s == 3) then s := 4;
             write(f); a(f,n); if (s != 2 && s != 3) then abort;
             close
a(f,n)  = if (n > 0) (read(f); if (s == 2) then s := 3; a(f,n-1))
```

while our approach places the test earlier and realises there is no need to track state at all

```
main(n) = if (n mod 2 ==0) (abort; write(f)); a(f,n); write(f); a(f,n); close
a(f,n)  = if (n > 0) (read(f); a(f,n-1))
```

6 Conclusion

We have described a powerful and theoretically well-founded process for performing compile-time and run-time verification that a program satisfies a particular resource usage policy specified by a DFA. At the heart of our approach is the approximation of a program's resource usage behaviour by a grammar. Various sophisticated analyses working on this grammar allow us to determine whether the original program satisfies the policy, or if it does not, where run-time tests

[5] Colcombet and Fradet do sketch how to extend their approach to treat the program as a context-free grammar, but this is not formalised and essentially collapses back to the treatment as a finite automata in order to perform state minimization.

and state tracking code need to be inserted into the original program. We believe our work provides a strong theoretical basis for automatic verification of resource usage policies.

References

1. Talpin, J., Jouvelot, P.: The type and effect discipline. In: Proc. of LICS'92, IEEE (1992) 162–173
2. Marriott, K., Stuckey, P., Sulzmann, M.: Resource usage verification. Technical report, University of Melbourne (2003) `www.cs.mu.oz.au/~pjs/ruv.ps.gz`.
3. Sudkamp, T.: Languages and Machines. Addison-Wesley (1988)
4. Schneider, F.B.: Enforceable security policies. Information and System Security **3** (2000) 30–50
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL'77, ACM (1977) 238–252
6. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proc. of POPL'00, ACM (2000) 54–66
7. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proc. of PLDI'01, ACM (2001) 59–69
8. Grossman, D., Morrisett, G., T. Jim, M.H., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: Proc. of PLDI'01, ACM (2001) 282–293
9. G. Tan, X.O., Walker, D.: Resource usage analysis via scoped methods. In: Foundations of Object-Oriented Languages (FOOL'10). (2003)
10. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Proc. of POPL'02, ACM (2002) 331–342
11. J. S. Foster, T.T., Aiken, A.: Flow-sensitive type qualifiers. In: Proc. of PLDI'02, ACM (2002) 1–12
12. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A system and language for building system-specific, static analyses. In: Proc. of PLDI'02, ACM (2002) 69–82
13. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: Proc. of PLDI'02, ACM (2002) 57–68
14. Chen, H., Wagner, D.: Mops: an infrastructure for examining security properties of software. In: Proc. of CCS'02. (2002) 235–244
15. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: A retrospective. In: Proc. of the 1999 New Security Paradigm Workshop. (1999)
16. Walker, D.: A type system for expressive security policies. In: Proc. of POPL'00, ACM (2000) 254–267
17. , Wright, A., Felleisen, M.:, A Syntactic Approach to Type Soundness. *Information and Computation* **115** (1994) 38–94
18. Thiemann, P.: Enforcing safety properties using type specialization. In: Proc. of ESOP'01. Volume 2028 of LNCS., Springer (2001)

A Resource Usage Approximation

Of course the foundation for our approach is the construction of a context-free grammar approximating the resource usage behaviour of a program. We now

$$\text{Evaluation Context } E ::= [] \mid E\ e \mid v\ E \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{open}\ (r :: Res(l)) \mid \mathsf{close}\ (r :: Res(l))$$

$$(\beta)\ (\lambda x.e)\ v \xrightarrow{\emptyset} [v/x]e \quad (\text{Context})\ \frac{e \xrightarrow{w} e'}{E[e] \xrightarrow{w} E[e']}$$

$$(\text{Open})\ \mathsf{open}\ (r :: Res(l)) \xrightarrow{open(l)} () \qquad (\text{Close})\ \mathsf{close}\ (r :: Res(l)) \xrightarrow{close(l)} ()$$

$$(\text{Rec})\ (\mathsf{fix}\ x\ \mathsf{in}\ (\lambda y.e))\ v \xrightarrow{\emptyset} (\lambda y.[\mathsf{fix}\ x\ \mathsf{in}\ (\lambda y.e)/x]e)\ v \qquad (\text{Let})\ \mathsf{let}\ x = v\ \mathsf{in}\ e \xrightarrow{\emptyset} [v/x]e$$

$$(\text{Ite1})\ \mathsf{if}\ True\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \xrightarrow{\emptyset} e_1 \qquad (\text{Ite2})\ \mathsf{if}\ False\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \xrightarrow{\emptyset} e_2$$

$$(\text{Reduce})\ \frac{e \xrightarrow{w} e'}{e \xrightarrow{w}{}^* e'} \quad (\text{Ref})\ e \xrightarrow{\emptyset}{}^* e \quad (\text{Trans})\ \frac{e \xrightarrow{w}{}^* e' \quad e' \xrightarrow{w'}{}^* e''}{e \xrightarrow{w\ w'}{}^* e''}$$

Fig. 5. Resource Usage Semantics

describe how this can be done for a simply-typed call-by-value language (like ML).

We assume resource usage primitives for opening (open) and closing (close) resources. Others such as read, write are left out for brevity. We assume that resources are always attached to resource usage primitives, e.g. open $(r :: Res(l))$. That is, the resource type represents must alias information. The typing rules are straightforward and left out for simplicity. We will sometimes make use of the following abbreviations:

$$\begin{array}{ll} \mathsf{let}\ f\ x = e\ \mathsf{in}\ [f/g]e' & \equiv \mathsf{let}\ g = (\mathsf{fix}\ f\ \mathsf{in}\ (\lambda x.e))\ \mathsf{in}\ e' \\ e; e' & \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e' \quad x \text{ is fresh} \end{array}$$

Note that all recursive expressions must be functions.

We introduce a small-step reduction semantics [17] to trace the sequence of resource usages. A single reduction step is written $e \xrightarrow{w} e'$ where reducing expression e to e' yields the trace w. We assume

$$\begin{array}{lll} \text{Usage Effects} & ue & ::= open(l) \mid close(l) \mid \ldots \\ \text{Traces} & w & ::= \epsilon \mid ue \mid w\ w \end{array}$$

where e.g. $open(l)$ is a primitive resource usage *effect* corresponding to resource usage primitive open used at abstract location l. In case there are no ambiguities, we omit the information about the particular resource and simply write e.g. *open* instead of $open(l)$. The reduction rules can be found in Figure 5. We are only interested in tracing resource usages, therefore we use () as a dummy value in case of rules (Open) and (Close).

We now give a resource usage logic presented in style of a type and effect system [1] which captures the resource usage behavior of a program.

$$\begin{array}{lll} \text{Usage Effects} & ue & ::= open(l) \mid close(l) \mid \ldots \\ \text{Effects} & \delta & ::= \emptyset \mid ue \mid \delta;\delta \mid \delta + \delta \mid \alpha \mid \mu\alpha.\delta \\ \text{Types} & \tau & ::= Res(l) \mid Int \mid Bool \mid \tau \xrightarrow{\delta} \tau \end{array}$$

$$(\text{Var})\ \frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau\&\emptyset} \qquad (\text{Open})\ \Gamma \vdash \mathsf{open}\ (r :: Res(l)) : Res(l)\&open(l)$$

$$(\text{Close})\ \Gamma \vdash \mathsf{close}\ (r :: Res(l)) : \tau\&close(l) \qquad (\text{SubEff})\ \frac{\Gamma \vdash e : \tau\&\delta \quad L(\delta) \subseteq L(\delta')}{\Gamma \vdash e : \tau\&\delta'}$$

$$(\text{Abs})\ \frac{\Gamma.x : \tau \vdash e : \tau'\&\delta}{\Gamma \vdash \lambda x.e : \tau \xrightarrow{\delta} \tau'\&\emptyset} \qquad (\text{App})\ \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\delta_3} \tau_2\&\delta_1 \quad \Gamma \vdash e_2 : \tau_1\&\delta_2}{\Gamma \vdash e_1\ e_2 : \tau_2\&\delta_1;\delta_2;\delta_3}$$

$$(\text{Fix})\ \frac{\Gamma.x : \tau\&\delta \vdash e : \tau\&\delta}{\Gamma \vdash \mathsf{fix}\ x\ \mathsf{in}\ e : \tau\&\delta} \qquad (\text{Let})\ \frac{\Gamma \vdash e : \tau\&\delta \quad \Gamma.x : \tau \vdash e' : \tau'\&\delta'}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : \tau'\&\delta;\delta'}$$

$$(\text{Ite})\ \frac{\Gamma \vdash e_1 : \tau_1\&\delta_1 \quad \Gamma \vdash e_2 : \tau\&\delta_2 \quad \Gamma \vdash e_3 : \tau\&\delta_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau\&\delta_1;(\delta_2+\delta_3)}$$

Fig. 6. Resource Usage Logic

Function arrows are annotated with effects. The empty effect is denoted by $\emptyset$. Primitive usage effects are as above. The effect operator ; represents the usual sequencing operator among effects whereas + represents choice. We assume that ; is associative, + is commutative, ; and + binds tighter than $\mu\alpha$. We generally assume that effects are *closed*, i.e. all effect variables α in a effect δ are bound by a $\mu\alpha$. The $\mu\alpha$ operator allows us to describe recursive effects where α ranges over effects. We often use δ^* as a short-hand for $\mu\alpha.(\delta;\alpha) + \emptyset$ assuming that $\alpha \notin fv(\delta)$.

Effects can equivalently be described by context-free grammars. W.l.o.g., we assume that all bound α's appearing in an effect δ are distinct. Then, we can associate a non-terminal symbol A to each effect variable α. Effects are translated via function $[\![\cdot]\!]$ which yields the starting symbol and a set of production rules.

$$
\begin{array}{ll}
[\![open(l)]\!] = (S, \{S \to open(l)\}) & [\![close(l)]\!] = (S, \{S \to close(l)\}) \\
[\![\alpha]\!] \quad\quad\ = (A, \emptyset) & [\![\emptyset]\!] \quad\quad\ = (S, \{S \to \epsilon\})
\end{array}
$$
$$
\begin{array}{l}
[\![\delta_1;\delta_2]\!] \ = \ (S, \{S \to S_1\ S_2\} \cup R_1 \cup R_2) \quad \text{where } [\![\delta_1]\!] = (S_1, R_1), [\![\delta_2]\!] = (S_2, R_2) \\
[\![\delta_1 + \delta_2]\!] \ = \ (S, \{S \to S_1, S \to S_2\} \cup R_1 \cup R_2) \quad \text{where } [\![\delta_1]\!] = (S_1, R_1), [\![\delta_2]\!] = (S_2, R_2) \\
[\![\mu\alpha.\delta]\!] \ = \ (A, \{A \to S'\} \cup R') \quad \text{where } [\![\delta]\!] = (S', R')
\end{array}
$$

We denote by $L(\delta)$ the set of possible sequences defined by δ, i.e. $L(\delta) = L(G)$ where $G = (\Sigma, N, S, R)$, $L(G) = \{w \mid S \Rightarrow^*_G w\}$ and $[\![\delta]\!] = (S, R)$. The set Σ of terminal symbols and the set N of non-terminal symbols are determined from the context.

We introduce typing judgments of the form $\Gamma \vdash e : \tau\&\delta$ to state that under environment Γ, expression e has type τ where the resource usage behaviour is described by effect δ. The typing rules are given in Figure 6.

We write $e \xrightarrow{w'}^*_n e'$ to denote a n-step reduction of expression e to e' yielding trace w' w.r.t. the rules in Figure 5. We can state that the prefix of a trace in the resource semantics must be part of a trace of the approximated semantics.

$$\text{(Var)} \quad \frac{x : \tau \in \Gamma \quad S \text{ fresh}}{\Gamma, x :: t \vdash_{inf} (\tau \mid S \mid \{S \to \epsilon\})}$$

$$\text{(Open)} \quad \frac{S \text{ fresh} \quad R = \{S \to open(l)\}}{\Gamma, (\texttt{open}\ (r :: Res(l)) :: Res(l)) \vdash_{inf} (Res(l) \mid S \mid R)}$$

$$\text{(Close)} \quad \frac{t \vdash \tau \quad S \text{ fresh} \quad R = \{S \to close(l)\}}{\Gamma, (\texttt{close}\ (r :: Res(l)) :: t) \vdash_{inf} (\tau \mid S \mid R)}$$

$$\text{(Abs)} \quad \frac{t \vdash \tau \quad \Gamma.x : \tau, e :: t \vdash_{inf} (\tau' \mid S' \mid R') \quad S \text{ fresh} \quad R = R' \cup \{S \to \epsilon\}}{\Gamma, (\lambda x :: t.(e :: t')) :: t \to t' \vdash_{inf} (\tau \xrightarrow{S'} \tau' \mid S \mid R)}$$

$$\text{(App)} \quad \frac{\begin{array}{c}\Gamma, e_1 :: t \to t' \vdash_{inf} (\tau_1 \xrightarrow{S_3} \tau_2 \mid S_1 \mid R_1) \quad \Gamma, e_2 :: t \vdash_{inf} (\tau_3 \mid S_2 \mid R_2) \\ S \text{ fresh} \quad R = R_1 \cup R_2 \cup [\![\tau_3 \preceq \tau_1]\!] \cup \{S \to S_1 S_2 S_3\}\end{array}}{\Gamma, ((e_1 :: t \to t')\ (e_2 :: t) :: t') \vdash_{inf} (\tau_2 \mid S \mid R)}$$

$$\text{(Fix)} \quad \frac{\begin{array}{c}t_1 \to t_2 \vdash \tau \quad \Gamma.x : \tau, (\lambda y :: t_1.e :: t_2) :: t_1 \to t_2 \vdash_{inf} (\tau' \mid S \mid R') \\ R = R' \cup [\![\tau' \preceq \tau]\!]\end{array}}{\Gamma, (\mathsf{fix}\ x :: t_1 \to t_2\ \mathsf{in}\ (\lambda y :: t_1.e :: t_2) :: t_1 \to t_2) :: t_1 \to t_2 \vdash_{inf} (\tau \mid S \mid R)}$$

$$\text{(Let)} \quad \frac{\begin{array}{c}\Gamma, e :: t \vdash_{inf} (\tau \mid S \mid R) \quad \Gamma.x : \tau, e' :: t' \vdash_{inf} (\tau' \mid S' \mid R') \\ S'' \text{ fresh} \quad R'' = R \cup R' \cup \{S'' \to SS'\}\end{array}}{\Gamma, (\mathsf{let}\ x :: t\ \mathsf{in}\ e' :: t') :: t' \vdash_{inf} (\tau' \mid S'' \mid R'')}$$

$$\text{(Ite)} \quad \frac{\begin{array}{c}\Gamma, e_1 :: t' \vdash_{inf} (\tau_1 \mid S_1 \mid R_1) \quad \Gamma, e_2 :: t \vdash_{inf} (\tau_2 \mid S_2 \mid R_2) \\ \Gamma, e_3 :: t \vdash_{inf} (\tau_3 \mid S_3 \mid R_3) \quad t \vdash \tau \quad S \text{ fresh} \\ R = R_1 \cup R_2 \cup R_3 \cup [\![\tau_2 \preceq \tau]\!] \cup \\ [\![\tau_3 \preceq \tau]\!] \cup \{S \to S_1 S_2, S \to S_1 S_3\}\end{array}}{\Gamma, (\mathsf{if}\ e_1 :: t'\ \mathsf{then}\ e_2 :: t\ \mathsf{else}\ e_3 :: t) :: t \vdash_{inf} (\tau \mid S \mid R)}$$

Fig. 7. Resource Usage Inference

Lemma 5. *Let* $\Gamma \vdash e : \tau \& \delta$ *such that* $e \overset{w}{\longrightarrow}{}^*_n e'$ *for some* n. *Then* $w\ w' \in L(\delta)$ *for some trace* w'.

For terminating expressions we can state a stronger statement. Each trace of a terminating expression occurs in the approximated semantics.

Theorem 10 (Resource Usage Soundness).
Let $\Gamma \vdash e : \tau \& \delta$ *such that* $e \overset{w}{\longrightarrow}{}^* v$ *for some value* v *and trace* w. *Then* $w \in L(\delta)$.

We present an inference algorithm, directly abstracting the resource usage behaviour of a program in terms of a context-free grammar. we assume that expressions are fully type-annotated and well-typed in the underlying system. The inference algorithm is formulated as a deduction system with clauses of the form $\Gamma, e :: t \vdash_{inf} (\tau \mid S \mid R)$ where environment Γ and the type-annotated expression ($e :: t$) are input values. As output values, we find a resource usage type τ and a pair (S, R). The context-free grammar $G = (\Sigma, N, S, R)$ describes the inferred usage behaviour of the expression where the set Σ of all primitive effects and the set N of non-terminals is determined by the context. In Figure 7, we provide a clause for each syntactic expression.

We introduce judgements $t \vdash \tau$ to generate a "fresh" resource usage type τ of "shape" t.

$$\begin{array}{ll}
\text{(Base)} & Int \vdash Int \qquad Bool \vdash Bool \\
\text{(Res)} & \quad Res(l) \vdash Res(l) \\
\text{(Arrow)} & \dfrac{t_1 \vdash \tau_1 \quad t_2 \vdash \tau_2 \quad A \text{ fresh}}{t_1 \rightarrow t_2 \vdash \tau_1 \xrightarrow{A} \tau_2}
\end{array}$$

By default, non-terminals A are constrained by the empty effect, i.e. $A \rightarrow \epsilon$. However, we omit to keep track of such production rules in the inference system. Instead, we add $A \rightarrow \epsilon$ to the inferred grammar for each non-terminal symbols A which does not appear on the left-hand side of a production rule.

We assume that in the initial environment all function arrows are annotated with non-terminal symbols. Note that this assumption is preserved by the inference rules. Then, we can translate structural constraints into production rules as follows:

$$\begin{array}{ll}
[\![Res(l) \preceq Res(l)]\!] & = \emptyset \\
[\![Int \preceq Int]\!] & = \emptyset \\
[\![Bool \preceq Bool]\!] & = \emptyset \\
[\![\tau_1 \xrightarrow{A} \tau_2 \preceq \tau_1' \xrightarrow{A'} \tau_2']\!] & = \{A' \rightarrow A\} \cup [\![\tau_1' \preceq \tau_1]\!] \cup \\
& \quad [\![\tau_2 \preceq \tau_2']\!]
\end{array}$$

In the following two theorems, Σ and N are determined by the context.

Theorem 11 (Soundness of Inference).
If $\Gamma, e :: t \vdash_{inf} (\tau \mid S \mid R)$ then $\Gamma \vdash e : \tau' \& \delta'$ for some type τ' and effect δ' where $L(\delta') = L(\Sigma, N, S, R)$ and $\tau = \phi(\tau')$ where ϕ is a mapping from non-terminal symbols to effect variables.

Theorem 12 (Completeness of Inference).
If $\Gamma \vdash \tau \& \delta$ then $\Gamma, e :: t \vdash_{inf} (\tau' \mid S \mid R)$ for some start symbol S and set of production rules R such that $L(\Sigma, N, S, R) \subseteq L(\delta')$.

The following example illustrates the inference process.

Example 13. Consider

```
main :: ()
main = let apply :: (Res(r)→())→Res(r)→Int→()
           apply f x n = if n=0
                         then f x
                         else f x; apply f x (n-1)
           g :: Res(r)→()
```

```
    g x = read(x); write(x)
    h :: Res(r)→()
    h x = close(x);
in  (apply g r1, apply h r2)
```

The (slightly simplified) abstraction to context-free grammars yields

$$
\begin{array}{ll}
\texttt{apply} & :: (Res(r) \xrightarrow{S_1} ()) \xrightarrow{S_2} Res(r) \xrightarrow{S_3} Int \xrightarrow{S_4} () \\
\texttt{g} & :: Res(r) \xrightarrow{S_g} () \\
\texttt{h} & :: Res(r) \xrightarrow{S_h} ()
\end{array}
$$

where

$$
\begin{array}{l}
S_1 \quad\ \rightarrow S_g \\
S_1 \quad\ \rightarrow S_h \\
S_2 \quad\ \rightarrow \epsilon \\
S_3 \quad\ \rightarrow \epsilon \\
S_4 \quad\ \rightarrow S_1\ S_4\ S_1 \\
S_g \quad\ \rightarrow read(r)\ write(r) \\
S_h \quad\ \rightarrow close(r) \\
Main \rightarrow S_4\ S_4
\end{array}
$$

Note that our analysis does not support resource polymorphism. Hence, the resource types of functions `g` and `f` must be identical. We are also lacking effect polymorphism which results in a less precise analysis. We plan to consider such extensions in future work.