

Program Slicing

Luca Carettoni
luca.carettoni@securenetwork.it

Andrea Naggi
andrea@naggi.it

ABSTRACT

Con lo sviluppo di sistemi software sempre più complessi ed eterogenei risulta necessario trovare delle tecniche di ausilio alla tradizionale attività di debugging e testing. La tecnica denominata *slicing*¹ si pone proprio in quest'ottica poiché nasce dall'osservazione che naturalmente i programmatori, durante queste fasi, cercano in qualche modo di astrarre parti del codice sotto analisi rispetto ad un particolare punto di interesse. Una *program slice*² è una "fetta" del programma principale che potenzialmente determina il valore di un insieme di variabili in un particolare punto del codice sorgente. Dalla definizione originale di Mark Weiser (1979) [22] sono state presentate molte implementazioni e tecniche ausiliarie per superare i problemi intrinseci nell'analisi statica del codice sorgente al fine di ottenere una *slice* che risultasse corretta rispetto ad alcuni parametri e nello stesso tempo contenente solamente le istruzioni realmente influenti. In questo articolo vogliamo quindi riassumere lo stato dell'arte a cui la ricerca è giunta in questi anni, accennando alla reale applicabilità del metodo in contesti produttivi.

1. INTRODUZIONE

Decomporre i programmi in moduli è sempre stato un approccio molto usato per far fronte alla crescente complessità dei sistemi. Anche la massima "*Divide et Impera*" è diventata, con la programmazione Object Oriented, un rimedio efficace per lo sviluppo e la manutenzione di software, evidenziando come sia importante focalizzare la propria attenzione su singoli componenti al fine di realizzare un buon prodotto. Ricavare una *slice* significa infatti ridurre un programma alle sole istruzioni che influenzano il valore delle variabili in un determinato punto del codice che viene denominato *slicing criterion*.

¹**slice** *vb sliced; slic-ing* **1** : to cut a slice from; also to cut into slices

(*The Merriam-Webster Dictionary*)

²**slice** {slis} *n* **1** : a thin flat piece cut from something

(*The Merriam-Webster Dictionary*)

Lo slicing criterion, nella sua accezione più classica, è definito come una coppia di valori (**numero-di-linea**, **variabile**). Lo *slicing* è quella tecnica in grado di realizzare in maniera **automatica** questa riduzione del codice sorgente iniziale, evidenziando solamente le istruzioni che contribuiscono alla determinazione del valore di una specifica variabile. Consideriamo come esempio il semplice frammento di codice:

```
1: read(n);
2: i := 1;
3: sum := 0;
4: product := 1;
5: while (i <= n) do
    begin
6:   sum := sum + 1;
7:   product := product + 1;
8:   i := i + 1;
    end
9: write(sum);
10: write(product);
```

Figura 1: Un primo esempio di *slicing*

Effettuando ora lo slicing con il seguente criterio (**10,product**), quello che vogliamo ottenere è un programma ridotto e coerente rispetto al punto di interesse considerato.

La slice sarà quindi costituita dalle istruzioni presenti nel programma precedente ad esclusione delle istruzioni alle righe **3**, **6** e **9**.

Nel 1979, Mark Weiser durante la sua tesi di PHD presso l'università del Maryland compì degli esperimenti su un gruppo eterogeneo di sviluppatori [22] facendo loro effettuare un'intensa attività di debugging su differenti programmi. Successivamente i soggetti vennero sottoposti ad un questionario in cui erano riportati singoli blocchi di istruzioni e in cui dovevano esprimere un giudizio sulla attinenza rispetto agli algoritmi appena analizzati.

Da questo semplice esperimento, Weiser confermò quanto di fatto era già stato notato in uno studio precedente [18]: tutte le persone, effettuando analisi correttiva sul codice sorgente di un programma, creano una rappresentazione mentale degli algoritmi, la quale li aiuta a comprendere meglio il meccanismo di funzionamento e a individuare il punto di *fault*; automatizzando questo processo si potrebbe quindi agevolare ulteriormente il programmatore. Dalla definizione originale di questa tecnica ad opera di Weiser, basata unicamente sull'analisi statica del codice sorgente (*static slicing*) sono state poi intraprese numerose ricerche per superare i li-

miti di questo metodo. Da un lato si è cercato di superare le problematiche inerenti a particolari costrutti dei linguaggi di programmazione come procedure, operatori di controllo del flusso, puntatori, array, etc. estendendo inoltre la tecnica ai nuovi approcci basati sul paradigma ad oggetti grazie alla definizione di nuovi algoritmi o di tecniche semplificate; dall'altro invece si è cercato di utilizzare ulteriori informazioni sull'input di una particolare esecuzione al fine di ridurre dinamicamente il codice (*dynamic slicing*) per poi effettuare considerazioni analoghe al caso statico.

Nel paragrafo 2 introdurremo brevemente alcune possibili applicazioni di questa tecnica; nel paragrafo 3 illustreremo l'algoritmo per lo slicing statico; nel paragrafo 4 verranno illustrati quattro approcci incrementali allo slicing dinamico, rimandando il lettore al paragrafo 5 per un breve accenno su altre tipologie di slicing. Infine nel paragrafo 6 analizzeremo gli strumenti di slicing disponibili agli sviluppatori.

2. CONTESTO APPLICATIVO

Come è possibile intuire il program slicing è in grado di assistere gli sviluppatori in numerose attività che vanno ben oltre il debugging del codice sorgente. In questo paragrafo cercheremo di analizzare altri possibili utilizzi del program slicing nelle numerose fasi di sviluppo di un programma.

Program Differencing: Spesso i programmatori si trovano di fronte al problema di rilevare le differenze tra due programmi. Esistono molte tecniche per trovare le differenze **testuali** tra programmi, ma queste spesso risultano insufficienti. Il program slicing può essere utilizzato per identificare le differenze **semantiche** tra due programmi. Sarà infatti sufficiente applicare un algoritmo di slicing basato su grafi delle dipendenze per portare alla luce le differenze tra i due programmi che vanno ad influenzare una data istruzione.

Program Integration: Dati un programma *Base* e due sue varianti *A* e *B*, ottenute modificando copie diverse di *Base*, lo scopo dell'integrazione di programmi è quello di determinare se le modifiche interferiscano tra loro e, se non lo fanno, di creare un programma integrato che incorpora le modifiche di entrambe le versioni *A* e *B*, insieme alle parti di *Base* che non sono state modificate da nessuna delle due. Per procedere all'integrazione è necessario per prima cosa identificare le modifiche apportate dalle due varianti usando il *Program Differencing* discusso in precedenza. La parte di *Base* da preservare è ottenuta identificando le porzioni isomorfe delle slice tra *A*, *B* e *Base*. Il programma unificato è ottenuto dall'unione delle componenti da preservare di *Base* con le differenze tra *Base* e *A* e con le differenze tra *Base* e *B*. Questo programma viene poi controllato per verificare che non vi siano interferenze.

Software Maintenance: gli addetti al mantenimento di un programma dopo il suo rilascio si trovano di fronte a problemi simili a quelli dei *program integrator*: devono capire a fondo il software esistente ed integrare le eventuali modifiche senza che queste abbiano un impatto negativo sulle parti di codice non modificate. Per questo compito è possibile utilizzare un tipo particolare di slice [8], chiamata *decomposition slice*. Una *decomposition slice* è in grado di catturare tutte le istruzioni che hanno a che fare con una variabile, ed è indipendente dalla posizione nel programma. Una *decomposition slice* può risultare particolarmente utile quando i programmatori sanno che una variabile *v* dovrà

cambiare valore nel corso dell'esecuzione del programma.

Testing: gli addetti al mantenimento di un programma spesso si trovano a dover effettuare dei *regression test*: cioè a ritestare il software dopo averlo modificato. Questo processo porta ad eseguire il programma in un gran numero di casi di test, anche per la più piccola delle modifiche. In letteratura sono stati presentati alcuni algoritmi che usano il program slicing per determinare quali componenti di un programma siano stati influenzati transitivamente da una modifica ad una istruzione *p*. Possono essere utilizzate slice multiple (*backward* e *forward*, vedi par. 5) oppure algoritmi che deducono i componenti da ritestare a partire da un grafo delle dipendenze. In alternativa è possibile utilizzare il *program differencing* discusso in precedenza per ridurre la dimensione del programma da ritestare.

Debugging: effettuare il debug dei programmi è sempre stato considerato un compito difficile. Spesso la parte più difficile del debugging sta proprio nello scoprire il bug. Questo è dovuto al fatto che nella porzione di codice da esaminare possono esserci molte istruzioni e non tutte hanno necessariamente effetto sull'istruzione che effettivamente crea il problema. Uno strumento che calcola le program slice è un aiuto fondamentale per chi deve effettuare il debugging di un programma. Permette al programmatore di concentrarsi solo sulle istruzioni che contribuiscono ad un malfunzionamento, rendendo più efficiente l'identificazione del punto di *fault* [13]. Il *dynamic slicing*, descritto nel paragrafo 4, è in particolar modo utile per effettuare debugging legato ad uno specifico test case, poichè mette in luce tutte le istruzioni che certamente influiscono su una variabile.

Software Quality Assurance: un particolare problema che gli auditor di software si trovano ad affrontare è la localizzazione del codice *safety critical* all'interno dell'applicazione e l'identificazione degli effetti che queste porzioni di codice hanno su tutto il sistema. Il program slicing può essere utilizzato a questi fini per selezionare le istruzioni che influenzano il valore di una variabile che fa parte di una porzione *safety critical* del codice. In secondo luogo le tecniche di program slicing possono essere utilizzate per certificare la *functional diversity* (per esempio per assicurarsi che non ci siano interazioni tra due o più componenti *safety critical* e che non ci siano interazioni tra componenti non *safety critical* e componenti che invece lo siano). Il program slicing viene utilizzato per la validazione della *safety* in questo modo: per prima cosa gli auditor identificano i componenti critici del sistema utilizzando una *fault tree analysis*; le porzioni di software che vengono invocate durante una condizione pericolosa vengono quindi identificate. A questo punto gli auditor possono localizzare le variabili del programma che sono indicatrici di una condizione di pericolo. Le program slice sono quindi estratte da queste variabili interessanti e usate per verificare che non ci siano interazioni tra componenti critici e componenti non critici utilizzando le tecniche di *slicing*, descritte nel paragrafo 5.

Reverse Engineering: il reverse engineering ha a che fare con il problema della comprensione del design attuale di un programma e con il modo in cui questo programma risulta differente dal design originale. Il program slicing risulta utile per concentrare gli sforzi sulle porzioni di programma che hanno subito più modifiche durante la fase di manutenzione. Per identificarle sarà sufficiente creare una rete di slice ordinate secondo la relazione *is-a-slice-of* [4]. Confrontando le due reti, quella tratta dal design originale e quella ricavata

dal programma dopo anni di manutenzione, saranno subito evidenti le differenze e si potranno concentrare su quelle gli sforzi di reverse engineering. Sempre per aiutare questo processo è stata introdotta una speciale nozione di slicing: l'*interface slicing* [5]. Spesso infatti per poter procedere con il reverse engineering è necessario identificare le astrazioni principali dei moduli di un programma e le interfacce tra di esse. Una *interface slice* è essenzialmente una forward slice, vedi paragrafo 5, presa rispetto ai nodi di ingresso in una collezione di procedure. Il suo scopo è quindi quello di isolare i comportamenti che un certo modulo esporta al software che lo contiene.

3. STATIC SLICING

Dopo aver illustrato le enormi potenzialità che questa tecnica potrebbe avere nel ciclo di sviluppo del software, ritorniamo sino alla sua prima apparizione in letteratura: nel 1979, Mark Weiser pubblica un articolo [22] in cui illustra la tecnica secondo un approccio di tipo statico. Weiser definisce il processo di computazione della slice usando solo informazioni statiche, senza fare quindi nessuna assunzione sull'input; la program slice S è un programma eseguibile e ridotto ottenuto da un altro programma P , rimuovendo parti di codice, in maniera da mantenere comunque una consistenza funzionale rispetto allo slicing criterion (n, V) .

Lo *slicing criterion*, per un particolare programma, definisce una sorta di finestra di osservazione del comportamento del codice stesso. Questa finestra è definita tramite il numero di riga (n) e il set di variabili da controllare (V). Il programma ridotto possiede quindi idealmente due caratteristiche fondamentali: è *minimale* ed *eseguibile*.

Minimale poichè contiene il minor numero di linee di codice tali da rappresentare tutti i punti di interesse per lo slicing criterion: non deve esistere un'altra slice, per lo stesso criterio (n, V) , con un minor numero di istruzioni. Questa proprietà è importante poichè influisce sulla reale utilità del metodo: una slice, relativamente breve, focalizza l'attenzione dello sviluppatore e del tester sulle parti fondamentali dell'algoritmo, rimuovendo quelle parti inessenziali che generano confusione e ridondanza. Purtroppo però è possibile formalmente mostrare che nessun algoritmo può sempre calcolare una slice con la minor cardinalità, in quanto non è possibile valutare l'equivalenza di due differenti parti di codice. Questa limitazione suggerisce di adattare il concetto di "dimensione della slice" specificatamente ad ogni singola analisi che si effettua. E' necessario evidenziare come il limite massimo sia comunque definito dal programma completo che può essere considerato una slice esso stesso.

La proprietà di eseguibilità è invece fondamentale per poter assicurare il corretto comportamento della slice a seguito dell'operazione di cancellazione di righe di codice dal programma originale. Il comportamento desiderabile della slice deve essere uguale a quello del programma originale per le parti considerate e per tutti i possibili valori di input. Definire una slice sul codice significa solamente guardare lo stesso sistema (e quindi lo stesso comportamento) secondo una particolare finestra di osservazione. Questo requisito, che appare ragionevole, è però troppo difficile da soddisfare poichè il programma potrebbe anche non terminare. Gödel e Turing dimostrando l'indecidibilità della terminazione di un programma, portano Weiser a definire un vincolo meno

stringente: il programma originale e la slice devono avere il medesimo comportamento se il programma originale termina. Con una slice dotata delle proprietà sopra illustrate è quindi possibile eseguire la slice e verificare il valore delle variabili considerate esattamente come se quel pezzo di codice fosse inserito all'interno di un'esecuzione che interessa tutto il programma.

L'approccio di Weiser viene supportato da un algoritmo che fa uso di una rappresentazione grafica intermedia del codice sorgente: il *Data Flow Graph* (DFG) su cui calcolare il set di istruzioni direttamente o indirettamente rilevanti, in accordo con le dipendenze di tipo data flow o control flow. Prima di analizzare questo algoritmo facciamo qualche breve considerazione sulle altre soluzioni implementative che, negli anni, sono state sviluppate. Oltre alle tecniche basate su Information-Flow Relation [6] sono estremamente interessanti quelle basate sul *Program Dependence Graph* (PDG) [15]. Dopo aver costruito il grafo delle dipendenze del programma, il problema del calcolo della slice è ridotto ad un semplice problema di raggiungibilità a partire dal nodo rappresentante il punto di interesse nel grafo del programma. La costruzione del Program Dependence Graph verrà spiegata in dettaglio nel paragrafo 4. Da questo grafo, la computazione della slice risulta essere un processo efficiente poichè è possibile riusare lo stesso grafo per qualsiasi slicing criterion sullo specifico programma; una volta ottenuta la slice è necessario trasformarla nuovamente in istruzioni. L'efficienza dell'implementazione deriva dal fatto che sebbene il processo di costruzione del PDG è $\Theta(n^2)$ sul numero di istruzioni, il calcolo della slice, che può essere ripetuto per diversi criteri, è lineare rispetto alle dimensioni del grafo (e quindi rispetto al numero di linee di codice).

In tutti gli approcci che abbiamo elencato, da un algoritmo base che considerava semplici programmi lineari³ si è poi passati ad introdurre tutti i costrutti particolari per specifici linguaggi. In letteratura sono quindi riportate soluzioni specifiche per sopperire a quei costrutti che causano problemi durante l'analisi statica: array [7], puntatori [7], goto [1], chiamate a procedure [20] piuttosto che nuovi paradigmi derivanti dalla programmazione a oggetti [14].

Algoritmo semplificato per straight line program

Per aiutare il lettore nella comprensione dell'algoritmo di slicing, secondo l'approccio classico di Weiser, definiremo anche noi inizialmente l'algoritmo per straight line program introducendo solo successivamente le modifiche al fine di adattarlo alla computazione della slice sull'esempio proposto nell'introduzione (vedi figura 1).

```
23.          foo := bar*2
Def(23) = {foo}
Ref(23) = {bar}
```

Figura 2: Gli insiemi REF e DEF.

Il *Control Flow Graph* di un programma P è un grafo orientato in cui ad ogni nodo è associata un'istruzione di P men-

³Programmi senza istruzioni che modificano il flusso (if, while, etc.)

tre ogni arco rappresenta il flusso di controllo in P stesso. Ogni nodo n nel grafo ha associati tre particolari insiemi: $REF(n)$, come set di variabili referenziate in una particolare istruzione, $DEF(n)$, come set di variabili definite in n e $REL(n)$, come set di variabili rilevanti per la particolare linea di codice. Con riferimento all'assegnamento in figura 2 possiamo vedere come vengono definiti gli insiemi REF e DEF .

L'algoritmo viene eseguito in modalità *backward* poichè il calcolo della slice avviene a partire dal punto su cui è definito il criterio, analizzando il grafo all'indietro. A partire dal Control Flow Graph e dagli insiemi appena definiti, calcoliamo gli insiemi REL e la slice tramite il seguente algoritmo iterativo rispetto al **criterio (n,v)**:

1. Inizializza \forall nodo, tutti gli insiemi REL a \emptyset .
2. Inserisci ogni variabile $\in \mathbf{v}$ in $REL(n)$
3. \forall nodo \mathbf{m} , predecessore di \mathbf{n} :
 $REL(m) = REL(n) - DEF(m)$
 $if (REL(n) \cap DEF(m)) \neq \emptyset then$
 $\{$
 $REL(m) = REL(m) \cup REF(m)$
 Includi m nella slice
 $\}$
4. Procedendo in maniera backward, ripeti il passo precedente sino al nodo origine del grafo.

Nell'algoritmo così definito si ipotizza che ogni istruzione abbia un singolo predecessore, ovvero che il programma sia strettamente lineare. Per contemplare la presenza di modificatori di flusso (istruzioni *if*, *while*, etc.) dobbiamo introdurre alcune modifiche: considerare un nuovo insieme chiamato *control set*, eventuali iterazioni dovute a cicli nel grafo e definire una regola per unire gli insiemi REL ; quest'ultima operazione è necessaria nei punti di congiunzione del grafo (*join point*) che corrispondono alle istruzioni che determinano la creazione di due o più branch nel codice. Il *control set* è un insieme che viene associato ad ogni nodo del grafo e contiene le etichette di tutti i nodi che sono direttamente rilevanti per l'esecuzione di quel punto del codice; nel caso di un'istruzione *if.else*, i nodi contenuti nel ramo dell'*if-selection* conterranno nel *control set* l'indicazione dell'istruzione *if* (il numero del nodo).

Nei join points, ovvero nei punti in cui risalendo il grafo due nodi hanno lo stesso predecessore, il REL set sarà definito come quell'insieme dato dall'unione dei REL set dei due nodi successivi considerando inoltre le variabili all'interno del predicato di controllo che risultano indirettamente rilevanti per il calcolo della slice. Questa approssimazione, che certamente aumenta la dimensione della slice, è necessaria in quanto a priori, durante l'analisi statica, non è possibile privilegiare un cammino rispetto ad un'altro. Per gestire l'eventuale presenza di loop nel grafo, dobbiamo anche contemplare la possibilità di iterare ulteriormente il nostro algoritmo al fine di portare a convergenza i REL set associati a nodi presenti nel loop. Hausler [16] ha comunque mostrato come il numero massimo di iterazioni è pari al numero di assegnamenti presenti nel loop: risultando finito potrà essere computato in maniera automatica.

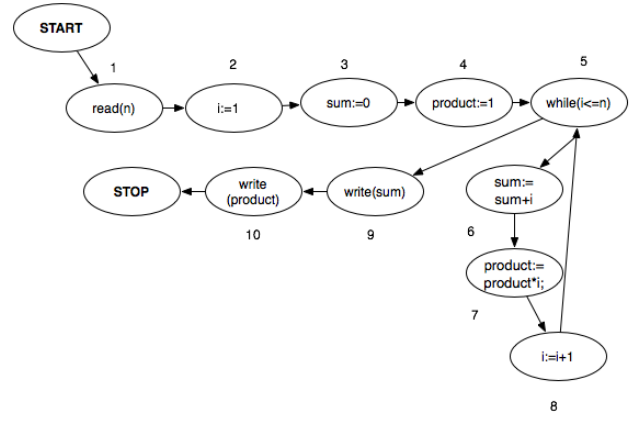


Figura 3: *Control Flow Graph* per il programma di esempio di figura 1

Dopo aver brevemente illustrato il funzionamento dell'algoritmo di slicing secondo l'approccio classico, vogliamo ora riportare un esempio di esecuzione di tale procedimento automatico sul codice che abbiamo presentato nell'introduzione (vedi figura 1). Dall'analisi del codice è possibile costruire il Control Flow Graph di figura 3 e su questo applicare il procedimento illustrato con il **criterio (10, product)**, a partire appunto dal nodo 10.

Nodo	DEF	REF	CTR	REL ⁰
1	{n}	\emptyset	\emptyset	\emptyset
2*	{i}	\emptyset	\emptyset	\emptyset
3	{sum}	\emptyset	\emptyset	{i}
4*	{product}	\emptyset	\emptyset	{i}
5*	\emptyset	{i, n}	\emptyset	{product, i}
6	{sum}	{sum, i}	5	{product, i}
7*	{product}	{product, i}	5	{product, i}
8*	{i}	{i}	5	{product, i}
9	\emptyset	{sum}	\emptyset	{product}
10*	\emptyset	{product}	\emptyset	{product}

Tabella 1: Prima iterazione dell'algoritmo di static slicing.

Nodo	DEF	REF	CTR	REL ¹
1*	{n}	\emptyset	\emptyset	\emptyset
2*	{i}	\emptyset	\emptyset	{n}
3	{sum}	\emptyset	\emptyset	{i, n}
4*	{product}	\emptyset	\emptyset	{i, n}
5*	\emptyset	{i, n}	\emptyset	{product, i, n}
6	{sum}	{sum, i}	5	{product, i, n}
7*	{product}	{product, i}	5	{product, i, n}
8*	{i}	{i}	5	{product, i, n}
9	\emptyset	{sum}	\emptyset	{product}
10*	\emptyset	{product}	\emptyset	{product}

Tabella 2: Seconda iterazione dell'algoritmo di static slicing.

Nelle tabelle 1 e 2, i nodi contrassegnati con il carattere * sono quelli che soddisfano la condizione indicata all'interno del predicato *if* nel punto 3 dell'algoritmo e che quindi com-

pongono la slice. E' da notare come solamente alla seconda iterazione (vedi tabella 2), considerando le variabili indirettamente rilevanti presenti nel predicato di controllo dell'istruzione *while*, la slice costruita è uguale a quella che ci si aspettava analizzando manualmente il codice, come visto nell'introduzione.

4. DYNAMIC SLICING

La nozione classica di *program slice* non tiene conto dei valori di input del programma; spesso però risulta particolarmente utile nel debugging tenere traccia del comportamento del programma nelle condizioni che hanno determinato un bug. Korel e Laski [12] affrontano il problema proponendo la controparte dinamica del program slicing classico introdotto da Weiser [23]. Lo slicing dinamico è in grado di trovare tutte le istruzioni che hanno **realmente** influito sul valore di una variabile per una *particolare esecuzione* di un programma. Per calcolare la slice dinamica Korel e Laski prendono spunto dall'algoritmo di Weiser per lo slicing statico introducendo una soluzione basata su equazioni *data-flow*. Le slice dinamiche sono calcolate a partire da una *execution history* (chiamata *trajectory* in [12]). La *execution history* è una lista delle istruzioni del programma registrate durante la sua esecuzione nell'ordine in cui esse sono state effettivamente eseguite. La *execution history* di un programma in

```

1:  read(n);
2:  i := 1
3:  while (i <= n) do
      begin
4:      if (i mod 2 = 0) then
5:          x := 17
      else
6:          x := 18
7:          i := i + 1
      end
8:  write(x)

```

Figura 4: Esempio 1 per il dynamic slicing

un certo caso di test viene indicata con $\langle x_1, x_2, \dots, x_n \rangle$ in cui x_i indica la *i-esima* istruzione del programma. Nel caso un'istruzione venga eseguita più volte, per esempio se si trova all'interno di cicli, le diverse istanze vengono differenziate da un numero in apice. In letteratura sono state presentate numerose soluzioni ai problemi più complessi legati allo slicing dinamico, ad esempio per la gestione di puntatori [10], array e, ovviamente, per supportare a pieno i linguaggi ad oggetti. Tuttavia ci concentreremo solo sulle soluzioni base, poichè le ottimizzazioni proposte riprendono la filosofia e gran parte del modus operandi di questi algoritmi.

Data Flow Problem

Korel e Laski definiscono una slice dinamica come un programma eseguibile ridotto. Le tre equazioni data-flow introdotte da Korel e Laski formalizzano le dipendenze tra le occorrenze di un'istruzione nella *execution history*. La relazione *Definition-Use* (*DU*) associa l'uso di una variabile con la sua ultima definizione. La relazione *Test-Control* (*TC*) associa la più recente occorrenza di un predicato di controllo con le occorrenze delle istruzioni che sono dipendenti da quel predicato. Le occorrenze della stessa istruzione sono legate dalla *Identity Relation* (*IR*).

$$\begin{aligned}
DU &= \{ (1^1, 3^3), (1^1, 3^7), (1^1, 3^{11}), (2^2, 3^3), \\
&\quad (2^2, 3^3), (2^2, 4^4), (2^2, 7^6), (7^6, 3^7), \\
&\quad (7^6, 4^8), (7^6, 7^{10}), (5^9, 8^{12}), (7^{10}, 3^{11}) \} \\
TC &= \{ (3^3, 4^4), (3^3, 6^5), (3^3, 7^6), \\
&\quad (4^4, 6^5), (3^7, 4^8), (3^7, 5^9), \\
&\quad (3^7, 7^{10}), (4^8, 5^9) \} \\
IR &= \{ (3^3, 3^7), (3^3, 3^{11}), (3^7, 3^3), \\
&\quad (3^7, 3^{11}), (3^{11}, 3^3), (3^{11}, 3^7), \\
&\quad (4^4, 4^8), (4^8, 4^4), (7^6, 7^{10}), (7^{10}, 7^6) \}
\end{aligned}$$

Figura 5: Relazioni Data Flow per il programma di figura 4 con input $n = 2$

In figura 5 sono riportate le relazioni data flow per il programma in figura 4 eseguito con input $n = 2$, quindi secondo la *execution history* $\langle 1^1, 2^2, 3^3, 4^4, 6^5, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12} \rangle$. (La *execution history* appena indicata può anche essere scritta secondo questa notazione, usata da Agrawal e Horgan: $\langle 1, 2, 3^1, 4^1, 6, 7^1, 3^2, 4^2, 5, 7^2, 3^3, 8 \rangle$ con intuibile significato degli apici.)

Le slice dinamiche sono calcolate in maniera iterativa, determinando set successivi S^i di istruzioni rilevanti. Per uno *slicing criterion* (x, I^q, V) , dove x indica l'input, I^q una particolare istruzione nella *execution history* e V una variabile, l'approssimazione iniziale di S^0 contiene le ultime definizioni delle variabili in V e i predicati di controllo della *execution history* da cui I^q dipende. L'approssimazione S^{i+1} è definita in questo modo:

$$\begin{aligned}
S^{i+1} &= S^i \cup A^{i+1} \text{ dove } A^{i+1} \text{ è costituita da:} \\
A^{i+1} &= \{X^p | X^p \in S^i, (X^p, Y^t) \in (DU \cup TC \cup IR) \text{ per alcune} \\
&\quad Y^t \in S^i, p < q\}
\end{aligned}$$

La relazione *IR* viene introdotta per un semplice motivo: le relazioni *DU* e *TC* attraversano la *execution history* solo in avanti, mentre la relazione *IR* è in grado di attraversarla in entrambe le direzioni, permettendo quindi di includere tutte le istruzioni e i predicati di controllo che sono necessari ad assicurare la terminazione di tutti i cicli nella slice, condizione necessaria affinché una slice sia eseguibile. Sfortunatamente l'attraversamento della *execution history* in senso inverso effettuato dalla relazione *IR* porta spesso all'inclusione di istruzioni non strettamente necessarie all'eseguibilità della slice, generando quindi slice sovradimensionate.

Program Dependence Graph

Utilizzato nell'approccio proposto da Ottenstein e Ottenstein [15], il *Program Dependence Graph*, abbreviato *PDG*, è un grafo che ha un nodo per ogni istruzione semplice (assegnamenti, letture, scritture, etc.) e un nodo per ogni espressione di un predicato di controllo (l'espressione condizionale di un costrutto *if-then-else*, *while*, etc.). Il *Program Dependence Graph* ha due tipi di archi orientati: archi *data-dependence* e archi *control-dependence*. Un arco *data-dependence* dal nodo v_i al nodo v_j implica il fatto che l'elaborazione effettuata al nodo v_i dipende direttamente dal valore calcolato al vertice v_j . Un arco *control-dependence* dal nodo v_i al nodo v_j indica che il nodo v_i potrebbe o meno essere eseguito a seconda del risultato booleano dell'espressione del predicato di controllo al nodo v_j .

```

begin
S1:   read(x);
S2:   if (x < 0)
      then
S3:     y := F1(x);
S4:     z := G1(x);
      else
S5:     if (x = 0)
          then
S6:       y := F2(x);
S7:       z := G2(x);
          else
S8:       y := F3(x);
S9:       z := G3(x);
          end if;
      end if;
S10:  write(y);
S11:  write(z);
end

```

Figura 6: Esempio 2 per il dynamic slicing

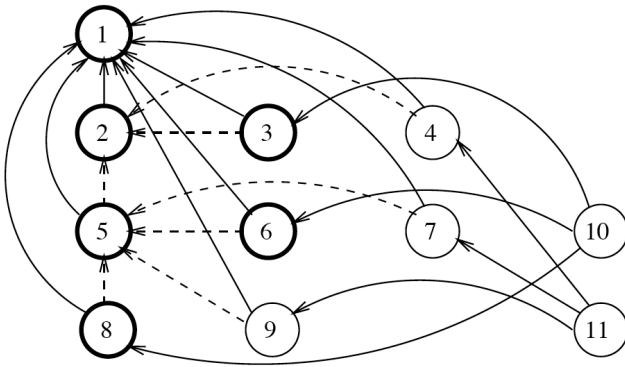


Figura 7: Program Dependence Graph per il programma di figura 6.

La figura 7 mostra il Program Dependence Graph del programma in figura 6 in cui gli archi data-dependence sono tracciati con linee continue e gli archi control-dependence con linee tratteggiate.

Poichè per il calcolo della slice dinamica non ci interessa distinguere tra i due tipi di archi, d'ora in avanti entrambi verranno indicati con linee continue.

Graph Reachability Problem

In letteratura sono stati proposti numerosi algoritmi oltre a quello di Korel e Laski per il calcolo di una slice dinamica. I più interessanti sono basati su un particolare tipo di grafo: il Program Dependence Graph e su alcune sue varianti. Agrawal e Horgan [3] introducono i primi algoritmi per il calcolo di una slice dinamica a partire da un grafo delle dipendenze. Questi quattro algoritmi rappresentano un raffinamento incrementale della tecnica, utile per comprendere al meglio la procedura di calcolo di una slice dinamica. Viene indicata la execution history del programma con la sequenza $\langle v_1, v_2, \dots, v_n \rangle$ dei nodi del Program Dependence Graph, aggiunti nell'ordine in cui essi vengono eseguiti. Le slice dinamiche, secondo Agrawal e Horgan [3], sono sempre riferite alla fine della execution history; per ottenere una

slice riferita ad un punto intermedio del programma è sufficiente tagliare la execution history alla istruzione desiderata e ignorare le successive. Agrawal e Horgan propongono quattro diversi approcci per il calcolo della slice dinamica: i primi due operano marcando alcune parti di un Program Dependence Graph come **eseguito** (nodi e archi rispettivamente). Entrambi questi approcci portano a slice dinamiche inaccurate, ma utili per comprendere l'algoritmo applicato. È interessante notare come il risultato del secondo approccio di Agrawal e Horgan, che mostreremo successivamente, porti esattamente alla stessa slice calcolata con le equazioni data-flow introdotte da Korel e Laski.

Primo Approccio

Il primo approccio è il più semplice e applica un algoritmo di static slicing ad un Program Dependence Graph ridotto ottenuto marcando i nodi effettivamente eseguiti.

To obtain the dynamic slice with respect to a variable for a given execution history, first take the projection of the Program Dependence Graph with respect to the nodes that occur in the execution history, and then use the static slicing algorithm on the projected Dependence Graph to find the desired dynamic slice. [3]

Dopo aver marcato il Program Dependence Graph l'algoritmo procede attraversando il grafo a partire dall'ultimo nodo in cui la variabile dello slicing criterion è stata definita e solo sui nodi marcati al passo precedente. Il set di nodi raggiunti è la slice dinamica cercata. Per costruzione questa slice dinamica contiene solo i nodi, quindi le istruzioni che sono state **veramente** eseguite. Purtroppo questa soluzione non è precisa, in quanto non è in grado di rilevare situazioni in cui effettivamente esiste un arco data dependence tra due nodi marcati v_1 e v_2 ma le definizioni del nodo v_2 non sono in realtà usate da v_1 . Le slice ottenute con questo approccio tendono ad essere sovradimensionate ancora più di quelle ottenute con le equazioni data flow di Korel e Laski, anche se sono decisamente facili da ottenere.

Secondo Approccio

Il secondo approccio risolve i limiti del primo marcando gli archi del Program Dependence Graph, invece dei nodi, ed attraversandolo successivamente solo sugli archi marcati.

Mark the edges of the Program Dependence Graph as the corresponding dependencies arise during the program execution; then traverse the graph only along the marked edges to find the slice. [3]

Ottenuto il grafo si procede con l'algoritmo di slicing dinamico selezionando i nodi raggiungibili a partire dal nodo che contiene l'ultima definizione della variabile a cui siamo interessati. È stato dimostrato [2] che, se il programma non ha cicli, questo approccio calcola sempre una slice accurata. In presenza di cicli può capitare che l'approccio includa più istruzioni del necessario. Per risolvere il problema legato ai cicli potremmo richiedere che l'algoritmo, prima di

marcare gli archi data dependence per una nuova occorrenza di un'istruzione nella execution history, rimuova ogni marca precedentemente apposta sugli archi in uscita da quel nodo. Purtroppo questa soluzione porterebbe a slice errate in altre situazioni, per ovviare alle quali viene introdotto il terzo approccio.

Terzo Approccio

L'approccio precedente porta a slice sovradimensionate poiché un'istruzione può comparire più volte nella execution history e ogni occorrenza di un'istruzione può avere un differente set di archi legati alle definizioni delle variabili usate dall'istruzione stessa, a seconda dell'iterazione del ciclo a cui appartiene.

Create a separate node for each occurrence of a statement in the execution history, with outgoing dependence edges to only those statements (their specific occurrences) on which this statement occurrence is dependent. [3]

Ogni nodo nel nuovo grafo delle dipendenze avrà al massimo un solo arco in uscita per ogni variabile usata in quell'istruzione. Questo nuovo grafo è chiamato *Dynamic Dependence Graph*. La figura 9 mostra un esempio della notevole dimen-

```

begin
S1:   read(n);
S2:   i := 1;
S3:   while (i <= n)
      do
S4:     read(x);
S5:     if (x < 0)
      then
S6:       y := F1(x);
      else
S7:       y := F2(x);
      end_if;
S8:     z := F3(y);
S9:     write(z);
S10:    i := i + 1;
      end_while;
end

```

Figura 8: Esempio 3 per il dynamic slicing

sione di un Dynamic Dependence Graph per un programma relativamente piccolo come quello di figura 8. Ottenuto il grafo si procede con l'algoritmo di slicing dinamico selezionando i nodi raggiungibili a partire dal nodo che contiene l'ultima definizione della variabile a cui siamo interessati.

Quarto Approccio

Poiché il Dynamic Dependence Graph è generalmente illimitato nel numero di nodi, in quanto essi sono pari agli elementi contenuti nella execution history e non è possibile stabilirne il numero a priori, prima della esecuzione reale, viene introdotto un metodo per ridurre le dimensioni del grafo, fondendone alcuni nodi.

Instead of creating a new node for every occurrence of a statement in the execution history,

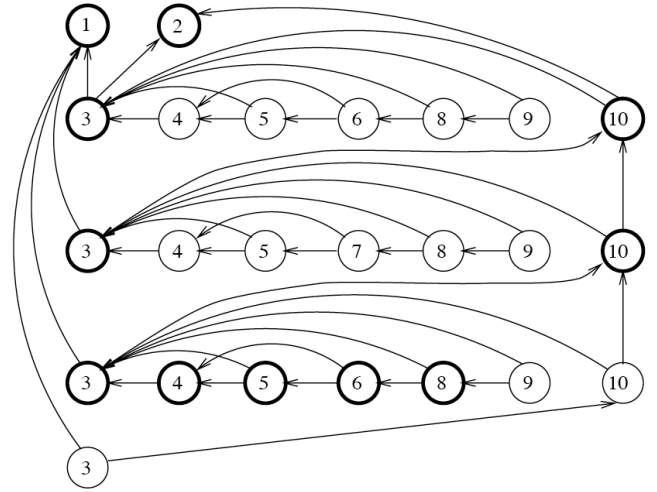


Figura 9: *Dynamic Dependence Graph* per il programma di figura 8 riferito al test-case (N=3, X=-4, 3, -2). I nodi in grassetto sono inclusi nella slice dinamica rispetto alla variabile Z alla fine della execution history

create a new node only if another node with the same transitive dependencies does not already exist. [3]

Il grafo così ottenuto viene chiamato *Reduced Dynamic Dependence Graph* (vedi figura 10) e per costruirlo è necessario tenere traccia di alcune informazioni aggiuntive, che verranno memorizzate in opportune tabelle:

- Per ogni variabile, la tabella *DefnNode* tiene traccia dell'ultimo nodo che l'ha definita.
- Per ogni predicato di controllo, la tabella *PredNode* tiene traccia dell'ultimo nodo che corrisponde a questo predicato nella execution history analizzata fin'ora.
- Per ogni nodo nel Reduced Dynamic Dependence Graph viene mantenuto un set chiamato *ReachableStmts* che contiene tutte le istruzioni di cui almeno una occorrenza è raggiungibile dal nodo in questione.

Ogni volta che un'istruzione S_i viene eseguita, viene determinato il set di nodi, D , che per ultimi hanno assegnato un valore alle variabili usate in S_i ed anche l'ultima occorrenza C del predicato di controllo da cui l'istruzione dipende. Se un nodo n , associato con S_i , esiste già e i suoi immediati discendenti sono gli stessi di $D \cup C$, allora il nodo S_i ed il nodo n vengono fusi. Altrimenti viene creato il nuovo nodo corrispondente a S_i . Ad ogni passo è inoltre necessario aggiornare le tabelle *DefnNode* e *PredNode* a seconda che l'istruzione esaminata sia una istruzione semplice o un predicato di controllo.

In presenza di dipendenze circolari la riduzione fin qui schematizzata non avrà luogo; per ogni iterazione di un ciclo che include una dipendenza circolare bisognerà creare un

nuovo nodo. Agrawal e Horgan hanno proposto una soluzione anche a questo problema: ogni volta che bisogna creare un nuovo nodo, per esempio per l'istruzione S_i , si controlla che nessuno dei suoi immediati discendenti, ad esempio il nodo v , abbia una dipendenza rispetto ad una occorrenza precedente di S_i e si controlla anche che ogni nodo immediatamente discendente dalla nuova occorrenza di S_i non sia raggiungibile da v . Se questa condizione non è verificata, allora è possibile fondere i nodi S_i e v , altrimenti bisognerà creare un nuovo nodo per S_i .

Applicando questo algoritmo alle tabelle descritte in precedenza, è sufficiente controllare che il set *ReachableStmts* da associare alla nuova occorrenza S_i sia un sottoinsieme dell'omonimo set associato a v . Se questa condizione è verificata, allora possiamo fondere la nuova occorrenza di S_i con v .

Una volta ottenuto il Reduced Dynamic Dependence Graph, per la data execution history, al fine di ottenere la slice dinamica associata ad una variabile *var* sarà sufficiente controllare a quale nodo sia associata *var* nella tabella *Defn-Node*: il set di istruzioni appartenenti all'insieme *ReachableStmts* associato a quel nodo sarà la slice dinamica cercata. Non sarà quindi necessario attraversare di nuovo il grafo per calcolarla.

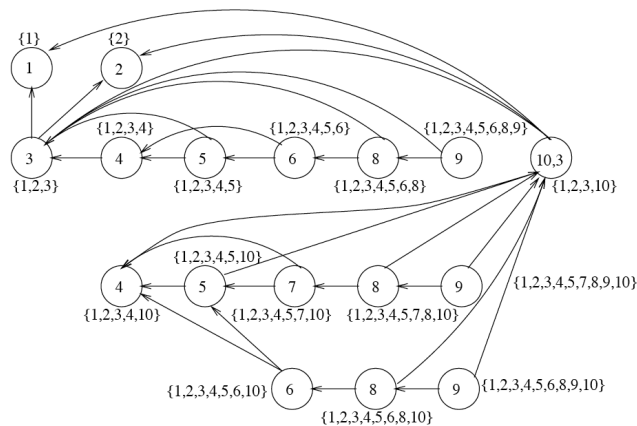


Figura 10: *Reduced Dynamic Dependence Graph* per il programma di figura 8 riferito al test-case ($N=3$, $X=-4$, 3 , -2). Ogni nodo è annotato con il set *ReachableStmts*

5. ALTRI APPROCCI ALLO SLICING

Parlando di static e dynamic slicing abbiamo già introdotto il concetto di analisi *backward*. Lo slicing può però essere implementato anche in modalità *forward*. Questo nuovo modo di procedere è utile quando dobbiamo capire quali parti, del codice sorgente, vengono influenzate da una determinata modifica; è quindi una tecnica utile nel caso di integrazione di codice piuttosto che per il debugging. Se l'analisi backward serve per rispondere alla domanda: "Quali istruzioni influenzano questo punto?", l'analisi di tipo forward risponde al quesito: "Quali istruzioni risentiranno di una modifica effettuata in un determinato punto del codice?". A livello implementativo, utilizzando il dependence graph, si tratta solamente di effettuare la copertura nella direzione opposta cui operano gli algoritmi precedentemente visti. Spesso a procedimenti di tipo backward vengono affiancate

iterazioni forward, dando origine a tecniche di slicing miste denominate *complete slicing*.

In letteratura sono poi stati definiti approcci misti di tipo statico e dinamico, chiamati *quasi-static*, in cui solamente parte degli input, per la creazione dell'execution history, è noto. Sempre considerando gli approcci misti, è da citare il *dicing* definito come una tecnica che effettua operazioni insiemistiche su diverse slice. L'applicazione classica è quella in cui si effettua la differenza tra due slice, calcolate su due slicing criterion differenti, al fine di evidenziare delle particolarità del software. Consideriamo, per esempio, un software difettoso che effettua il conteggio di caratteri e parole all'interno di un testo in cui il valore dei caratteri risulta corretto mentre quello delle parole errato. Su questo semplice programma possiamo ricavare due diverse backward slice a partire dall'istruzione di stampa del numero di caratteri e di parole ed applicare *dicing* per capire tutte le istruzioni che determinano il conteggio delle parole ma non quello dei caratteri; tra di esse ci saranno le istruzioni che determinano il malfunzionamento dell'applicazione.

Per completezza riportiamo anche un approccio differente, chiamato *chopping*, che risulta utile quando si desidera capire l'influenza di una modifica nell'applicazione generale. Queste operazioni sono comuni durante lo sviluppo di software in ambiente collaborativo poichè solitamente si effettua program integration ad intervalli molto brevi. Definendo un criterio del tipo $\text{chop}(t,s)$, la slice conterrà tutti i punti del programma che trasmettono gli effetti di una modifica in un punto t rispetto ad un punto del codice s . Questo procedimento è realizzato tramite l'intersezione di una forward slice presa rispetto a t e una backward slice rispetto a s .

6. TOOL

Dopo aver analizzato le motivazioni e le tecniche utilizzate per realizzare la program slice, vogliamo ora cercare di tracciare il punto della situazione rispetto ai tool realmente disponibili agli sviluppatori. E' importante sottolineare come, sebbene la tecnica sia decisamente interessante e potrebbe realmente rendere più efficiente la fase di testing e di debugging, in realtà i tool disponibili non sono ancora abbastanza maturi da poter essere usati in un contesto reale di sviluppo di software. Il funzionamento stesso dei tool e le limitazioni sulla tecniche di slicing implementate li rendono quindi dei buoni strumenti per sperimentare slicing su quelle che vengono definite "toy application" ma poco applicabili su progetti reali anche a causa dei limiti introdotti sui costrutti dei linguaggi usabili. Tutti i tool realizzano la slice attraverso analisi statica; particolari costrutti presenti nel codice sorgente analizzato (array, puntatori, etc), che di fatto determinano una serie di approssimazioni, spesso non vengono considerati oppure conducono a slice sovradimensionate.

Passiamo quindi ora in rassegna i principali strumenti:

Wisconsin e CodeSurfer [21] [9] : Da un lavoro di ricerca dell'università del Wisconsin nasce questo tool che viene ora commercializzato in un prodotto decisamente più corposo chiamato CodeSurfer. Permette static slicing (backward e forward) e chopping su codice sorgente in ANSI C e C++ (alpha support). Il tool,

per eseguire lo slicing, costruisce il dependence graph e permette inoltre di esplorarlo. E' disponibile per i sistemi Windows, Linux e Solaris.

Unravel [11] : Sviluppato dal *National Institute of Standards and Technology (NIST)* è un prototipo di program slicer per ANSI C. Permette di eseguire static backward slicing attraverso un'interfaccia decisamente primitiva ma funzionale. Sviluppato su SunOS è ora disponibile per tutte le piattaforme unix-like.

Indus, Bandera e Kaveri [17] : Uno dei progetti maggiormente interessanti. *Indus* è il motore di slicing che effettua static slicing (backward e forward) su sorgenti Java. Negli intenti del gruppo di ricerca che sviluppa Indus questo tool dovrebbe diventare il primo di una serie di strumenti per analizzare codice Java. A livello implementativo il motore di slicing utilizza il linguaggio intermedio *Jimple*: tutti i comandi Java sono convertiti in molteplici istruzioni *Jimple* e la slice viene ricavata su questo nuovo set, per poi essere mappata all'indietro al termine della computazione. Indus permette di selezionare un parametro di visibilità su cui considerare la slice, definito in termini di classi, metodi o blocchi di istruzioni; inoltre permette due modalità di esecuzione differenti che privilegiano il mantenimento del valore della variabile nello slicing criterion (post-execution) secondo la definizione di Weiser [22] piuttosto che la raggiungibilità dell'istruzione definita nel criterio stesso (pre-execution). *Bandera*, tool per model checking, integra Indus in maniera da fornire allo sviluppatore uno strumento sempre più versatile. *Kaveri* è invece un front-end per l'ambiente di sviluppo OpenSource *Eclipse* [19]. Data l'enorme importanza che questo IDE sta assumendo negli ultimi anni, la presenza di un plugin per effettuare slicing, integrato nell'ambiente di sviluppo, sembra essere il punto di contatto tra questa tecnica e il suo reale utilizzo in produzione.

7. CONCLUSIONI

Nel corso dell'articolo abbiamo presentato una panoramica sulle tecniche statiche e dinamiche dello slicing che possono essere trovate in letteratura. Partendo dal nostro studio, vogliamo cercare di riportare qualche considerazione di carattere generale su questa particolare quanto interessante tecnica.

Fare slicing in maniera automatica è sicuramente un buon approccio per cercare di ridurre la crescente complessità dei sistemi. La letteratura, in questi vent'anni dalla prima pubblicazione, è fiorente di attività di ricerca e di soluzioni teoriche per cercare di calcolare la slice in maniera efficiente e con buoni risultati. A fronte di queste basi teoriche non risultano però altrettanto interessanti gli strumenti a disposizione dello sviluppatore che abbiamo potuto provare; essi risultano spesso incompleti ed in versione prototipale, non disponibili per tutti i linguaggi di programmazione e per tutti i costrutti di tali linguaggi. Alcuni problemi e limitazioni, come quelli intrinseci nel metodo statico, sono superati teoricamente con alcune approssimazioni durante l'esecuzione dell'algoritmo di slicing ma, a livello pratico, conducono alla costruzione di slice con un alto numero di righe.

La scelta di quale tecnica di slicing adottare, tra le innumerevoli presenti (statico, dinamico, backward, forward, chopping, etc.), è fondamentale per il risultato che si vuole ottenere. Come presentato, slicing può essere usato in diversi momenti dello sviluppo di un software e non solo durante il debugging; risulta quindi di fondamentale importanza scegliere la tecnica che più si adatta al contesto applicativo. Presentando i diversi algoritmi implementativi abbiamo poi mostrato come quelli basati su grafi (*PDG* in particolare) risultino più efficienti e più semplicemente implementabili in strumenti dedicati.

A nostro avviso, eventuali sviluppi futuri della ricerca dovrebbero indagare sulle tecniche di approssimazione al fine di contenere le dimensioni della slice ottenuta oltre a focalizzarsi sulla realizzazione di strumenti efficaci e ben integrati negli ambienti di sviluppo del software per fare in modo che questa tecnica possa essere adoperata universalmente durante il processo produttivo.

8. REFERENCES

- [1] H. Agrawal. On slicing programs with jump statements. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 302–312, New York, NY, USA, 1994. ACM Press.
- [2] H. Agrawal and J. R. Horgan. Dynamic program slicing. technical report. Technical report, Software Engineering Research Center - Purdue University, West Lafayette, Indiana, November 1989.
- [3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25/6, pages 246–256, White Plains, NY, June 1990.
- [4] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [5] J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*, pages 509–518, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [6] J.-F. Bergeretti and B. A. Carrè. Information-flow and data-flow analysis of while-programs. *ACM Trans. Program. Lang. Syst.*, 7(1):37–61, 1985.
- [7] D. Binkley and K. Gallagher. Program slicing. *Advances in Computers*, 43, 1996.
- [8] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Softw. Eng.*, 17(8):751–761, 1991.
- [9] I. GrammaTech. Codesurfer. <http://www.grammatech.com/products/codesurfer/overview.html>.

- [10] E. H. S. Hiralal Agrawal, Richard A. DeMillo. Dynamic slicing in the presence of unconstrained pointers. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 60–73, New York, NY, USA, 1991. ACM Press.
- [11] D. W. B. Jim Lyle, Dolores R. Wallace. Unravel program slicing tool. <http://hissa.nist.gov/unravel/>. National Institute of Standards and Technology Information Technology Laboratory.
- [12] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
- [13] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization. *Empirical Softw. Engg.*, 7(1):49–76, 2002.
- [14] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 495–505, Washington, DC, USA, 1996. IEEE Computer Society.
- [15] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.
- [16] P. Hausler. Denotational program slicing. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, volume 2, pages 486–494. Software Track, 1989.
- [17] V. P. Ranganath et al. Kaveri user guide (v 0.6). <http://projects.cis.ksu.edu/projects/indus/>.
- [18] B. Shneiderman. Exploratory experiments in programmer behavior. In *International J. of Computer and Information Sciences*, 1976.
- [19] E. Team. Eclipse ide. <http://www.eclipse.org/>.
- [20] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
- [21] W. University. Wisconsin program slicer. <http://www.cs.wisc.edu/wpis/html/>.
- [22] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [23] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.