Construction of the System Dependence Graph for Web Application Slicing

Filippo Ricca and Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ricca, tonella}@itc.it

Abstract

The computation of program slices on Web applications may be useful during debugging, when the amount of code to be inspected can be reduced, and during understanding, since the search for a given functionality can be better focused. The system dependence graph is an appropriate data structure for slice computation, in that it explicitly represents all dependences that have to be taken into account in slice determination.

In this paper, the main problems related to the construction of the system dependence graph are considered. With no loss of generality, solutions are presented with reference to the server side programming language PHP and to the client side language Javascript. Most of the difficulties concern event and hyperlink handling, dynamic generation of HTML code, and direct access to HTML elements by client code. An example of Web application is analyzed, supporting the feasibility of the approach.

1 Introduction

Program slicing is a static analysis technique for the extraction of the program statements relevant to a specified computation. Program slicing was introduced for the first time by Weiser [7] and has now many applications such as debugging, code understanding, program testing, reverse engineering, software maintenance, reuse, software safety and metrics. Two excellent surveys on program slicing are [1, 6]. A Web application consists of a set of static HTML pages displayed to the user and (possibly) of server side programs, which perform some computation, finally resulting in the production of dynamic pages transmitted to the browser for display. Moreover, HTML pages may embed client procedures which are either executed during page loading or in response to graphical interface events.

In [5] we have extended the notion of slicing to Web applications, and described the dependences that have to be taken into account. In particular, nesting control dependences for the HTML code and data flows from HTML code to server programs, as well as call and parameter-in dependences, have been considered. Examples are given with reference to a pseudo-language for the server side programs. In this paper, we focus on a real world language, widely used in the development of Web applications: PHP. Moreover, we consider also the case in which an HTML page includes client side code (ignored for simplicity in [5]), written in Javascript. Correspondingly, several problems, not dealt with in [5], have to be handled, such as the representation of the graphical events triggering execution of client code, or transfer of control to the server, the reference to HTML elements by client code through access paths, and the usage of session variables. Dynamic generation of HTML code (not considered in [5]) will be also discussed, leading to a novel notion: *dynamic slicing* applied to Web applications. In fact, in presence of "extreme" HTML code generation, a purely static analysis is very limited, while dynamic slicing remains still applicable.

Web application slicing is conducted on an extension of the program representation used with traditional software, the System Dependence Graph (SDG) [2]. Its basic elements are similar to those introduced in [4] for data flow testing. We describe an approach for solving the difficulties in the construction of the SDG for Web applications, based on the definition of Web specific dependences and nodes. Call dependences are categorized either as returning or non-returning, and graphical interface events and event loop are explicitly represented in the SDG. Results of slicing an executable example written in PHP/HTML/Javascript are also presented.

The remainder of this paper is organized as follows: the next section gives some background information on Web applications. Section 3 introduces our approach to Web application slicing. It is ended by a subsection on dynamic slicing. The usage of slicing for the analysis of a real Web application is presented in Section 4. Section 5 concludes the paper.

2 Web applications

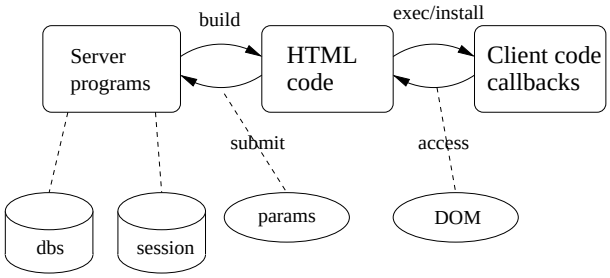

Figure 1. *Typical organization of a Web application.*

Slicing Web applications is more complex than slicing traditional programs, because data flows are not limited to the transmission of values of variables from statement to statement. In fact, HTML pages can be generated dynamically by Web applications, and data values may be embedded in the generated code. In turn, dynamically generated pages can reference other dynamic pages, to which data values are transmitted from the former.

A Web application is a special case of client-server system, in which the Web server plays the role of the server, the Web browser plays the role of the client, and a fixed communication protocol (the HTTP protocol) is established between the client and the server. Fig. 1 depicts the typical organization, in which the only non optional component is the HTML code. Static Web sites consist only of a set of fixed Web pages written in HTML and stored in the file system. They are transmitted to the Web browser upon request. Dynamic Web applications include also a set of server programs, which *build* (part of) the HTML code to be displayed by the browser. In turn, the HTML code can activate the execution of a server program by means of a `SUBMIT` input within an HTML element of type `FORM` or an `HREF`. Data flows from a server program to the HTML code are achieved by embedding values of variables inside the HTML code, as the values of the attributes of some HTML elements. In the opposite direction, the basic mechanism for data value propagation is by means of form parameters. *Hidden* parameters are constant values that are just transmitted to the server (possibly recording the values of some previous computation), while non hidden input parameters are gathered from the user. Parameter passing is strictly by value and the invocation of the server program is a control transfer without return. Server programs can exploit persistent storage devices (such as data bases) to record values and to retrieve data necessary for the construction of the HTML page. Moreover, session specific data (session variables) can be stored at the server side and maintained across successive executions. Cookies act similarly to session variables, with the only difference of being stored at the client side instead of the server. Such data are used to identify the ongoing interaction and to record data that need to survive past the end of execution of server programs.

Finally, more dynamism can be included in a Web application by means of client side code, which can be either executed during the loading of the HTML page inside the browser, or in response to a graphical interface event (e.g., mouse click). In the latter case, callbacks are installed by declaring that they are reacting to a graphical event on a given HTML element. Some HTML attributes (e.g., `ONCLICK, ONBLUR`) are available for this purpose. Then, the occurrence of the graphical event triggers their execution, similarly to the graphical user interfaces of traditional software. An event loop dispatching the events to the respective callbacks is in fact implicitly activated. The client code (possibly executed in response to an interface event) can read and write the values of the attributes of the HTML elements, by exploiting the Document Object Model (DOM), giving a standard interface to them. For example, the access path `document.Form0.x.value` can be used to access the value of variable `x` inside form `Form0`. This represents the basic data flow mechanism between HTML code and client code.

While usually callbacks return the control to the event loop after completing their execution, when the related interface event is a click on a `SUBMIT/HREF` HTML element associated with a server program, control is transferred to the server program and never returns back to the event loop. In other words, `SUBMIT/HREF` calls are no-return transfers of control. The typical usage of client side callbacks is form validation and computation of field values (e.g., conversions, totals, etc.). Instead of transferring any values entered by the user to the server programs, only valid values are transferred, thus delegating the validity check to a piece of code running on the client.

Several server side languages are available for the construction of a Web application (e.g., PHP, Java, Perl, VBscript, etc.). The same is true for the client side code (e.g., Java, Javascript, etc.). In this paper, we will consider PHP for the server and Javascript for the client. Such a choice is by no means restrictive, since these languages offer all the basic features available in the others.

3 Web application slicing

According to its original definition [7], a *program slice* is a reduced, executable program obtained from a given program by removing statements, so that it replicates part of the behavior of the initial program. The main requirements in this definition are that a slice be still a legal program and that its behavior be preserved with respect to a computation of interest.

Similar constraints are enforced in introducing the notion of slice for Web applications. Slicing a Web application results in a Web application which exhibits the same behavior as the initial Web application in terms of information of interest displayed to the user, programs generating it (in case it is dynamic) and interaction with the user (client side code).

Definition: *a* Web application slice *is obtained from a given set of Web pages and server programs by removing HTML and server/client code statements, so that part of the behavior of the initial Web application is replicated.*

The criterion for slice computation is an information item of interest displayed in a given Web page, and the resulting slice reproduces the same information item, if the user performs the same navigation actions and provides the same input in the original and in the sliced Web application.

Program slices can be computed by incrementally adding consecutive sets of transitively relevant statements. Statements that directly affect the computation at another statement are connected to the latter by a dependence relation, which can be explicitly represented in a graph called System Dependence Graph (SDG) [2]. Different kinds of dependences are distinguished in the SDG (control, intraprocedural flow, call, parameter-in, etc.) and the problem of static slicing can be restated in terms of a two-step reachability problem in the SDG. In the following, we discuss the kinds of dependences that hold for Web applications and we define an SDG valid for Web applications, that can be used to slice them. Definitions already given in [5] are reported here for completeness.

3.1 Nesting/control dependences

Control dependences for traditional programs connect the predicate tested at a conditional or loop statement to the instructions the execution of which directly depends on the truth value of the predicate. This continues to hold for the server/client side code, but Web applications are characterized by an additional kind of control dependence, which is found in the HTML code. In fact, an HTML statement can be interpreted correctly by the browser only if all enclosing HTML tags are available. Therefore a nesting dependence holds between a tag and all directly enclosed statements.

Definition: *a* nesting dependence *holds between two HTML statements if the latter is nested inside the former.*

Figure 2 contains a fragment of PHP and HTML code, in which the first 6 lines are executed by the server, being included within the tags `<?` `?>`, while the last 4 lines are HTML statements directly passed to the browser. The server execution consists of testing the value of variable `$x`, and if it contains the string `"xx"`, the two variables `$y` and `$z` are assigned a new value. The HTML code inserts a form in the page being constructed. Such a form includes a text field, named `w` and initialized with `"ww"`, for user input, and a submit button which executes the page `aa.php`.

```
cd.php:
1  <?
2  if ($x == "xx") {
3      $y = "yy";
4      $z = "zz";
5  }
6  ?>
7  <FORM method="POST" action="aa.php">
8      <INPUT TYPE="Text" NAME="w" VALUE="ww"><BR>
9      <INPUT TYPE="Submit"><BR>
10 </FORM>
```

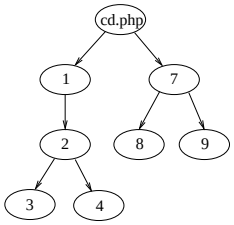

Figure 2. *Fragment of PHP/HTML code and related nesting/control dependences.*

Control dependences for this code fragment are shown at the bottom. Node `cd.php` is connected to the top level instructions. When the execution of a statement is conditional to another one (e.g., the assignment at line 3 and the `if` at line 2), there is a control dependence between the two. Statements which close a scope (e.g., the closed bracket at line 5) have no associated nodes, in that the control dependence relation makes them unnecessary.

3.2 Data dependences

Data dependences for traditional programs hold when a statement defines the value of a variable, which is used at another statement after being propagated to it along a definition clear path (i.e., a path containing no redefinition of the given variable). In addition to such a situation, which still can be found in server/client code, data flows may occur in a Web application from server/client side statements to the HTML code. On the other side, a variable definition at an HTML statement can reach a server/client instruction only thorugh a procedure invocation, as a submission parameter, or through a DOM element access.

Definition: *a* data dependence *holds between two statements if the former defines the value of a variable which is used by the latter, and a definition clear path exists between the two.*

Figure 3 contains an example of two kinds of data dependences: a data dependence internal to PHP code, and

dd.php:

```
1   <?
2   $x = $z;
3   if ($x == "a") {
4       $y = "1";
5   } else {
6       $y = "2";
7   }
8   ?>
9   <FORM method="POST" action="aa.php">
10      <INPUT TYPE="Text" NAME="w" VALUE="<?print $y?>"> <BR>
11      <INPUT TYPE="Submit"><BR>
12  </FORM>
```

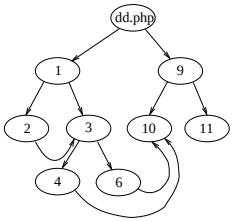

Figure 3. *Fragment of PHP/HTML code and associated control dependences (straight lines) and data dependences (curve lines).*

aa.html:

```
    ...
1   <FORM method="POST" NAME="Eform" action="bb.php">
2       <INPUT TYPE="Text" NAME="x" VALUE="Thank "> <BR>
3       <INPUT TYPE="Submit"><BR>
4   </FORM>
    ...
7   <SCRIPT LANGUAGE="JavaScript">
8       document.Eform.x.value = document.Eform.x.value + "you";
9   </SCRIPT>
```

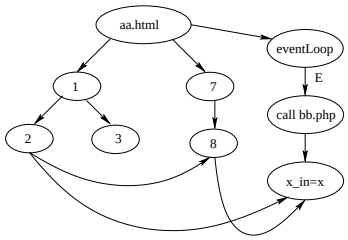

Figure 4. *Data dependence mediated by the DOM.*

two other ones from two PHP statements to an HTML statement. The server side statement 2 defines the value of `$x`, which is subsequently used (without intermediate redefinition) at statement 3. The two server side assignments to variable `$y` at lines 4 and 6 have a data dependence with the HTML statement at line 10, where the value of the input variable `w` is obtained from the value of `$y`. The effect is that a text field is showed to the user inside the dynamic page `dd.php` with an initial value equal to `$y`. The collected input is then stored inside variable `w`. It should be noted that this is the main communication mechanism between server side statements and HTML variables, allowing the propagation of values from code executed on the server to Web pages. A variant of this mechanism is often exploited by Web applications to transmit a value from a server program to another server program via the generated Web page. It is obtained by means of a *hidden* HTML variable, the value of which is filled in by a first server program, and is transmitted to another server program, invoked via form submission. In the example of Figure 3, this can be achieved by replacing the input type at line 10 (`Text`) with `Hidden`. The form will not prompt the user for any input, and the value inserted by `dd.php` will be transmitted to `aa.php`.

Figure 4 shows the case of a data dependence between an HTML statement and a client code statement, achieved by means of the DOM. The Javascript code fragment from line 7 to line 9 is executed by the browser when these lines are loaded. The input variable `x` of the form `Eform`, loaded by the browser previously, is both used and defined at line 8. It is referenced as a DOM element, reached from `document`, the DOM element associated with the whole page. Then, the value of `x` is passed to `bb.php` as an input parameter, when the user clicks on the related submit button (the event loop, handling interface events is explained below).

3.3 Call dependences

When a server side program is invoked from an HTML statement (e.g., via form submission or `HREF`), control of execution is transferred to the server and never returns back to the Web page issuing the call. In other words, a server program invoked from an HTML statement cannot produce any effect on the variables of the calling page, since the invocation is a no-return invocation. A consequence is that the *calling context problem*, i.e., the problem of keeping the data flows associated to the different call sites separated, is absent, since call sites are not affected by the invocation.

Calls issued within server (or client) code, such as from a PHP procedure to another PHP procedure (or from a Javascript function to another Javascript function), similarly to traditional software, are subject to the calling context problem (see [2] for an algorithm handling such cases).

Invocations within the event loop can be issued before page loading is complete, if the user clicks on some interface widget while page loading is in progress. This means that it is not possible to assume that all HTML and client statements interpreted during page loading have been encountered when a callback is activated. The consequence in the construction of the SDG is that data dependences having a callback parameter as target need be computed in a non-killing (flow insensitive) way. That is, when a statement redefines the value of a variable, it is not possible to consider the previous definitions as invalid for the callback parameters, since execution could be interrupted by the user before the invalidating statement is loaded, and the actual parameter value is the previous one. The visible result in the SDG consists of some extra data dependence edges, which end

at parameter-in nodes, and are due to the non-killing nature of the related statements. An example of such an edge connects node 2 to the parameter-in node `x_in` in Figure 4.

Definitions: (1) *a* call dependence *holds between each statement of type* call *and the server/client program or procedure invoked. (2) A* parameter-in *dependence holds between any actual parameter of a call and the respective formal parameter of the invoked program or procedure. (3) A* parameter-out *dependence holds between any value returned from an invocation and the related variable in the call site.*

```
aa.php:
1   <?
2   z = "aa";
3   ?>
4   <FORM method="POST" action="bb.php">
5       <INPUT TYPE="Submit"><BR>
6       <INPUT TYPE="Text" NAME="x" VALUE="<?print $z?>"><BR>
7       <INPUT TYPE="Text" NAME="y" VALUE="<?print $z?>"><BR>
8   </FORM>

bb.php:
9   <?
10  $xx = $x;
11  $yy = $y;
12  ?>
```

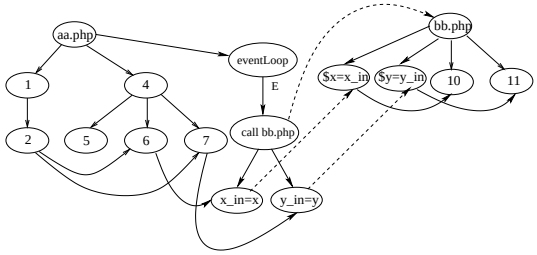

Figure 5. *Fragment of PHP/HTML code and associated control/data dependences. Call/parameter-in dependences are depicted with dashed lines.*

In Figure 5, the HTML form at line 4 installs `bb.php` as the response to a click on the `Submit` button of this form. Consequently, the event loop of this page contains only one call, triggered by the click event (indicated generically with `E`). Since this is a no-return invocation, it is indicated with a curved dashed line. Straight lines are used for normal procedure invocations. Two input variables are submitted by the form and accessed by the server program invoked: `x` and `y`. They are associated with two parameters (`x_in` and `y_in`), and with two parameter-in dependences (straight dashed lines in the figure).

3.3.1 Session variables

Session variables hold values that are maintained across different executions of PHP server programs, provided that such executions are triggered during a same Web navigation session. They can be handled similarly to global variables

```
aa.php:
1   <?
2   session_start();
3   session_register("GlobalX");
4   $GlobalX = "xx";
5   ?>
6 <A HREF="bb.php?<?= SID ?>"> bb </A>

bb.php:
9   <?
10  session_register("GlobalX");
11  print $GlobalX;
12  ?>
```

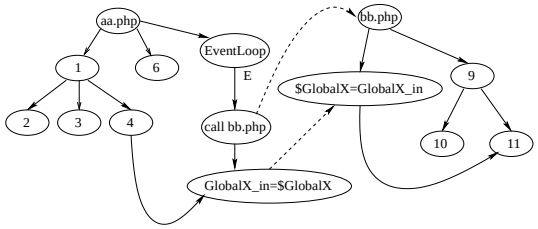

Figure 6. *Invocation with session variables.*

of traditional programs. Their representation in the SDG is obtained by means of additional input and output parameters, making the transmission of values possible from procedure to procedure. Output parameters are necessary only when the invocation returns to the call site, where the effect of the call has to be propagated. On the contrary, for no-return invocations (curved dashed lines) only input parameters (and parameter-in dependences) have to be created.

In Figure 6, a session is started at line 2, and variable `$GlobalX` is declared to be a session variable. Consequently, the access to this variable from another server program (`bb.php`, line 11), successively invoked, results in the value set at line 4. In the associated SDG, a data dependence connects node 4 to the input parameter `GlobalX_in`, and a parameter-in dependence connects the latter to the respective parameter inside `bb.php`. No parameter-out dependence is necessary, since the call is non-returning.

3.3.2 Document Object Model (DOM)

Figure 7 is related to the possibility of having data dependences that cross the boundaries of the procedures, without being associated to parameters or global variables. Interface events can trigger either the execution of the Javascript function `set_zero` on the client (loss of focus from the text field at line 7), or the transfer of control to the server program `bb.php` (click on the `Submit` button, line 6). Correspondingly, the event loop contains two call nodes. The input variable `w`, which is assigned the value 3 at line 7, is redefined inside the Javascript function `set_zero` (line 3). Although here this variable is out of scope (*invisible*), it can still be accessed by means of the DOM (`document.Eform.w.value`). Invisible variables that are ac-

aa.html:

```
...
<SCRIPT LANGUAGE="JavaScript">
    function set_zero() {
        document.Eform.w.value = "0";
    }
</SCRIPT>
...
<FORM method="POST" NAME="Eform" action="bb.php">
    <INPUT TYPE="Text" ONBLUR="set_zero();" NAME="w" VALUE="3"> <BR>
    <INPUT TYPE="Submit"><BR>
</FORM>
```

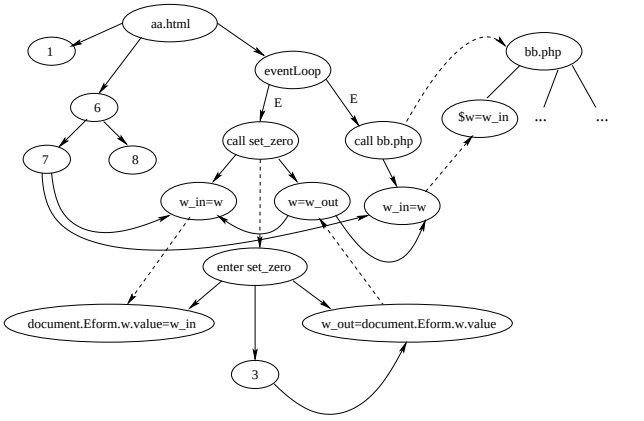

Figure 7. *Access to an invisible variable via the DOM.*

aa.html:

```
...
<SCRIPT LANGUAGE="JavaScript">
    function set_zero() {
        var x;
        x = document.Eform.w.value;
...
        document.Eform.w.value = "0";
    }
</SCRIPT>
...
<FORM method="POST" NAME="Eform" action="bb.php">
    <INPUT TYPE="Text" NAME="w" VALUE="3"> <BR>
    <INPUT TYPE="Submit" ONCLICK="set_zero();"><BR>
</FORM>
```

bb.php:

```
<?
$x = $w;
print $x;
?>
```

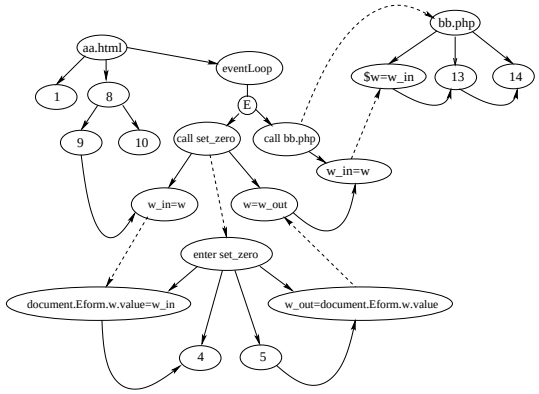

Figure 8. *Double call in the event loop.*

cessed via the DOM can be handled by introducing an additional input and an additional output parameter to transmit their values into and out of the called procedure. This is the reason for the two additional parameters, `w_in` and `w_out`, of function `set_zero` indicated in the SDG. In this case, resolution of the invisible variable reference based on the DOM can be achieved by means of a simple static analysis of the source. The two names, `w` at line 7, and `document.Eform.w` at line 3, refer to a same variable, because line 7 is inside a top level form named `Eform`. More generally, HTML variables referenced by client code can be located by traversing the document elements listed in the DOM access path. This may become difficult in presence of "extreme" HTML code generation (see below).

3.3.3 Double callback

Interface events may trigger the execution of more than one callback in sequence. To represent such cases in the SDG, an additional node is introduced for the graphical event, instead of the edge labeled *E*. Its children are the related calls. The example in Figure 8 associates the Javascript function `set_zero` and the PHP program `bb.php` to the click of the mouse on the `Submit` button (respectively, at lines 10 and 8). The effect of such double association is that the Javascript function will be executed first, and the PHP program will receive the control of the execution then. The node labeled *E* in the event loop is connected to such two calls, properly ordered. Note that there is no data dependence between node 9 and the input parameter `w_in` of `bb.php`, because the call to `bb.php` is always preceded by a call to `set_zero` which redefines the value of `w`, and no alternative path exists from 9 to the call of `bb.php`.

As with slicing of traditional software, the need of including variable declarations for the variables included in a slice is not represented explicitly with a specific dependence, and is assumed implicitly. This applies to the declaration of `x` at line 3 of Figure 8, in case `x` is used in the slice, but there are other situations reducible to this one. The declaration of the session variables and the start-up of the session (lines 2-3 of Figure 6) are another example.

3.3.4 HTML code generation

Figure 9 shows an example where the HTML code of the dynamic page produced in output has not a fixed structure. There are some fixed parts, the form opened at line 1 and closed at line 8, and the submit button created at line 7, but the input text fields of the form are created dynamically at lines 2-6, within a loop iterating a number of times, `$N+1`, which is also computed at run time (not shown). The names of the variables associated with these inputs are constructed dynamically from the loop index (`x0, ..., xN`). Their initial value is equal to the loop index (`0, ...,`

```
aa.php:
1   <FORM method="POST" action="bb.php">
    ...
2   <? for ($i=0; $i<=$N; $i++) {
3       $y[$i]="x" . $i;
4       $v[$i]=$i; ?>
5       <INPUT TYPE="Text" NAME="<? print $y[$i] ?>" VALUE="<? print $v[$i] ?>"> <BR>
6   <? } ?>
7   <INPUT TYPE="Submit"><BR>
8   </FORM>
    ...

bb.php:
12  <?
13  $sum = 0;
14  for ($i=0; $i<=$N; $i++) {
15      $z[$i]="x" . $i;
16      $v[$i]=$HTTP_POST_VARS[$z[$i]];
17      $sum=$sum+$v[$i];
18  } ?>
```

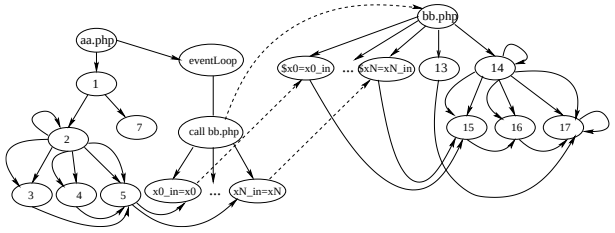

Figure 9. *Generation of HTML code from PHP.*

N). Consequently, the number of parameters passed to the server program `bb.php` is variable, function of `N`. On the server side, the program `bb.php` constructs dynamically the names of the parameters from `$N`, assigning them to the array `$z` (line 15). Then, parameter values are retrieved (line 16) and accumulated in variable `$sum` (line 17).

Construction of the SDG for this code fragment is remarkably difficult, due to the dynamic generation of a part of the code itself, namely the input elements of an HTML form and the parameters of a server program. Figure 9 shows what an "ideal" code analyzer is expected to generate, that is, a variable number of parameters (ellipsis) with an associated set of data and parameter-in dependences. In its most general form, the problem is that the code under analysis generates the code actually interpreted by the browser. Such a generation process includes constant strings (such as `" VALUE="`), concatenated with strings computed dynamically. Consequently, the code generation instructions (lines 1, 5, 7, 8 of Figure 9) build the strings that are the real object of the analysis. For a static analysis of the source, the variable parts of such strings have to be reduced to constants, whenever possible, or otherwise conservative assumptions have to be made about them. A safe approximation of the set of constants that possibly replace each string variable can be obtained, for example, by means of copy-constant propagation or symbolic execution techniques. In the example of Figure 9, the names of the input variables, `$y[$i]`, cannot be determined by copy-constant propagation. They are therefore labeled as *nonconstant* values, and the names of the associated variables are *unknown*. A conservative choice about such variables is to assume that their name could collapse with any other name in scope, and to represent them with a single parameter node in the invocations. Better approximations could be obtained with ad-hoc analyses.

3.4 Dynamic slicing

Dynamic slicing [3] is achieved by considering an execution trace in addition to the source code of an application. The transitive closure of the dependences between elements in the trace gives the dynamic slice, which can be eventually mapped to a subset of the original source statements. In order to produce the trace, an input vector with values for all input variables has to be selected. The result of slicing is a subprogram computing the same value of a given variable at a given statement for the input vector associated with the execution trace.

For a Web application, the execution trace consists of the sequence of statements executed on the server/client, plus the set of HTML pages encountered during the navigation. Some of them may be static, but some may also have been generated dynamically. Differently from static slicing, when dynamic slices are computed the HTML code generated by the server programs needs not be approximated, being completely known. Therefore, all of the problems outlined in the previous section with reference to the dynamic generation of the HTML code (Figure 9) disappear with dynamic slicing. The price to pay is a loss of generality. In fact, the slice reproduces the original behavior of the Web application only for the considered input vector.

To map the execution trace of a Web application to its source code, it is necessary to introduce an additional dependence, associated with HTML code generation. When an HTML statement is reached during dynamic slicing, the original source code statement that built it has to be included in the dynamic slice.

Definition: *a* build dependence *holds between a server program statement and an HTML statement if the former generates the latter.*

The tracer to use for dynamic slicing of Web application will have to be able to trace server/client statement execution, to store the HTML pages taversed during the navigation, and to trace the association between each server statement generating HTML code and the code generated.

Figure 10 shows the execution traces obtained by invoking `aa.php` (its code is given in Figure 9) with `$N=1`, inserting the values 12 and 13 in the two input text fields of the HTML page produced by `aa.php` and displayed by the browser, and finally submitting the form to `bb.php` (the code of which is also given in Figure 9). The execution trace of `aa.php` is enriched with the build dependence information, while the trace of `bb.php` is shown with information

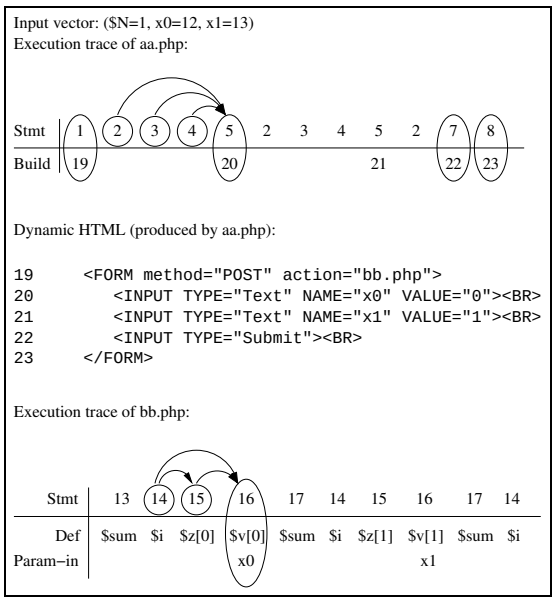

Figure 10. *Computation of a dynamic slice.*

on the variables defined at each statement and with the input parameters retrieved by each statement. The dynamic HTML code generated by `aa.php` is in Figure 10.

If a dynamic slice on variable `$v[0]` at the end of the execution of `bb.php` is determined, the statements encircled in Figure 10 are obtained. The first statement to include in the slice is 16, since it defines the value of `$v[0]`, and no redefinition occurs until the end of execution. Data and control dependences lead to including statements 14 and 15. Moreover, statement 16 accesses the parameter `x0`, which is passed to `bb.php` at statement 20. This statement is generated by `aa.php` at statement 5, which is thus also included in the dynamic slice. Data and control dependences of `aa.php` lead to the inclusion of statements 2, 3, 4. Finally, the analysis of the HTML code generated by `aa.php` results in the need for the slice to build also lines 19, 22 and 23, because of nesting control dependences in the HTML code (line 22 is necessary to make the form executable). Correspondingly, the statements 1, 7, 8 of `aa.php` are included in the slice, being responsible of generating the HTML lines 19, 22, 23.

4 An Online Travel Agency Web Application

Let us consider the Web application for a travel agency shown in Figure 11 (already presented in [5] in pseudo-VBscript). This simplified Web application, obtained by abstracting the main characteristics present in a set of real travel agency applications downloaded from the Web, was implemented by the authors using PHP and Javascript.

In this Web application, users can select hotels, cars

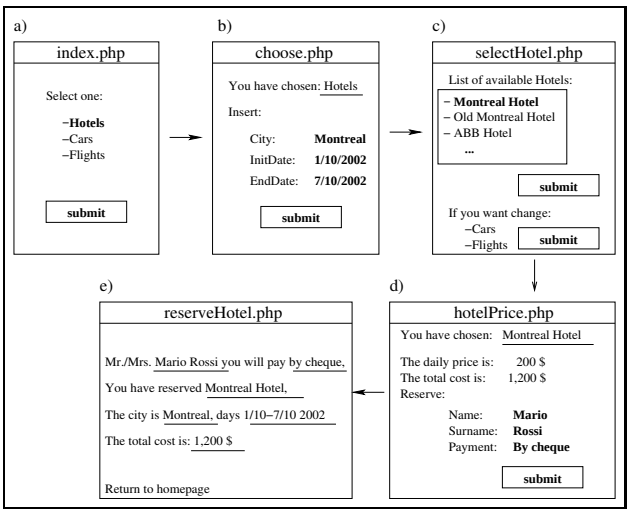

Figure 11. *An on-line Travel Agency application.*

and flights and reserve/rent/book them on-line. For the sake of simplicity, the pages displaying error messages are not reported. When users select an option of interest from the page `index.php` (Figure 11.a), a dynamic page `choose.php` is created, as shown in Figure 11.b. If, for example, the option *Hotels* was selected, users are asked to input the city where they want to stay and the arrival and departure dates. After clicking the submit button, the dynamic page `selectHotel.php` (Figure 11.c) appears, showing a list of available hotels in the chosen city. Users can select one of them or change the option of interest passing from hotels to cars or flights. If users decide to select a hotel and press the related submit button, the dynamic page `hotelPrice.php` is created, as shown in Figure 11.d. This page contains the chosen hotel, its daily price and the total cost for the entire stay. In the same page the user can insert name, surname and payment type to reserve a room in the selected hotel. The dynamic page `reserveHotel.php` (Figure 11.e) is then displayed. It contains a simple report confirming the reservation. Now users can return to the homepage and select cars or flights.

The values inserted by the users inside input forms are stored in session variables and are used by the Travel Agency application to propose default values in successive insertions. For example, dates (arrival and departure) and city inserted by the users to reserve a hotel are proposed as a default in the reservation of cars or flights.

Figure 12 contains a portion of the PHP source code of the server program `choose.php`. The content of the dynamic page produced by this server program depends on the selection made in the previous page, `index.php`. The value for such a selection is stored in variable `$x`. If the option *Hotels* was selected, the test at line 30 succeeds, while tests at lines 68 and 90 fail. Lines between 32 and 46

```
Choose.php:
20 <? include "session_variables.php";
21      $tempInitDate = mktime(0, 0, 0, 1, 1, 2002);
22      $tempEndDate = mktime(0, 0, 0, 1, 1, 2002); ?>
24 <HTML><HEAD><TITLE> Choose Hotels, Cars and Flights </TITLE>
26 <? include "validate_date.jsp"; ?> </HEAD><BODY>
28 you have choosen: <? print $x; ?> <BR>
30 <? if ($x == "Hotels") {
32      if ($GlobalpickupCity != "") {
33          $tempCity = $GlobalpickupCity;
34          $tempInitDate = $GlobalpickupDate;
36          $tempEndDate = $GlobaldropoffDate;
38      }
40      if ($GlobaldestCity != "") {
42          $tempCity = $GlobaldestCity;
44          $tempInitDate = $GlobaldestDate;
46      } ?>
50 Insert: <BR>
54 <FORM method="POST" action="selectHotel.php?<? = SID ?>">
56      City: <INPUT type="Text" name="city" size=20 value="<? print $tempCity ?>" >
58      InitDate: <INPUT type="Text" name="initDate" size=20
                value="<? print date(d/m/Y, $tempInitDate) ?>" onblur="check_date(this)" >
60      EndDate: <INPUT type="Text" name="endDate" size=20
                value="<? print date(d/m/Y, $tempEndDate) ?>" onblur="check_date(this)" >
62      <INPUT type="Submit"><BR>
64 </FORM>
66 <? } ?> // this PHP bracket closes the 'if' at line 30
68 <? if ($x == "Cars") {
...
88 <? } ?>
90 <? if ($x == "Flights") {
...
110 <? } ?>
120 </BODY></HTML>
```

Figure 12. *Page* `choose.php`.

recover (when available) the values inserted previously, to rent a car or book a flight. They become the default values of the form generated at lines 54-64. Before invoking (line 54) the server program `selectHotel.php`, the dates inserted by the user are checked by the Javascript function `check_date.php` (see Figure 13), called when there is a loss of focus from the text fields (lines 58 and 60).

Figure 14 contains the PHP source code of the server program `selectHotel.php`. In this program a connection with the MySQL database TravelAgency is established (lines 151-153). The array `$result` (line 155) is assigned the list of hotels available in the city chosen by the user in the previous page. This list is recovered by the SQL query at line 154, on the table *Hotel*, consisting of the columns *name, city* and *price*. The session variables `$Global-city`, `$GlobalinitDate` and `$GlobalendDate` are assigned their values at lines 162-173, after converting the dates to timestamps. The list of available hotels is printed inside a pull-down menu of the page being generated (lines 201-204). The two forms, respectively for submitting the chosen hotel to `hotelprice.php`, and for changing option and passing to cars or flights, are created at lines 199-207 and 209-218 (portion not shown).

Figure 15 contains the PHP instructions of the server programs `hotelPrice.php` and `reserveHotel.php`. The first program recovers the daily price of the hotel chosen by the user, by means of a query (line 224)

```
validate_date.jsp:
300 <SCRIPT LANGUAGE="JavaScript">
302     function check_date(field){
304         var checkstr = "0123456789"; var Datevalue = ""; var DateTemp = "";
306         var seperator = "/"; var day; var month; var year; var leap = 0; var err = 0; var i;
308         DateValue = field.value;
310         for (i = 0; i < DateValue.length; i++) { /* Delete all chars except 0..9 */
312         if (checkstr.indexOf(DateValue.substr(i,1)) >= 0) {
314             DateTemp = DateTemp + DateValue.substr(i,1); } }
316         DateValue = DateTemp;
318         year = DateValue.substr(4,4);
320         month = DateValue.substr(2,2);
322         if ((month < 1) || (month > 12)) { err = 21; }
324         day = DateValue.substr(0,2);
326         if (day < 1) { err = 22; }
328         if ((year % 4 == 0) || (year % 100 == 0) || (year % 400 == 0)) { leap = 1; }
330         if ((month == 2) && (leap == 1) && (day > 29)) { err = 23; }
332         if ((month == 2) && (leap != 1) && (day > 28)) { err = 24; }
334         if ((day > 31) && ((month == "01") || (month == "03") || ... )) { err = 25;}
336         if ((day > 30) && ((month == "04") || (month == "06") || ... )) { err = 26; }
340         if (err == 0) {
341             field.value = day + seperator + month + seperator + year;
342         } else {
343             alert("Date is incorrect!"); }
345     }
346 </SCRIPT>
```

Figure 13. *Javascript Function* `check_date`.

on the database *TravelAgency*, table *Hotel*, and stores it into variable `$priceForDay` (lines 226 and 242). This value is used at line 244 to compute the total cost of the stay. At line 245, the total cost of the hotel is stored into the session variable `$GlobalCostHotel`. The server program `reserveHotel.php` accesses the session variables stored previously to build a report confirming the reservation.

4.1 Slice on variable `$GlobalCostHotel`

The SDG of the Travel Agency application has been built, and a slice at the output node 296 on variable `$GlobalCostHotel` of `reserveHotel.php` has been determined. The resulting PHP/HTML/Javascript/SQL statements have grey background in the Figures 12, 13, 14, 15. At the end of each user interaction, this slice displays the same value of variable `$GlobalCostHotel` as the initial Web application. All statements in the original Web application involving the variables `name`, `surname`, and `payment` have been removed, being not relevant for the computation of the total hotel cost.

It is interesting to note that the SQL query at line 224 (Figure 15) contributes to the slice. In fact, the hotel daily price (line 242) depends on the result of the query (dependences: 224 $\rightarrow$ 225, 225 $\rightarrow$ 226, 226 $\rightarrow$ 242). In this simple example, the code for inserting values into the database accessed at line 224 is not included. Consequently, slicing can stop at line 224, as regards the database values, assuming that the database is already in place, with all necessary tables filled in. A more complex Web application may include also dynamic pages for the insertion of values into a database. Slice computation has to take database

SelectHotel.php:

```
150 <? include "session_variables.php"; ?>
151 <? $dbhost = "mysql.itc.it"; $dbname = "TravelAgency"; $dbuser = "TravelAgency"; $dbpass = "ttttaaaa";
152     $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die ("impossible connecting with MySQL");
153     mysql_select_db($dbname, $conn) or die ("Impossible selecting database $dbname");
154     $query = "SELECT name FROM Hotel WHERE (city = '$city');";
155     $result = mysql_query($query, $conn) or die("error: " . mysql_error()); ?>
160 <? $Globalcity = $city; ?>
162 <? $posFirstSlash = strpos($initDate, '/');
163     $day = substr($initDate, 0, $posFirstSlash);
164     $posSecondSlash = strpos($initDate, '/', $posFirstSlash+1);
165     $month = substr($initDate, $posFirstSlash+1, $posSecondSlash-($posFirstSlash+1));
166     $year = substr($initDate, $posSecondSlash+1, 4);
167     $GlobalinitDate = mktime (0, 0, 0, $month, $day, $year);
168     $posFirstSlash = strpos($endDate, '/');
169     $day = substr($endDate, 0, $posFirstSlash);
170     $posSecondSlash = strpos($endDate, '/', $posFirstSlash+1);
171     $month = substr($endDate, $posFirstSlash+1, $posSecondSlash-($posFirstSlash+1));
172     $year = substr($endDate, $posSecondSlash+1, 4);
173     $GlobalendDate = mktime (0, 0, 0, $month, $day, $year); ?>
190 <HTML><HEAD><TITLE> Select Hotel </TITLE></HEAD><BODY>
191 List of free Hotels:
195 Select one:
199 <FORM method="POST" action="hotelPrice.php?<? = SID ?>">
200     <SELECT type="text" name="selectedHotel" size=1> <BR>
201     <? while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
202         foreach ($line as $col_value) {
204             print "<OPTION> $col_value"; } } ?>
206     <INPUT TYPE="Submit"><BR>
207 </FORM>
208 If you want change:
...
219 </BODY></HTML>
```

Figure 14. *Page* `selectHotel.php`.

hotelPrice.php:

```
220 <? include "session_variables.php"; ?>
221 <? $dbhost = "mysql.itc.it"; $dbname = "TravelAgency"; $dbuser = "TravelAgency"; $dbpass = "ttttaaaa";
222     $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die ("impossible connecting with MySQL");
223     mysql_select_db($dbname, $conn) or die ("Impossible selecting database $dbname");
224     $query = "SELECT price FROM Hotel WHERE (name = '$selectedHotel');";
225     $exec = mysql_query($query, $conn) or die("error: " . mysql_error());
226     $result = mysql_fetch_row($exec);
230     $GlobalHotel = $selectedHotel; ?>
240 <HTML><HEAD><TITLE> Hotel costs</TITLE></HEAD><BODY>
241 you have choosen: <? print $selectedHotel; ?> <BR>
242 <? $priceForDay = $result[0];
243     $daysDifference = ($GlobalendDate - $GlobalinitDate)/(60*60*24);
244     $totalCost = $priceForDay * $daysDifference;
245     $GlobalCostHotel = $totalCost; ?>
246 The daily price is: <? print $priceForDay; ?> <BR>
247 The total cost is: <? print $totalCost; ?> <BR>
250 Reserve: <BR>
252 <FORM method="POST" action="reserveHotel.php?<? = SID ?>">
253     Name: <INPUT TYPE="Text" NAME="name" SIZE=20 ><BR>
254     Surname: <INPUT TYPE="Text" NAME="surname" SIZE=20 ><BR>
255     Payment: <INPUT TYPE="Text" NAME="payment" SIZE=20 ><BR>
256     <INPUT TYPE="Submit"><BR>
257 </FORM>
258 </BODY></HTML>
```

reserveHotel.php:

```
280 <? include "session_variables.php"; ?>
282 <HTML><HEAD><TITLE> Reserve Hotel </TITLE></HEAD><BODY>
290 <? print "Mr/Mrs "; print $name; print " "; print $surname;?>
291 <? print " you will pay by "; print $payment;?> <BR>
292 <? print "you have reserved: "; print $GlobalHotel;?> <BR>
293 <? print "the city is: "; print $Globalcity; ?>
294 <? print "<BR> days: "; print date("d/m/Y", $GlobalinitDate); ?>
295 <? print " - "; print date("d/m/Y", $GlobalendDate); ?>
296 <? print "<BR> the total cost is: "; print $GlobalCostHotel; ?> <BR>
299 <A HREF="index.php?<? = SID ?>">Return to homepage</A></BODY></HTML>
```

Figure 15. *Pages* `hotelPrice.php` *and* `reserveHotel.php`.

insertion statements into account, since they originate additional data dependences. The SDG representation of the application does not change, since it is possible to conservatively treat each database table column as a global variable, without distinguishing among the different records. The only additional problem is that the SQL code has to be analyzed as well. Since it is generated dynamically, in a similar way as the HTML code, its analysis may be troublesome (and in general infeasible), and approximations obtained by constant propagation or symbolic execution may be necessary. With reference to the SQL query at line 224, a slicer which handles also the SQL code should detect the usage of the column `name` of table `Hotel`. Such a value can be considered equivalent to a global variable (for example named `_db.TravelAgency.Hotel.name`), and as such it becomes an additional input parameter of `hotelPrice.php`. The SQL statements inserting records into table `Hotel` have an indirect (mediated by parameter-passing) data dependence with the query at line 224, and thus would be included in the slice. A similar argument can be made for the SQL query at line 154.

5 Conclusions and future work

The main problems in the construction of the SDG for a Web application are the presence of code to be analyzed that is generated dynamically and the multiplicity of the involved programming languages. In our future work we will investigate techniques for the static approximation of the HTML/Javascript and SQL code generated by the PHP programs. Dynamic slicing will be also studied in more detail. Our future efforts will converge toward the implementation of a Web slicer, working on Web applications written in PHP/HTML/Javascript and including SQL queries.

References

[1] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[2] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transaction on Programming Languages and Systems*, pages 26–61, 1 1990.

[3] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[4] C.-H. Liu, D. C. Kung, P. Hsia, and C.-T. Hsu. An object-based data flow testing approach for web applications. *International Journal of Software Engineering and Knowledge Engineering*, 11(2):157–179, April 2001.

[5] F. Ricca and P. Tonella. Web applications slicing. In *Proceedings of the International Conference on Software Maintenance*, pages 148–157, Firenze, Italy, 2001.

[6] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[7] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.