Int J Artif Intell Educ (2014) 24:162–188
DOI 10.1007/s40593-014-0014-z

RESEARCH ARTICLE

Analysing Student Programs in the PHP Intelligent Tutoring System

Dinesha Weragama · Jim Reye

Published online: 4 February 2014

Abstract Programming is a subject that many beginning students find difficult. The PHP Intelligent Tutoring System (PHP ITS) has been designed with the aim of making it easier for novices to learn the PHP language in order to develop dynamic web pages. Programming requires practice. This makes it necessary to include practical exercises in any ITS that supports students learning to program. The PHP ITS works by providing exercises for students to solve and then providing feedback based on their solutions. The major challenge here is to be able to identify many semantically equivalent solutions to a single exercise. The PHP ITS achieves this by using theories of Artificial Intelligence (AI) including first-order predicate logic and classical and hierarchical planning to model the subject matter taught by the system. This paper highlights the approach taken by the PHP ITS to analyse students' programs that include a number of program constructs that are used by beginners of web development. The PHP ITS was built using this model and evaluated in a unit at the Queensland University of Technology. The results showed that it was capable of correctly analysing over 96 % of the solutions to exercises supplied by students.

Keywords PHP · Intelligent Tutoring System · Knowledge base · Program analysis

Introduction

Programming is a subject that needs to be studied by students in many disciplines. Students in a first-level programming class vary widely in their prior knowledge of relevant subject matter. The differing knowledge levels of these students need to be taken into account when designing a course to teach programming. Intelligent Tutoring Systems (ITSs) are suitable means of addressing the diverse prior knowledge of these individuals.

D. Weragama (✉) · J. Reye
Science and Engineering Faculty, Queensland University of Technology, Brisbane, Australia
e-mail: d.weragama@qut.edu.au

J. Reye
e-mail: j.reye@qut.edu.au

With the current popularity of the World Wide Web, more and more students are showing an interest in learning to develop web pages. Many resources that teach the basics of web development can be found online ("PHP Tutorial," undated; "W3Schools online web tutorials," undated). However, these do not customise their instruction based on the diverse prior knowledge of the students. No ITSs that teach web development have been published in the literature. The skills required to develop web pages are somewhat different from the skills required to develop stand-alone applications (described later). Such differences need to be taken into account when developing an ITS to teach web programming.

Programming in any form is a practical subject which many beginners find very difficult (Miliszewska and Tan 2007; Truong et al. 2003). Therefore, any course that teaches programming needs to include practical programming exercises. The students' solutions to these exercises must be analysed and appropriate feedback should be provided in order to maximise learning. For this to be possible, a particular programming language needs to be selected. Of the many available web development languages, PHP is one of the most popular ("TIOBE Programming Community Index for December 2012,"). Therefore, PHP has been selected as the language taught by this ITS.

A major challenge that is encountered when designing an ITS that teaches programming is that a programming exercise rarely has a unique solution. This is exemplified by the simple PHP program segments shown in Table 1 (Weragama 2013). All three of these programs are solutions to an exercise asking the students to first display the text 'Welcome!' on a web page, then add 3 to whatever the value existing in the variable $y and store it into a variable $x and finally display the value of variable $x on the web page. There is no way of knowing which of these methods a student will use when answering the exercise. In order to teach effectively, the ITS should be capable of identifying all these solutions as semantically equivalent and therefore identifying them as correct.

Researchers have taken different approaches to solving this problem as described later in this paper. However, these methods are difficult to use for analysing PHP programs for several reasons. Some of the methods are very specific to a certain programming language. Others can identify some alternative solutions but have difficulty as the number of possibilities increase. The programs in Table 1 show another problem encountered when dealing with PHP programming. It is possible to intermingle HTML and PHP code within each other and any analysis process needs to be capable of handling such code. The rest of this paper describes the design of a knowledge base that can handle alternative PHP programming code solutions to a

Table 1 Different methods of writing the same program

Program a	*Program b*	*Program c*

Welcome!
<?php
$x=$y+3;
echo($x);
?>
	<?php
	echo('Welcome!');
	$x=$y+3;
	echo($x);
	?>
		Welcome!
		<?php
		$z=$y+1+2;
		$x=$z;
		echo($x);
		?>

single programming exercise and results of an evaluation of the PHP ITS, which implements this knowledge base.

The main emphasis of this research is on analysing alternative solutions to programming exercises correctly rather than on the pedagogical aspects of the ITS. However, the pedagogical aspects have been addressed (briefly) in order to show that the knowledge base designed in this manner can effectively be used in an ITS to teach PHP programming.

The rest of the paper provides details of how the PHP ITS is used to analyse alternative solutions to the same exercise. The next section describes the program analysis process used by the ITS, in detail. This is followed by a section that outlines the interfaces of the PHP ITS and a section that describes how the system was evaluated under practical use and presents the results of the analysis. The second-last section compares the PHP ITS to related work while the final section discusses the conclusions of this research and offers some suggestions for future work.

Program Analysis

The knowledge base of the PHP ITS has been designed to handle alternative solutions to a given programming exercise using concepts of Artificial Intelligence (AI) to model states and changes of state. It uses a set of predicates based on First Order Logic, and a set of rules and actions based on AI approaches, in order to analyse programs written by students.

Figure 1 shows a schematic representation of how the AI problem is formulated in this ITS. A state is represented by a set of facts and each fact is specified as an instance of a predicate. Each exercise specification contains an Overall Goal and an Initial State. The Overall Goal is the set of facts that need to be present in the final state of the program in order for it to be identified as correct. The Initial State contains a set of facts that represent the state of the exercise before the student's program is analysed. This is an empty set for exercises where the student needs to write the entire code to create a

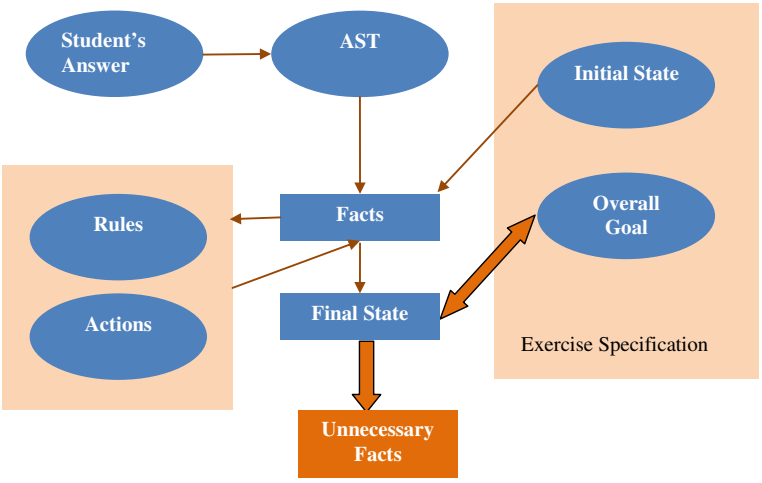

Fig. 1 Basic program analysis (Weragama 2013)

web page. However, the PHP ITS contains some gap exercises, where the student is required to complete a part of the code while the other part of the code is supplied by the exercise. This means that a certain set of facts are already present before the student's program segment is analysed. Such facts are given as the Initial State.

Once a student submits a solution to an exercise, it is converted into an Abstract Syntax Tree (AST) to make it easier to analyse (Weragama and Reye 2012a, b). This AST is then walked through node by node, creating facts that are relevant to the functionality of each node. The rules that are defined in the knowledge base are activated when certain facts are created in the knowledge base, to create additional facts. Similarly, the actions defined are activated when certain types of nodes are encountered, resulting in more facts. The final set of facts obtained in this manner, after analysing all the nodes of the AST, is known as the final state.

The final state is then compared against the Overall Goal to see if all the facts in the Overall Goal are present in the final state. If so, the student's program is identified as correct. If some facts in the Overall Goal are not present in the final state, these facts are used to identify the errors in the student's program and to provide appropriate feedback.

A common mistake made by many beginning students is to include unnecessary code in their programs. Such unnecessary statements are identified by maintaining a set of related statuses throughout the fact creation process. A new status is created each time a significant change in the set of facts occurs. When a new fact is created, a check is made to see whether it is dependent of any facts created during a previous status. If so, a link is created between the current status and this previous status. This status flow is analysed as a final step to identify any statuses that do not contribute to the Overall Goal of the program. If any such statuses are present, the program statements that resulted in the creation of these statuses are identified as unnecessary to achieve the Overall Goal and relevant feedback is provided.

It can be seen that this method of program analysis depends on the facts created during the AST walking process. This means that it should be possible to use the knowledge base created here to analyse programs written in other 3GL programming languages. The amount of additional work involved would depend on the number of differences between the AST produced by (whatever) the other language and PHP. Although this seems to be the case, the current work does not investigate this possibility.

Example of Program Analysis

This section discusses the analysis process of an example PHP program in detail. As discussed above, the knowledge base used for program analysis consists of a set of predicates, rules and actions. The exact predicates, rules and actions that come into effect are very much dependent on the type of program statements used so this analysis will consider only those relevant to the example. Consider a gap exercise where the student has been supplied code that creates a form with a textbox and a submit button. The student is required to write code to add 5 to the value entered in the textbox and display the result when the form is submitted. For simplicity, they are permitted to assume that a numeric value was entered in the textbox so it is not necessary to validate this data.

Initial State

Since this is a gap exercise, the Initial State will specify the facts that are created as a result of the form definition. The unique ids of the objects are assigned by the system during their creation and the names of the textbox and submit button are provided by the Initial State. Therefore, the facts shown in Fig. 2 are created in the system at this point.

The value stored in an HTML input element is accessed through PHP using a super-global array. In order to understand how this is handled in the knowledge base, it is first necessary to explain how PHP arrays in general are modeled. The relevant predicates and their relationships are shown in Fig. 3.

An array is actually a collection of objects, and each element in the array behaves in exactly the same method as a normal variable. Therefore, an array element is modeled as a sub-type of the *Variable* object called an *ArrayVariable*. Ordinary variables that are accessed using a name only are modeled as another sub-type of the *Variable* object called a *SimpleVariable*. Each *Variable* has a unique id given by the *HasVariableId* predicate and a value given by the *HasValue* predicate. Each *SimpleVariable* also has a name given by the *HasName* predicate.

An *ArrayVariable* is actually a relationship between an *Array* and a *Key* so it is a reification (or objectification) of a predicate that relates these two object types. The corresponding predicate is known as *HasElement*. When accessing an array element through PHP, it is possible to use any form of expression within the brackets after the array name. Therefore, the *Key* is related to an *Expression* through the *HasKeyExpression* predicate.

The array itself can be one of two sub-types: a *UserDefinedArray* or a *PreDefinedArray*, where the later is a standard part of the PHP language. Each *UserDefinedArray* has a name given by the *HasArrayName* predicate. Several types of *PreDefinedArrays* are found in PHP but for the purpose of this description, consider only *FormArrays* which are arrays that are created for accessing values entered in HTML forms. Two types of *FormArrays* are possible, based on the method of data access used by the form: *$_GET* and *$_POST*.

At this point it is necessary to explain a sub-type of the *Expression* object known as a *SimpleExpression*. These are actually *VariableExprs* and *LiteralExprs*. Each *VariableExpr* is connected to the corresponding *VariableId* through the *HasVariable* predicate. Each *LiteralExpr* is connected to another type of object known as a *Literal* which has a value given by the *HasLitValue* predicate.

```
HasMethod(FormId1,'POST')
HasInputElement(FormId1,InputId1)
HasInputName(InputId1,'textbox')
HasInputType(InputId1,'TEXT')
HasInputElement(FormId1,InputId2)
HasInputName(InputId2,'submit')
HasInputType(InputId2,'SUBMIT')
```

Fig. 2 Facts created as a result of the form definition

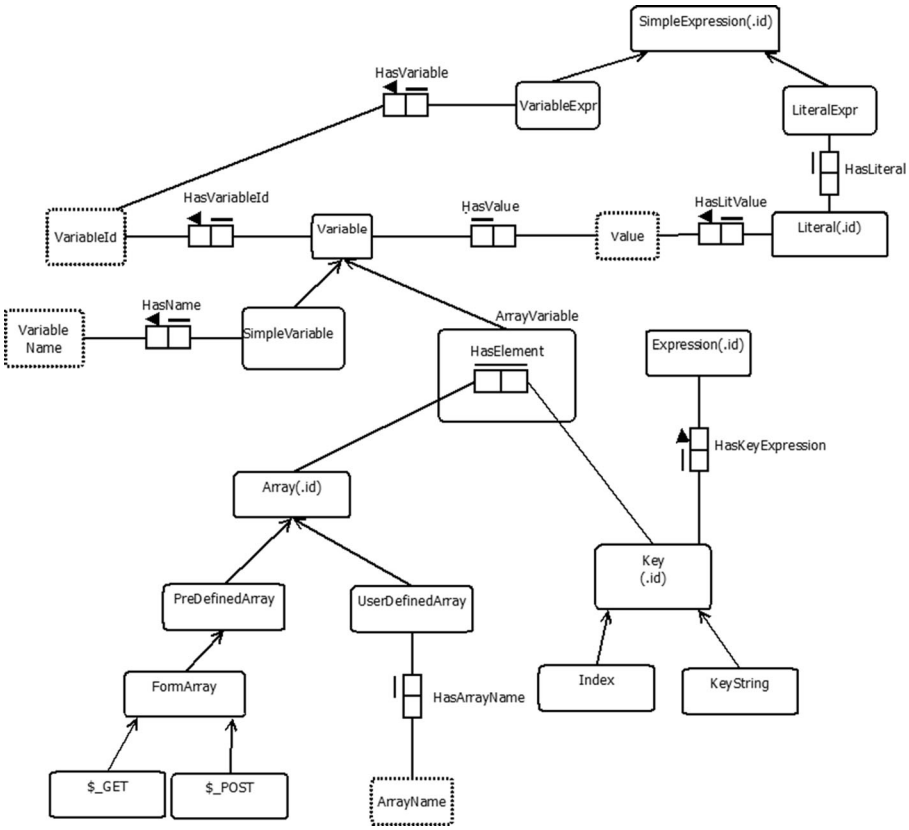

Fig. 3 Predicates relevant to array processing

The Initial State of the exercise specification also contains facts that are relevant to the *FormArray* created as a result of the form, in addition to the facts directly related to the elements of the form. These facts are shown in Fig. 4. It should be noted here that although the *ArrayElements* are created as variables, they are not assigned values in the Initial State of the program. The reason for this is that they will only contain values once the form is submitted and the supplied code does not specify any form submissions.

```
HasFormArray(FormId1,ArrId1)
HasVariableId(HasElement(ArrId1,KeyId1),VarId1)
HasKeyExpression(KeyId1,KeyExprId1)
HasLiteral(KeyExprId1,LitId1)
HasLitValue(LitId1,'textbox')
HasVariableId(HasElement(ArrId1,KeyId2),VarId2)
HasKeyExpression(KeyId2,KeyExprId2)
HasLiteral(KeyExprId2,LitId2)
HasLitValue(LitId2,'submit')
```

Fig. 4 Initial state facts related to the *FormArray*

Overall Goal

The next part of the exercise specification is the Overall Goal. This specifies the required output of the program in terms of the predicates described above. In this exercise, the value entered in the textbox should be increased by 5 and displayed on the web page, only when the form is submitted. PHP uses a pre-defined function 'isset' to check whether a variable has been assigned a value. In order to check whether a form has been submitted, this function is called with the parameter set to a name of an input element within the form, usually the submit button. Therefore, it is first necessary to see how pre-defined function calls are modeled in order to model the Overall Goal.

A call to any function, whether pre-defined or user-defined, is modeled as a *FunctionCall* object with a unique id. A *FunctionCall* needs to call some *Function* and this relationship is given by the *CallsFunction* predicate. The *Function* has a name given by the *HasFunctionName* predicate. A function can have any number of parameters given by the *HasParameter* predicate that takes in three arguments, the *FunctionId*, a *ParamPosition* and a *ParameterVariableId*. Since the parameters in a function definition must always take the form of variables, they are modeled as a special sub-type of the *Variable* object known as a *ParameterVariable*.

A function call must contain the same number of parameters as the function that it calls. These are modeled using the *HasParamExpression* predicate. This takes in the *FunctionCallId*, the *ParamPosition* and an *ExpressionId* as parameters. An *ExpressionId* is used here since the actual parameters within a function call can be any form of expression. A *FunctionCall* can have a return value given by the *HasReturnValue* predicate. Very often, a call to a function can be replaced by an expression. Therefore, a *FunctionCall* is related to a sub-type of *Expression* known as *FunctionExpr* through the *HasFunctionCall* predicate.

Using these predicates, it is possible to now define the Overall Goal of the program. The Overall Goal needs to specify that the output should only occur when the form is submitted so it is *conditional*. Such conditional goals are specified using implications. The Overall Goal for this exercise is given in Fig. 5. In broad terms, the left hand side of the implication specifies that the 'isset' function should take in a parameter with the value 'submit' and should

```
(HasFunctionName(FUNCID1,'isset')
∧ CallsFunction(FUNCCALLID1,FUNCID1)
∧ HasParamExpression(FUNCCALLID1,1,VAREXPRID1)
∧ HasVariable(VAREXPRID1,VARID1)
∧ HasVariableId(HasElement(ARRID1,KEYID1),VARID1)
∧ HasKeyExpression(KEYID1,KEYEXPRID1)
∧ HasLiteral(KEYEXPRID1,LITID1)
∧ HasLitValue(LITID1,'submit')
∧ HasReturnValue(FUNCID1,True))→
      HasVariableId(HasElement(ARRID1,KEYID2),VARID2)
      ∧ HasKeyExpression(KEYID2,KEYEXPRID2)
      ∧ HasLiteral(KEYEXPRID2,LITID2)
      ∧ HasLitValue(LITID2,'textbox')
      ∧ HasValue(VARID2,VALUE1)
      ∧ Add(VALUE1,5,VALUE2)
      ∧ OnPage(VALUE2,1)
```

Fig. 5 Overall goal

return the value True. In other words, the variable associated with the submit button should have a value or the form should have been submitted. The right hand side of the implication specifies that, if this is the case, the value of the variable associated with the textbox, or in other words the value entered in the textbox before the form was submitted, should be increased by 5 and the resultant value should be displayed. The *OnPage* predicate is a special predicate that is used to specify any data displayed on the web page. The first parameter specifies the displayed data and the second parameter specifies the position of the displayed data.

Walking the AST

The next step in the program analysis process is to convert a solution to the exercise submitted by a student into an AST. For the purpose of explanation consider a very common solution to this exercise as shown in Table 2.

The first node encountered during the AST walking process of this program is a conditional node corresponding to the *if* statement. The condition of this selection statement is a *FunctionExpr* which in turn calls a function so the following facts corresponding to this function call are created. Note that the *VariableExpr* corresponding to the parameter of the function call accesses the *ArrayVariable* corresponding to the submit button that has already been created during the Initial State.

```
HasFunctionCall(FuncExprId1,FuncCallId1)
CallsFunction(FuncCallId1,FuncId1)
HasFunctionName(FuncId1,'isset')
HasParamExpression(FuncCallId1,1,VarExprId1)
HasVariable(VarExprId1,VarId2)
```

In this case, the *FunctionCall* accesses a pre-defined function. The knowledge base stored information regarding the number of parameters and functionality of pre-defined functions. This information is now accessed to create the rest of the predicates that correspond to the 'isset' function, resulting in the following facts.

```
HasParameter(FuncId1,1,ParamVarId1)
HasReturnExpression(FuncId1,RetExprId1)
```

Table 2 A solution to example exercise

Program

`if(isset($_POST['submit']))`
`{`
`  echo($_POST['textbox']+5);`
`}`

```
HasValue(varId1,True)←
       HasFunctionCall(funcExprId1,funcCallId1)
       ∧ CallsFunction(funcCallId1,funcId1)
       ∧ HasFunctionName(funcId1,'isset')
       ∧ ValueOf(funcExprId1,True)
       ∧ HasParamExpression(funcCallId1,1,paramExprId1)
       ∧ HasVariable(paramExprId1,varId1)
```

Fig. 6 Rule to set the value of the parameter variable when the 'isset' function returns True (Weragama 2013)

Processing within the *if* part of the selection structure progresses only if the condition within the selection statement is True. Therefore, the value of the *FunctionExpr* can be taken to be true within the *if* part. Each *Expression* has a value given by the *ValueOf* predicate so the following fact is True within the *if* section.

```
ValueOf(FuncExprId1,True)
```

At this point, a special rule within the knowledge base is activated. Although many rules within the knowledge base are quite general, this rule is specific to the 'isset' function since it is extensively used during forms programming using PHP. It specifies that, if the 'isset' function returns True, the value of the variable that is the parameter to the 'isset' function will also be set to True. This is because the 'isset' function returns True only if the variable has been assigned a value. The rule used in this case is shown in Fig. 6.

Since the system now contains facts that correspond to all the premises of the rule, it is activated to create a fact that corresponds to the conclusion so the following fact is created.

```
HasValue(VarId2,True)
```

Considering the semantics of PHP form processing, all variables corresponding to input elements in the form are set when the submit button is set to True, or in other words, when the form is submitted. Another rule, shown in Fig. 7, is used to set symbolic values to these variables at this point since the actual input values are not known. For convenience, the rule sets each variable to the name of the corresponding *InputElement*. The resulting fact in this case is shown below. Note that, since the value of the *FunctionExpr* is True only within the if part of the selection statement, these facts are also only True within this scope.

```
HasValue(varId2,inputName2)←
       HasInputElement(formId1,inputElementId1)
       ∧ HasInputName(inputElementId1,inputName1)
       ∧ HasInputType(inputElementId 1,'SUBMIT')
       ∧ HasFormArray(formId1,formArrayId1)
       ∧ HasVariableId(HasElement(formArrayId1,keyId1),varId1)
       ∧ HasKeyExpression(keyId1,exprId1)
       ∧ ValueOf(exprId1,inputName1)
       ∧ HasValue(varId1,True)
       ∧ HasVariableId(HasElement(formArrayId1,keyId2),varId2)
       ∧ HasKeyExpression(keyId2,exprId2)
       ∧ ValueOf(exprId2,inputName2)
```

Fig. 7 Rule to set the values of all form variables once the form is submitted (Weragama 2013)

Next, the AST nodes corresponding to the statements within the *if* part of the selection statement are analysed. The only node here corresponds to an *echo* statement, which activates an action. The corresponding *Display* action is shown in Fig. 8. It can be seen that this action takes in an *ExpressionId* as an argument so an expression is created for whatever needs to be displayed. In the case of the program in Table 2, the expression is an addition. This has been modeled as an *AddExpr* which is a sub-type of the *CalculateExpression* object, which in turn is a sub-type of the *Expression* object. Many *CalculateExpressions* contain two sub-expressions on either side so the object is again a reification of a predicate that relates the two sub-expressions.

In this case, the sub-expression on the left hand side of the *AddExpr* is a *VariableExpr* and the one on the right hand side is a *LiteralExpr* so the following facts are created.

```
HasId(AddExpr(VarExprId2,LitExprId1),ExprId1)
HasVariable(VarExprId2,VarId1)
HasLiteral(LitExprId1,LitId3)
HasLitValue(LitId3,5)
```

It can be seen from Fig. 8 that, in order to find the pre-conditions of the action, it is necessary to find the value of the expression with id ExprId1. A set of rules have been defined in the knowledge base to find the value of many types of expressions. Figure 9 shows some of these rules which are useful for this explanation.

These rules are now activated due to facts that currently exist in the system, resulting in the following facts.

```
ValueOf(VarExprId2,'textbox')
ValueOf(LitExprId1,5)
ValueOf(ExprId1,value1) where Add('textbox',5,value1)
```

Note that *Add(x,y,z)* is a predicate which is true if the result of adding x and y is z. Such predicates are necessary since it is often necessary to deal with symbolic values during program analysis as described above.

Now, the pre-conditions of the *Display* action are satisfied. Note that rC is a variable which does not occur as a direct result of analysis of the student's solution. It is a variable which holds a running counter showing the current position where any text is displayed on a web page. This is necessary in cases where the order of display of elements is important. Since the pre-conditions are now satisfied, facts relevant to the effect of the action are now created as below.

```
OnPage(value1,1)
HasValue(rC,2) where Add(1,1,2)
```

```
Action(Display(expressionId),
PRECOND: ∃ value,rC,x
          (ValueOf(expressionId,value))
         ∧ HasValue(rC,x))
EFFECT:  OnPage(value,x)
         ∧ Add(x,1,y)
         ∧ HasValue(rC,x) ← HasValue(rC,y))
```

Fig. 8 *Display* action (Weragama 2013)

```
ValueOf(literalExpr,v) ←HasLiteral(literalExpr,literalId)
                        ∧ HasLitValue(literalId,v)

ValueOf(variableExpr,v) ←HasVariable(variableExpr,VarId)
                        ∧ HasValue(varId,v)

ValueOf(AddExpr(exprIda,exprIdb),v) ←  ValueOf(exprIda,va)
                                       ∧ ValueOf(exprIdb,vb)
                                       ∧ Add(va,vb,v)
```

Fig. 9 Rules for finding *ValueOf* expressions

All the nodes of the AST have now been analysed, resulting in the final state of the program as shown in Fig. 10. Note that, in the interest of space, only facts relevant to comparing against the Overall Goal are presented here. The final stage of program analysis is to compare this final state against the Overall Goal given in Fig. 5. It can be seen that, all the facts in the Overall Goal are present in the final state when FUNCID1=FuncId1, FUNCALLID1=FuncCallId1, VAREXPRID1=VarExprId1, VARID1=VarId2, ARRID1=ArrId1, KEYID1=KeyId2, KEYEXPRID1=KeyExprId2, LITID1=LitId2, KEYID2=KeyId1, VARID2=VarId1, KEYEXPRID2=KeyExprId2 and LITID2=LitId1. Therefore, the student's program is identified as correct.

Alternative Solutions to the Exercise

The above section discussed how the program in Table 2 is analysed by the PHP ITS. However, it is quite likely that a student entered another, equally correct solution. The program in Table 3 shows another example of a correct solution to the exercise. On inspection, it can be seen that the first difference between the programs occur within the *if* statement. Therefore, the analysis is the same as in the previous case up to this point.

```
(HasFunctionName(FuncId1,'isset')
∧ CallsFunction(FuncCallId1,FuncId1)
∧ HasParamExpression(FuncCallId1,1,VarExprId1)
∧ HasVariable(VarExprId1,VarId2)
∧ HasVariableId(HasElement(ArrId1,KeyId2),VarId2)
∧ HasKeyExpression(KeyId2,KeyExprId2)
∧ HasLiteral(KeyExprId2,LitId2)
∧ HasLitValue(LitId2,'submit')
∧ HasReturnValue(FuncId1,True))→
       HasVariableId(HasElement(ArrId1,KeyId1),VarId1)
       ∧ HasKeyExpression(KeyId1,KeyExprId1)
       ∧ HasLiteral(KeyExprId1,LitId1)
       ∧ HasLitValue(LitId1,'textbox')
       ∧ HasValue(VarId1,'textbox')
       ∧ Add('textbox',5,value1)
       ∧ OnPage(value1,1)
```

Fig. 10 Final state

In the program in Table 2, a *Display* action was immediately activated once within the *if* section. In the program in Table 3, this is replaced by an assignment statement, which is again modeled as an action as shown in Fig. 11. The arguments of this action are the name of the variable on the left hand side of the assignment statement, and the id of the expression on the right hand side. This expression can take any form but the pre-condition specifies that it should have a value. The value of any expression can be found using one or more of the rules used for this purpose. The effect in this case is dependent on whether the variable on the left hand side of the assignment statement already exists or not. If it does, the value of this variable is updated to the value of the expression. If the variable doesn't exist, it is first created and the relevant predicates to set the name and value are then created (In PHP, variables are not declared. Each variable is created when you first assign a value to it).

In the case of the program in Table 3, the expression on the right hand side is a *VariableExpr* and it accesses the *ArrayVariable* corresponding to the textbox so the following fact is created.

```
HasVariable(VarExprIdr,VarId1)
```

Now, using the rules in Fig. 9, the *ValueOf* this expression is found as below.

```
HasValue(VarExprIdr,'textbox')
```

The pre-conditions of the *Assign* action are now satisfied. In this case, a variable with the name on the left hand side, $a, does not exist so it is created, so the following facts come into existence.

```
HasName(VarIda,'a')
HasValue(VarIda,'textbox')
```

Next, the node corresponding to the *echo* statement is analysed. In this case, the expression that is within the statement is somewhat different from the previous case so the following facts are created.

```
HasId(AddExpr(VarExprId2,LitExprId1),ExprId1)
HasVariable(VarExprId2,VarIda)
HasLiteral(LitExprId1,LitId3)
HasLitValue(LitId3,5)
```

Again using the rules in Fig. 9, the following facts are created.

```
ValueOf(VarExprId2,'textbox')
ValueOf(LitExprId1,5)
ValueOf(ExprId1,value1) where Add('textbox',5,value1)
```

Table 3 Another solution to example exercise

Program

`if(isset($_POST['submit']))`
`{`
`  $a=$_POST['textbox'];`
`  echo($a+5);`
`}`

```
Action(Assign(x,expressionId),
PRECOND: ∃ value ValueOf(expressionId,value)
EFFECT:  When ∃ variableId  (HasName(variableId,'x'):
                    HasValue(variableId,_) ← HasValue(variableId,value)
      ∧ when ¬∃ variableId(HasName(variableId,'x'):
                          Generate(newVariableId)
                          HasName(newVariableId,'x')
                          HasValue(newVariableId,value)
                          HasInitialValue(newVariableId,value))
```

Fig. 11 *Assign* action (Weragama 2013)

So it can be seen that the resulting final state is exactly the same as in the previous example. This means that, even though the form of the expressions used in the program in Table 3 are different from those used in the program in Table 2, the program is still identified as correct. Other types of expression are handled in a similar manner using different rules in order to identify all semantically equivalent programs as correct.

Incorrect Solutions to the Exercise

To show the other side of the coin, we now describe how incorrect solutions to the exercise are handled by the knowledge base. Table 4 shows two incorrect solutions to this exercise. *Program a* displays the value entered in the textbox without incrementing it and *Program b* contains an unnecessary program statement where the value of the textbox is stored in a variable but the variable is never used.

The analysis of *Program a* continues as in the case of the program in Table 2. The only difference occurs when the Display action is activated. Here, the expression created is just a VariableExpr referring to the ArrayVariable corresponding to the textbox. No calculations are performed so the ValueOf the expression is the value stored in the textbox. Therefore, the final state of the program is as shown in Fig. 12. On comparing this against the Overall Goal in Fig. 5, it can be seen that a component of the goal, the Add fact, is missing so that value in the OnPage fact is incorrect. This is used to identify the fact that the value displayed on the web page is incorrect and appropriate feedback is provided. Similarly, based on which components of the goal, or which sub-goals, are missing, specific feedback about the error is provided.

Identifying the error in *Program b* is handled somewhat differently. In this case, the Overall Goal is fully satisfied by the final state. However, there is a statement that does

Table 4 Incorrect solutions to exercise

Program a	*Program b*

`if(isset($_POST['submit']))`
`{`
`  echo($_POST['textbox']);`
`}`
	`if(isset($_POST['submit']))`
	`{`
	`  $a=$_POST['textbox'];`
	`  echo($_POST['textbox']+5);`
	`}`

```
(HasFunctionName(FuncId1,'isset')
∧ CallsFunction(FuncCallId1,FuncId1)
∧ HasParamExpression(FuncCallId1,1,VarExprId1)
∧ HasVariable(VarExprId1,VarId2)
∧ HasVariableId(HasElement(ArrId1,KeyId2),VarId2)
∧ HasKeyExpression(KeyId2,KeyExprId2)
∧ HasLiteral(KeyExprId2,LitId2)
∧ HasLitValue(LitId2,'submit')
∧ HasReturnValue(FuncId1,True))→
      HasVariableId(HasElement(ArrId1,KeyId1),VarId1)
      ∧ HasKeyExpression(KeyId1,KeyExprId1)
      ∧ HasLiteral(KeyExprId1,LitId1)
      ∧ HasLitValue(LitId1,'textbox')
      ∧ HasValue(VarId1,'textbox')
      ∧ OnPage('textbox',1)
```

Fig. 12 Final state for incorrect solution

not contribute in any form to achieving this final state. As briefly described earlier, a series of statuses and their relationships are maintained during program analysis, in order to identify such unnecessary statements. If the Overall Goal is satisfied, these statuses are checked to see whether any exist that are not related to the status where the Overall Goal is satisfied. Any such statuses are identified as being created by unnecessary program statements.

The Initial State of any program creates a new status known as status 0. All facts created during the initial status are related to status 0. In this case, the creation of the *ArrayVariables* happens in this status so they are related to status 0. A new status is created each time a selection statement is encountered so a new status, status 1 is created for the *if* statement. The condition within this *if* statement refers to a variable created during status 0 so a link is maintained between status 0 and status 1.

Next, a new status, status 2 is created for the *if* part of the selection statement. Since this is a part of the main selection statement, it is linked to the relevant status, status 1. Any statuses resulting from program statements within the *if* part of the selection statement are linked to status 2. In this case, the first assignment statement creates a new status, status 3. Since it uses the value from a variable created in status 0, a link is created between these statuses. The *echo* statement within the *if* part creates a new status, status 4. Since this again accesses a variable created in status 0, these statuses are linked. The final status flow for *Program b* is shown in Fig. 13.

Status 4 is the status where the Overall Goal is satisfied. When considering the status flow, it can be seen that all statuses, except status 3 have links that terminate in status 4. Status 3 however, is at the end of a flow and does not link to status 4. This means that it does not contribute to the Overall Goal. Therefore, the statement that created this status, the assignment, is an unnecessary program statement. It is therefore identified as an error and appropriate feedback is provided.

In the case of the program in Table 2, the status flow is similar except for the fact that status 3 does not exist since there is no corresponding statement. Therefore, the status flow shows statuses that are all linked to the status where the Overall Goal is satisfied so no unnecessary program statements are present.

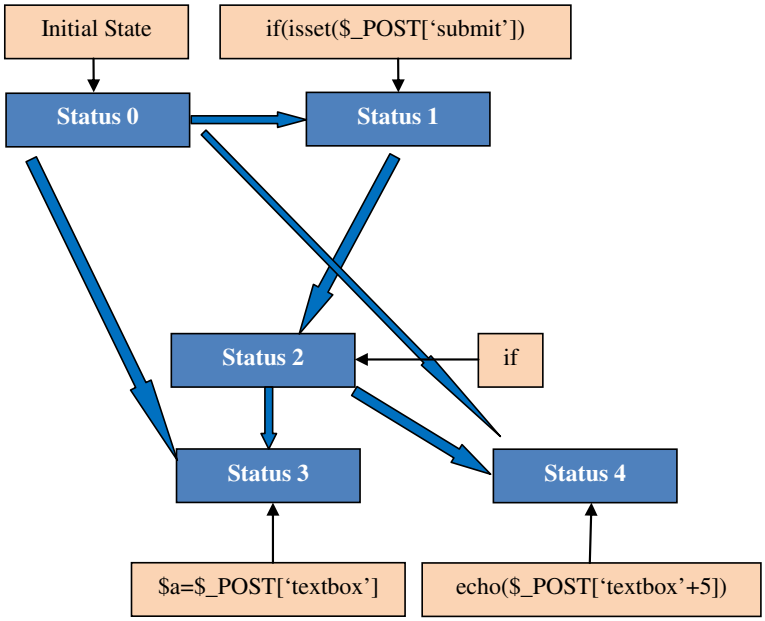

Fig. 13 Status flow for *Program b*

Types of Constructs Handled by the PHP ITS

The preceding parts of this paper describe how the PHP ITS handles different types of expressions, selection structures and pre-defined functions. The knowledge base is capable of handling several more types of constructs that are frequently encountered during novice PHP programming. This section summarizes how some of these constructs are handled.

Selection structures allow for many variations in writing programs. Conditions within selection structures allow for a multitude of semantically equivalent programs. For example, the condition $x>10 can be written as $x>=11, !($x<=10), (10<$x) and many other ways. It is necessary that the knowledge base be capable of recognizing all these variations. This is done using a set of rules that are used to convert between equivalent expressions. The number of rules is very small compared to the number of possible ways in which the expression can be written. The possibilities of writing equivalent selection structures are further increased by the fact that it is possible to write many independent *if* statements, *if-else* statements, nested *if* statements and *switch* statements that are semantically equivalent. Once such a program is converted to a set of facts as described above, it is possible to flatten out the set of facts by removing all levels of nesting. The Overall Goal in this case is also specified as a set of facts where all forms of nesting are removed. This makes it possible to identify the program as correct, no matter what levels of nesting are used.

In order to explain this further, consider an example where the student needs to write a program segment to display the grade based on a given mark. If the marks are greater than 80, the grade is an 'A'. If the marks are between 60 and 79, the grade is a 'B' and if

the marks are between 50 and 59 the grade is a 'C'. In all other cases, the grade is a 'F'. Figure 14 shows how the overall goal for this case is set up. Assume that the initial state is set up so that the VariableId of the variable holding marks is VarId1 and its initial value is val_m.

A correct program to achieve this objective can be written in a multitude of ways. Table 5 shows two correct solutions to the exercise, although they may not be ideal. The overall goal for this exercise is specified using a fully flattened state. In other words, each possible condition is enumerated separately, with the corresponding result for each such condition. In order to understand how *Program a* is analysed, consider its elseif ($marks>=60) condition. The 'else' in this case implies that the condition opposite to the condition of the corresponding 'if' (i.e. $marks>80) is satisfied here. This means that, when analyzing this section of code, LessThan(val_m,80) becomes a component of the predicate to the premise of the implication, where val_m is taken to be the initial value of the mark. There is also an additional 'if' condition (i.e. $marks>=60). The predicate corresponding to this, GreaterThanOrEqual(val_m,60) is also a component of the premise since the grade is 'B' only obtained when this condition is satisfied too. Therefore, the complete premise of the implication is a combination of these two predicates, i.e. GreaterThanOrEqual(val_m,60) ∧ LessThan(val_m,80). Therefore, the program analysis also results in a flattened state and the corresponding sub-goals are seen to be satisfied. Therefore, both these programs are identified as correct by the system.

The above program considered the use of pre-defined functions. User defined functions are handled in a similar manner but special methods are necessary to analyze the functions themselves for correctness. In this case, hierarchical planning concepts in AI are utilized to handle the function definitions. The requirements of the function are given as a set of conditions of sub-plans which needs to be satisfied in order for the function to be identified as correct. Once the function is identified as correct, a special predicate is created and the Overall Goal checks for this predicate.

A similar method is utilized for analyzing loops. Again, conditions of sub-plans are used to check the correctness of the loop before checking for the correctness of the entire program (Weragama and Reye 2013). However, loops pose some additional problems. It is possible to classify loops based on their functionality (Reye et al. 2013). All types of loops cannot be handled by the present knowledge base. Definite loops, where the number of iterations are known before the actual execution of the loop and

```
HasName(VarId2,'grade')
∧
(GreaterThanOrEqual(val_m,80)
                    → HasValue(VarId2,'A')
∧
GreaterThanOrEqual(val_m,60) ∧LessThan (val_m,80)
                    → HasValue(VarId2,'B')
∧
GreaterThanOrEqual(val_m,50) ∧LessThan (val_m,60)
                    → HasValue(VarId2,'C')
∧
LessThan(val_m,50)
                    → HasValue(VarId2,'F'))
```

Fig. 14 Overall goal for nested selection structure

Table 5 Correct solutions to nesting exercise

Program a	*Program b*

`if($marks>=80)`
`{`
`  $grade='A';`
`}`
`elseif ($marks>=60)`
`{`
`  $grade='B';`
`}`
`elseif ($marks>50)`
`{`
`  $grade='C';`
`}`
`else`
`{`
`  $grade='F';`
`}`
	`if($marks>=80)`
	`{`
	`  $grade='A';`
	`}`
	`if ($marks<80 && $marks>=60)`
	`{`
	`  $grade='B';`
	`}`
	`if ($marks<60 && $marks>50)`
	`{`
	`  $grade='C';`
	`}`
	`if($marks<50)`
	`{`
	`  $grade='F';`
	`}`

also loops that perform some action on each item in a collection independently can be handled. These account for over 50 % of loops encountered in real-world programming (Stavely 1993).

Another difficulty mentioned when handling PHP programs is the possibility of interleaving PHP and HTML code within each other. This problem has been handled by conducting the AST walking process in two steps instead of one. During the first step, all PHP statements are analysed. Some PHP sections could result in displaying text on the HTML web page but could depend on values of PHP variables. These statements are converted into the relevant HTML form. In the second step, all HTML statements, including those that result from the execution of PHP statements, are analysed.

Therefore, the knowledge base of the PHP ITS is capable of handling a large portion of the constructs encountered by beginners of PHP programming. It is difficult to say how much effort will be needed to expand the knowledge base to handle new constructs. Further research needs to be carried out in order to determine this.

The PHP Intelligent Tutoring System

The PHP Intelligent Tutoring system uses the knowledge base described above to analyze student's solutions to PHP exercises. It is implemented using a web interface where students are given the opportunity to select a PHP exercise. They can then enter a solution to the exercise and ask for feedback. Figure 15 shows the main interface of the PHP ITS where the students can enter solutions to exercises. The top section displays a description of the exercise. The left hand pane gives any pre-defined code and provides space for the student to enter a programming solution. It also contains buttons to check the answer to the solution and perform other editing functions. The right hand pane contains the feedback provided by the system. If the student's solution does not contain

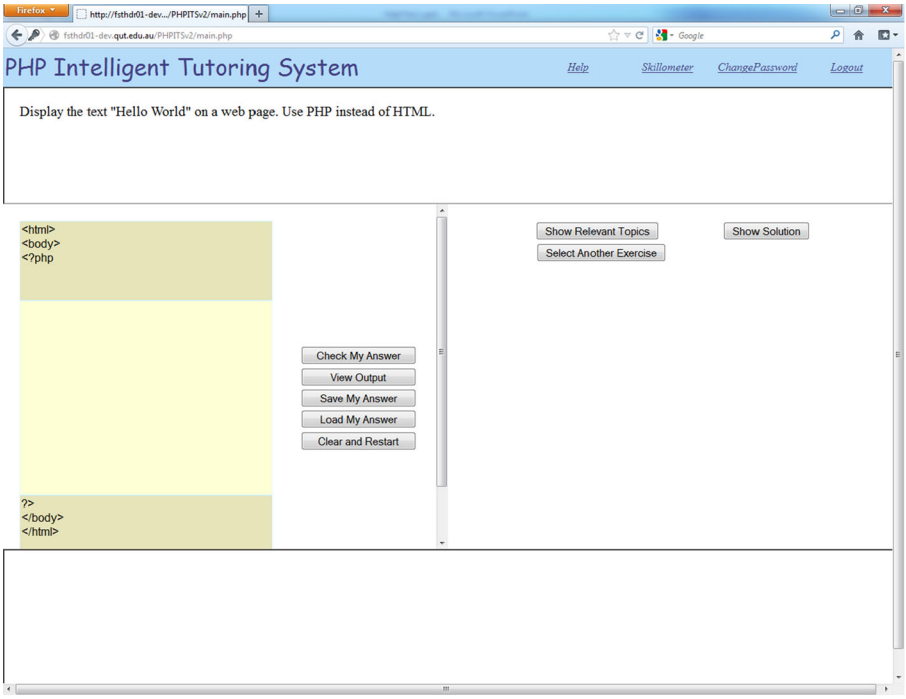

Fig. 15 The main interface of the PHP ITS (Weragama 2013)

any syntax errors, the bottom section displays the web page created by the student's code.

The system analyses the solution given by the student using the above procedure and states whether it is correct or not. If the solution is incorrect, the student can request four further levels of help regarding the error. The error message displayed depends on the sub-goals that were not satisfied, or in other words, the exact predicates in the Overall Goal that are not present in the final state. Therefore, students can obtain feedback based on the specific errors that they have made. For example, in the example exercise described in the "Program analysis" section, assume the student's program displayed the result of adding 3 to the value entered in the textbox instead of 5. This would mean that the OnPage fact in the overall goal in Fig. 5 would not be present in the final state of the system. The student can ask for two levels of help on 'What is wrong?' and two further levels of help on 'How do I fix it?'. The four help messages displayed by the system in this case are shown in Table 6. Each

Table 6 Help messages for each level

Help Type	*Level*	*Help Message*
What is wrong?	1	Your program does not display the necessary text.
What is wrong?	2	Your program does not display the result of adding 5 to the value entered in the text box.
How do I fix it?	1	Include statements to add 5 to the value entered in a textbox and display it.
How do I fix it?	2	Include the statement 'echo($_POST['textbox']+5);' in your code.

student can decide which level of help is best for him/her and ask for no further help once he/she has corrected the specific error.

The knowledge level of each student is maintained using a Bayesian Belief Network. The subject matter taught by the system is broken down into topics and each sub-goal is linked to one or more topics. The knowledge level of each topic is updated based on whether the student achieved each sub-goal or not. This is done using a simplified version of the model proposed by Reye (2004).

This student model is used to suggest the next best exercise for each student based on the topics covered by the exercises and the knowledge level of the current student of each of these topics. The concept used here is that of the Zone of Proximal Development (ZPD) as described by Vygotsky (1978). According to this concept, the ZPD is the area where a student can comfortably learn. This is slightly higher than the student's current level of knowledge but not too high. This means that the best exercise for a student to achieve optimal learning is one that contains slightly more topics than those already learned by him/her. Using this concept, the most suitable exercise is taken to be the one with the least number of unknown topics but containing at least one unknown topic. Although the system offers this suggestion, the students are free to select another exercise if they wish.

A series of web links are also maintained in the system. Each topic is linked to one or more of these web links. When an error is identified, the topics that are linked to the specific sub-goal that is not satisfied is used to find web links that are relevant to solving the current problem. These links are displayed on the feedback page, along with information about the error. Even before a student makes an error, s/he can ask for links to relevant web pages to solve a particular exercise. In this case, all topics linked to all sub-goals of that particular exercise are used to display all corresponding web links.

Currently, the PHP ITS contains 27 exercises which have been added to cover the concepts described above. Adding new exercises to the system is somewhat time consuming and cannot be done by a person without technical knowledge of the ITS. The main reason for this is that the Overall Goal for each exercise needs to be specified in terms of the set of predicates defined in the knowledge base. Any person adding exercises should therefore be familiar with the representations used in the knowledge base.

Evaluation

The PHP Intelligent Tutoring System (PHP ITS) was evaluated using two groups of postgraduate students at the Queensland University of Technology. The evaluation was conducted during two semesters. During Semester 1, 2012, the first group of 19 students used the system. The system was then improved based on the experiences of the administrators as well as feedback obtained from students using a questionnaire and a focus group discussion. The improved system was then evaluated on a second group of 15 students during Semester 2, 2012. Feedback obtained during this round of evaluation was used to further improve the system.

During the evaluation process, the students were required to work through exercises using the PHP ITS. No formal classes were provided for the students. However, some supporting web links as well as a suggested textbook were provided. Exercises were released in weekly batches during the first six weeks of the semester. Each batch

consisted of a set of exercises that covered new topics as well as a topics from previous exercises.

Each student's actions when using the system were recorded. These included the actual action performed, the date and time, the solution they submitted when they were seeking feedback and the result of the program analysis. This data was used to analyze the system's capability of correctly analyzing programs written by students.

Results and Discussion

Data was obtained for a total of 27 exercises covering 34 topics. A total of 1,062 solutions submitted by students were present in the interaction data. Of these, 384 solutions were found to be syntactically very similar to other programs. After eliminating syntactically similar programs to avoid bias in the analysis, 678 solutions were analysed.

A program was considered to be analysed correctly if the program conformed to the specifications given in the exercise and the system identified it as correct, or if the program did not conform to the specifications and the system identified it as incorrect. Although the final version of the PHP ITS showed an analysis accuracy of 96.31 %, the results obtained from the two earlier versions were somewhat less accurate. Of the 678 solutions analysed, 546 were analysed correctly by those versions. This is 80.53 % of the total number of exercises analysed.

It is useful to see whether the exercise itself affected the percentage of correctly analysed solutions. Figure 16 shows the percentage of correctly analysed solutions for each exercise, for the first two versions (combined) of the system. It can be seen that 15 of the 27 exercises were analysed correctly more than 80 % of the time. In fact, 5 exercises were analysed correctly 100 % of the time. Only 2 exercises were analysed correctly less than 60 % of the time.

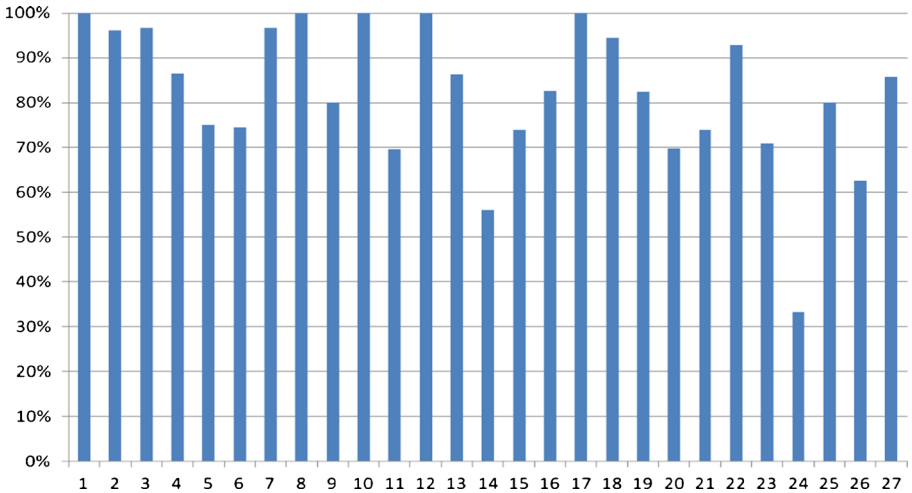

Fig. 16 Percentage of correctly analysed solutions for each of the 27 exercises by the first two versions of the system

As mentioned previously, the PHP ITS is continuously being improved. The data captured was from the system at the time the student used it. Some of the data is from the first round of evaluation while the other part of the data is from the second round. System improvements were carried out after both these rounds as well as when an error was identified due to a student inquiry. Therefore, it was important to see how the re-analysis would behave when using the third version of the knowledge base which resulted from the system improvements mentioned above. In order to test this, the programs that were incorrectly analysed during the empirical evaluation were re-analysed using the final version of the system.

Figure 17 shows the results of this analysis. It can be seen that the analysis accuracy has increased markedly to 96.31 %. All 27 exercises were now analysed correctly more than 80 % of the time and 16 exercises were analysed accurately 100 % of the time.

The exercises that are not being analysed correctly at this time were investigated to see the cause of the errors. It turned out that there were two main issues related to these errors. First, some students were using constructs which were not part of the PHP language that the PHP ITS was designed to teach. Since the knowledge base does not handle these constructs, it is unable to analyze the programs correctly. The students using the PHP ITS were not restricted to learning PHP from a single source. Therefore, they may have come across many different approaches during their studies. It is not practically possible to cater for all these variations. However, a large portion of the possible variations have been handled by the PHP ITS.

The second reason for inaccurate program analysis was a technical difficulty encountered when handling string literals. Due to implementation issues, strings were stored in the predicates using single quotes. However, as PHP permits the use of both single and double quoted literals, it has proved difficult to standardize the storage method. Although every effort has been taken to minimize such errors, there are still instances where the ITS finds it difficult to accept one form instead of the other.

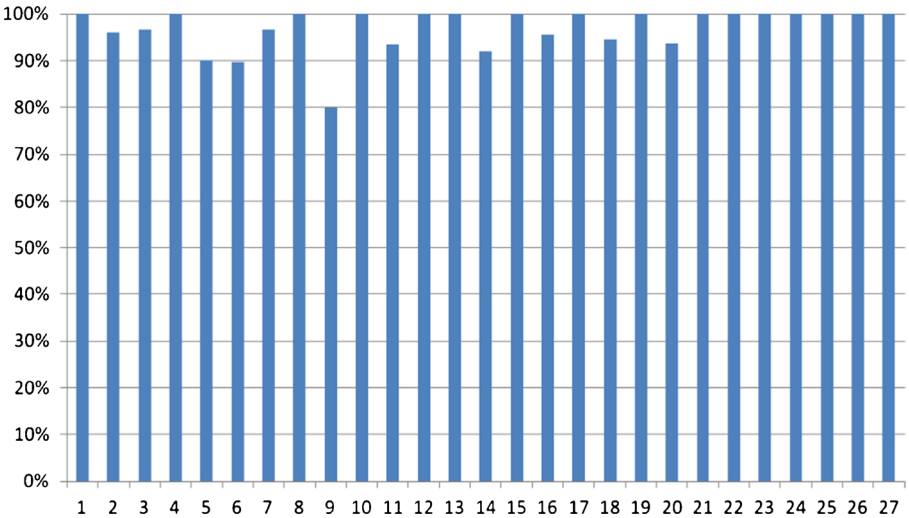

Fig. 17 Percentage of correctly analysed solutions for each of the 27 exercises by the third version of the system

The incorrectly analyzed programs were further investigated to see the nature of the incorrect analysis: whether correct programs were analysed as incorrect or incorrect programs were analysed as correct. Students are more likely to be upset if their correct programs were analysed as incorrect. However, analyzing incorrect programs as correct is equally damaging to learning since this misleads the students. The results showed that 81.54 % of the incorrectly analysed programs were incorrect programs being analysed as correct.

One premise of this work is that students' programs sometimes contain programming statements which are unnecessary for achieving the goal of the program. Of the 678 programs analysed during this study, 109 contained statements which were unnecessary. This is 16.08 % of the total number of programs and is therefore a considerable issue for novice programmers. Any guidance that can be given regarding such unnecessary statements is likely to help the students write better programs.

When considering the Intelligent Tutoring System itself, it is important to see whether the students found the analysis of their computer programs by the system useful. In order to check this, the interaction data that was stored was used to see how many attempts the students made at each exercise based on the feedback given by the system. This was accomplished by first finding the total number of times each student asked the system to check their solution for each exercise. These totals were then averaged for each student to find the average number of times each student asked for their solutions to each exercise to be checked by the system. Finally, an overall average was found by averaging the means for each student. The results showed that, on average, students asked the system to analyse their solutions 2.85 times with a standard deviation of 1.38. However, when considering the data for each question, it could be seen that the total number of attempts ranged from a minimum value of 1 to a maximum value of 30. Another measure of whether the students found the analysis process useful is whether they gave up on the exercises. The results showed that the students gave up on the exercises 5.49 % of the time.

The evaluation process also investigated whether the PHP ITS was conducive to learning the PHP scripting language. The main measure used here was a comparison of the pre- and post-test scores obtained by the students. Figure 18 shows the average pre- and post-test scores obtained by the students using the first two versions of the system.

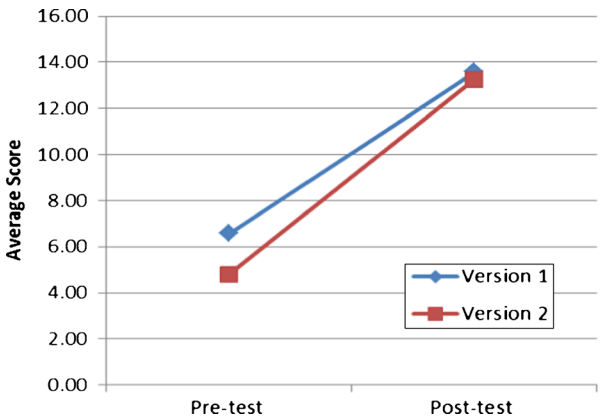

Fig. 18 Average pre-test and post-test scores

It can be seen that there is an increase in the score, indicating that students have learned PHP by using the system. A one-tailed *t*-test with a 95 % confidence interval was carried out between the pre- and post-test scores for both versions to see whether the increase was significant. Both versions returned a p-value less than 0.001. Therefore, it can be concluded that the students' test score increased significantly by using the PHP ITS.

Comparison to Related Work

Many methods of knowledge engineering have been used in the past to analyse computer programs. Of these, maintaining libraries of possible bugs is one of the least effective since it is almost impossible to store a list of all possible bugs in a program since the number of bugs is practically infinite. Additionally, maintaining bug libraries requires the use of empirical studies in order to identify common student errors. Another common problem is that bug libraries do not transfer well between different populations of students (Ohlsson and Mitrovic 2007). The program analysis methods used in the PHP Intelligent Tutoring System do not depend on bug libraries in any way and are free of the issues inherent in this method. Researchers have used many other approaches to try to solve the problem of program analysis without depending on extensive bug libraries.

The cognitive tutors, which are highly successful tutors that teach programming in Pascal, LISP and Prolog (Anderson et al. 1995; Corbett 2001; Corbett and Anderson 1992), use symbolic production rules to solve this problem. An ideal solution to each exercise is stored as a set of production rules incorporating the underlying skills that are required to solve the exercise. The students' solution is compared against these rules and any discrepancy is identified as an error in the program. The ideal solution can consist of several sets of production rules, but this becomes cumbersome as the program gets more complex and more alternative sets of rules need to be stored. Therefore, this method is only suitable for analysing small computer programs. The method of program analysis used in the PHP ITS does not depend on an ideal solution but on a set of generalized predicates. This means that many alternative solutions can result in the same set of predicates and therefore, the method is suitable for handling diversity in student programs. Cognitive tutors also provide feedback on each keystroke made by a student since such tutors work by matching the student's program to an ideal solution. This method of feedback reduces the flexibility of programming by students, thereby limiting their chances to explore. The PHP ITS does not impose such rules but allows students to complete the program in any order they want and ask for feedback only if they require it.

Constraint-Based Modelling (CBM) (Ohlsson and Mitrovic 2006) is another approach that has been utilised to solve this problem of bug libraries. This method uses a set of if-then conditions known as relevance conditions and satisfaction conditions. The relevance conditions are checked to see if they are satisfied once a student submits a solution to an exercise. If any such conditions are matched, the corresponding satisfaction condition is checked to see whether this is also satisfied. If not, an error is identified. Since no ideal solution to an exercise is stored, CBM is more capable of handling alternative solutions to a single exercise. Although this method has been used

very successfully to create ITSs that teach database concepts, its use in the programming domain is very limited (Holland et al. 2009). Only a preliminary evaluation of the J-LATTE system, which uses CBM, can be found in the literature and this covers only three types of problems: expression printing, predicate methods and simple iteration. The PHP ITS on the other hand covers many more programming constructs, including web form handling, and has been evaluated over a longer period of time with positive results.

A major problem in using symbolic rules for knowledge representation is that maintainability becomes very difficult as the number of rules increases (Hatzilygeroudis and Prentzas 2004). Limitations of human inferencing make it difficult to create a set of rules which are consistent under all possible circumstances. In order to avoid this problem, another approach is sometimes used for program analysis. This involves first converting the student's solution into some standardized form and then comparing it against a solution stored in a similar standardized form or using the same standardization technique to convert the stored solution (Gegg-Harrison 1991; Hong 2004; Jin et al. 2012; Rivers and Koedinger 2012). Although many of these methods are capable of handling very small programs, the power of some of them against more complex programs is as yet undemonstrated. Also, some of these methods can only be used for analysing programs written in a particular programming language since they utilise concepts that are specific to that language. On the other hand, the PHP ITS uses a knowledge base which covers programming concepts used in many procedural and web programming languages. Its analysis methods do not depend on concepts that are limited to a particular language, such as Prolog Programming Concepts (Hong 2004). Therefore, the concepts here should be extendible to other 3rd Generation programming languages although no effort has been made here to evaluate this possibility.

The Talus system (Murray 1987) is an automatic program debugging system which uses program verification techniques to detect and correct bugs in computer programs written in LISP. It uses Boyer-Moore Logic together with theorem proving and heuristic reasoning to detect and correct bugs. Its capabilities are limited to analysing recursive programs in a functional programming language. It cannot handle imperative programs written in procedural languages as can the PHP ITS. Also, the fact that the PHP ITS does not depend on any heuristics makes it more robust in its analysis.

Granularity based approaches are another method used for program analysis in Intelligent Tutoring Systems. These utilise granularity hierarchies which are directed graphs where nodes represent strategies. Two types of links, abstraction and aggregation, are possible between these nodes. These hierarchies are used to recognize plans in students' programs and provide appropriate feedback. This allows analysing programs and providing feedback at different levels of detail. Also, the same code can be viewed differently, based on the task at hand. The SCENT tutor, uses these concepts to analyse LISP programs (McCalla et al. 1992; Palthepu et al. 1995). One main disadvantages of this approach is that it ignores supposedly irrelevant objects. Very often, such irrelevant objects are of great interest when considering an educational context. The ability of the PHP ITS to identify irrelevant program statements and provide appropriate feedback is therefore very useful for educational purposes.

Another approach to automated program analysis is that of intention based analysis. This form of analysis is based on the fact that, in order to better understand the errors

made by a student, it is useful to first model the intention of the student when s/he was writing the program. One of the oldest and most famous systems to use this approach was PROUST (Johnson 1990; Johnson and Soloway 1985). Here, solutions to exercises are stored as program plans. A student's solution is analysed to identify which program plan it matches most closely. Error messages are provided based on the variation of the program from the plan. However, as the number of programming plans stored in the system became larger, it became more difficult to identify which plan the student was using. PROUST was not coupled with a teaching module or a student module and was termed as a "program debugger" and not an ITS. The CPP-Tutor (Naser 2008), C-Tutor (Song et al. 1997), Prolog Intelligent Tutoring System (PITS) (Looi 1991) and Java Intelligent Tutoring System (JITS) (Sykes 2007) are newer ITSs that uses the concept of intention based analysis to find errors in computer programs written by students. Although this seems very promising in terms of learning, the ability of the systems to identify the actual intention of the student when writing the program is unclear. Many use heuristic approaches to find the most closely matched solution from a number of possible solutions. This means that many different solutions need to be stored as well as analysed to identify the intention of the student. The PHP ITS does not depend on heuristics for any part of the diagnosis and therefore, the element of doubt in the analysis is reduced considerably.

Conclusions and Future Research

The PHP Intelligent Tutoring System is a system that has been designed to help students learn basic PHP programming on their own. It provides students with exercises to solve and provides feedback regarding the correctness of their solution. In doing so, it analyses PHP programs written by students to see if they match the specifications of the exercise. This is a difficult task since programming problems have many correct solutions. The knowledge base of the PHP ITS uses concepts from Artificial Intelligence (AI) and First-Order Logic to try to solve this problem.

The PHP ITS converts a student program into a set of facts and compares it against an overall goal in order to see if it is correct. Errors are identified by comparing the facts resulting from the student program against the facts in the overall goal. Appropriate feedback is provided to the student based on which exact facts are missing.

The program analysis process of the PHP ITS does not cover all aspects of programming in PHP. It is aimed at novice students wanting to learn the programming language, so it does not delve into more advanced topics such as classes and objects. It covers the basics of displaying data on web pages, selection structures, certain types of loops, pre-defined and user-defined functions, arrays and HTML form processing. It would be useful for the system to cover a wider range of PHP topics but this would require considerable effort since the knowledge representation for each of these concepts have to be considered separately.

The PHP ITS was evaluated using two sets of postgraduate students at the Queensland University of Technology. The solutions submitted by the students were stored together with the results of the analysis. These results were then examined manually to see whether the program analysis process was correct. The results of this manual analysis showed that the PHP ITS analysed the solutions correctly over 96 % of

the time. However, the accuracy should be tested by further empirical evaluations of the system. Such evaluations could also investigate the interesting question as to what level of accuracy must be achieved by the system in order to maintain student confidence. While 100 % accuracy is the ideal, it may be that a system that is 95 % accurate (say) is sufficient in the complex domain of computer programming. I.e. a system that is mostly correct is better than no system at all.

The main emphasis of this research was on the program analysis process. However, a complete ITS needs to have many more components in order to fully support student learning. The interfaces and tutoring aspects of the current PHP ITS are at a basic level and their contribution to the learning process have not been tested here. More evaluation work could be carried out to judge the understandability of the error messages. It would also be useful to see the perception of the users who used the system in such aspects as ease of use and response speed. The overall learner satisfaction of the system would be a useful measure in understanding further improvements to the system.

This research evaluated the program analysis process of the PHP ITS and its potential for successfully teaching the PHP scripting language to novices. The results showed that the students' test scores increased significantly after using the system, indicating that it was successful in teaching the subject, at least to beginning students.

References

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: lessons learned. *The Journal of Learning Sciences, 4*(2), 167–207. doi:10.1207/s15327809jls0402_2.

Corbett, A. T. (2001). Cognitive computer tutors: solving the two-sigma problem. In M. Bauer, P. J. Gmytrasiewicz, & J. Vassileva (Eds.), *8th International Conference on User Modeling Lecture Notes in Computer Science, UM 2001* (Vol. 2109, pp. 137–147). Sonthofen: Springer Verlag. doi:10.1007/3-540-44566-8.

Corbett, A. T., & Anderson, J. R. (1992). Student modeling and mastery learning in a computer-based programming tutor. In C. Frasson, G. Gauthier, & G. I. McCalla (Eds.), *2nd International Conference on Intelligent Tutoring Systems* (608). Berlin: Springer. Retrieved from http://repository.cmu.edu/cgi/viewcontent.cgi?article=1088&context=psychology.

Gegg-Harrison, T. S. (1991). Learning prolog in a schema-based environment. *Instructional Science, 20*(2), 173–192. doi:10.1007/BF00120881.

Hatzilygeroudis, I., & Prentzas, J. (2004). Knowledge representation requirements for Intelligent Tutoring Systems. In J. C. Lester, R. M. Vicari, & F. Paraguacu (Eds.), *7th International Conference on Intelligent Tutoring Systems* (Vol. 3220, pp. 87–97). Berlin: Springer.

Holland, J., Mitrovic, A., & Martin, B. (2009). J-LATTE: a constraint-based tutor for Java. In S. C. Kong, H. Ogata, H. C. Arnseth, C. K. K. Chan, T. Hirashima, F. Klett, J. H. M. Lee, C. C. Liu, C. K. Looi, M. Milrad, A. Mitrovic, K. Nakabayashi, S. L. Wong, & S. J. H. Yang (Eds.), *17th International Conference on Computers in Education* (pp. 142–146). Hong Kong: Asia-Pacific Society for Computers in Education.

Hong, J. (2004). Guided programming and automated error analysis in an intelligent Prolog tutor. *International Journal of Human-Computer Studies, 61*(4), 505–534. doi:10.1016/j.ijhcs.2004.02.001.

Jin, W., Barnes, T., Stamper, J., Eagle, M., Johnson, M., & Lehmann, L. (2012). Program representation for automatic hint generation for a data-driven novice programming tutor. In S. Cerri, W. Clancey, G. Papadourakis, & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 304–309). Berlin: Springer.

Johnson, W. L. (1990). Understanding and debugging novice programs. *Artificial Intelligence, 42*(1), 51–97. doi:10.1016/0004-3702(90)90094-G.

Johnson, W. L., & Soloway, E. (1985). PROUST: knowledge-based program understanding. *IEEE Transactions on Software Engineering, SE-11*(3), 267–275. doi:10.1109/TSE.1985.232210.

Looi, C. K. (1991). Automatic debugging of prolog programs in a prolog intelligent tutoring system. *Instructional Science, 20*(2–3), 215–263. doi:10.1007/BF00120883.

McCalla, G., Greer, J., Barrie, B., & Pospisil, P. (1992). Granularity hierarchies. *Computers & Mathematics with Applications, 23*(2–5), 363–375. doi:10.1016/0898-1221(92)90148-B.

Miliszewska, I., & Tan, G. (2007). Befriending computer programming: a proposed approach to teaching introductory programming. *Issues in Informing Science & Information Technology, 4*, 277. http://proceedings.informingscience.org/InSITE2007/IISITv4p277-289Mili310.pdf.

Murray, W. R. (1987). Talus: automatic program debugging for intelligent tutoring systems. *Computational Intelligence, 3*(1), 1–16.

Naser, S. S. A. (2008). Developing an intelligent tutoring system for students learning to program in C++. *Information Technology Journal, 7*(7), 1055–1060. http://docsdrive.com/pdfs/ansinet/itj/2008/1055-1060.pdf.

Ohlsson, S., & Mitrovic, A. (2006). Constraint-based knowledge representation for individualized instruction. *Computer Science and Information Systems, 3*(1), 1–22. http://www.cosc.canterbury.ac.nz/tanja.mitrovic/comsis.pdf.

Ohlsson, S., & Mitrovic, A. (2007). Fidelity and efficiency of knowledge representations for intelligent tutoring systems. *Technology, Instruction, Cognition & Learning., 5*, 101–132.

Palthepu, S., McCalla, G., & Greer, J. (1995). *Granularity in reverse engineering*. Canada: ARIES Laboratory, Department of Computer Science, University of Saskatchewan.

PHP Tutorial. (undated). Retrieved May 13, 2010, from http://www.w3schools.com/php/default.asp.

Reye, J. (2004). Student modelling based on belief networks. *International Journal of Artificial Intelligence in Education, 14*(1), 63–96. http://www.ijaied.org/pub/956/file/956_Reye04.pdf.

Reye, J., Weragama, D., Hartanto, B. (2013). *Loops and collections*.

Rivers, K., & Koedinger, K. (2012). A canonicalizing model for building programming tutors. In S. Cerri, W. Clancey, G. Papadourakis, & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 591–593). Berlin: Springer.

Song, J. S., Hahn, S. H., Tak, K. Y., & Kim, J. H. (1997). An intelligent tutoring system for introductory C language course. *Computers & Education, 28*(2), 93–102. doi:10.1016/s0360-1315(97)00003-1.

Stavely, A. M. (1993). An empirical-study of iteration in applications software. *Journal of Systems and Software, 22*(3), 167–177.

Sykes, E. (2007). Developmental process model for the Java Intelligent Tutoring System. *Journal of Interactive Learning Research, 18*(3), 399–410.

TIOBE Programming Community Index for December 2012. Retrieved December 18, 2012, from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Truong, N., Bancroft, P., & Roe, P. (2003). A web based environment for learning to program. *Proceedings of the 26th Australasian Computer Science Conference, 16*, 255–264. http://crpit.com/confpapers/CRPITV16Truong.pdf.

Vygotsky, L. (1978). Interaction between learning and development. *Readings on the Development of Children* (pp. 34–41).

W3Schools online web tutorials. (undated). Retrieved July 25, 2011, from http://www.w3schools.com/.

Weragama, D. (2013). Intelligent tutoring system for learning PHP. Doctoral dissertation, Queensland University of Technology, QLD, Australia.

Weragama, D., & Reye, J. (2012a). Design of a knowledge base to teach programming. In S. Cerri, W. Clancey, G. Papadourakis, & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 600–602). Berlin: Springer-Verlag.

Weragama, D., & Reye, J. (2012b). Designing the knowledge base for a PHP tutor. In S. Cerri, W. Clancey, G. Papadourakis, & K. Panourgia (Eds.), *11th International Conference on Intelligent Tutoring Systems* (Vol. 7315, pp. 628–629). Berlin: Springer.

Weragama, D., & Reye, J. (2013). The PHP intelligent tutoring system. In H. C. Lane, K. Yasef, J. Mostow, & P. Pavlik (Eds.), *16th International Conference Artificial Intelligence in Education* (Vol. 7926, pp. 583–586). Berlin: Springer.