Int J Artif Intell Educ (2015) 25:118–156
DOI 10.1007/s40593-014-0030-z

RESEARCH ARTICLE

Designing a Knowledge Representation Approach for the Generation of Pedagogical Interventions by MTTs

Luc Paquette · Jean-François Lebeau · Gabriel Beaulieu · André Mayers

Published online: 8 October 2014

Abstract Model-tracing tutors (MTTs) have proven effective for the tutoring of well-defined tasks, but the pedagogical interventions they produce are limited and usually require the inclusion of pedagogical content, such as text message templates, in the model of the task. The capability to generate pedagogical content would be beneficial to MTT frameworks, as it would lessen the task-specific efforts and could lead to the capability of providing more sophisticated pedagogical interventions. In this paper, we show how Astus, as an MTT framework, strive to attain a higher level of automation when generating pedagogical interventions compared to other MTT frameworks such as TDK and CTAT's MTTs. This is achieved by designing a knowledge representation approach in which each type of knowledge unit has a clearly defined semantic on which the MTT's pedagogical module can rely on. We explain how this knowledge representation approach is implemented as a knowledge-based system in ASTUS and show how it allows the development of MTTs that can automatically generate the pedagogical content required to provide next-step hints and negative feedback on errors. Multiple small-scale experiments were conducted with computer science undergraduate students in order to obtain a preliminary assessment of the effectiveness of Astus's pedagogical interventions.

Keywords Model-tracing · Knowledge representation · Pedagogical intervention · Next-step hints · Negative feedback

Introduction

Model-tracing tutors (MTTs) have proven successful for the tutoring of well-defined tasks such as LISP programming (Anderson et al. 1989), middle-school mathematics

L. Paquette (✉)
Teachers College, Columbia University, New York, NY, USA
e-mail: paquette@tc.columbia.edu

J.-F. Lebeau · G. Beaulieu · A. Mayers
Université de Sherbrooke, Sherbrooke, Québec, Canada

(Koedinger and Anderson 1993; Aleven et al. 2009a, b) and physics (VanLehn et al. 2005). Their main distinguishing feature is their capability to follow learners on a step-by-step basis by tracing their actions against an executable model of the task. This allows MTTs to provide pedagogical interventions such as flag feedback (indicating whether the learner's step is correct or not by highlighting it in green or red), next-step hints and negative feedback on errors.

Research on MTTs has slowed in recent years as researchers have focused their efforts on defining or improving other paradigms for developing intelligent tutoring systems (ITSs) that address some of the limitations of MTTs. Constraint-based tutors (Mitrovic 2010) can be used to model ill-defined tasks (Mitrovic and Weerasinghe 2009) such as writing SQL queries (Mitrovic 1998; Mitrovic and Ohlsson 1999). The example-tracing approach (Aleven et al. 2009a, b) is an efficient way to produce tutors with similar behaviors to MTTs with less modeling effort (Aleven et al. 2006). Machine learning has been used to model ill-defined tasks such as logic proofs (Barnes et al. 2008).

We believe there is still room to improve MTTs by giving them the capability to provide more sophisticated pedagogical interventions. Originally, the design of MTTs and other ITSs was to be very modular (Wenger 1987, Woolf 2009). MTTs would possess a sophisticated pedagogical module capable of producing pedagogical interventions that are adapted to the learners and to the different learning situations they encounter. Although some modern ITS frameworks, such as Gift (Sottilare et al. 2012), aim for the implementation of a sophisticated pedagogical module that is independent of the task, this modularity has proven challenging to implement in MTT frameworks. An alternative that has been used by frameworks such as the Cognitive Tutors (Anderson et al. 1990) is to include pedagogical content in the model of the task. With this approach, appropriate interventions are selected using task-independent processes, but using task-specific pedagogical content, such as text message templates and common errors (buggy rules), to instantiate those interventions.

Despite this approach's success (Anderson et al. 1995), the types of interventions that can be produced by the pedagogical module and their adaptation to different learning situations is limited by their use of task-specific pedagogical content that must be created before the tutor's execution. To address this limitation, Neil Heffernan (Heffernan et al. 2008) suggested the inclusion of pedagogical knowledge to the Cognitive Tutors' model of the task. This approach allows the production of pedagogical behaviors that are closer to that of human tutors, but the process of creating a model of the pedagogical knowledge increases the task-specific efforts required to author an MTT, a factor that already limits the use of current MTT frameworks.

The aim of our research is to improve the sophistication of the pedagogical modules of MTTs while minimizing the pedagogical content that must be included in the model of the task. To achieve this, we developed a knowledge-based system (KBS) based on a knowledge representation approach where each type of knowledge unit has a clearly defined semantic on which the MTT's modules can rely on. This allows the pedagogical module to interpret the model of the task, thus it is able to automatically generate interventions as well as the pedagogical content used to instantiate them.

We implemented this system in Astus, a MTT framework[1] (Paquette et al. 2010) and we showed how Astus is able to automatically generate pedagogical content for two

[1] A brief overview of the process of developing a MTT using Astus is presented in appendix A.

types of interventions: next-step hints and negative feedback on errors. Although both those types of intervention are provided by other MTT frameworks such as TDK (Anderson and Pelletier 1991) and CTAT (Aleven et al. 2006; Aleven et al. 2009a, b), those frameworks require that their pedagogical content (hint messages and common errors) be included in the model of the task whereas Astus's pedagogical module automatically generates it. In addition, we consider the automatic generation of those interventions to be a necessary first step towards building a pedagogical module capable of generating more sophisticated interventions.

There have been multiple attempts in the past to enable ITSs to automatically generate pedagogical interventions and content, but this approach has not been widely adopted by modern frameworks. Rickel (1988) proposed a framework (TOTS) that would allow ITSs for well-defined tasks to generate interventions by using a KBS based on procedural networks (Sacerdoti 1975). Unfortunately, we found no examples of how the TOTS framework has been used to author tutors and of the interventions generated by TOTS. Steve (Rickel and Johnson 1999) uses a similar approach to generate explanations in an ITS that helps students learn to perform physical tasks such as operating complex machinery. Likewise, REACT (Hill and Johnson 1993), a trainer for operators of deep-space communication stations, uses a similar KBS to implement impasse-driven tutoring (Hill and Johnson 1995) that includes generated pedagogical interventions on errors. Finally, the GIL tutoring system (Reiser et al. 1992) is able to generate interventions for LISP programming tasks, using a production system specially extended for that purpose.

Stamper et al. (2013) also worked on the problem of generating next-step hints within a logic proof tutor. Their approach consists in generating Markov Decision Processes (MDPs) to graph the states transitions in a task case using traces of previous learners and experts. Using those MDPs, the tutor is able to determine which step a learner should execute next. This approach is especially useful for ill-defined domains, where multiple sequences of steps can be used to successfully perform the task case. This allows the tutor to generate hints regarding the step that should be executed next. This approach greatly differs from Astus's. Whereas Astus uses task-independent processes to generate pedagogical content from an expert defined model of the task, Stamper et al. (2013) use a task-independent process to generate MDPs that are used to instantiate task-specific hints.

In this paper, we first present a brief summary of production systems, a KBS that is usually associated with MTT frameworks. Then, we examine how a KBS can be adapted for the generation of pedagogical interventions, and apply the resulting system to the generation of next-step hints in Astus. We go on to show how Astus can also generate negative feedback on errors. Finally, we present the results of small-scale experiments conducted to provide an initial evaluation of the interventions generated by Astus and reveal possible improvements.

Production Systems

Production systems are classically used by MTT frameworks to model a task (Aleven 2010) for two main reasons. First, production rules can be considered as the base unit of procedural memory and can thus be used to model the procedural knowledge of an ideal learner. Classical MTTs make the hypothesis that such a model is efficient for

tutoring (Anderson et al. 1990). Second, production rules are modular and expressive. The author of a classical MTT can design a set of production rules, for which the execution produces a list of the steps[2] required to perform a task, without being constrained by the structure of the production rules themselves.

In production systems, the task is modeled using two main structures: working memory elements (WMEs) and production rules. WMEs are declarative knowledge units describing the objects in working memory (WM) by specifying a set of attributes whose values are references to other WMEs or primitive data (numbers, strings, etc.). Production rules are IF-THEN structures, where the IF part specifies conditions describing the WM state required to execute the rule and the THEN part specifies the rule's actions (modifications to WM or steps in the learning environment). Instances of the WME are used to describe the MTT's interpretation of the learner's mental representation of the task. This allows the MTT to interpret the learner's action by matching production rules against the content of WM to find a chain of rules that explains the executed step.

The practical expressivity of WMEs and production rules make them a powerful tool for modeling the task, but it can be difficult for MTTs that use them to generate pedagogical interventions. Their expressivity makes it difficult for the MTT to analyze their content, which greatly limits how the MTT can use the content of the model of the task to produce pedagogical interventions. Although some interventions, such as flag feedback, can be achieved without requiring additional pedagogical content, this is usually not the case. For example, to provide next-step hints, classical MTTs require that specific hint templates be associated with each of the model's production rules. Figure 1 shows how a rule taken from CTAT's (Aleven et al. 2006) fraction addition MTT[3] uses the "construct-message" keyword to include next-step hints in the model of the task.

Knowledge-Based System

Increasing the sophistication of the pedagogical interventions produced by an MTT framework requires that its pedagogical module be able to access rich pedagogical content regarding the task. One way to achieve this without increasing the task modeling efforts is to use structures that allow the automatic generation of pedagogical content. To implement this approach in Astus, we implemented a KBS based on a knowledge representation approach where the model of the task reflects the teacher's instructions rather than the procedural knowledge of an ideal learner. This system allows the creation of models of the task that can be used to: 1) generate pedagogical content, by interpreting the modeled instructions, and 2) trace the learners' steps, by instantiating the procedure described by the instructions. Using its KBS, Astus is able to implement a pedagogical module that has the capability to automatically generate pedagogical interventions by interpreting the model of the task.

In this section, we present three features of Astus's KBS that allows it to generate pedagogical interventions: its hierarchical procedural knowledge structure, its explicit procedural knowledge units and its semantically rich declarative knowledge units. We

[2] A step is an atomic action in the learning environment that modifies the state of the task (VanLehn 2006).
[3] http://ctat.pact.cs.cmu.edu

```
(defrule same-denominators
  ?problem <- (problem (given-fractions ?f1 ?f2) (subgoals))
  ?f1 <- (fraction (denominator ?denom1))
  ?f2 <- (fraction (denominator ?denom2))
  ?denom1 <- (textField (value ?d &:(neq ?d nil)))
  ?denom2 <- (textField (value ?d))
=>
  (bind ?sub (assert (add-fractions-goal (fractions ?f1 ?f2))))
  (modify ?problem (subgoals ?sub))
  (construct-message
  "[The two fractions have the same denominator, so no need to convert them.]")
)
```

Fig. 1 Example of a production rule taken from CTAT's fraction addition MTT

describe the aspects of classical production rule based KBSs that are relevant for each feature. Then we explain how the feature is implemented in Astus's KBS. Finally, using next-step hints as an example, we show how these features allow Astus to generate increasingly complete pedagogical interventions. The examples presented throughout this section are taken from MTTs for the insertion of elements into an AVL tree[4] (Paquette et al. 2013) and subtraction (Paquette et al. 2010).

Hierarchical Procedural Knowledge Structure

The hierarchical structure of procedural knowledge is used to locate specific knowledge units during the global process of performing a task. When included in an MTT, this allows the MTT to intervene by providing the learner with information regarding his/her progress toward performing the task.

When modeling a task for a classical MTT, the production rules are designed to be chained in a specific order that models the learner's cognitive processes. The result of the execution of these chains forms an implicit hierarchy that is determined by the content of the rules: the rules' application conditions are designed to match specific actions resulting from other rules. Since the chaining is not explicitly described in the rule syntax, and the MTT cannot interpret the task-specific content of production rules, it is difficult for the MTT to infer the hierarchical structure of the model's procedural knowledge.

One way to make the hierarchical structure of the procedural knowledge explicit in production systems is to introduce the concept of goals. In such a system, the application condition of every rule must specify a goal that will be achieved by its execution, and the rule's actions can add goals to WM. Goals have been used in production systems such as CTAT's Cognitive Tutors to control rule chaining (Fig. 1), but they are not required by the rule syntax (goals are standard WMEs). By making goals mandatory and including them in the rule syntax, they become an explicit part of the MTT, allowing it to easily analyze the hierarchical structure of the rules to produce a procedural graph of the interactions between the rules and the goals (Fig. 2).

Astus

Rather than using production rules as its base procedural knowledge unit, Astus's KBS uses goals and procedures organized in a graph similar to a procedural network

[4] AVL Trees are a self-balancing binary search tree in which the heights of the two child subtrees of any nodes differ at most by one (Adelson-Velskii and Landis 1962).

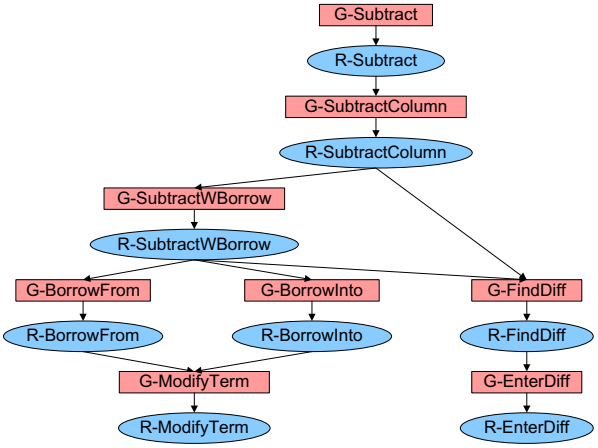

Fig. 2 Example of a procedural graph for the subtraction task. Rules are represented as ovals prefixed with "R-" and goals are rectangles prefixed with "G-"

(Sacerdoti 1975). A goal represents the intent of performing a task or a sub-task, whereas a procedure represents a particular way to achieve a goal. Because procedures are associated with a unique parent goal and because their syntax is designed to make explicit the sub-goals they instantiate, Astus can produce a procedural graph very similar to the one shown in Fig. 2. Instead of rules (ovals prefixed with "R-"), Astus uses procedures (goals are depicted as rectangles and procedures as ovals).

More specifically, Astus's procedural graphs can be compared to hierarchical task networks (HTNs) as defined by Erol et al. (1994) As their procedural networks ancestors, HTNs represent procedural knowledge in a more declarative form than production systems (Sacerdoti 1975). This allows HTNs to more easily monitor and reason on the execution of the procedural knowledge. As such, using a hierarchical structure of procedural knowledge in Astus has a beneficial impact on the generation of pedagogical interventions. It allows Astus's pedagogical module to examine how the different procedural knowledge units interact with each other. This information can then be used when generating pedagogical content for interventions such as next-step hints.

When comparing Astus's procedural graph to HTNs, *tasks* (as defined by HTNs) like goals represent what needs to be done. As Astus's goals are achieved by procedures, an HTN's *coumpound* tasks are performed trough *methods* that decompose a task into sub-tasks and *primitive* tasks are performed by *planning operators* that correspond to steps in an MTT. The main difference between Astus and HTNs is in how they instantiate their plans. The knowledge units in HTNs are designed to enable a sound and complete search for plans. In Astus, the procedural graph is assumed to be conforming to the experts' intent (as an expert system). As such, the procedural graph is used to implement a simple plan recognition algorithm known as *model tracing* (Anderson et al. 1990; Astus's implementation is described in the *model tracing in Astus* section of this paper).

An additional argument for the use of procedural graphs comes from Sierra (VanLehn 1990), a theory explaining the origin of the learners' procedural errors. Sierra showed evidence suggesting that learners use a hierarchical structure of goals to regulate their problem solving process. According to this theory, the emergence of

some of the learners' errors is difficult to explain without the use of such a structure. Astus's procedural graph meets Sierra's criterion for a hierarchical goal structure and can thus provide useful information when generating pedagogical content regarding the learners' errors.

Next-Step Hints

Although the information contained in the procedural graph is useful when generating hints, such hints would be incomplete. At most, a template, such as "In order to [parent goal], you need to [sub-goals]", could be defined and applied to procedures (e.g. Fig. 3) to provide information about which goals need to be achieved, but those hints would provide no information about how to organize those goals. Is it necessary to achieve all the goals? Is achieving one of them enough? Is the order of the goals relevant? Should the goal be achieved more than once? To answer these questions, the procedural knowledge units need to provide additional information regarding the organization of their sub-goals.

Explicit Procedural Knowledge Units

In order for the MTT to automatically generate pedagogical content regarding the task's procedural knowledge units, it must have the capability to understand their behavior. Their semantic must thus be explicit to the MTT so it can obtain information about their content and infer the outcomes of executing them.

In production systems, the rule execution order is implicit in the content of the model's production rules.[5] In order for a rule to be fired, its application condition must match the content of WM. Either the condition matches the initial state of WM and the rule can be executed at the beginning of the task, or the execution of an available rule modifies the content of WM in a way that allows additional rules to match.[6] Thus, the rules are executed following a specific organization that is implicit in their application conditions and their actions.

To make the organization of the rules' execution explicit in the MTT, the KBS needs to allow the MTT to interpret the rules' application conditions and the effects of their execution on WM. This would allow the MTT to infer the rule organization by associating the outcomes of a rule's action and the application conditions of other rules. This information could then be used to generate pedagogical interventions.

Astus

In Astus, procedures fall into two main categories: primitive and complex. Primitive procedures have no sub-goals; instead they are associated to the different type of steps that can be performed in the learning environment. Complex procedures are associated to a general type of sub-goals organization that is known by the MTT's pedagogical module. For each procedure added in the model of the task, the author can further customize the organization of its sub-goals.

[5] The author can also specify static priorities that are used when two or more rules can be executed simultaneously

[6] The WM effects of a rule can also prevent rules that were previously available from firing

Formalizing the procedural knowledge by the use of complex procedures that describe the organization of their sub-goals is analogous to the way procedural knowledge is treated in VanLehn's Sierra theory (VanLehn 1990). According to this theory, learners use three main control structures when performing tasks: AND, OR and FOR-EACH goals. Astus's complex procedures extend these basic structures to offer unordered, partially ordered and totally ordered sequences, conditional repetitions (loops), for-each repetitions (over a sequence of objects), conditional selections and nondeterministic selections (over a set of objects).

Next-Step Hints

Knowing the semantic of each procedural knowledge unit allows Astus to provide hints that refer to the organization of a procedure's sub-goals. Since each procedure has a specific type whose structure is known to Astus, it can interpret the procedure's content to produce its pedagogical interventions. As such, Astus can generate next-step hints by using an appropriate template, based on the type of procedure the hint refers to.

Using the additional information available from the procedure type, Astus can generate hints for the procedures presented in Fig. 3. For example, when provided with the additional knowledge that the procedure *PInsert* is a totally ordered sequence, the hint can indicate that all of the procedure's sub-goals must be achieved in the correct order:

```
In order to insert an element, you need to do the following in
the given order:
1) insert in a sub-tree
2) update
3) check for imbalances
```

Similarly, a hint can be generated for *PInsertSubTree* using the knowledge that the procedure is a conditional selection of a sub-goal:

```
In order to insert in a sub-tree, you need to either insert to
the left or insert to the right.
```

Finally, *PSubtract* is the repetition of a single sub-goal:

```
In order to subtract, you need to repeatedly subtract a
column.
```

Although these hints provide instructions regarding the sub-goals' organization, there is still room for improvement. For example, when providing a hint for a selection, the MTT should be able to explain when each sub-goal should be selected; and, when generating a hint regarding a repetition, the MTT should be able to indicate how many times the sub-

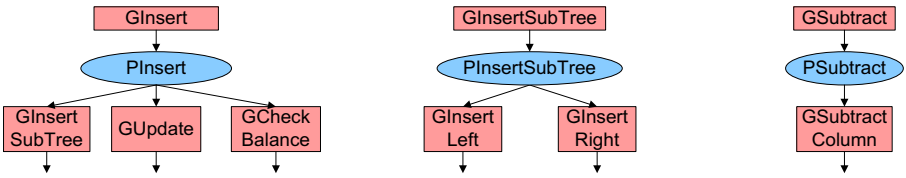

Fig. 3 The procedural graph of three procedures for which Astus can generate next-step hints

goal should be repeated. In order to include this information in the hints it generates, the MTT must be able to interpret the content of the procedural knowledge units in more detail. This requires that the MTT be able to interpret how procedural knowledge units access and manipulate the content of WM or Astus's counterpart, the knowledge base (KB).

Semantically Rich Declarative Knowledge Units

In order to automatically generate interventions giving instructions on how to execute specific procedural knowledge units, the MTT needs to be able to interpret their full content, not just their hierarchical structure and the organization of their execution. The MTT needs to have access to information regarding the declarative knowledge that is manipulated by the procedural knowledge. For production rules, this implies that the MTT needs to be able to analyze the application condition of a rule to determine which objects from WM are useful, and how the rule manipulates those objects.

To manipulate objects from WM, production systems offer one type of declarative knowledge unit: working memory elements (WMEs). WMEs are expressive, as a WME is composed of a set of untyped attributes, but their structure is not a rich source of information regarding their usage. Since the attributes are not typed, the MTT cannot analyze the links between the different types of WMEs. Thus, it is not possible to distinguish the role of a specific WME to determine whether it models the task case, a mental calculation, a mental representation of the learning environment or a goal.

Astus

Three main features have been added to Astus's KBS in order to extend the classical representation of declarative knowledge in MTTs. First, declarative knowledge units can be of three types: concepts, relations and functions. Second, the manipulation of declarative knowledge units is restricted to a small number of task-independent operators whose semantics is known to the MTT. Third, task-specific manipulations of declarative knowledge units are done through a fixed interface that the MTT can interpret. Since the semantics of these features is known to the MTT, it has the capability to include them in the pedagogical interventions it generates.

The first type of declarative knowledge unit defined in Astus is a concept: a pedagogically relevant abstraction used to model the task case's objects. Concepts are defined by a set of features that are essential to the description of an object. Each feature has a type that is either primitive data (e.g. integers, decimals, strings) or a concept. For example, in our MTT for the insertion of elements into an AVL tree, the concept "Node" has only one feature: the node's content (an integer). The use of an explicit structure for concepts allows Astus to generate pedagogical content describing instances of concepts by referring to the values of their features.

Astus allows inheritance between concepts in order to further specify the type of an object. For example, the "Node" concept can be specialized as a "BinaryNode" which adds two new features to the object: a left and a right pointer to the node's sub-trees. Likewise, the "AVLNode" concept is a specialization of a "BinaryNode" containing a balance factor. Concept instantiation can be either asserted or inferred from the values of an object's features. For example, a "BinaryNode" object can be instantiated as a "Leaf" when both of its pointers are null. When generating pedagogical content, the

instantiation of a concept can be used to explain to the learner that an object is an interesting occurrence of a more general concept.

In addition to concepts, Astus's KBS allows authors to define relations and functions to model connections between objects. A function is defined by a list of arguments and an image. For example, in our AVL MTT, the function "parentOf" has one argument, a node, and its parent node as an image. Similarly, relations are defined by lists of places. For example, the relation "childOf" identifies whether a specific node is a child of another one. This relation has two places, two nodes, and is instantiated if the value of the first place is a child of the value of the second one. As for concept instantiation, functions and relations can be either asserted or inferred (see Appendix A). The separation of declarative knowledge units in different types (concept, function, relation) provides Astus with information about the role of each unit that can be integrated in the generated pedagogical content.

In order for procedures to interact with the content of the KB, Astus provides task-independent operators and an interface for task-specific manipulations. The task-independent operators are Boolean operators to express conditions and navigation operators to access the features of an object. For example, the navigation operator "→" can be used to obtain the value of a feature called "content" from an instance of the concept "Node" ("node→content"). Boolean operators can be used to check whether an object is an instance of a specific concept (*isA*), whether two variables refer to the same object (*same*), whether two or more objects are linked by a given function or relation (*exists*), and also to check the order of two objects (*greater*, *lesser*). These operators can be combined using logical operators (*and*, *or*, *not*). As the semantic of those operators is well known by Astus, pedagogical content can be generated to explain it. For example, Astus can generate messages describing the structure of logical conditions (Paquette et al. 2012a, b).

Task-specific manipulations of the KB are done through a query interface specified by Astus's KBS. This interface allows procedures to specify the information they want to retrieve from the KB. In particular, it can specify the type of declarative knowledge to retrieve by using the name of the concept, relation or function. Additionally, the query interface specifies whether all instances of the desired knowledge units are to be retrieved or whether a unique instance is sought. The use of this query interface allows Astus to generate pedagogical content explaining how the execution of a procedure manipulates declarative knowledge.

For example, in our AVL tree MTT, the query interface allows a procedure to retrieve all instances of the concept "Leaf" by using the instruction "all (Leaf)". Likewise, the query "unique (Leaf)" retrieves an instance of the concept "Leaf" and ensures that it is the only instance of this concept in the KB. Queries can also be used to retrieve instances via relations and functions. For functions, the query will retrieve the image of a function's instance. For example, "unique (parentOf, [node])" will retrieve the image of the function parentOf, a function taken from our AVL tree MTT taking a node as its argument and having a node (the parent of the argument) as its image, for the argument "node". A relation query will retrieve the objects associated with the free place (the one that is not constrained by the query). For example, for the relation childOf, a relation taken from our AVL tree MTT defined by two places (a child node and a parent node), "all (childOf, [_, parentNode]" will retrieve all the children of a node, whereas "unique (childOf, [childNode, _])" will retrieve the unique parent of a node. It is also possible to filter the results of a query according to a logical condition, using the keyword "where". For example, the query "all (Node, where { not

(same ($e→leftPointer, nullPointer)) })" will retrieve a set containing all of the instances of the concept "Node" and will then filter the result to find the ones for which the feature "leftPointer" is not the same as the variable "nullPointer".[7] The result of the query will thus be all of the nodes that have a child to the left.

The fact that all accesses to declarative knowledge are made through a fixed query interface allows Astus to interpret how procedures access and manipulate the content of the KB. Although Astus does not have the capability to interpret how the requested information is produced, it can generate pedagogical content explaining what knowledge to retrieve from the KB and what manipulation should be executed on that knowledge. For example, Astus can access the content of a query to explain to the learner that he/she needs to retrieve all the nodes that are leaves (task-independent access), but it cannot explain why a specific node is a leaf, since this instantiation is done through task-specific processes.

Next-Step Hints

Astus can improve the interventions it generates by providing information about how procedures access the KB and how they manipulate the retrieved objects. In particular, the hints generated can specify the objects that should be passed as arguments for each sub-goal of a procedure and detail the conditions specified by the procedure.

With this information, Astus can generate more complete hints. First, we can look at the *PInsert* procedure and its parent goal *GInsert*:[8]

```
Goal 'GInsert' eng-name 'insert an element' {
    param 'tree' type 'Tree' eng-name 'tree'
    param 'element' type 'int' eng-name 'element to insert'
}

Totally ordered sequence 'PInsert' achieves 'GInsert' {
    goal 'GInsertSubTree' using 'tree → root', 'element'
    goal 'GUpdate' using 'tree'
    goal 'GCheckBalance' using 'tree → root'
}
```

This procedure makes very limited use of Astus's interface for the manipulation of the KB objects. The only manipulation is to access the root of the tree in which the insertion is taking place. For this procedure, Astus can improve the generated hint by indicating which objects should be used as arguments for the sub-goals:

```
In order to insert an element, you need to do the following in
the given order:
1) insert in a sub-tree using the root of the tree and the
element to insert
2) update using the tree
3) check for imbalances using the root of the tree
```

[7] The "where" clause is evaluated for each node retrieved by the query, with the variable "$e" taking the value of specific nodes.

[8] The goal's definition shows an example of how the textual names used by the MTT to generate its messages are encoded by the author as part of the goals, concepts, functions and relations, using the keyword "eng-name".

For a conditional procedure such as *PInsertSubTree*, being able to interpret the access to the KB allows Astus to detail the conditions used to decide which sub-goals should be executed. These conditions are defined in the procedure:

```
Conditional 'PInsertSubTree' achieves 'GInsertSubTree' {
    if lesser('element', 'node → content')
        goal 'GInsertLeft' using 'node', 'element'
    if greater('element', 'node → content')
        goal 'GinsertRight' using 'node', 'element'
}
```

By analyzing the content of these conditions, the MTT can include them in its hints:

In order to *insert in a sub-tree*, you need to either *insert to the left*, if the *element to insert* is less than the *content* of the *node*, or *insert to the right*, if the *element to insert* is greater than the *content* of the *node*.

Finally, the hint generated for *PSubtract* can also be improved using the information contained in the procedure:

```
Ordered For-Each 'PSubtract' achieves 'GSubtract' {
    iterator 'column' from 'subtraction problem → columns'
    goal 'GSubtractColumn' using 'column'
}
```

In order to *subtract*, you need to *subtract a column* for each of the *columns* of the *subtraction problem*.

The previous examples make use of all the information available in the procedures to generate next-step hints. In order to continue improving the interventions generated by Astus, it needs to have access to information regarding the execution context for specific instances of the procedures. We must thus define how Astus traces the learners' steps and ensure that the tracing process can be easily interpreted.

Model Tracing in Astus

As for the design of Astus's KBS, we must ensure that the process used by Astus to trace the learners' steps provides information that is relevant to the automatic generation of pedagogical content. The main purpose of this information should be to allow the contextualization of the generated interventions to specific states of the task performing process. The result of the tracing process must thus provide information not only regarding the learners' currently possible steps, but also about those performed in the past and those that should be executed in the future. This is done by building a tree containing the previously executed procedural knowledge units, the units relevant to the possible next steps and those related to what should be executed in the future. The tracing process should also provide information regarding the declarative knowledge

used to perform the task. For this reason, it should implement mechanisms to store the result of the evaluation of the procedure's queries.

When tracing the learner's steps in Astus, the procedural knowledge contained in the procedural graph is instantiated to produce an episodic tree (Fig. 4), a task tree dynamically generated according to the state of the KB. To do this, starting from a task's main goal, the procedure associated with this goal is executed. Its queries and conditions are evaluated to determine the sub-goals that are instantiated and their arguments. Depending on the procedure's type, the sub-goals can be instantiated as currently executing (E) or waiting (W). The same process is applied recursively for each of the new executing goals.

For example, when executing a sequence procedure, depending on its order constraints, all its sub-goals might be instantiated in the executing state or some might be instantiated as waiting (*PConvertToFloatingPoint* in Fig. 4). In the case of conditional procedures, the execution of the procedure will instantiate only one sub-goal, depending on the evaluation of the conditions contained in the procedure (Fig. 5a). As a final example, a *for-each* procedure will evaluate the sequence of objects on which its sub-goal should be repeated and will instantiate one sub-goal for each of the objects from the sequence. Depending on whether or not the order is important for the iteration's execution, the episodic tree might contain only one executing goal, with the other marked as waiting (Fig. 5b).

Astus uses the episodic tree to trace the learner's steps. In this tree, the leaves are either primitive procedures or goals waiting to be expanded in the future. When the learner performs a step, it is compared to each of the tree's primitive procedures to find a match.[9] If no match is found, the step is marked as off-path and is considered incorrect. If a match is found, the step will either be attributed to a known error, if one of its parent procedures is marked as erroneous, or be considered a correct step. When the step is considered correct, the state of the primitive procedure associated with it is changed to completed / achieved (C) and the tree is updated accordingly.

Starting from the newly completed primitive procedure (one of the tree's leaves), its parent goal is also marked as achieved. Then, the individual goals and procedures are updated, going upward through the tree towards the root. A goal is marked as achieved if its child procedure is completed. Procedures are updated according to their types. A conditional procedure is marked as completed once the selected sub-goal is achieved. For sequences, sub-goals marked as waiting are executed if their order constraints are satisfied, and the procedure is completed when all of the sub-goals are achieved. For-each procedures are updated like sequences. A loop procedure is marked as completed when its iteration condition is met; otherwise a new instance of the sub-goal is created. A task case is considered performed once the tree's root goal is achieved.

Next-Step Hints

Using the content of the episodic tree allows Astus to contextualize the hints it generates according to the current state of the task case. This includes taking into account the results of conditional expressions when selecting a sub-goal and adapting hints for procedures that are partially completed.

[9] If more than one match is available for a step, this will cause an ambiguity that will be resolved in the following steps.

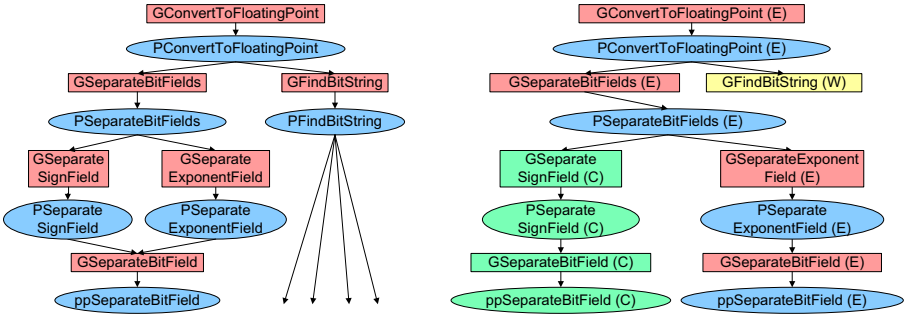

Fig. 4 Procedural graph from a floating point number conversion MTT (*left*) and its instantiation as an episodic tree (*right*). Units marked (E) are currently executing, (W) are waiting and (C) are completed / achieved

For a partially completed sequence procedure such as *PInsert*, the generated hint can be modified to focus on the next goal the learner should achieve. If the first sub-goal (*GInsertSubTree*) has already been successfully achieved (Fig. 6), the generated hint can notify the learner and explain what his/her next objective should be:

`You are currently doing insert an element, you already did insert in a sub tree, your next objective should be to update using the tree.`

For a conditional procedure (*GInsertSubTree*), only one of its sub-goals is instantiated in the episodic tree (Fig. 6). As such, it is not necessary for the generated hint to list all the possible conditions. The hint can focus the learner's attention on the currently executing goal:

`In order to insert in a sub-tree, you need to insert to the right since the element to insert is greater than the content of the node.`

Finally, for the procedure *PSubtract*, a for-each procedure, the MTT can offer additional information concerning the number of sub-goals left to achieve:

`In order to subtract, you need to subtract a column for each of the columns of the subtraction problem. You have already achieved this for the first 2 columns. You must repeat the process for the 2 remaining ones.`[10]

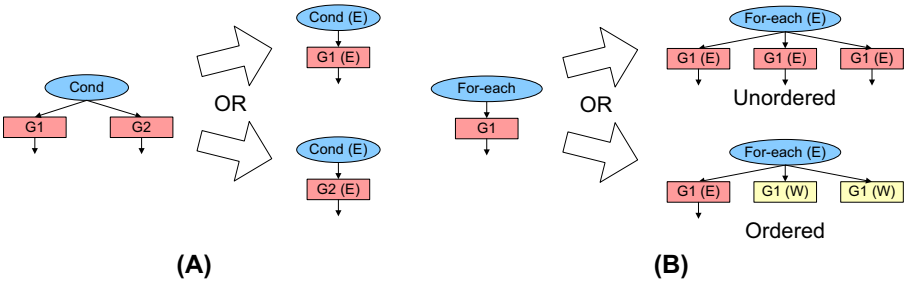

Fig. 5 Instantiation in an episodic tree of a conditional procedure (**a**) and a for-each procedure (**b**)

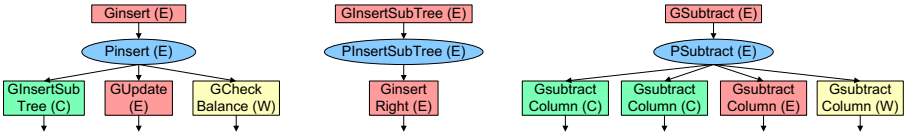

Fig. 6 Episodic trees associated with the three generated hints

The examples we provided to illustrate the generation of next-step hints are a small subset of the templates that can be used to generate hints. The instructions given for any of the procedures can be adjusted to specific learning situations. One hint may provide information regarding the procedure's queries whereas another does not, or a hint may or may not specify the sub-goals' arguments. Hints may be specific to the current state of the task case or more abstract. The choice of the template's content depends on the pedagogical strategy applied by the MTT.

The hints provided by Astus are currently generated using task-independent templates filled directly with task-specific content extracted from the model of the task. This method limits the readability of the generated hints. One step towards improving them would be to use natural language generation techniques.

Negative Feedback

A criticism that is often made about the use of negative feedback in ITSs is that, although it can be useful to provide this type of intervention, its high modeling efforts (Johnson 1990) usually outweigh its benefits (McKendree 1990; Corbett et al. 1997). The capability for an MTT framework to be able to provide negative feedback without requiring any knowledge engineering efforts would thus be valuable as the benefits of this intervention would now outweigh its cost.

The same features that allowed Astus to automatically generate next-step hints can also be used to automatically generate negative feedback on many of the learners' errors. To do this, we took inspiration from the Sierra theory (VanLehn 1990), and we designed a method by which Astus can diagnose some of the learners' errors by determining the origin of their off-path steps.

Sierra

Sierra is a theory to explain the origin of learners' procedural errors (VanLehn 1990). It proposes plausible cognitive processes that could lead learners to execute incorrect steps. According to this theory, errors can be observed when learners face impasses – situations in which their current knowledge of the task is insufficient to perform it – and try to repair them. The combination of an impasse and a repair strategy determines the learner's erroneous behavior.

Three types of repair strategy are proposed: no-op, back-up and barge-on. When the no-op strategy is applied, the learner ignores the goal he/she does not know how to achieve. The back-up strategy is very similar to no-op, but, instead of simply ignoring his/her current goal, the learner returns to a previously unfinished goal and resumes from that point. Finally, when the barge-on strategy is applied, the learner modifies his/her procedural knowledge in order to resolve the impasse.

The Sierra theory has been validated by creating a computational model of learning that includes the impasse and repair processes. This model was applied to the subtraction task and has successfully modeled the acquisition of multiple errors observed in learners' behavior. Despite this success, little effort has been made to incorporate elements of the Sierra theory into ITSs. Applying this theory in Astus would improve the resulting MTTs by allowing them to diagnose many of the learner's errors without the need for erroneous knowledge units, thus reducing the effort required to provide negative feedback on error similar to that offered by REACT's impasse-driven tutoring (Hill and Johnson 1995).

Error Diagnosis

We have previously shown (Paquette et al. 2012a, b) how Astus's KBS is compatible with the assumptions formulated in Sierra and how Astus can automatically disrupt procedural knowledge units to model erroneous behaviors analogous to those resulting from Sierra's impasse and repair process. In this section, we describe how we took inspiration from Sierra to design a method allowing the diagnosis of many of the learners' errors.

In Astus, the episodic tree contains all the steps that are predicted by the MTT's executable model of the task. When an off-path step is executed, Astus attempts to diagnose the learner's error by manipulating the content of the episodic tree to try to instantiate a step that matches the learner's off-path step. To generate a diagnosis, the MTT starts by searching the tree to identify all the complex procedures that might be the source of the learner's impasse. The result is an ordered list of procedures, with the first ones being those closest to the steps contained in the tree. Figure 7 illustrates the process of constructing this list. A depth-first search, exploring the branch that explains the learner's last step and the currently executing branches, is carried out, starting from the tree's root (*Goal1*). The branch that explains the last step (the second instance of *pp2*) is searched first, as its incorrect execution might explain errors in which the learner considers a procedure, such as *Loop*, as not completed even though in fact it is. Steps performed prior to that one are ignored. The complex procedures encountered during the search are added to the ordered list after their sub-goals have been completely searched.

For each procedure identified by this search, Astus interpolates the steps resulting from its incorrect execution. Figure 8 shows examples of interpolation applied to the episodic tree from Fig. 7. First, the learner might use a *barge-on* repair to modify the condition of the *Loop* procedure, thus repeating its sub-goal (*Goal6*). Likewise, he/she might modify the conditional procedure's (*Cond*) conditions (*barge-on*) and achieve the wrong sub-goal (*Goal8*). Finally, he/she might use the *back-up* repair to avoid having to achieve a current goal (*Goal3*) and instead try to achieve a sub-goal (*Goal4*) that is still waiting for the completion of a previous one.

When interpolating procedures, the MTT can also apply a process similar to the *barge-on* repair strategy to modify how the procedures access the KB. When querying the KB to access an instance of a function or a relation, the interpolation process can replace the accessed relation or function by a similar one (one that is defined over the same concepts). In our subtraction MTT, this can cause the function *differenceOf*, which takes two integers as its arguments and has one integer as its image, to be

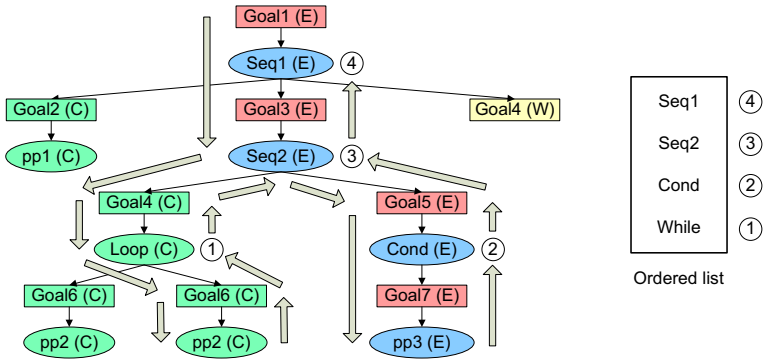

Fig. 7 Construction of an ordered list of all the procedures that might have been the source of the learner's impasse. The next correct step is *pp3*; *pp1* and both instances of *pp2* have already been performed

replaced by the similar function *sumOf*. Likewise, the interpolation can swap two arguments that are of the same type. For example, in our AVL MTT, the function *heightDifference* takes two sub-trees as argument and has one integer as its image (the difference in height for two sub-trees). When applying the argument swap interpolation, Astus will inverse the two sub-trees to produce behaviors such as 5 - 6=-1 instead of 6 - 5=1 (with 5 and 6 being the value of the heights for the two sub-trees passed as arguments). Astus can also apply a *barge-on* repair when using the access operator (→) and access the wrong feature of a concept. This can only be applied if both the original feature and the incorrect one are references to objects of the same type. In our AVL MTT, this repair can have the effect of accessing a node's left pointer (node→leftPointer) instead of its right pointer (node→rightPointer).

Astus uses the set of all interpolated steps to try to find a match for the learner's off-path step. If such a match is found, the branch of the episodic tree containing the interpolated step is used as a diagnosis of the learner's error. It is possible that multiple interpolated steps match the learner's off-path step. In our current implementation, such an ambiguity is resolved by using the interpolation that is closest to a leaf of the episodic tree as the diagnosis. Doing so decreases the chance of diagnosing as an error a mental step that the leaner has correctly performed, but increases the chance of diagnosing as an error a mental step that the learner has not performed yet. In the future, more sophisticated methods will be explored to improve the diagnosis process.

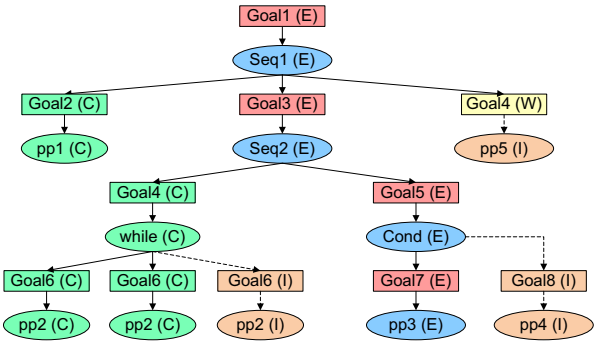

Fig. 8 Result of the interpolation. The interpolated goals are marked with an (I)

Once a diagnosis has been produced, the MTT can use it to react to the learner's step by providing negative feedback.

Table 1 contains a list of all the different types of error that Astus can diagnose. For each error, we provide 1) the knowledge unit that might have been incorrectly executed, 2) the possible sources of the error (repair types) and 3) a description of the observed error. We note that some types of error can be associated to many types of repairs. This is mainly the cases when the observed error can be associated to executing the wrong sub-goal. In those situations, it is possible to identify which incorrect sub-goal has been executed and which one should have been executed instead, but it is not possible to accurately identify what caused the incorrect sub-goal to be executed. The learner might possess an incorrect knowledge of which sub-goal to execute (*barge-on*) or he/she might be unable to execute the correct sub-goal in this context, thus leading him/her to avoid the correct sub-goal (*no-op* or *back-up*) and execute the incorrect one. For example, a *violated order constraint* error can be caused by an incorrect knowledge of the order constraint (*barge-on*), but could also be caused by the learner not knowing how to execute the correct sub-goal. In that situation, the learner might continue

Table 1 Summary of the statistical analysis for our first next-step hint experiment

Name	Knowledge Unit	Repair Type	Description
Path disruption	Path query	Barge-on	The learner accesses the wrong feature of an object with two or more features of the same concept.
Relation/function swap	Relation/Function query	Barge-on	The learner applies the incorrect relation or function. The incorrect relation or function must have places or parameters that are compatible with those of the correct relation or function.
Relation/function place/parameter swap	Relation/Function query	Barge-on	The learner inverts two of the relation's or function's places or parameters.
Incorrect condition	Conditional procedure	Barge-on/no-op/back-up	The learner executes a sub-goal even though its associated conditional expression is evaluated as false.
Violated order constraint	Sequence procedure	Barge-on/no-op/back-up	The learner executes a sub-goal even though it still has one or more unsatisfied order constraints.
Incorrect object order	For-each procedure	Barge-on/no-op/back-up	The learner executes the sub-goal on a specific object even though other objects should be processed before this one.
Incomplete for-each procedure	For-each procedure	Barge-on/no-op/back-up	The learner did not execute the procedure's sub-goal on all the relevant objects.
Loop early stop	Loop procedure	Barge-on	The learner stopped repeating the procedure's sub-goal too early.
Continued loop	Loop procedure	Barge-on	The learner repeats the procedure's sub-goal even though the end of the loop has been reached.
No loop	Loop procedure	Barge-on	The learner executed the procedure's sub-goal only once.

executing the sub-goals in the right order after ignoring the one that should have been executed (*no-op* or *back-up*).

Episodic Tree Search Optimization

Searching through the episodic tree to identify the source of a learner's off-path step is a time consuming process. Depending on the size of the current episodic tree and the size of the procedural graph, a very large number of interpolations are possible. For each procedure in the episodic tree, Astus needs to interpolate the result of executing any of the alternative sub-goals or any sub-goals that are currently waiting because of ordering constraints. This process is repeated recursively on each interpolated sub-goals until the primitive procedures are reached. In addition, the result of incorrectly executing any of the procedure's queries must be interpolated. Astus needs to generate a set of all the combination of possible query disruptions and then interpolate each of the procedure's sub-goals using all the possible disruptions. This process is repeated recursively for every interpolated procedure until the interpolation tree has been completely explored. In the worst case scenario, the number of interpolated branch can increase exponentially for every interpolated procedure. As such, it is critical to optimize the process of searching through the episodic tree by pruning interpolation branches when there is no possibility for them to lead to the learner's off-path step.

In order to prune the interpolation tree, we implemented two methods that rely on information that we pre-computed from the procedural graph. First, for every goal and procedure contained in the procedural graph, we compute a set of primitive procedures that are accessible from that knowledge unit. For example, in the episodic tree shown in Fig. 8, every primitive procedure from *pp1* to *pp5* can be accessed from *Goal 1* and *Seq 1*, whereas only *pp3* and *pp4* can be accessed from *Goal 5* and *Cond* and *pp2* is the only primitive procedure that can be accessed from *Goal 4* and *While*. Using this information, Astus can avoid interpolating goals or procedures if there is no possibility for them to result in executing a primitive procedure of the same type as the learner's off-path step. For instance, if, in the episodic tree presented in Fig. 8, the learner executed an off-path step of type *pp2*, Astus would avoid interpolating from *goal 4* and *goal 5* since *pp2* is not accessible from them.

The second optimization allows Astus to avoid disrupting queries that are not related to the arguments of the executed off-path step. In order to implement this optimization, an analysis of the data flow is executed on the procedural graph. This analysis allows Astus to associate to each query in the procedural graph a set of all the primitive procedures whose arguments can be affected by it. When interpolating the episodic tree to identify the source of the student's off-path step, Astus won't disrupt queries that won't affect the arguments of the off-path step.

Feedback Generation

When Astus diagnoses an off-path step as an error, it can generate negative feedback. To obtain this behavior, we use Astus's capability to generate interventions by examining the content of the task model. In this section, we give examples of negative feedback generated by our MTT for the insertion of elements in an AVL tree (Fig. 9).

Figure 9 illustrates a task case where the value 18 needs to be inserted into an existing AVL tree. The learner has reached the node containing the value 15 and must decide on which side of this node to continue the insertion process. Figure 10 shows part of the episodic tree for this specific state of the task case. The procedure *PInsertSubTree* is a conditional procedure that determines whether the new value should be inserted to the left or to the right of the current node. As the value 18 is greater than 15, the learner has to insert to the right and the goal *GInsertRight* is instantiated by *PInsertSubTree*.

The next step for the learner is to create a new node (*ppCreateNode*) to the right of the current one. If he/she executes the similar step of creating a new node to the left, Astus will try to diagnose this off-path step by interpolating new branches in the episodic tree. Starting from the procedure *PInsertSubTree* (Fig. 10), Astus will instantiate the goal *GInsertLeft* to interpolate the behavior of incorrectly executing *PInsertSubTree*. This interpolation will lead to an instance of the primitive procedure *ppCreateNode* that corresponds to the learner's step. Having found a match for the off-path step, Astus will use this interpolation as the diagnosis for the learner's error.

To produce feedback on errors, Astus defines error types, specified by the type of the incorrectly executed procedure and the repair that was applied to it (Table 1), and associates a feedback template to each of them. In the above example, the type of the error is an *incorrect condition* error on the *PInsertSubTree* procedure. As the learner did not fully understand when to insert to the left or to the right, he/she incorrectly chose to insert 18 to the left of the current node (*GInsertLeft*). Using this diagnosis, Astus can provide feedback by instantiating the corresponding template:

```
You should [correct sub-goal name] instead of [used sub-goal
name] since [condition for the correct goal].
```

Using the content taken from *PInsertSubTree* (page 11), the negative feedback can be instantiated (Fig. 9 shows how this feedback is communicated to the learner):

```
You should insert to the right rather than insert to the left
since the element to insert is greater than the content for
node.
```

A second example of an error can be observed in our AVL MTT when the learner calculates a node's balance factor. This is done by subtracting the height of the right sub-tree from the height of the left sub-tree (left height - right height). A common error occurs when the learner does not remember which height to subtract from the other and does the opposite subtraction (right height - left height). This type of error is referred to as a *function parameter swap* in Table 1.

When interpolating the episodic tree to diagnose this error, Astus will encounter the procedure *PUpdateBalanceFactor*. This procedure queries the KB for an instance of the function *heightDifference* that has two pointers to sub-trees as arguments and the difference of their heights as image. While interpolating, Astus will try to incorrectly execute this query by inverting the function's two arguments. This will result in the error of subtracting right height - left height. Using this diagnosis, Astus can instantiate

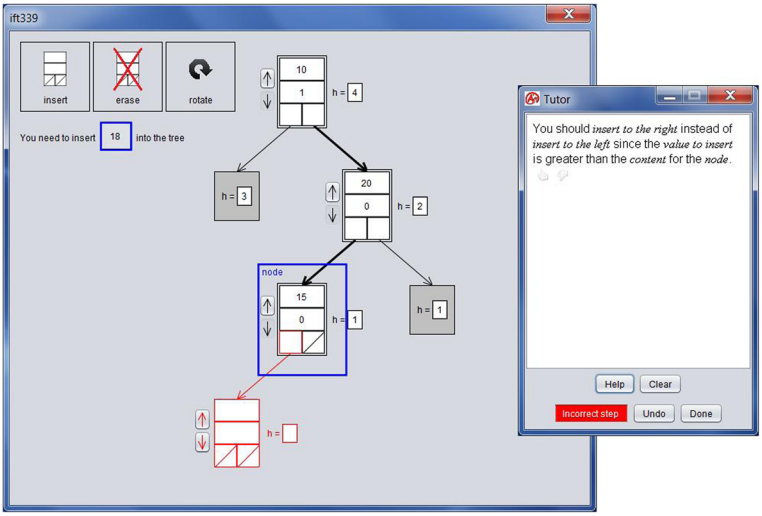

Fig. 9 Our MTT for the insertion of elements into an AVL tree. The learner has performed an off-path step that was diagnosed by the MTT

a feedback template:

`While trying to [goal name], you have inverted the [pair of arguments] for the [function name]. You should have used [correct arguments].`

The instantiation of this template will result in the following feedback:

`While trying to update the balance factor, you have inverted the first term and the second term for the height difference. You should have used the left pointer as the first term and the right pointer as the second term.`

Astus can adapt the templates used to generate its negative feedback according to the desired pedagogical strategy. An experiment by McKendree (1990) evaluated the effectiveness of goal-oriented feedback on error and feedback explaining the reason

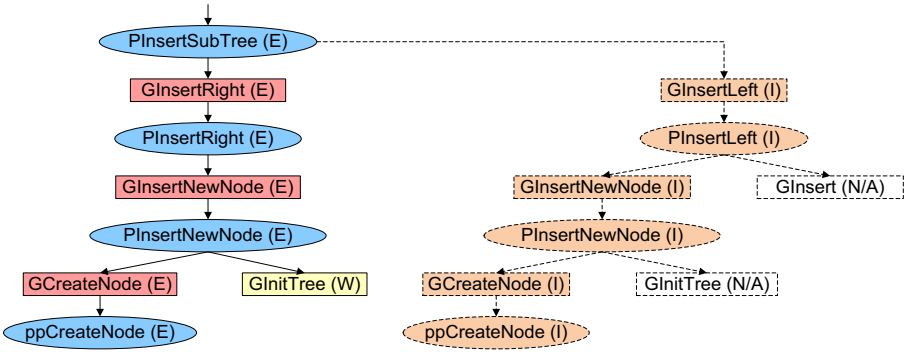

Fig. 10 Part of the episodic tree for the insertion of 18 and the interpolated branch for the off-path diagnosis

for an error. According to this experiment, goal-oriented feedback is the most useful for the learner as it helps him/her correct his/her mistake, and it also seems to have beneficial effects on subsequent encounters with the problematic knowledge. It can be combined with feedback explaining the reason for the error, a type of feedback that can improve later performance, but does not indicate how to correct the error. The feedback we provide uses a combination of pointing to the correct goal and explaining the cause of the error.

The format we chose is also advantageous in situations where Astus's diagnosis does not accurately identify the source of the learner's error. In those situations, we expect learners will still benefit from the MTT's interventions since Astus provides goal-oriented feedback that not only explains the error, but also indicates what should be done next.

Empirical Studies

In this section, we present the results of five small-scale experiments conducted with students from the computer science department at the Université de Sherbrooke. The purpose of the experiments was to obtain an initial assessment of the effectiveness of the interventions generated by Astus.

Next-Step Hints

We conducted a preliminary evaluation of the next-step hints generated by Astus in the context of a floating point number conversion MTT that was used in a system programming course at the University of Sherbrooke. This section presents a summary of three experiments.

First Experiment

The objective of our first experiment (Paquette et al. 2011) was to compare learning gains and student assessment for next-step hints generated by Astus to those for hints authored by a teacher. In this experiment, 38 students were randomly divided into two groups: the first group received teacher-authored hints[10] (TH) and the second received framework-generated hints (FH). Both groups received flag feedback.

Of the 38 students, four did not show up for the experiment and two did not go to the correct classroom and thus did not complete the study under the planned condition. In total, 19 students completed the study in the TH group and 15 in the FH condition.

The experiment was conducted in one session of 1 h and 50 min separated into five activities: explanation of how to use the MTT (10 min), pretest (20 min), using the MTT (50 min), posttest (20 min), and appreciation survey (10 min). When using the MTT, students were presented with 10 task cases they had to perform in a specific order. Students were asked to perform as many cases as they could in the allowed time. The intent was to provide more cases than students would perform in 50 min to control the time spent using our MTT rather than the number of cases.

[10] The teacher-authored hints were encoded as text templates similar to those used by Cognitive Tutors.

The pretest and posttest each consisted of seven questions evaluating the knowledge practiced using the MTT. The two tests had the exact same structure, but with minor differences in the numbers used. This was done to prevent students from answering the posttest using their memory of the pretest while still evaluating the same knowledge. The tests were graded out of a total of 20 points.

An appreciation survey was used to evaluate the students' opinion regarding the next-step hints. Students were first asked to rate, on a scale from 1 to 4, their appreciation of five characteristics of the hints: whether the hints helped them complete the case, whether the hints helped them understand the task, whether the hints were easy to understand, whether the hints hindered their understanding and an overall appreciation score for the hints. The average of the five ratings was used as an appreciation score, where a higher score indicates a higher appreciation. Then, students were asked to choose which hints they preferred from four pairs. The two hints from each pair referred to the same situation, one was framework generated while the other one was teacher authored. Students were not informed of the origin of the two hints they were asked to rate. The average of four ratings, ranked from 0 to 5, was used as a preference score. A score of 0 indicates a strong preference for the first hint while a score of 4 indicates a strong preference for the second hint. The pairs were created by randomly selecting one hint from each of the generated hint types: loop, conditional, sequence with one sub-goal (sequence 1) and sequence with more than one sub-goal (sequence N).

No data was collected regarding the number of help request for each student since logging of the students' actions had not been implemented in Astus yet.

Table 2 contains the results of four statistical tests that were conducted on the students' pretest and posttest scores. First, we used a two-sample *t*-test to compare the pretest scores for the two conditions (left of Fig. 11). No statistically significant differences were found between the students' pretest scores for the FH condition (M=8.77; SD=6.14) and the TH condition (M=11.11; SD=4.71). The test had low statistical power (22.67 %). The medium effect size (d=0.43) seems to indicate that a significant difference would have been found with a more powerful test.

Paired t-tests were used to test the learning gains between the pretests and posttests for both conditions. Both the FH and the TH conditions showed significant gains. The FH condition's pretest (M=8.77; SD=6.14) and posttest (M=13.13; SD=4.56) scores indicate a large effect size (d= *0.79)*, and so do the TH condition's pretest (M=11.11; SD=4.71) and posttest (M=14.95; SD=4.15) scores (d=0.86).

We conducted an analysis of covariance (ANCOVA), with the pretest scores as the covariate, in order to compare the learning gains of each group by comparing their posttest scores. Using an ANCOVA analysis allows us to compare the posttest scores while controlling for differences in the students' pretest scores across the two groups. The ANCOVA did not show significant differences between the posttest scores for the FH (M_{aj}=13.832) and TH (M_{aj}=14.396) conditions. The power of this test is low, but the very small effect size (η^2_p=0.0075) and the adjusted means do not indicate any important difference between the two conditions.

A two-sample *t*-test showed no significant differences between the students' hint appreciation scores (t (28)=0.358, p=0.723) for the FH (M=3.24; SD=0.54) and the TH (M=3.17; SD=0.46) conditions. The power of the test is low (6.40 %) and its effect size is small (d=0.13). One student's data was rejected due to invalid answers.

Table 2 Summary of the statistical analysis for our first next-step hint experiment

	Stat	*p*	Effect size	Power
Pretest scores	t (32)=−1.258	0.218	d=0.43	22.67 %
Learning gain (FH)	t (14)=−3.485	0.004**	d=0.79	89.65 %
Learning gain (TH)	t (18)=−4.926	< 0.001***	d=0.86	97.49 %
ANCOVA	F (1, 31)=0.234	0.632	η^2_p=0.0075	7.56 %

Students were asked to give their comparative appreciation for four pairs of hints (framework vs. teacher). One-sample t-tests were used to evaluate whether the answers were different from a normal distribution centered at the score 2 (meaning that neither of the two hints were preferred). Table 3 and Fig. 11 (right) presents the results of those analyses. For the loop procedure, the teacher's hint was significantly more appreciated. The generated hints were significantly more appreciated for the conditional and for the multiple sub-goals sequence procedure. Finally, for the one sub-goal sequence, no significant difference was found. Data from four students were rejected due to invalid answers.

Second Experiment

Our first experiment did not find any significant difference between framework-generated and teacher-authored hints. However, because this experiment did not include a control group where no next-step hints were provided to the students, we could not determine whether the students' learning gains were a result of the hints they received. The observed gains could simply be caused by the activity of performing task cases using an MTT. To determine whether next-step hints were helpful, we conducted a second experiment (Paquette et al. 2012a, b) comparing the learning gains of an MTT without next-step hints (flag feedback only) to those of an MTT that also provided framework-generated hints. A group of 32 students was randomly divided into two sub-groups: 16 students used an MTT that provided no hints (NH) and 16 students used an MTT with framework-generated hints (WH). Both versions of the MTT provided flag feedback.

Our second experiment reused the same design as our first one, except students used the MTT for 40 min instead of 50. This was done in order to reduce the number of

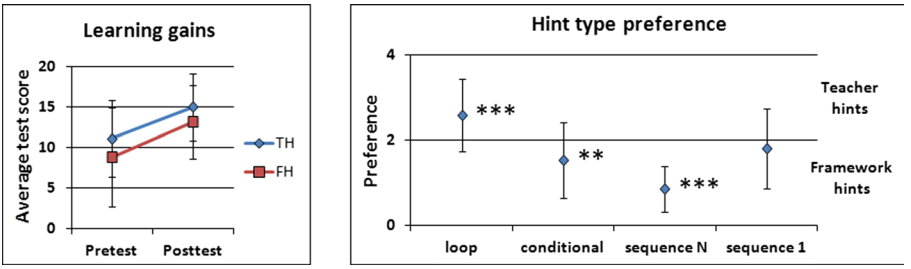

Fig. 11 The results of our first next-step hint experiment. The '*' character indicates statistical significance (** for p<0.01 and *** for p<0.001)

Table 3 Results for the comparison of hint appreciation

	Stat	*M*	*SD*	*p*	Preferred hint	Effect size	Power
Loop	t (31)=3.913	2.59	0.86	< 0.001***	teacher	d=0.69	96.56 %
Conditional	t (29)=2.904	2.47	0.88	0.007**	generated	d=0.53	80.11 %
Sequence N	t (30)=11.998	3.15	0.53	< 0.001***	generated	d=2.16	100.0 %
Sequence 1	t (30)=−1.147	1.81	0.94	0.260	neither	d=0.21	19.89 %

students who were able to perform all 10 task cases. The same pretest and posttests were reused, but this time half the students received the first test as pretest and the second as posttest while the order was reversed for the other half. Although both tests are very similar, this was done to counter balance any effect the difficulty of the tests might have on measured learning gains. The results of our second experiment are summarized in Table 4 and shown in Fig. 12 (left).

A two-sample *t*-test showed no statistically significant differences between the students' pretest scores for the NH (M=8.13; SD=2.34) and the WH (M=8.82; SD=4.16) conditions. Although no significant differences were found, the standard deviation of the WH condition is much higher than the one for the NH condition.

The learning gains between the pretests and posttests were validated using paired t-tests. Both conditions showed significant gains. The NH condition's pretest (M=8.13; SD=2.34) and posttest (M=12.09; SD=3.30) scores indicate a large effect size (d=1.35), and so do the WH condition's pretest (M=8.81; SD=4.16) and posttest (M=14.44; SD=4.06) scores (d=1.37). The effect sizes are very similar even though the mean learning gain is higher for the WH (5.63) condition when compared to the NH (3.96) condition. This lack of difference results from the difference in standard deviations between the two conditions. The effect size for the WH condition would have been higher if its standard deviations were closer to those of the NH condition.

A one-tailed analysis of covariance (ANCOVA), with the pretest scores as the covariate, showed a significant difference between the posttest scores for the NH (M_{aj}=12.31) and WH (M_{aj}=14.22) conditions. The effect size is η^2_p=0.091.

Third Experiment

To further validate the results of our second experiment, we conducted a third one using the same experimental design as our second experiment. In this experiment, 33 students were randomly assigned to two groups: 16 for the NH condition and 17 for the WH one. The results of our third experiment are summarized in Table 5 and shown in Fig. 12 (right).

A two-sample *t*-test showed no statistically significant differences between the students' pretest scores for the NH (M=10.97; SD=4.28) and the WH (M=11.50; SD=5.24) conditions.

The learning gains between the pretests and posttests were validated using paired t-tests. Both conditions showed significant gains. The NH condition's pretest (M=10.97; SD=4.28) and posttest (M=13.19; SD=4.25) scores indicate a medium effect size (d=0.52), and the WH condition's pretest (M=11.50; SD=5.24) and posttest (M=15.26;

Table 4 Summary of the statistical analysis for our second next-step hint experiment

	Stat	*p*	Effect size	Power
Pretest scores	t (23.629)=0.576	0.570	d=0.20	8.50 %
Learning gain (NH)	t (15)=6.213	< 0.001***	d=1.35	99.89 %
Learning gain (WH)	t (15)=5.550	< 0.001***	d=1.37	99.91 %
ANCOVA	F (1, 29)=3.057	0.046*	η^2_p=0.091	39.40 %

SD=4.43) scores indicate a large effect size (d=0.77). The higher effect size for the WH condition suggests it yielded higher learning gains than the NH condition.

A one-tailed analysis of covariance (ANCOVA), with the pretest scores as the covariate, showed no significant differences between the posttest scores for the NH (M_{aj}=13.83) and WH (M_{aj}=15.26) conditions. The effect size is η^2_p=0.086.

Interpretation of the Results

The results from our first experiment suggest that, for our floating point number conversion MTT, the nature of the next-step hints (framework-generated or teacher-authored) did not have a significant impact on learning gains. This is supported by the fact that both conditions had significantly higher posttest scores, with similar effect sizes (d=0.79 and d=0.86). Additionally, the ANCOVA comparing the posttest scores of the two conditions, using the pretest scores as a covariate, did not show a significant difference and had a very low effect size (η^2_p=0.0075). The similar slopes for the two conditions in the graphical visualization of the students' scores also suggest similar learning gains (left of Fig. 11).

The second experiment was conducted to ensure that the use of next-step hints had a significant effect on learning gains and that the learning gains observed in our first experiment were not merely a consequence of using an MTT to perform task cases. Both conditions had significant learning gains, with very similar effect size (d=1.35 and d=1.37) even though the mean gain between pretest and posttest was higher for the WH condition (M=5.63) than for the NH condition (M=3.93). The similar effect size might be due to the higher standard deviation for WH's pretest scores (SD=4.16) than

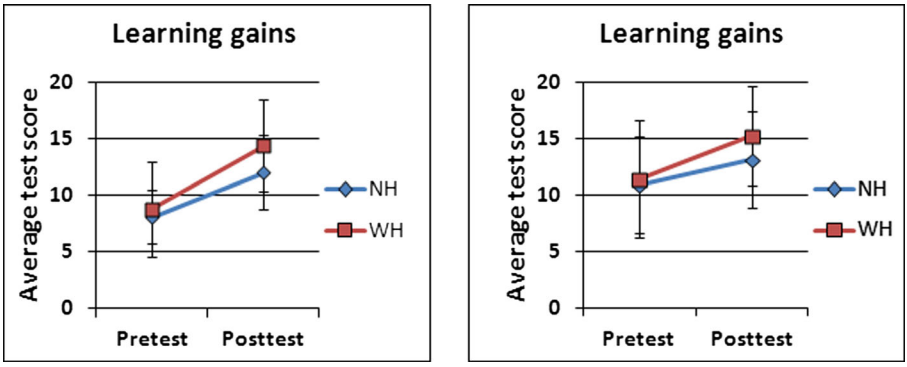

Fig 12 The results of our second (*left*) and third (*right*) next-step hint experiments

Table 5 Summary of the statistical analysis for our third next-step hint experiment

	Stat	*p*	Effect size	Power
Pretest scores	t (31)=0.318	0.753	d=0.11	61.00 %
Learning gain (NH)	t (15)=2.970	0.010**	d=0.52	49.46 %
Learning gain (WH)	t (16)=4.401	< 0.001***	d=0.77	86.64 %
ANCOVA	F (1, 30)=2.818	0.052	η^2_p=0.086	36.40 %

for NH's pretest scores (SD=2.34). The effect size for WH would have been higher if its standard deviation had been closer to that of the NH condition. The result of the one-tailed ANCOVA was significant, with an effect size of η^2_p=0.091, which seems to indicate that the framework-generated hints provided to the students did improve learning gains. Finally, the steeper slope for the WH condition in Fig. 12 (left) also suggests that the next-step hints were beneficial.

We conducted a third experiment to try to reproduce the results of our second experiment. Both the NH and the WH conditions showed significant learning gains. Although the higher effect size for WH (d=0.77) seems to indicate that the hints had a positive impact on the students' learning gains, the one-tailed ANCOVA was not statistically significant, but its effect size (η^2_p=0.086) was very similar to that obtained in our second experiment (η^2_p=0.091) and the results of the test were close to a statistically significant difference (p=0.052). These two facts suggest that a more powerful test could have found a significant difference. Finally, the steeper slope for the WH condition in Fig. 12 (right) suggests greater learning gains when receiving next-step hints.

Overall, the results of our three initial experiments seem to suggest that our framework-generated hints have a positive impact on learning gains, but their low statistical power means that additional studies will be required to validate this result. Our validation would benefit from additional larger scale experiments that included more students and where the students would use the MTT for longer period of time. We would also be interested in improving our empirical validation by reproducing similar results for different tasks.

Negative Feedback

We conducted two experiments in a data structure course, using an MTT for the insertion of elements into an AVL tree (Fig. 9). Our aim with the first experiment was to evaluate whether the diagnoses produced by our MTT were accurate and whether the provided negative feedback helped the learner. The second experiment was designed to analyze logs of the students' interactions with the MTT.

First Experiment

Our MTT was used by 45 students randomly divided into two groups. The first 23 students used an MTT that provided both negative feedback on error and flag feedback (WF condition), whereas the remaining 22 students received only flag feedback (NF

condition). The students were first asked to do a pretest (20 min), then to use the MTT (30 min) and finally to do a posttest (20 min). When using the MTT, students were presented with 10 task cases they had to perform in a specific order. Students were asked to perform as many cases as they could in the allowed time.

Two versions of the test, each consisting of 13 questions, were used as pretest and posttest. The two tests had the same structure, but with minor differences in the provided AVL trees. The tests were graded out of a total of 40 points. Half the students received the first version as pretest and the second as posttest, whereas the order was reversed for the other half.

At the end of the experiment, students from the WF condition were asked to answer a series of questions regarding their perception of the quality of the negative feedback they received. Out of 23 students, 22 said they had received feedback while using the MTT[11]. Of those 22, 2 answered that they received very few feedback interventions (1–3), 11 received few (4–9), 8 often received feedback (9–15) and 1 very often (16+). Students were also asked to indicate whether they agreed or disagreed with five statements regarding the feedback (Table 6).

A two-sample *t*-test showed no statistically significant differences between the students' pretest scores for the WF (M=20.02; SD=6.32) and NF (M=23.41; SD=8.67) conditions. The low statistical power (31.06 %) and the medium effect size (d=0.45) for this test seem to indicate that a significant difference could have been found with a more powerful test.

Paired t-tests were used to assess the learning gains between pretest and posttest. Both the WF and NF conditions showed significant gains. The WF condition pretest (M=20.02; SD=6.32) and posttest (M=27.61; SD=7.44) scores indicate a large effect size (d=1.09) and the NF condition pretest (M=23.41; SD=8.67) and posttest (M=28.27; SD=9.22) scores indicate a medium effect size (d=0.54). The results are illustrated in Fig. 13.

A one-tailed analysis of covariance (ANCOVA), with the pretest scores as the covariate, did not show a significant difference between the posttest scores for conditions WF (M_{aj}=29.066) and NF (M_{aj}=26.749). The power of this test is low (44.25 %), with an effect size of η^2_p=0.049. Table 7 presents a summary of the analysis of the learning gains.

Second Experiment

We conducted a second experiment designed to retrieve logs of the students' interactions with the MTT and compile statistics relevant to our diagnosis of errors and the negative feedback provided by the MTT. We randomly divided a class of 34 students into two groups: 18 students used an MTT providing negative feedback on error (WF) and 16 used an MTT that provided only flag feedback (NF). Although the students did a pretest and a posttest, we did not use this data as the pretest scores were too strong (more than 25 % of the students had perfect or almost perfect scores) and did not leave any room for improvement. This might be due to the fact that the students had a related assignment that was due the week after the experiment.

[11] This data was collected through survey since logging of the students' actions had not been implemented in Astus yet.

Table 6 Students' responses to the feedback assessment questionnaire

	Strongly disagree	Disagree	Agree	Strongly agree
1. Their content corresponded to my error	0	2	13	7
2. They helped me perform the task case	0	3	8	11
3. They helped me learn how to perform the task	1	6	12	3
4. They were easy to understand	1	8	8	5
5. They hindered my understanding of the task	15	4	3	0

We analyzed logs from 30 students: 18 from the WF condition and 12 from the NF condition (the logs from 4 students were lost due to issues uploading their data to our servers). In total, the students performed 192 task cases (113 for WF and 79 for NF) and executed 1,120 off-path steps (576 for WF and 544 for NF) on 585 different instances of errors.[12] Our MTT was able to diagnose as errors, and provide negative feedback for, 381 out of the 576 off-path steps executed by students from the WF condition (66.15 %).

A two-sample *t*-test (t (28)=−1,719, p=0.097) showed a marginal difference in the number of off-path steps per student for the WF (M=32.00; SD=17,38) and the NF (M =45.33; SD=25.21) conditions. The statistical power of this test was low (34.21 %), and effect size was medium (d=0.62).

We examined the number of correct steps executed by a student after an off-path step. For the WF condition, the students corrected their errors on the next attempt 65.62 % of the times when they received negative feedback on their error and 38.97 % of the times when they did not receive negative feedback. Overall, the chance of correcting an error, whether or not the MTT provided negative feedback, was 56.60 %. For the NF condition, the students received no negative feedback and corrected their error on the next attempt 47.61 % of the time.

Interpretation of the Results

The two experiments we conducted had for objectives to assess the quality of the negative feedback generated by Astus and its effect on learning. In our first experiment, we asked the students to evaluate the quality of the negative feedback they received (Table 6). Almost all of the students who received feedback (20 out of 22) agreed that the feedback they received accurately identified their errors, but this result is attenuated by the fact that students' subjective report of the accuracy of feedback can be unreliable since students tend to respond as they expect the experimenters want them to. Additional experiments will be required to formally evaluate the accuracy of Astus's diagnoses.

In addition, the students' evaluation of the negative feedback they received suggested many improvements that could be made to increase the efficiency of our feedback. Although most students (19 out of 22) answered that the feedback helped them perform the task cases, about a third (7 out of 22) also answered that the feedback

[12] We consider consecutive off-path steps as being caused by the same error instance.

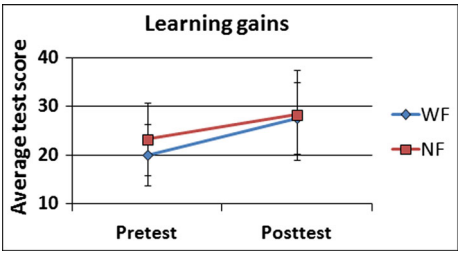

Fig 13 The results of pretest and posttest results for our negative feedback experiment

did not help them learn (Table 6). This might be due to the fact that they had difficulty understanding the feedback (9 out of 22). With this in mind, it would be necessary for us to improve the readability of the feedback generated by our framework.

The pretest and posttest scores of our first experiment were used to evaluate whether the learners' performance was improved by negative feedback on error. Although the ANCOVA did not show a statistically significant difference between the two conditions, the t-tests evaluating the learning gains yielded a higher effect size (d=1.09) for the WF condition than for the NF condition (d= 0.54) (Table 7). This is illustrated by the steeper slope for condition WF on the graphical representation of the learners' scores (Fig. C13).

The higher learning gains for the WF condition might be explained by its lower average score on the pretest, but the effect size for the ANCOVA, which controls for pretest scores, was η^2_p=0.049. Additional experiments will be required to assess whether the difference in learning gain between the two conditions is meaningful. The observed effect size (η^2_p=0.049) is about half that obtained for similar experiments measuring the learning gains of next-step hints (η^2_p=0.091 and η^2_p=0.086). This is consistent with what we expected, as next-step hints are on-demand help, whereas negative feedback is targeted at specific learning situations and is only available on errors for which a diagnosis is possible. As such, our negative feedback can only help learners when there is a plausible explanation for their errors, and is mainly targeted at learners who have a minimal understanding of how to perform the task. It will not help learners who have greater difficulty since many of their errors will be the result of trial and error.

Although the ANCOVA's effect size is consistent with the expected value, a power analysis indicated that 122 students would have been required for the test to have a statistical power of 80 %. This is much higher than the number of students per course in the computer science department at Université de Sherbrooke, and it considerably reduced our chances of finding a statistically significant difference.

Table 7 Summary of the statistical analysis for our negative feedback experiment

	Stat	p	Effect size	Power
Pretest scores	t (43)=−1.503	0.140	d=0.45	31.06 %
Learning gain (WF)	t (22)=−7.400	< 0.001***	d=1.09	99.88 %
Learning gain (NF)	t (21)=−4.252	< 0.001***	d =0.54	67.98 %
ANCOVA	F (1, 43)=2.187	0.074	η^2_p=0.049	44.25 %

Logs of the students' interactions with our MTT in our second experiment showed that our MTT was able to diagnose about two-thirds (66.15 %) of the students' off-path steps. As such, our MTT was able to provide negative feedback for many of the learners' off-path steps, but this number might include diagnosis on minor slips or errors related to using the learning environment. In those situations, the feedback will not be helpful to the learner.

We examined how often students were able to correct an off-path step on their next attempt. For the WF condition, we observed a higher percentage of correct steps following an off-path step for which negative feedback was provided (65.62 %) than for one where only flag feedback was provided (38.97 %). This difference could be due to the fact that slips, errors related to the learning environment and errors that are easy to correct are often diagnosed by Astus. To verify whether this is the case, we compared the total number of correct attempts following an off-path step across both conditions. If the negative feedback did not contribute to learners being able to correct their errors, the percentage of correct steps following an off-path step should be similar for both conditions. We observed a higher percentage of correct steps for the WF condition (56.60 %) than for the NF condition (47.61 %). This suggests that negative feedback did help learners correct some of their error more quickly.

Overall, the use of negative feedback seems to have had a positive impact on the students' use of our MTT. The number of off-path steps per student was reduced in the WF condition and the students in the WF condition seem to have been able to correct their errors more quickly. Additional experiments will be required to further validate those results.

Discussion and Conclusion

The idea of building ITSs with sophisticated pedagogical module having the capability to automatically generate pedagogical interventions has been proposed by researchers in the past (Wenger 1987, Rickel 1988) and many systems implemented this idea to varying extents. Steve (Rickel and Johnson 1999) has the capability to generate explanations for the tutoring of physical tasks, REACT (Hill and Johnson 1993) provides negative feedback regarding the learner's impasse and GIL (Reiser et al. 1992) is able to generate next-step hints and negative feedback for LISP programming.

Astus's main contributions reside in 1) the integration of the functionalities of those past systems in an MTT framework that can automatically generate interventions regardless of the task and 2) an initial empirical validation of the effectiveness of Astus's pedagogical interventions. This was obtained thanks to the design of a knowledge representation approach where the model of the task in a format closer to that of the teacher's instructions. This approach allows Astus to both trace the learners' steps and automatically generate pedagogical content that can be used to provide interventions.

In this paper, we have shown how three main features of Astus's KBS contribute to providing information that can be used to generate pedagogical content. First, Astus's hierarchical procedural knowledge provides information regarding the location of specific knowledge units in the global process of performing the task. Second, the semantic of each type of procedural knowledge unit is explicit, thus allowing Astus to understand their behavior and infer the outcomes of executing a specific knowledge unit. Third, declarative knowledge units have a rich semantic that provides Astus with information about how

procedural knowledge manipulates declarative knowledge. In addition, the combination of those three features allows Astus to implement a tracing process allowing it to contextualize the pedagogical interventions it generates to specific states of the task case.

We validated the usefulness, for the purpose of generating pedagogical content, of the information made available by Astus's KBS by implementing the automatic generation of next-step hints. We showed how each of this system's main features allows it to generate increasingly more complete hints. We presented the results of three small-scale experiment that seem to suggest that the next-step hints generated by Astus can be effective and we briefly discussed how additional experiments will allow us to further validate this result.

We also showed that Astus has the capability to automatically generate negative feedback on error by manipulating the episodic tree used when tracing the learners' steps. Astus achieves this behavior by interpolating incorrect executions of the task's procedural knowledge to determine whether they provide plausible explanations for the learner's off-path steps.

The use of negative feedback in MTTs has been debated in the past because of its high knowledge engineering cost required to produce this type of intervention (Corbett et al. 1997). On the other hand, Astus's capability to diagnose off path steps and generate negative feedback does not require any additional knowledge engineering from the MTT's author. Initial results of the evaluation of the effectiveness of Astus's negative feedback are encouraging, although additional larger scale experiments will be required to validate them.

Until now, the next-step hints and the negative feedback generated by Astus have only been used by learners in the context of computer science undergraduate courses. We are currently working towards the development of two additional MTTs. The first one is for the trauma nursing core course where learners need to correctly asses and treat a patient arriving at the trauma ward. The second one is for an undergraduate genetics course where the learners need to perform a DNA restriction analysis.

Preliminary observations of the hints and negative feedback generated by Astus for those new MTTs suggest that the method we use to generate interventions generalizes to those tasks. As these two MTT require improvements to Astus's KBS (description of those improvements is out of the scope of this paper), the generation of interventions must be adapted to include the KBS's new features. For the trauma nursing core course, improvements have been made to provide the procedural knowledge with more sophisticated access to the KB. For the genetics course, a heuristic-based selection procedure type was added. Both of those improvements and others that are needed to improve the range of domain supported by Astus's KBS are not yet supported by the intervention generation method described in this paper.

Promising future work related to the generation of pedagogical interventions includes their adaptation to specific learners and learning situations. Since the pedagogical content of an intervention is dynamically generated while the learner performs a task case, it would be possible to modify the generated content depending on the current learning situation; something that is much harder to achieve when the intervention's content is provided by a teacher before the execution of the MTT.

We are also interested in studying how Astus's KBS can be used to generate additional types of interventions such as worked examples (McLaren et al. 2008; McLaren and Isotani 2011), self-explanation prompts (Aleven and Koedinger 2008; Conati and VanLehn 1999) and analogies with other task cases (Ohlsson 2008). The

generation of such interventions by Astus would aid in investigating their effectiveness for the tutoring of various tasks and their interaction when combined in sophisticated pedagogical strategies.

Although recent research have lost their focus on the model-tracing approach to intelligent tutoring, we believe that current research have not reached the upper limit of what can be attained using MTTs. The research presented in this paper aims to explore how the sophistication of an MTT's pedagogical module can be improved by allowing MTTs to automatically generate pedagogical content. We hope that, as we discover new ways to expand the pedagogical behaviors of MTTs, the method we develop will facilitate the discovery of similar approaches for other intelligent tutoring paradigms that address different limitations of MTTs such as their high development cost.

Acknowledgment This research was supported by a NSERC doctoral fellowship. We would like to thank Mikaël Fortin for providing the teacher-authored next-step hints that were used with our floating point number conversion MTT, Richard St-Denis for allowing us to use this MTT in his class and Jean Goulet for allowing us to use the AVL tree MTT in his class. We would also like to thank the anonymous reviewers, Paul Moncuquet and Ryan S. Baker for their comments and suggestions.

Appendix A – Developing A MTT Using Astus

Our aim is to require development efforts on par with building a Cognitive Tutor using CTAT (Aleven et al. 2006; Aleven et al. 2009a, b). To this end, we assume that the authors have programming skills and we offer them a modeling language[13] in an Eclipse-based developpement environment. In addition, the Astus framework offers multiple tools allowing authors to vizualize the task's declarative and procedural knowledge at "loadtime" and to inspect the KB and the episodic tree at runtime.

Although tools have been developed to ease Astus's MTT development process, evaluating the efforts in a formal context has not been yet part of our research. Our research priority was to ensure that the KBS we developed allows the generation of a range of pedagogical content and pedagogical interventions.

Developing a MTT in Astus is usually done in four main phases. First, the author defines the task's declarative knowledge. Second, the learning environment is designed and implemented. Third, the procedural graph is defined. Fourth and final, a set of task cases is created. This appendix presents a brief overview of each of those phases.

Defining Declarative Knowledge

The development process starts with the modeling of the concepts, relations and functions relevant to the task. The concepts are the first knowledge units to be defined since they describe the objects the learner will perceive and manipulate when performing a task in the learning environment. As such, other knowledge units such as goals, procedures, functions and relations will manipulate concepts. For example, in the subtraction task, a *column* would be a concept having three attributes: a top *term*, a

[13] Implemented as an embedded domain-specific language (EDSL). The actual syntax is different than the more abstract syntax given here, mainly because of the limitations of the Groovy Builders library (http://groovy.codehaus.org/Builders).

bottom *term* and a *difference*.

```
Concept 'Column' eng-name 'column' {
    attribute 'top' type 'Term'
    attribute 'bottom' type 'Term'
    attribute 'result' type 'Difference'
}
```

Whereas the concepts are the basic building blocks of the learning environment, functions and relation may help describe the objects. They are also manipulated by the goals and procedures. For example, in the subtraction task, we could have a *nextColumn* function with a *Column* argument and a *Column* image:

```
function 'nextColumn' eng-name 'next column' {
   argument 'current' type 'Column'
   image 'next' type 'Column'
}
```

Relations are defined like functions, but with a list of places instead of a list of arguments and an image.

Implementing the Learning Environment

The Views

Once enough declarative knowledge has been modeled, the author can design the learning environment. For each concept that the learner can perceive or manipulate, the author defines a *view*. A view is composed of two scripts[14] that define how to build and update the visualization of the concept in the learning environment. First the *build* script is executed each time a view is instantiated for an object that instantiate the concept (it is possible to have multiple views of the same object). Second, the *update* script is executed each time the visualized object is modified by a step (as described below).

The role of the scripts is not only to manage the visualization of the learning environment, but also to maintain a corresponding model of the latter that can be manipulated by the MTT's modules. This model is formed of a view tree where each view can have *sub-views* associated to objects linked to the visualized object through features, functions or relations. For each view, the model also contains a set of *components* and a set of *handlers*. For example, a view tree could have a root view containing a textfield (with a 'ValueChange' handler), a button (with a 'ButtonPress' handler) and a sub-view that itself only has a label (with no handlers).

The Steps

To obtain changes in the views, steps that the learner can perform need to be added to the model. The starting points of steps are *primitive procedures* which are the leaf

[14] Written using an EDSL derived from Groovy Swing Builder (http://groovy.codehaus.org/Swing+Builder).

vertices of the procedural graph and the first to be added to the model of the task. As each procedure is associated to a goal, the corresponding goals must be added first. Goals are defined by a set of parameters that becomes local variables of the procedures. For example, in the subtraction task, the goal GChangeTerm has two parameters:

```
Goal 'GChangeTerm' eng-name 'change the value' {
    param 'term' type 'Term' eng-name 'term'
    param 'newValue' type 'int' eng-name 'new value'
}
```

Two different kinds of scripts need to be linked to a primitive procedure to form a step. The first script is the *KB script.* This script, which is executed every time the step is performed, produces modifications to the KB that are reflected in the learning environment through updated or newly built views. Those modifications include adding objects (as instances of concepts), changing or removing objects, and (un)-asserting relations and functions on the latter. For example, the KB script of the primitive procedure PPChangeTerm in the subtraction task changes a term:

```
Primitive 'PPChangeTerm' achieves 'GChangeTerm' {
   change(term, newValue)
}
```

In order for to recognize that a learner has performed a step, *interactions scripts* are associated to the primitive procedures. Those scripts contain a pattern of interactions that is similar to a regular expression. The interactions included in the pattern are either handlers or composite interactions (that usually represent reusable inputs and selections). They also contain a set of binding between the variables of the procedure and extraction mechanisms that extract objects that become the arguments of the step. For example, the *interaction script* associated to PPChangeTerm's pattern is an atom:

```
Do 'PPChangeTerm' {
   pattern 'a1' <- interaction(Term, writeValue)
   argument 'term' <- owner(a1)
   argument 'newValue' <- extract(a1)
}
```

The sole interaction a1 is triggered when the *writeValue* handler is used in a Term's view. Then, the *owner*, the object visualized by the view containing the handler, is extracted and bound to the term variable of the primitive procedure. Finally, the default extraction mechanism extract the object produced by the handler that triggered the interaction, in this case, an integer object that corresponds to the new value of the textfield in the Term's view.

The patterns found in some steps are more complex, containing sequences, repetitions and selections of interactions, some of them possibly interleaved. The use of interaction scripts allows Astus to not only recognize complex patterns of interactions

that are linked to a single step, but also to generate them in order to demonstrate a step with mouse movements, mouse clicks and keyboard strokes.

Defining the Procedural Knowledge

The next phase is for the author to define the procedural knowledge required to perform the task. This is mainly accomplished by defining *complex procedures* and the goals they achieve. The different types of complex procedure are presented in the section named "explicit procedural knowledge units" of the main paper and examples of complex procedure definitions are provided in sub-section "next-step hints" of the "semantically rich declarative knowledge units" section of the paper.

In many tasks, defining the complex procedures will require the addition of new concepts, functions and relations to the model. In particuliar, those for which the instantiation is inferred. Astus offer two different mechanisms of inference. The first one is to associate a script to the concept, relation or function that will be executed when a condition or a query in the complex query refers to one of the latter. The second mechanism is a Jess[15] production rule for which the THEN-PART instantiate the concept, the function, or the relation when the IF-PART is matched against the content of the KB.

For example, in the subtraction task, the procedure PCalcDiff queries the subtract function which has an associated script:

```
sequence 'PCalculateDif' achieves 'GCalculateDiff' {
  goal 'GWriteTerm' using unique(subtract, ['column → top',
     'column → bottom'])
}

function 'subtract' eng-name 'subtract' {
  argument 'op1' type Integer
  argument 'op2' type Integer
  image 'diff' type Integer

  apply : { op1 - op2 }
}
```

Finally, another example, in the AVL-tree task, is the Leaf concept instantiated via a production rule:

```
(defrule Leaf
    (Pointer (object ?nullPointer) (id "null") )
    (BinaryNode (object ?node) (left ?leftCell) (right
     ?rightCell))
    (Cell (object ?leftCell) (value ?nullPointer))
    (Cell (object ?rightCell) (value ?nullPointer))
  =>
    (isa ?node "Leaf")
)
```

[15] Jess, the Rule Engine for the Java Platform (http://herzberg.ca.sandia.gov/)

Defining the Task Cases

The final phase of the development process is for the author to create tasks cases that the learners will have to perform. To define a case, the author needs to specify a root goal and the initial state of the KB. The initialization script of the KB, similar to the KB script of a primitive procedure, instantiate concepts, functions and relations that are case specific. For example, in the subtraction task:

```
case '5031-1689' goal 'Gsubtract' {
  init(5031, 1680)
}
```

Where in it is an auxiliary routine that is called in each of the task case definition:

```
def init(minuend, subtrahend) {
  def cols = []

  while (subtrahend != 0 || minuend != 0) {
    def c = object(Column, top: object(Term, value:minuend % 10),
          bottom: object(Term, value:subtrahend % 10),
          result: object(Difference))

    cols.add(c)
    minuend = minuend / 10
    subtrahend = subtrahend / 10
  }

  fact(firstColumn, [cols[0]])
  fact(lastColumn, [cols[cols.size() - 1]])

  for (i in (cols.size() - 2)..0) {
    fact(nextColumn, [cols[i], cols[i+1]])
  }
}
```

References

Adelson-Velskii, G. M., & Landis, E. M. (1962). An Algorithm for the Organization of Information. *Soviet Mathematics Doklady, 3*, 1259–1262.

Aleven, V. (2010). Rule-Based Cognitive Modeling for Intelligent Systems. In Nkambou R., Bourdeau J. & Mizoguchi R. (Eds.) Advances in Intelligent Tutoring Systems, 33–62.

Aleven, V., & Koedinger, K. R. (2008). An Effective Metacognitive Strategy: Learning by Doing and Explaining with a Computer-Based Cognitive Tutor. *Cognitive Science, 26*, 147–179.

Aleven, V., McLaren, B. M., & Sewall, J. (2009a). Scaling Up Programming by Demonstration for Intelligent Tutoring Systems Development: An Open-Access Website for Middle School Mathematics Learning. *IEEE Transactions on Learning Technologies, 2*(2), 64–78.

Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. *In Proceedings of Intelligent Tutoring Systems, 2006*, 61–70.

Aleven, V., McLaren, B.M., Sewall, J., Koedinger, K.R. (2009). Example-Tracing Tutors: A New Paradigm for Intelligent Tutoring Systems. International Journal of Artificial Intelligence in Education, Special Issue on "Authoring Systems for Intelligent Tutoring Systems", 105–154.

Anderson, J. R., Boyle, C. F., Corbett, A., & Lewis, M. W. (1990). Cognitive Modeling and Intelligent Tutoring. *Artificial Intelligence, 42*, 7–49.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill Acquisition and the LISP Tutor. *Cognitive Science, 13*, 467–505.

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences, 4*, 167–207.

Anderson, J.R., Pelletier, R. (1991). A Development System for Model-Tracing Tutors. In Proceedings of the International Conference of the Learning Sciences, 1–8

Barnes, T., Stamper, J., Croy, M., & Lehman, L. (2008). A Pilot Study on Logic Proof Tutoring Using Hints Generated from Historical Student Data. *In Proceedings of Educational Data Mining, 2008*, 197–201.

Conati, C., & VanLehn, K. (1999). Teaching Meta-Cognitive Skills: Implementation and Evaluation of a Tutoring System to Guide Self-Explanation while Learning from Examples. *Proceedings of Artificial Intelligence in Education, 1999*, 297–304.

Corbett, A.T., Koedinger, K.R., Anderson, J.R. (1997). Intelligent Tutoring Systems. In Handbook of Human-Computer Interaction (2nd ed.), 849–874.

Erol, K., Hendler, J., & Nau, D. S. (1994). UCMP: A Sound and Complete Procedure for Hierarchical Task-Network Planning. In *Proceedings of the International Conference of Artificial Intelligence Planning Systems* (pp. 249–254).

Heffernan, N. T., Koedinger, K., & Razzaq, L. (2008). Expanding the Model-Tracing Architecture: A 3rd Generation Intelligent Tutor for Algebra Symbolization. *International Journal of Artificial Intelligence in Education, 18*, 153–178.

Hill, R.W., Johnson, L.W. (1993). Impasse-Driven Tutoring for Reactive Skill Acquisition. In Proceedings of the Conference on Intelligent Computer-Aided Training and Virtual Environement Technology

Hill, R. W., & Johnson, L. W. (1995). Situated Plan Attribution. *International Journal of Artificial Intelligence in Education, 6*, 35–66.

Johnson, L. W. (1990). Understanding and Debugging Novice Programs. *In Artificial Intelligence, 42*, 51–97.

Koedinger, K., & Anderson, J. R. (1993). Effective Use of Intelligent Software in High School Math Classrooms. *Proceedings of Artificial Intelligence in Education, 1993*, 241–248.

McKendree, J. (1990). Effective Feedback Content for Tutoring Complex Skills. *Human-Computer Interaction, 5*, 381–413.

McLaren, B. M., & Isotani, S. (2011). When is it Best to Learn with All Worked Examples? *Proceedings of Artificial Intelligence in Education, 2011*, 222–229.

McLaren, B.M., Lim, S., Koedinger K.R. (2008). When and How Often Should Worked Examples be Given to Students? New Results and a Summary of the Current State of Research. Proceedings of the Cognitive Science Conference

Mitrovic, A. (1998). A Knowledge-Based Teaching System for SQL. *In Proceedings of ED-MEDIA, 1998*, 1027–1032.

Mitrovic, A. (2010). Modeling Domains and Students with Constraint-Based Modeling. In Nkambou R., Bourdeau J. & Mizoguchi R. (Eds.) Advances in Intelligent Tutoring Systems, 63–80.

Mitrovic, A., & Ohlsson, S. (1999). Evaluation of a Constraint-Based Tutor for a Database Language. *Artificial Intelligent in Education, 10*, 238–256.

Mitrovic, A., & Weerasinghe, A. (2009). Revisiting Ill-Definedness and the Consequences for ITSs. *In Proceedings of Artificial Intelligence in Education, 2009*, 375–382.

Ohlsson, S. (2008). Computational Models of Skill Acquisition. The Cambridge Handbook of Computational Psychology, Cambridge University Press, 359–395

Paquette, L., Lebeau, J.-F., Mayers, A. (2010). Authoring Problem-Solving Tutors: A Comparison Between CTAT and ASTUS. In Nkambou R., Bourdeau J. & Mizoguchi R. (Eds.) Advances in Intelligent Tutoring Systems, 377–405.

Paquette, L., Lebeau, J.-F., & Mayers, A. (2012a). Automating the Modeling of Learners' Erroneous Behaviors in Model-Tracing Tutors. *In Proceedings of User Modeling, Adaptation and Personalization, 2012*, 316–321.

Paquette, L., Lebeau, J.-F., Mayers, A. (2013). Diagnosing Errors from Off-Path Steps in Model-Tracing Tutors. In Proceedings of Artificial Intelligence in Education 2013

Paquette, L., Lebeau, J.-F., Beaulieu, G., & Mayers, A. (2012b). Automating Next-Step Hints Generation Using ASTUS. *In Proceedings of Intelligent Tutoring Systems, 2012*, 201–211.

Paquette, L., Lebeau, J.-F., Mbungira, J. P., & Mayers, A. (2011). Generating Task-Specific Next-Step Hints Using Domain-Independent Structures. *In Proceedings of Artificial Intelligence in Education, 2011*, 525–527.

Reiser, B.J., Kimberg, D.Y., Lovett, M.C., Ranney, M. (1992). Knowledge Representation and Explanation in GIL, An Intelligent Tutor for Programming. In Computer-Assisted Instruction and Intelligent Tutoring Systems, 111–149

Rickel, J. (1988). An Intelligent Tutoring Framework for Task-Oriented Domains. *In Proceedings of Intelligent Tutoring Systems, 1988*, 109–115.

Rickel, J., & Johnson, L. W. (1999). Animated Agents for Procedural Training in Virtual Reality: Perception, Cognition, and Motor Control. *Applied Artificial Intelligence, 13*, 343–382.

Sacerdoti, E. D. (1975). *A Structure for Plans and Behavior*. New York: Elsevier.

Sottilare, R.A., Goldberg, B.S., Brawner, K.W., Holden, H.K. (2012). A Modular Framework to Support the Authoring and Assessment of Adaptive Computer-Based Tutoring Systems (CBTS). In Proceedings of the Interservice/Industry Training, Simulation, and Education Conference.

Stamper, J., Barnes, T., Croy, M., & Eagle, M. (2013). Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. *International Journal of Artificial Intelligence in Education, 22*(1), 29–41.

VanLehn, K. (1990). Mind Bugs: The Origin of Procedural Misconceptions. MIT Press

VanLehn, K. (2006). The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education, 16*, 227–265.

VanLehn, K., Lynch, C., Schultz, K., Shapiro, J. A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., & Wintersgill, M. (2005). The Andes Physics Tutoring System: Lessons Learned. *International Journal of Artificial Intelligence in Education, 15*, 147–204.

Wenger, E. (1987). Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge. Morgan Kaufmann Publishers

Woolf, B. (2009). Building Intelligent Interactive Tutors: Student-Centered Strategies for Revolutionizing E-Learning. Morgan Kauffman Publishers