The Generation of 'Critical Problems' By Abstract Interpretations of Student Models.

Rick Evertsz*
Scientia Ltd,
150 Brompton Road,
London, SW3 1HX,
United Kingdom.

Abstract

As the range of models which a tutoring system can capture is extended, efficient diagnosis becomes more difficult. We have implemented a partial solution to this problem, in the form of a 'Critical Problem' generator. We argue that great diagnostic power can be obtained by generating discriminating problem examples. In general, efficient diagnosis is just not possible without such a hypothesis-testing capability. We describe a program, PG, which given a pair of production rule models and a description of the class of problems which the student must solve, generates an abstract specification of the problems which discriminate between those two hypotheses. The key to this problem lies in the realisation that we are only interested in the abstract mapping between a model's inputs and outputs; from the point of view of generating a Critical Problem, the intermediate processing of the model is irrelevant.

1. Introduction

In the realm of ICAI, the 'student model' is generally recognised as an essential prerequisite for adaptive tutoring. It embodies the machine's view of what the student 'knows' - without it, the tutoring system cannot make pertinent tutorial decisions. The space of models which a tutoring system must consider is non-trivial. Current student modelling research is concerned with extending this space through the use of machine learning techniques (cf. ACM, [Langley and Ohlsson, 1984]). Whilst the machine learning approach does extend the potential coverage of the modelling system, the larger search space leads to poor runtime performance - as it stands this method could not form part of a tutoring system which runs in real-time. Interestingly, research on electronic fault diagnosis has run into similar problems. Extending the range of models to the multiple-fault case, leads to a mushrooming of an already large search space [de Kleer, 1986]. Thus for diagnostic domains, a major research goal is to provide the machine with *efficient* methods of searching ever larger spaces.

Through the medium of a computer program called **PG** (**P**roblem **G**enerator), we have developed an approach to controlling the diagnostic search space, whose roots lie in the methodology of science - where the experimental method is employed to distinguish between competing hypotheses. The central tenet is that great diagnostic power can be obtained by generating discriminating problem examples. Armed with the ability to test hypotheses about what the student is doing, the modeller acquires the same powerful leverage available to the scientist who can generate experiments to test his/her theories.

Imagine a situation in which a tutoring system is trying to ascertain how a student deals with fraction subtraction problems of the form: $\mathbf{W - X^y/_z}$. The student has just solved the problem $\mathbf{8 - 4^3/_4}$, and come up with an answer of $\mathbf{4^3/_4}$. The tutor searches for an account of the student's error, and produces two possibilities:

> **Model-1:** If the first whole number has no associated fraction, then subtract the second whole number from the first, and write down the fractional part of the second term.
>
> **Model-2:** If the first whole number has no associated fraction, then the answer is the whole of the second term (i.e. discard the first term).

The tutoring system tries to obtain more information by setting the student a problem of a similar type (i.e. of the form $\mathbf{W - X^y/_z}$). Instantiating this problem template with a random set of numbers (subject to the constraint that it form a legal fraction subtraction problem), the tutor sets the following problem next: $\mathbf{4 - 2^7/_8}$. Unfortunately, in the current situation this problem provides no diagnostic power whatsoever. In order to discriminate between the above two models, the machine must generate a problem which will produce a different answer for each.

We call a problem which discriminates between two hypotheses a 'Critical Problem'. IDEBUGGY [Burton, 1982] adopts an essentially heuristic method of generating Critical Problems. Each primitive bug has an associated set

* This work was funded by SERC, through the UK Alvey programme.

of problem templates which, when instantiated, are likely to produce problems which reveal the presence of that bug. Not all of the problems will have this feature, therefore the system needs to run the model on the problem to make sure that it does the required job. Though quite effective, IDEBUGGY's approach has one major drawback. The system implementor must specify the problem templates for each of the bugs in the database of models. This means that any changes to the models must be echoed in their associated problem templates. This reliance on an outside party to maintain the problem templates, means that the technique is only useful where there is a fixed, unchanging library of bugs.

Sleeman [1981] describes a system which attempts to generate such problem templates automatically, by running the rules backwards. His work is the only previous attempt to generate problems on the basis of the *form* of the rules in the models, rather than on the basis of 'canned' procedures attached to each rule. However, it can only be applied to a very limited range of production rule models, and even then it only works when in partnership with the user (who must supply the all-important termination condition). It can only be applied to models where Working Memory only ever contains one element, and all of the rules have one condition element only. There is no provision for multiple LHS conditions, or negated condition elements. The algorithm also ignores the relationships between variables in the LHS. In the simple algebra examples, tackled by the program, inter-variable constraints can be ignored. Under such conditions, the program can generate a discriminatory template, where the two models contain the same rules, but ordered differently. In the more general case, where the two models contain different rules (limited to one pair only), the algorithm is incapable of generating a discriminating template. Evertsz [1989] presents a critique of the algorithm, and shows why any algorithm, which analyses the rules in the reverse direction, is doomed to failure because it cannot capture crucial aspects of the model's computational behaviour (in particular, conflict resolution).

Our approach to the problem is to have PG reason about the *abstract* input/output behaviour of the candidate models. This approach borrows from the methodology of experimental science. Scientists formulate experimental hypotheses in terms of variables which are either manipulable or measurable. These variables summarise the internal, unobservable workings of the system being tested. It is the descriptive economy of a theory, expressed solely in terms of observables, which simplifies the problem of proposing an informative experiment (for example, compare the simplicity of the General Gas Law with the more complicated Kinetic Theory of Gases). From the hypothesis-testing perspective, the intermediate processing performed by a model is only relevant in so far as it defines the mapping between inputs and outputs. The hypothesis-tester can only manipulate inputs and observe the student's outputs.

PG takes a pair of student models, expressed in an OPS-like production system, and through a process termed 'Abstract Interpretation' derives an abstract specification of the problems which discriminate between the two models. This abstract specification can be instantiated, by a process of constraint satisfaction, to yield a Critical Problem.

2. Abstract Interpretation

Student models are designed to take *concrete* values as inputs, and produce other concrete values as outputs. For example, a model which computes the factorial of a number, would be given a concrete number, **N**, as input, and would output another number, **N!**. Now, imagine that our goal is to characterise the general behaviour of that model. We do not want to know what its output would be for some specific, concrete input; rather, we want some general description of the model's output for some abstract, non-specific **N**. This abstract description can then be compared with that for the competing model to see whether they are non-equivalent (if the two models are equivalent, in terms of their input/output behaviour, then no Critical Problem exists).

Abstract Interpretation is a commonly employed solution to the problems of automatic program analysis. The general idea is to glean information about a program by running it on *abstract specifications* of data objects, rather than the data objects themselves. Cousot and Cousot [1977] first introduced the notion of the Abstract Interpretation of imperative languages. We have developed an algorithm for the Abstract Interpretation of production systems, which takes as input a pair of production rule models and an abstract description of the class of legal inputs to those models, and produces a description of the set of Critical Problems. The key difference between an Abstract Interpreter and a normal production rule interpreter, is the former's ability to run in an abstract rather than a concrete domain.

Note that it is not necessary that PG find *all* I/O mappings for the two models. It is sufficient to find just one from which a Critical Problem can be generated. Although we do not require that PG find all I/O mappings, we do desire that its search strategy be 'complete'. That is, in the limit, it should be able to find all I/O mappings, stopping when either a discriminating output description is found, or there are no more I/O mappings for the two rulesets. Furthermore, we require that the I/O mappings be 'sound'. In other words, PG should never derive abstract I/O mappings encompassing concrete I/O mappings which the models would never actually compute. For example, if a model only computes the I/O mapping: **(x + y)** for the two input variables **x**, and **y**, then we would not expect PG to derive the I/O mapping: **(x - y)**. The mapping, **(x - y)**, is *unsound* because it embraces concrete I/O mappings which are never computed by the ruleset (e.g if **x** is **5** and **y** is **2**, then the abstract mapping would erroneously predict an output of **3**). In other words, for any ruleset, if **A** is the set of possible I/O mappings in the concrete domain, and **B** is the set of possible instantiations (in the same concrete domain) of PG's derived abstract I/O mappings, then it should never be the case that: $\exists(x \in B) x \notin A$.

3. Production System Execution in the Abstract Domain

Each operation of a production system has its abstract equivalent in PG. For example, pattern matching is replaced by unification so that rules can match abstract working memory elements (i.e. ones containing uninstantiated variables). In general, a model, running in the concrete domain, can follow a number of different paths to a halt state, depending on what the initial input looks like. The input, together with the rules in the model, define the particular *path* taken through the space of possible routes from start state to halt state. PG characterises this set of potential paths by running on abstract data. For example, consider a model which takes a number, **N**, and outputs whether it is even or odd. The set of all possible paths through the production system is indexed by the members of the union of the set of all even numbers and the set of all odd numbers (i.e. the infinite set of integers). However, when run in the abstract, this infinite set is summarised by the two abstract paths:

$$\forall(n)\ EVEN(n) \supset OUTPUT(EVEN).$$

$$\forall(n)\ ODD(n) \supset OUTPUT(ODD).$$

During a run, PG collects the outputs along each of these paths, and terminates with a description of the abstract mapping between inputs and outputs. In principle, PG can search all paths, however in practice it searches the paths one at a time until a Critical Problem is found.

3.1 Input Specifications

PG is provided with a description of the class of inputs which the model can consider (termed a set of 'Input Specifications'). It *is* possible to abstractly interpret a ruleset without an Input Specification, however, the search space becomes unmanageable. In principle, any ordered subset of the LHS patterns could form a 'meaningful' abstract input to a model (by 'meaningful input' we mean one which is capable of matching a non-empty subset of the LHS patterns). For example, according to this criterion, a ruleset with five distinct LHS patterns is capable of spawning **5!** Input Specifications, i.e. 120. In fact, there is nothing to stop us including duplicate abstract input elements, in which case the space of Input Specifications is infinite.

In the context of a tutoring system, the set of Input Specifications is precisely the collection of problem templates from which the tutoring system generates problems to pose to the student. These problem templates describe the set of *legal* problems for the domain being tutored. For example, a problem template in the subtraction domain might be: 'One number minus another, where the first number is greater than or equal to the second'. However, we should not expect our tutoring system to have the following template: 'One number, **M**, minus another, **S**, where the units digit of **M** is less than that of **S**, and **M≥S**'. This is because the reference to the relationship between the units digits of **M** and **S** has nothing to do with the legality of the problem; it is there to test for a particular subskill, i.e. borrowing. The units test is an implicit embodiment of the expectation that the borrowing subskill can be faulty, and implies that it is worth setting problems which require a borrow, as these will reveal any flaws in the student's algorithm. PG is only given Input Specifications which encode information about problem legality. To do otherwise, would be to bias its search for Critical Problems.

3.2 Abstract Rule Instantiation

PG keeps a record of the current state of Working Memory down each path being explored, and a record of the instantiations fired. It also maintains a note of the current tree-depth of the problem-solving process. Every time a rule fires, it moves one level deeper into the tree. The level marker is used to rename variables in a production rule, so that they do not clash with variables of the same name, in other parts of the tree. PG also maintains a record of the variable bindings, from cycle to cycle (termed an 'environment'). Working Memory will contain terms with uninstantiated variables, however, PG can partially instantiate them by retrieving their values from the environment. In addition to an environment of variable bindings, PG maintains the set of 'constraints' on the variables in the environment. For example, if a firing rule requires that **?x** be less than **?y**, then this constraint must be carried forward to subsequent cycles. PG also records any outputs, as these will be used to build the I/O mappings of the ruleset. The role played by these various data structures will become clearer as the algorithm is developed in subsequent sections.

When PG is run on an Input Specification, the level marker is initialised to zero, and the patterns in the Input Specification are added to Working Memory, after *renaming* the variables therein. Any constraints in the Input Specification are also renamed, and are then added to PG's constraint set. For example, if the Input Specification were: **(?x - ?y), (> ?x ?y)**, then Working Memory would contain the single element: **(?x.0 - ?y.0)**, and the constraint set would consist of: **(> ?x.0 ?y.0)**. PG is now ready to commence the first Recognise-act Cycle. The Recognise-act Cycle begins with the incrementing of the level marker to **1**.

The lefthand sides of the rules are *unified* with the contents of Working Memory. Our unification algorithm is augmented to allow function expressions to be unified with constants or other function expressions. For example, when asked to unify the function expression: **(*times ?x.1 5)** with: **20**, it generates the equality constraint: **(= (*times ?x.1 5) 20)**.

The abstract rule instantiations are passed to a resolution theorem prover which assesses their consistency. For example, if the Input Specification defines **?x** to be odd, then the theorem prover will reject any instantiation of a rule whose lefthand side contains the constraint: **EVEN(?x)**, because **EVEN(?x)** $\wedge$ **ODD(?x)** is unsatisfiable for all values of **?x**. Resolution theorem provers seem ideally suited to the job of spotting

unsatisfiable sets of constraints, as they are geared to finding contradictions.

Our production rule language includes 'negated patterns'; these are patterns which require that no matching element be present in Working Memory. If a negated pattern unifies with some Working Memory element, then PG tries to build the minimal set of constraints (termed a 'Negation Nullifier') which would prevent the negated pattern from matching that item. The instantiation is said to be valid if the Negation Nullifier is consistent with the constraints collected so far. This is illustrated in the following scenario.

4. A Simple Scenario

The following tiny ruleset has been created to illustrate how a negation nullifier is built, and to show the unification of a function expression with a constant. In the concrete domain, it starts with an input of the form: **(add ?x ?y)**, where the variables are both natural numbers, and deposits the sum of **?x** and **?y** in working memory. The rule **finish** fires if that sum is zero, not equal to **?x**, and the original **?x** and **?y** are equal; it outputs **?x** and halts. Clearly, there is no way that this ruleset can ever output anything, because the formula: **?x+?y=0** $\wedge$ **?x=?y** $\wedge$ **?x≠0** $\wedge$ **NATNUM(?x)** $\wedge$ **NATNUM(?y)** is unsatisfiable. In the following scenario, we shall see how the attempt to create a negation nullifier leads to the predicted contradiction. Constraints and function expressions begin with an asterisk, and negated patterns are prefixed with a tilde.

Input Specification:
((add ?x ?y) (*natnum ?x) (*natnum ?y))

Rules:

```
(define-rule start
  if (add ?x ?y)
  then (sum-x-y (*plus ?x ?y)))

(define-rule finish
  if (add ?x ?x) & (sum-x-y 0) & ~(sum-x-y ?x)
  then (*output ?x) & (*halt))
```

At the start of the run, Working Memory is initialised with the single element: **(add ?x.0 ?y.0)**, and the constraint set is initialised to: **(*natnum ?x.0)** $\wedge$ **(*natnum ?y.0)**. The Recognise-act Cycle begins, and the instantiation of **start** is added to the Conflict Set. The instantiation of **start** contains the following environment: **(?y.1/?y.0 ?x.1/?x.0)**, where *?Z/?Y* denotes the binding of *?Z* to *?Y*. The instantiation fires and deposits **(sum-x-y (*plus ?x.1 ?y.1))** in Working Memory.

Working Memory:
((sum-x-y (*plus ?x.1 ?y.1)) (add ?x.0 ?y.0))
Constraint Set:
((*natnum ?x.0) $\wedge$ (*natnum ?y.0))
Environment:
(?y.1/?y.0 ?x.1/?x.0)

On the next cycle, **start** is instantiated, but is rejected because it has already fired. The two positive patterns in **finish** are satisfied by the contents of Working Memory as follows: the first pattern, **(add ?x.2 ?x.2)**, unifies with **(add ?x.0 ?y.0)** and adds the bindings, **(?x.0/?y.0 ?x.2/?x.0)**, to the current environment. The second pattern, **(sum-x-y 0)**, unifies with **(sum-x-y (*plus ?x.1 ?y.1))** generating the equality constraint: **(= (*plus ?y.0 ?y.0) 0)**. The two Abstract Pattern Instantiations are now paired by merging their environments and constraints.

Merged Environment:
(?x.0/?y.0 ?x.2/?x.0 ?y.1/?y.0 ?x.1/?x.0)
Merged Constraints:
((= (*plus ?y.0 ?y.0) 0))

The negated pattern, **~(sum-x-y ?x.2)**, unifies with the Working Memory element, **(sum-x-y (*plus ?x.1 ?y.1))**, and produces the Abstract Negated Pattern Instantiation shown below.

Negated Pattern:
~(sum-x-y ?x.2)
WM Element:
(sum-x-y (*plus ?x.1 ?y.1))
Bindings:
(?x.2/(*plus ?x.0 ?y.0))

The single binding is turned into an equality constraint, **(= ?x.2 (*plus ?x.0 ?y.0))**, and then negated, **¬(= ?x.2 (*plus ?x.0 ?y.0))**; this is the Negation Nullifier. The Negation Nullifier is then instantiated in terms of the current environment, and passed to the theorem prover, along with the current set of constraints (instantiated). Note that the combined set of constraints is instantiated in terms of the current environment, before passing it to the theorem prover. This explains why the constraint, **(*natnum ?x.0)**, disappears when the current constraint set is conjoined with the Negation Nullifier. When it is instantiated it becomes: **(*natnum ?y.0)**, which then gets deleted because it is a duplicate constraint.

Negation Nullifier:
¬(= ?x.2 (*plus ?x.0 ?y.0))
Current Environment:
(?x.0/?y.0 ?x.2/?x.0 ?y.1/?y.0 ?x.1/?x.0)
Current Constraint Set:
((= (*plus ?y.0 ?y.0) 0) ∧ (*natnum ?x.0) ∧
(*natnum ?y.0))

Instantiated Negation Nullifier:
¬(= ?y.0 (*plus ?y.0 ?y.0))
Combined Constraint Set:
¬(= ?y.0 (*plus ?y.0 ?y.0)) ∧
(= (*plus ?y.0 ?y.0) 0) ∧
(*natnum ?y.0))

The theorem prover rejects the combined constraint set, because it is unsatisfiable. Informally, if **?y.0** is a natural number, which results in zero when added to itself, then **?y.0** can only be zero; however, this contradicts the first constraint, which states that **?y.0** is not equal to **?y.0** + **?y.0**. So, the interpreter halts because there are no other rules to fire.

5. Abstract Conflict Resolution and Rule Firing

Once the Conflict Set is computed, PG applies abstract versions of the conflict resolution principles Recency and Specificity. Applying Recency to abstract instantiations is not problematic, because PG maintains an abstract version of Working Memory for each path. Specificity pertains to the lefthand sides of rules, thus its definition is the same as that for an interpreter operating in the concrete domain.

In the concrete domain, conflict resolution chooses a unique instantiation to fire, and only that path is followed subsequently. However, in the abstract domain the conflict resolution principles can only be used to *order* the instantiations in the Conflict Set. This is a subtle, but crucial point. Imagine a situation in which the Conflict Set contains two instantiations, I_1 and I_2. After applying conflict resolution, I_1 is found to be the winner. However, in the concrete domain, there may be cases where, because of the particular values in Working Memory, I_1 cannot be instantiated. In such instances, I_2 will win, because it is the only member of the Conflict Set. The Abstract Interpreter must take this into account, otherwise it will unintentionally exclude valid paths. The instantiation, I_2, wins precisely when its constraints are satisfied and those of I_1 are not. For example, if I_1 includes the constraint, **(*even ?x)**, but I_2 does not, then I_2 will be chosen whenever **?x** is not even. If the Input Specification does not specify whether **?x** is even or odd, then both instantiations are valid, because PG has no evidence to the contrary. Were it to simply choose I_1 on the basis of Specificity, then it would loose the information pertaining to the path followed when **?x** is odd.

PG solves this dilemma by explicitly attaching an 'exclusion clause' to I_2. The exclusion clause consists of the *negation* of the set of constraints in I_1. Thus, I_2 inherits the constraint, **¬(*even ?x)**, which in effect says that I_2 can only fire if **?x** is not even. Note that PG passes the constraints of I_2, together with the exclusion clause, to the theorem prover, which may find them to be unsatisfiable. In this case, PG rejects the instantiation altogether.

Abstract Rule Firing is quite simple, and merely involves depositing any righthand side elements in Working Memory (without instantiating any of the variables therein). Any 'output' elements in the righthand side are recorded, so that the path always has a list of the abstract outputs produced so far.

6. Generating Critical Problems

As described so far, Abstract Interpretation is carried out on a single model. In fact, PG interprets the pair of candidate models in tandem. Recall that its overall goal is to find a *single* input for which each model produces a *different* output. A naive way to achieve this goal is to find *all* of the I/O mappings for each model, and to then sift through these until a pair of mappings, with an overlapping input but non-overlapping output, is found. However, it is more economical to only associate the paths in one model with those paths in the other which have overlapping abstract inputs. This strategy enables PG to discard unpromising paths early on. Returning to our even/odd example, if some path in model-1 specifies that the input is even, whilst another path in model-2 specifies that it is odd, then there is no point developing these paths further. They will never yield a Critical Problem, because there is no concrete input which is both even and odd.

PG employs a heuristic which guides it down the most promising paths. Before running the pair of models, it performs a *dependency analysis* on the lefthand side and righthand side patterns of the rules in each model. This analysis produces a graph which encodes how the lefthand side patterns are triggered by the righthand side actions. This enables it to make informed guesses about what rules lie on a path between the start state and some intermediate state. Because different rules tend to produce different behaviour, it is often the case that a path in one model, which passes through a rule which is not in the other model, will yield a different output. PG uses the dependency analysis to suggest a path in one model which goes through some rule which is not in the other model.

Once PG has found a non-equivalent pair of abstract I/O mappings, it has only to instantiate the I/O mappings. This entails finding a concrete input, which satisfies the combined constraints of both I/O mappings, but produces a different output in each case. Let the set of constraints for model **i** be denoted by $\mathbf{C_{i1}} \wedge \ldots \wedge \mathbf{C_{im}}$, and let $\mathbf{O_i}$ be the output for that model. For the two candidate models, **i** and **j**, PG's goal is to find some instantiation of the following expression: $\mathbf{C_{i1}} \wedge \ldots \wedge \mathbf{C_{im}} \wedge \mathbf{C_{j1}} \wedge \ldots \wedge \mathbf{C_{jn}} \wedge \mathbf{O_i} \neq \mathbf{O_j}$. We have implemented a simple constraint-satisfaction procedure which finds solutions to such expressions; however, this aspect of PG has received comparatively little attention, and we would not want to make any great claims in this direction. There are a number

of constraint-satisfaction procedures which could more efficiently perform the function of instantiating such expressions (cf. Davis, [1987]).

7. Evaluation in the Domain of Fraction Subtraction

We have evaluated PG on a set of empirically-derived production rule models from the domain of fraction subtraction. These are based on data collected from 29 children [Evertsz, 1982]. In all runs so far, PG is able to find a Critical Problem if one exists. This success is dependent on its axiomatisation of the constraints and function expressions used in the production rule models. PG's reasoning about the abstract computational behaviour of the models is sound, provided that it is supplied with an adequate *semantics* for the predicates and functions in the models.

8. Discussion

Despite PG's success in the domain of fraction subtraction, the current implementation embodies a fundamental limitation. It is our feeling that this limitation is unimportant for most domains, but we highlight it here nevertheless.

Currently, there is one aspect of the computational behaviour of production systems which PG is unable to capture: I/O mappings defined in terms of loops. For example, consider a pair of student models which both compute the function 'factorial'. Both models loop until the original number reaches zero, and then multiply the intermediate numbers together (e.g. **5!** → **5, 4, 3, 2, 1, 0** → **1*1*2*3*4*5** → **120**). However, one model performs the multiplication by *successive addition*, whilst the other simply calls the function ***mult**. Such a situation throws PG into an infinite loop. The first time round the loop, both models output **1**. On subsequent cycles, the base value of **1** gets wrapped in ever deeper nestings of ***mult** in one case, and an equivalent composition of ***plus**'s in the successive addition case. To overcome this limitation in a general way, one could make use of mathematical induction, as in the Boyer/Moore theorem prover [Boyer and Moore, 1979].

A general-purpose Abstract Interpreter of production systems should be capable of handling such loops. However, many procedural skills do not incorporate such looping behaviour. Therefore, there seems little point in adding this functionality until we encounter a domain which requires the extra inductive machinery.

Acknowledgements

Many thanks to Marc Eisenstadt, Mark Elsom-Cook and Tim O'Shea for providing great support during the development of this work.

I am grateful to Richard Joiner for commenting on earlier drafts of this paper.

References

[Boyer and Moore, 1979] Boyer, R.S. and Moore, J.S. *A Computational Logic*. Academic Press, Inc., 1979.

[Burton, 1982] Burton, R.R. *Diagnosing bugs in a simple procedural skill*. in 'Intelligent Tutoring Systems', Sleeman and Brown (eds), Academic Press, pp157-183, 1982.

[Cousot and Cousot, 1977] Cousot, P., & Cousot, R. *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixpoints*. 4th POPL, pp238-252, 1977.

[Davis, 1987] Davis, E. *Constraint Propagation with Interval Labels*. Artificial Intelligence, **32**, pp281-331, 1987.

[de Kleer, 1986] de Kleer, J. *An Assumption-based TMS*. Artificial Intelligence, **28**, pp127-162, 1986.

[Evertsz, 1982] Evertsz, R. *A Production System Account of Children's Errors in Fraction Subtraction*. CAL Report #28, The Open University, 1982.

[Evertsz, 1989] Evertsz, R. *The Role of the Crucial Experiment in Student Modelling*. Forthcoming Doctoral Dissertation, The Open University.

[Langley and Ohlsson, 1984] Langley, P. & Ohlsson, S. *Automated Cognitive Modeling*. Proceedings of AAAI-84, pp193-197, 1984.

[Sleeman, 1981] Sleeman, D. *A Rule-based Task Generation System*. Proceedings of 7th International Joint Conference on Artificial Intelligence, pp882-886, 1981.