Quality Issues in the Formal Refinement of KBS

Dr. A. N. Clark
Department of Computing
University of Bradford
Bradford, West Yorkshire, BD7 1DP, UK
e-mail: a.n.clark@comp.brad.ac.uk
tel.: +44 1274 385133

February 22, 1999

To appear in the International Journal of Expert Systems: Research and Applications, 1999.

Abstract

This paper defines a formal framework for the development of Knowledge Based Systems and shows how it can be used to enhance the quality of KBS products. A rule language is defined and given a semantics using non-deterministic state transition systems. The language is used to define the following components of KBS development: specification; implementation; refinement; completeness; consistency; decomposition; and recomposition. The formal framework is used to define and establish properties of three example applications. The paper concludes with a proposal for a KBS development process that uses the framework to facilitate Quality Control.

1 Overview

One of the major aims of Quality Control during software development is to produce a system that meets its specification. The work described in this paper contributes to this area of Quality Control by identifying a simple and widely applicable model for KBS and its development process, by defining properties of KBS specifications, implementations and development steps all of which contribute to the Quality Control process, and by giving examples of Quality Control being applied to various steps in the development of three simple applications.

The literature in the area of KBS development, for example (Rushby, 1988), (Lopez, 1990), (O'Keefe, 1987) and (Culbert, 1987), describes the development of these systems as often being *ad hoc*, where rules are added, modified and deleted until the system produces an acceptable result. In many cases these systems are not specified, but are developed interactively until they produce acceptable outcomes for as many executions as possible. There is often no clear development path leading from a specification to an implementation; as a consequence there is no way of knowing how complete the implementation is with respect to the desired outcomes.

Computer programs perform complex series of atomic steps in order to transform a given input into a particular output. The steps available to a program are given in terms of the hardware that the program runs on. The steps and the data they affect usually have nothing whatsoever to do with the application area addressed by the program. The aim of a specification for a program is to state exactly what the program will do while stating as little as possible about the steps by which the program will achieve it. Given such a specification, it is the developer's task to produce a program that performs all of the steps necessary to achieve the specified results.

We take the following view of software specification and development. A software *specification* is a description of inputs and outputs to be achieved by any implementation meeting the specification. All of the possible inputs and outputs must be precisely described along with the operations used to manipulate them. The specification need not describe any sequences of steps that use the operations to translate a given input into one of its specified outputs. A software *implementation* associates each input/output pair with a sequence of atomic operations corresponding to a calculation that computes the output from the input. The atomic operations are the instructions performed by a machine (concrete or abstract). In general, the operations defined by a software specification will be very much more abstract than those performed by an implementation calculation. A software implementation cannot translate a given input to a given output unless it associates them by means of a sequence of atomic operations.

Software development is a process starting with a software specification and ending with a software implementation. There are various ways of performing the process, for example:

- By gradually adding partial sequences of steps to the specification until an implementation is achieved.

- By starting with all the possible implementations that can be constructed and gradually discarding those that are undesirable.

- By translating specifications into other specifications or translating implementations into other implementations. In general, translations between specifications or implementations (collectively referred to as *system descriptions*) must be shown to preserve some properties.

Even though there are many possible ways to implement the software development process, there are a number of features common to all of them. Firstly, a specification associates a number of the inputs and outputs of the intended system; any description derived from the specification by a software development process must ensure that all of the desired input/output associations are preserved. This property is termed *completeness* and Quality Control must ensure that completeness is preserved throughout the development process in order that no specified behaviour is lost.

Secondly, a specification defines a collection of operations. Each operation is used to manipulate the inputs and outputs. These operations place restrictions on the changes permissible for certain data values in implementations. Each implementation meeting a specification must obey these restrictions and only exhibit sequences of actions using the operations in an allowed order. Also, any of the outputs calculated by an implementation for a given input must have been specified in the first place. This property is termed *consistency* and Quality Control must ensure that consistency is preserved throughout the development process in order that the implementation behaves as desired.

Finally, a development process is intended to produce an implementation that can be executed within reasonable resource limits. Each step in the development process must be seen to reduce the amount of resources required to produce an output for a given input. This property is termed *refinement* and Quality Control must ensure that each step in the development process is a refinement in order that the implementation is acceptable.

A useful distinction between *conventional* and *knowledge based* software is that the former is usually specified in terms of a deterministic specification and implemented in terms of deterministic calculations; whereas the latter is usually specified in terms of a non-deterministic specification and implemented (at least initially) in terms of non-deterministic calculations. Conventional specifications associate at most a single output with each input whereas a KBS specification may be far less specific and associate more than one output with each input. A KBS specification leads to a *search space*, characteristic of many KBS applications. When a search space is flattened to produce calculations, the resulting calculations are branching and result as traces from the execution of a non-deterministic state transition system.

Using non-determinism as the basic characteristic feature of KBS software has implications as far as the software development process is concerned and how the quality of the process is controlled. Whereas conventional specifications state exactly what is required in terms of input/output associations, not all of the associations defined by a KBS specification may be required. A KBS specification must include a statement describing which of the possible associations are legal and which should be disregarded. Furthermore, the KBS specification is associated with a search space which, when flattened, produces the non-deterministic calculations of a KBS implementation. In general, it is not practical to perform all of these calculations in order to achieve only the required input/output associations. Some of the calculations must be thrown away as part of the development process. The initial implementation that calculates all of the input/output pairs non-deterministically is referred to as a *stupidity based system*. The development process must find some way of including mechanisms that preserve only the desirable calculations thereby producing a *knowledge based system*. This is to be contrasted with conventional software development where such mechanisms are not required since the problem is simply to translate a given collection of input/output pairs into an implementation representing a collection of complete and consistent calculations.

Quality Control of the KBS software development process is concerned with increasing the likelihood of producing a correct implementation with acceptable performance. Quality Control of KBS development must check for the following:

- Each step in the development process may delete calculations that compute outputs from given inputs providing that either the specification does not require them or some other calculation already computes them.
- Each step in the development process may change how input/output pairs are computed providing that the calculations are consistent with any restrictions imposed by the specification. Restrictions include the specification of legal operations, the number of steps in a calculation or the number of resources a calculation may use.
- Each step in the development process must not introduce further non-determinism for a given input *i.e.* each step must remove unwanted possible calculations or shorten existing calculations.

The rest of this paper is organised as follows: §2 describes a formalism for defining rule based systems and gives a precise meaning to the components specification, implementation, modification *etc.* §2 also defines a collection of properties that Quality Control of the development process for rule based implementations must preserve or achieve. §3 gives three examples of simple KBS applications, some steps in their development and the Quality Control of the steps. Finally, §4 reviews the material, makes some conclusions with respect to Quality Control of KBS development, analyses related work and suggests areas for further work.

2 Rule based systems

This section describes a formalism for specifying, implementing and modifying rule based systems. This is a modification of non-deterministic conditional ground term rewriting systems (Jounanaud & Dershowitz, 1990) where we extend terms with some useful features for pattern matching and extend the matching process using equational theories. See (Beynon-Davies, 1993), (Lucas

& Van Der Gaag, 1991) or (Luger & Stubblefield, 1989) for a general introduction to rule based systems; see (Goguen, 1976) for an introduction to equational algebra.

2.1 Syntax and semantics

Let V and F be two disjoint sets of variables and function symbols respectively. Each function symbol $f \in F$ has a positive integer arity $arity(f)$.

Definition 1 *A term t is one of the following:*

1. *A variable v. The distinguished variable $\_$ is used when the value is of no interest.*
2. *A term t annotated with an asterisk t^*.*
3. *A variable v and a term t', $v : t'$.*
4. *A variable v and a term t', $v@t'$.*
5. *A function symbol f applied to a sequence of terms $(t_1, \ldots, t_n)$, $f(t_1, \ldots, t_n)$ where $arity(f) = n$.*

A *ground* term contains no variables. The set of variables in a term t is $vars(t)$. A *simple* term is either a variable or a function symbol applied to simple terms. A *substitution* θ is a set of mappings from variables to ground terms $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ and is applied to a variable $\theta(v_i)$ to produce the associated ground term t_i. A substitution is uniquely extended to a term substitution homomorphism:

$$\theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n))$$

As described in (Clark, 1996), the expressivity of rule based systems is greatly increased when the matching is performed modulo equational theories. An equational theory defines the function symbols, their arities and sorts necessary in order to define the terms described above. The equational theory also contains a collection of equations that define equalities between algebraic terms.

Definition 2 *A presentation of an equational theory for lists defines $[]$ as the empty list, $[\_]$ as the singleton list constructor, $\_ +\!\!+ \_$ as the list concatenation operator and the following equations:*

$$(1)\ [] +\!\!+ l = l = l +\!\!+ []$$

$$(A)\ (l_1 +\!\!+ l_2) +\!\!+ l_3 = l_1 +\!\!+ (l_2 +\!\!+ l_3)$$

(1) defines $[]$ as the left and right identity of $\_ +\!\!+ \_$ and (A) defines $\_ +\!\!+ \_$ to be associative. The equations define $[1] +\!\!+ []$ and $[1]$ to be the same lists and

$[1] +\!\!+ ([2] +\!\!+ [3])$ and $([1] +\!\!+ [2]) +\!\!+ [3]$ to be the same lists (in general the parentheses will be omitted).

These equations are taken into account in the equality relation between two list terms and will be used when matching terms against values from the carrier of any algebra satisfying the equational theory.

Definition 3 *A presentation of an equational theory for sets contains $\emptyset$ as the empty set, $\{\_\}$ as the singleton set constructor, $\_ \cup \_$ as set union and the following equations:*

$$\{x\} \cup \{x\} = \{x\}$$

$$(1)\ \emptyset \cup s = s$$

$$(C)\ \ s_1 \cup s_2 = s_2 \cup s_1$$

$$(A)\ \ s_1 \cup (s_2 \cup s_3) = (s_1 \cup s_2) \cup s_3$$

Matching list terms is referred to as $A1$-matching and matching set terms is referred to as $AC1$-matching. Similarly, for bags: $<>$ is the empty bag, $< \_ >$ is the singleton bag constructor, $\_ \frown \_$ is bag union and the equations are $AC1$ (but not union for singleton bags). The notations $[t_1, \ldots, t_n]$, $\{t_1, \ldots, t_n\}$ and $< t_1, \ldots, t_n >$ are used as sugar for the concatenation of singleton lists, sets and bags respectively.

Definition 4 *A boolean expression e is one of the following:*

1. *true.*
2. *false.*
3. *$R(t_1, \ldots, t_n)$ where R is an n-ary relation symbol and t_i are simple terms.*
4. *A conjunction of two boolean expressions $e_1 \& e_2$.*
5. *A disjunction of two boolean expressions $e_1 | e_2$.*
6. *A negation of a boolean expression $\neg e$.*

Definition 5 *A rule r is $t_1 \Rightarrow t_2$* **when** *e where t_1 and t_2 are terms and e is a boolean expression such that $vars(t_2) \subseteq vars(t_1)$ and $vars(e) \subseteq vars(t_1)$.*

The boolean condition of a rule will be omitted when it is *true*. t_1 is referred to as the *antecedent* of the rule and t_2 as the *consequent*. Two rules r_1 and r_2 are sequenced to produce a new rule $r_1; r_2$. When $r_1; r_2$ is applied it performs the non-deterministic transition defined by r_1 followed by the non-deterministic transition defined by r_2. A rule is used to rewrite a ground term by applying it; this is represented as $r(t)$. When it is defined, the result of applying a rule to a ground term can be viewed in two ways: $r(t)$ is the set of all ground terms produced by rewriting t in all possible ways as defined by r; or, $r(t)$ is one of the possible ground terms selected non-deterministically from all the possible ways of rewriting t using r. We explicitly make the distinction between the two views when it is required. The following rule:

$$r_1 = [v_1] + v_2 \Rightarrow v_2$$

is applied to a non-empty list and decapitates it. The following rule:

$$r_2 = v_1 + {v_2}^* + v_3 \Rightarrow v_1 + [v_2] + v_3$$

is applied to a list and reduces an arbitrary number of repetitions of the same element down to a single occurrence. The term ${v_2}^*$ will match a non-empty list of any length whose elements are all the same. For example, if the list is $[2, 2, 2]$ then the term matches the list and binds v_2 to 2. The term will also match singleton lists, for example $[2]$, but not empty lists or lists whose elements are not all the same. When r_2 is applied to the list $[1, 2, 2, 3]$ the possible outcomes are: $[1, 2, 2, 3]$ and $[1, 2, 3]$. The term $v : t$ matches any ground term that also matches t and binds v to the ground term in addition to binding any of the variables in t:

$$r_3 = v_1 : (v_2 + [1]^* + v_3) \Rightarrow v_1 + v_1$$

The rule r_3 is applied to a list containing at least one sublist that is a sequence of 1's. The entire list is bound to the variable v_1 and the consequent appends the list to itself. For example:

$$r_3([2, 1, 1, 1, 3]) = [2, 1, 1, 1, 3, 2, 1, 1, 1, 3]$$

The term $v@t$ is used to rule out certain values. The term t contains variables bound elsewhere in a rule. Given a ground term t', t' matches t and consequently binds t' to v when all substitutions θ such that $\theta(t) = t'$ are disjoint from the substitutions used in the rest of the rule. This mechanism can be used to rule out repetitions:

$$r_4 = (v_1 + [v_2] + v_3, v_4@(\_ + [v_2] + \_)) \Rightarrow (v_1 + v_3, v_4)$$

The rule r_4 is applicable to pairs of lists. When it is applied, the first component of the pair is matched to the pattern $v_1 + [v_2] + v_3$ and arbitrarily selects an element v_2 from the list. The second component of the pair must match the pattern $v_4@(\_ + [v_2] + \_)$. This pattern matches lists from which no element t can be arbitrarily selected such that t is the same value as is bound to v_2 from the first component of the pair, *i.e.* v_4 can be bound only to lists that do not contain v_2.

A rule set R is either empty $\emptyset$, a singleton rule $\{r\}$ or a union of two rule sets $R_1 \cup R_2$. The set of all sequences of rules in a rule set R is denoted R^*, for example

$$\{r_1, r_2, r_3\}^* = \{r_1, r_1; r_1, r_1; r_2, \ldots, r_2; r_1, r_2; r_2, \ldots\}$$

Definition 6 *The semantics of rule sets is defined as:*

$$
\begin{array}{l}
[\![\emptyset]\!] = \{\} \\
[\![\{r\}]\!] = [\![r]\!] \\
[\![R_1 \cup R_2]\!] = [\![R_1]\!] \otimes [\![R_2]\!] \\
P \otimes Q = P \cup Q \cup \{t_1 +\!\!+ [(t_2,t_3)] +\!\!+ [(t_3,t_4)] +\!\!+ t_5 \mid t_1 +\!\!+ [(t_2,t_3)] \in P \otimes Q \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [(t_3,t_4)] +\!\!+ t_5 \in P \otimes Q\}
\end{array}
$$

where $P \otimes Q$ is the smallest set satisfying the equation.

The meaning of a rule set is all of the possible sequences of state transformations produced by non-deterministically applying the rules in the set any number of times to any ground term. The operator $\otimes$ is used to concatenate the sequences of transformations from two different rule sets: each sequence is concatenated on a shared ground term t_3. If $[(t_1, \_)] +\!\!+ \_ +\!\!+ [(\_, t_2)] \in [\![R]\!]$ then let $t_1 \mapsto^* t_2$ be a calculation relative to R. Where $[(t_1, t_2)] \in [\![R]\!]$ then let $t_1 \mapsto t_2$ be a calculation relative to R. A rule set R may be viewed as a non-deterministic function applicable to ground terms $R(t)$ the result of which is viewed either as the entire set of possible outcomes of applying rules in R^* (see below) or as a single element selected at random from that set.

Definition 7 *The semantics of terms is defined as:*

$$
\begin{array}{l}
[\![\_]\!] = \{(x,\emptyset) | x \in G\} \\
[\![v]\!] = \{(x,\theta) | x \in G \wedge \theta = \{v \mapsto x\}\} \\
[\![v^*]\!] = \{([x,\ldots,x],\theta) | x \in G \wedge \theta = \{v \mapsto x\}\} \\
[\![v : t]\!] = \{(x,\theta) | (x,\theta') \in [\![t]\!] \wedge \theta = \theta' \cup \{v \mapsto x\}\} \\
[\![v@t]\!] = \{(x, \theta \cup \{v \mapsto x\}) | \{(x,\theta_1),\ldots,(x,\theta_n)\} \in [\![t]\!]/G \wedge \theta = not(\theta_1 \cup \ldots \cup \theta_n)\} \\
[\![f(t_1,\ldots,t_n)]\!] = \{(f(x_1,\ldots,x_n),\theta) | (x_i,\theta_i) \in [\![t_i]\!] \wedge \theta = \bigcup_{i=1,n} \theta_i \wedge consistent(\theta)\}
\end{array}
$$

The meaning of a term t is a set of values (t', θ) where t' is a ground term and θ is a substitution. The set G contains all possible ground terms. The notation P/Q denotes a set constructed by partitioning the set P containing values (t, θ) into a collection of sets:

$$\{\{(t_1,\theta_{11}),(t_1,\theta_{12}),\ldots\},\{(t_2,\theta_{21}),(t_2,\theta_{22}),\ldots\},\ldots\}$$

for every term $t_i \in Q$. The operator *not* applies to a substitution and marks all of the mappings in the substitution as being *illegal*, *n.b.* all mappings not illegal are *legal*. Applying *not* to the same mappings more than once has no further effect. The predicate *consistent* holds for all substitutions that, for all variables v, contains at most one legal mapping, $v \mapsto t$, and for each illegal mapping, $v \mapsto t'$, the value t' cannot also be legal, $t \neq t'$.

Definition 8 *The semantics of boolean expressions is defined as:*

$[\![true]\!]_\theta = \mathbf{true}$
$[\![false]\!]_\theta = \mathbf{false}$
$[\![R(t_1, \ldots, t_n)]\!]_\theta = (\theta(t_1), \ldots, \theta(t_n)) \in R$
$[\![e_1 \& e_2]\!]_\theta = ([\![e_1]\!]_\theta)\,\mathbf{and}([\![e_2]\!]_\theta)$
$[\![e_1 | e_2]\!]_\theta = ([\![e_1]\!]_\theta)\,\mathbf{or}([\![e_2]\!]_\theta)$
$[\![\neg e]\!]_\theta = \mathbf{not}([\![e]\!]_\theta)$

The meaning of a boolean expression is either **true** or **false** and is produced by evaluating the expression with respect to a substitution θ.

Definition 9 *The semantics of a single rule is defined as:*

$$\begin{aligned}
[\![t_1 \Rightarrow t_2 \textbf{ when } e]\!] = \{[(x_1, x_2)] | &(x_1, \theta_1) \in [\![t_1]\!] \wedge \\
&(x_2, \theta_2) \in [\![t_2]\!] \wedge \\
&[\![e]\!]_{\theta_1 \cup \theta_2} \wedge \\
&consistent(\theta_1 \cup \theta_2)\}
\end{aligned}$$

$$[\![r_1; r_2]\!] = [\![r_1]\!] \oplus [\![r_2]\!]$$
$$P \oplus Q = \{l_1 + [(t_1, t_2)] + [(t_2, t_3)] + l_2 | l_1 + [(t_1, t_2)] \in P \wedge [(t_2, t_3)] + l_2 \in Q\}$$

The meaning of a single rule is a set of sequences of pairs produced by taking the meanings of antecedent and consequent terms and pairing up ground terms with consistent mapping sets for which the boolean expression holds. Where rules are sequenced using _; _ the meaning is constructed by concatenating the meanings of the individual rules.

2.2 Specification, refinement and modularity

The development of a rule based system can be viewed as a process whose input is a specification of *what* the system must achieve and whose output is an implementation describing *how* the system achieves it. Quality Control is an activity that minimises the risk of the development process going wrong by checking at each stage of the process that certain properties have been achieved or retained. The development process goes wrong when the output of the development process is incomplete, inconsistent or is not an acceptable refinement of the input to the process. This section gives precise definitions for the components of the development process, the activities that the process performs and the properties to be maintained by Quality Control.

2.2.1 Specification and implementation

The specification of a rule based system is (T, I, O, M, S) where T is the set of ground terms, some of which are states, over which the rule based system will compute, $I \subseteq T$ is the set of initial states, $O \subseteq T$ is the set of terminal states, M is a rule set describing the atomic moves that can be applied to the values in T and $S \subseteq I \times O$ is a set of initial/terminal state pairs that are the inputs and outputs for the specified system.

An implementation for a rule based system is a rule set R. There are two conditions which R must satisfy if it claims to implement the rule based system specification (T, I, O, M, S): consistency and completeness. Informally, an implementation is consistent when it only uses the moves that are allowed by the specification and when all of its halting calculations starting with an initial state $t_1 \in I$ and producing a terminal state $t_2 \in O$ are allowed by the specification. Informally, an implementation is complete when it performs calculations satisfying all of the pairs in S. These definitions are made precise as follows:

Definition 10 *An implementation R of a rule based specification* (T, I, O, M, S) *is* complete *when*

$$\forall (t_1, t_2) \in S \ \exists [(t_1, \_)] +\!\!+ \_ +\!\!+ [(\_, t_2)] \in [\![R]\!]$$

Definition 11 *An implementation R of a rule based specification* (T, I, O, M, S) *is* consistent *when*

- $\forall [(t_1, \_)] +\!\!+ \_ +\!\!+ [(\_, t_2)] \in [\![R]\!] \ (t_1 \in I) \wedge (t_2 \in O) \Rightarrow (t_1, t_2) \in S$
- $\forall r \in R \ \exists m \in M^* \ \forall t \in T \ r(t) \subseteq m(t)$

A specification must be shown to be *complete* and this is essentially the same as showing that the specification when viewed as an implementation is complete. A specification is viewed as an implementation simply by viewing the set of legal moves as a rule base. This is referred to as the *initial* implementation.

Definition 12 *A specification* (T, I, O, M, S) *is* complete *when the initial implementation M is complete with respect to the specification.*

Definition 13 *A specification* (T, I, O, M, S) *is* consistent *when the initial implementation M is consistent with respect to the specification.*

The implementations of rule based systems are developed by applying a sequence of modifications. A modification may introduce new rules, delete old rules or replace rules with new ones. The idea of a modification is to reduce the amount of non-determinism present in the rule base system. Initially, a specification is shown to be complete. This guarantees that all of the specified input/output pairs are computed by applying some sequence of moves. The specification does not indicate which particular sequence of moves to use. The initial implementation of a specification uses the moves at random to guarantee fairness. In general, there will be many sequences of moves not leading to a solution. Some sequences may never terminate at all. A modification will throw away sequences which:

- do not lead to a solution; or
- never terminate; or

- lead to a solution in an unacceptable number of moves.

Definition 14 *Given an implementation R of a rule based system, a* modification $R \longmapsto R'$, *producing rule set R', is:*

1. *A* deletion *when $R' \cup \{r\} = R$.*
2. *An* addition *when $R \cup \{r\} = R'$.*
3. *A* replacement *when $R' \cup \{r\} = R \cup \{r_1, r_2, \ldots, r_n\}$*

2.2.2 Refinement of rule based systems

The development process from software specification to implementation is generally referred to as *refinement* (Morgan, 1990). The particular formalisms used to represent software descriptions and to transform them will differ from application to application, but in general refinement is defined to preserve or achieve various properties at each step in the development process. Morgan (1990) describes a refinement system based upon viewing a software system in terms of preconditions on its inputs and postconditions on its outputs, both of these are described using predicate calculus. A refinement step is a modification to the system description weakening the preconditions and strengthening the postconditions. Such a view of software is very general and abstracts away from the computational mechanisms used to compute the specified relations. This paper can afford to take a more restricted view of software since the computational mechanism is known in advance *i.e.* rule transitions. The problem addressed here is not the search for an implementation mechanism, but the *control* of a given implementation mechanism. A specification is viewed as a description of a non-deterministic system without any regard for the efficiency of the calculations it performs. Refinement is viewed as the process by which a specification is incrementally modified in order to increase the efficiency of the calculations by reducing the number of steps that are taken and reducing the amount of choice available at each step.

The meaning of a rule set, $[\![R]\!]$ is defined as a set of all the chains that can possibly be performed when it is used to translate a ground term. The meaning may also be viewed as a directed graph, (N, E, s, t), as follows:

$$
\begin{array}{l}
N = T \\
\forall x_1, x_2 \in N \;\; \_ \mathbin{+\!\!+} [(x_1, x_2)] \mathbin{+\!\!+} \_ \in [\![R]\!] \Leftrightarrow \exists e \in E \;\; s(e) = x_1 \wedge t(e) = x_2 \\
\forall e_1, e_2 \in E \;\; t(e_1) = s(e_2) \Rightarrow \exists e \in E \;\; s(e) = s(e_1) \wedge t(e) = t(e_2)
\end{array}
$$

where each ground term is defined as a node of the graph, each transition is an edge and E is the smallest set satisfying the conditions. A graph also contains the transitive closure of the edges between nodes, where $e_1; e_2$ represents an edge from $s(e_1)$ to $t(e_2)$ when $t(e_1) = s(e_2)$. Graph *homomorphisms* are used to transform one graph into another. Given two graphs, $G_1 = (N_1, E_1, s_1, t_1)$

and $G_2 = (N_2, E_2, s_2, t_2)$, a graph homomorphism $\phi : G_1 \to G_2$ is a pair of mappings: $\phi_n : N_1 \to N_2$ and $\phi_e : E_1 \to E_2$ for which the following two diagrams commute:

$$
\begin{tikzcd}
E_1 \arrow[r, "s_1"] \arrow[d, "\phi_e"'] & N_1 \arrow[d, "\phi_n"] \\
E_2 \arrow[r, "s_2"'] & N_2
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
E_1 \arrow[r, "t_1"] \arrow[d, "\phi_e"'] & N_1 \arrow[d, "\phi_n"] \\
E_2 \arrow[r, "t_2"'] & N_2
\end{tikzcd}
$$

A refinement is defined to be a syntactic transformation on a rule set. The transformation is constrained by the following relationship between the semantics of the rule set before and after the transformation. Let R_1 and R_2 be rule sets, the transformation applied to R_1 in order to produce R_2 is a *refinement*, $R_1 \rightsquigarrow R_2$, when there exists a graph homomorphism ϕ such that $\phi : [\![R_2]\!] \to [\![R_1]\!]$, *i.e.* some possible values have been "forgotten" (since they are not required); some transitions have been "forgotten" (since they do not compute a desired result or they are duplicated elsewhere); the result of the modification is *consistent* (when the result does something it agrees with the original rule set); and the number of edges between nodes is reduced (computations between values are made more efficient).

The following is a simple example of a graph transformation for which there exists a homomorphism. The rule set R_1 gives rise to tree-shaped calculations with the states $1, 2, 3, 4, 5, 6, 7$ and single-step transitions a, b, c, d, e, f. Completeness defines that any modification must contain states 4 and 7. The rule set R_2 is a modification of R_1 giving rise to states $1, 2, 4, 5, 6, 7$ and transitions a, b, d, e, g:

$$
[\![R_1]\!] =
\begin{tikzcd}
 & & 1 \arrow[dl, "a"'] \arrow[dr, "b"] & & \\
 & 2 \arrow[dl, "c"'] \arrow[d, "d"] & & 6 \arrow[d, "e"'] \arrow[dr, "f"] & \\
3 & 4 & & 5 & 7
\end{tikzcd}
\qquad
[\![R_2]\!] =
\begin{tikzcd}
 & 1 \arrow[dl, "a"'] \arrow[d, "b"'] \arrow[dr, "g"] & \\
2 \arrow[d, "d"'] & 6 \arrow[d, "e"'] & 7 \\
4 & 5 &
\end{tikzcd}
$$

the modification is a refinement because the following pair of mappings:

$$\phi_n = \{1 \mapsto 1, 2 \mapsto 2, 4 \mapsto 4, 5 \mapsto 5, 6 \mapsto 6, 7 \mapsto 7\}$$

$$\phi_e = \{a \mapsto a, b \mapsto b, d \mapsto d, e \mapsto e, g \mapsto b; f\}$$

is a homomorphism $\phi : [\![R_2]\!] \to [\![R_1]\!]$ and the completeness criterion is satisfied. The effect of the modification is to remove state 3, thereby reducing non-determinacy at state 2. Non-determinacy at state 6 is also reduced, seemingly at the expense of increasing non-determinacy at state 1 although on closer inspection this is not the case since the new edge g expands to a pair $b; f$ in $[\![R_1]\!]$. The length of the calculation reaching state 7 is reduced by introducing the new edge g.

Definition 15 *A modification m between two rule sets R_1 and R_2 is a* refinement *$R_1 \rightsquigarrow R_2$ when the following diagram commutes:*

$$\begin{tikzcd} R_1 \arrow[r, "{[\![\cdot]\!]}"] \arrow[d, "m"'] & G_1 \\ R_2 \arrow[r, "{[\![\cdot]\!]}"'] & G_2 \arrow[u, "\phi"'] \end{tikzcd}$$

where $[\![.]\!]$ is the semantic function for rule sets, G_1 and G_2 are the meanings of R_1 and R_2 represented as graphs and ϕ is a graph homomorphism.

The definition of refinement given above allows rule set modifications to increase the efficiency of an implementation by deleting states and transitions and by "telescoping" sequences of steps in a rule based calculation down into a single step. Since the refinement step is allowed to forget information, the definition of completeness must be used at each stage in order to ensure no vital information is forgotten. Since a refinement step can only ever produce a new implementation consistent with the result of the previous step, then providing the original implementation is consistent with the specification the result of applying any refinement step will also be consistent.

2.2.3 Modularity of rule based system refinement

Any development process is enhanced by the ability to decompose the job into a number of separate units that can be developed individually and then composed when each has been completed. Modular development can facilitate Quality Control since the measurement of progress with respect to smaller units is much easier than with larger units. However the Quality Control procedures must also be applied to the decomposition and recomposition steps in order to verify that they are appropriate. Quality Control of modularity therefore rests on a precise definition of the steps and components that are involved in breaking a large specification or implementation down into separate units and then recomposing them at some later date.

The decomposition of rule based systems can occur with respect to two (semi-) independent criteria:

- Syntactic decomposition of a rule set into different rule sets where the ground terms rewritten by the rule sets do not change; and

- Semantic decomposition of the set of ground terms into different sets where the rule set that rewrites the ground terms does not change.

In order to define a method for KBS development it is necessary to be precise regarding the meaning of decomposition and subsequent recomposition. This section defines a framework for these activities and makes use of structural properties of ground terms arising from equational theories to show how modularity can be achieved for the applications that follow.

Given a rule set R and two rule sets R_1 and R_2 we would like to define the conditions under which R is the composition of R_1 and R_2. Consider the semantics of these rule sets with respect to the same set of ground terms. The graphs $\llbracket R_1 \rrbracket$ and $\llbracket R_2 \rrbracket$ are both *contained* in the graph $\llbracket R \rrbracket$. Furthermore, the graph $\llbracket R \rrbracket$ is the *smallest* graph that contains both of the component graphs. This can be defined precisely, by stating that $\llbracket R \rrbracket$ is the co-product of $\llbracket R_1 \rrbracket$ and $\llbracket R_2 \rrbracket$.

Definition 16 *Let $\llbracket R_1 \rrbracket$ and $\llbracket R_2 \rrbracket$ be two graphs. A co-product of these graphs $\llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket$ is a graph with two injection graph homomorphisms $i_1 : \llbracket R_1 \rrbracket \to \llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket$ and $i_2 : \llbracket R_2 \rrbracket \to \llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket$ such that for any graph G with homomorphisms $f : \llbracket R_1 \rrbracket \to G$ and $g : \llbracket R_2 \rrbracket \to G$ there exists a unique graph homomorphism $u : \llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket \to G$ such that the following diagram commutes:*

$$\begin{tikzcd}
\llbracket R_1 \rrbracket \arrow[dr, "i_1"] \arrow[ddr, "f"'] & & \llbracket R_2 \rrbracket \arrow[dl, "i_2"'] \arrow[ddl, "g"] \\
& \llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket \arrow[d, "u"] & \\
& G &
\end{tikzcd}$$

If the decomposition of rule sets is syntactic and not semantic then we can use the definition of co-products to provide a framework for modularity in system development. For example, consider the decomposition of a rule set R into R_1 and R_2 such that $\llbracket R \rrbracket = \llbracket R_1 \rrbracket + \llbracket R_2 \rrbracket$. The rule sets are refined independently using $\phi_1 : \llbracket R_1' \rrbracket \to \llbracket R_1 \rrbracket$ and $\phi_2 : \llbracket R_2' \rrbracket \to \llbracket R_2 \rrbracket$. Finally, the refined rule sets are combined to produce R' such that $\llbracket R' \rrbracket = \llbracket R_1' \rrbracket + \llbracket R_2' \rrbracket$. Then by the definition of co-products there exists a unique $u : \llbracket R' \rrbracket \to \llbracket R \rrbracket$ making the following diagram commute:

$$\begin{tikzcd}
\llbracket R'_1 \rrbracket \arrow[d, "\phi_1"'] \arrow[dr] & & \llbracket R'_2 \rrbracket \arrow[d, "\phi_2"] \arrow[dl] \\
\llbracket R_1 \rrbracket \arrow[dr] & \llbracket R' \rrbracket \arrow[d, "u"] & \llbracket R_2 \rrbracket \arrow[dl] \\
& \llbracket R \rrbracket &
\end{tikzcd}$$

Unfortunately, this definition does not universally apply when the decomposition is semantic because the ground term sets that label the nodes in the component graphs ($\llbracket R_1 \rrbracket$ and $\llbracket R_2 \rrbracket$) are not sub-sets of the ground term set that labels the co-product graph. The rest of this section presents a framework for investigating semantic decomposition defined in terms of of projection mappings from the ground term set of a rule set R to ground term sets of rule sets R_1 and R_2 that are semantic decompositions of R.

Definition 17 *The semantics of a rule set R may be parameterised with respect to a set of ground terms T where T is closed with respect to all of the possible ground terms produced by rewriting any element $t \in T$. The semantics is represented as $\llbracket R \rrbracket_T$. The graph $\llbracket R \rrbracket_T$ contains all edges $n \to n$ that loop on a node. These edges correspond to applying the identity rule to all ground terms in T.*

Parameterising a rule set with respect to ground terms that it rewrites allows us to express modifications to the ground terms during semantic decomposition. A method of semantic decomposition for any rule set R whose ground terms are the carrier of a freely generated monoid (for example lists or sets) is to systematically forget elements from the freely generating set. Imagine running a rule set R that rewrites lists of integers and ignoring all occurrences of the integer 1. Ignoring all elements from two subsets of the freely generating set will produce a semantic decomposition of R.

Definition 18 *Let $\llbracket R_1 \rrbracket$ and $\llbracket R_2 \rrbracket$ be two graphs. A product of these graphs is a graph $\llbracket R_1 \rrbracket \times \llbracket R_2 \rrbracket$ with two graph homomorphisms $\pi_1 : \llbracket R_1 \rrbracket \times \llbracket R_2 \rrbracket \to \llbracket R_1 \rrbracket$ and $\pi_2 : \llbracket R_1 \rrbracket \times \llbracket R_2 \rrbracket \to \llbracket R_2 \rrbracket$ such that for any graph G with graph homomorphisms $f : G \to \llbracket R_1 \rrbracket$ and $g : G \to \llbracket R_2 \rrbracket$ there exists a unique graph homomorphism*

$u : G \to \llbracket R_1 \rrbracket \times \llbracket R_2 \rrbracket$ *such that the following diagram commutes:*

$$\begin{tikzcd}
& G \arrow[ddl, "f"'] \arrow[d, "u"] \arrow[ddr, "g"] & \\
& \llbracket R_1 \rrbracket \times \llbracket R_2 \rrbracket \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
\llbracket R_1 \rrbracket & & \llbracket R_1 \rrbracket
\end{tikzcd}$$

Consider two rule sets R_1 and R_2 with respect to ground term sets T_1 and T_2 giving rise to graphs $\llbracket R_1 \rrbracket_{T_1}$ and $\llbracket R_2 \rrbracket_{T_2}$. Let $\llbracket R \rrbracket_T$ be a product of these two graphs. Let $\phi_1 : \llbracket R'_1 \rrbracket_{T'_1} \to \llbracket R_1 \rrbracket_{T_1}$ and $\phi_2 : \llbracket R'_2 \rrbracket_{T'_2} \to \llbracket R_2 \rrbracket_{T_2}$ be two refinements and let $\llbracket R' \rrbracket_{T'}$ be a combination of the refined graphs. By the definition of products there is a unique refinement $\phi : \llbracket R' \rrbracket_{T'} \to \llbracket R \rrbracket_T$ such that the following diagram commutes:

$$\begin{tikzcd}
& \llbracket R' \rrbracket_{T'} \arrow[dl] \arrow[d, "u"] \arrow[dr] & \\
\llbracket R'_1 \rrbracket_{T'_1} \arrow[d, "\phi_1"'] & \llbracket R \rrbracket_T \arrow[dl] \arrow[dr] & \llbracket R'_2 \rrbracket_{T'_2} \arrow[d, "\phi_2"] \\
\llbracket R_1 \rrbracket_{T_1} & & \llbracket R_2 \rrbracket_{T_2}
\end{tikzcd}$$

The diagram above shows the framework for semantic decomposition. Starting with a rule set R we semantically decompose it to product R_1 and R_2 by projecting the term set T to product T_1 and T_2. Next, we refine R_1 and R_2 to product R'_1 and R'_2 respectively. Finally, we recompose the rule sets to produce R' that is guaranteed to be a refinement of R providing all the components of the diagram are correctly defined.

This section concludes by showing how we can use the properties of freely generated monoids to construct products of rule sets whose ground term sets are lists, sets and bags. These definitions will be used in the applications of rule set refinement that follow.

Definition 19 *Let T be a set of ground terms. Let $L(T)$ be the freely generated monoid of lists whose elements are drawn from T, i.e. $L(T) = (T^*, +\!\!+, [])$ where T^* is the carrier of the monoid (a set of sequences of elements of T), $+\!\!+$ is* *list append and [] is the empty list. Similarly, let $S(T)$ be the freely generated monoid of sets based on T and $B(T)$ be the freely generated monoid of bags based on T. Since all the results that follow are equivalent for $L(T)$, $S(T)$ and $B(T)$ we will use $M(T)$ to represent any monoid freely generated based on T.*

Definition 20 *Let $M_1 = (S_1, \oplus, e_1)$ and $M_2 = (S_2, \otimes, e_2)$ be two monoids. A monoid homomorphism $\phi : M_1 \to M_2$ is a mapping from S_1 to S_2 such that:*

- $\phi(e_1) = e_2$; *and*
- $\phi(v_1 \oplus v_2) = \phi(v_1) \otimes \phi(v_2)$

Consider two sets T_1 and T_2 such that $T_1 \subseteq T_2$. An element of the carrier of $M(T_2)$ can always be projected to become an element of the carrier of $M(T_1)$ by 'forgetting' elements of T_2 that are not elements of T_1.

Definition 21 *Let $T_1 \subseteq T_2$ be two sets of ground terms and let $m : T_2 \to T_1 \cup \{e\}$ be a mapping defined as follows:*

$$m(t) = \begin{cases} t & when\ t \in T_1 \\ e & otherwise \end{cases}$$

The mapping m can be uniquely extended to be a homomorphism from $M(T_2)$ to $M(T_1)$ that forgets the elements of T_2 not also in T_1. In general, a monoid homomorphism based on the projection between two sets is represented as $\Pi(T_1, T_2) : M(T_2) \to M(T_1)$.

Theorem 1 *Let R be a rule set that rewrites ground terms that are elements of the carrier of a monoid. Let T_1 and T_2 be two ground term sets and let $M_1 = M(T_1)$, $M_2 = M(T_2)$ and $M_3 = M(T_1 \cup T_2)$ be freely generated monoids based on the corresponding sets. The graph $[\![R]\!]_{M_3}$ is a product of graphs $[\![R]\!]_{M_1}$ and $[\![R]\!]_{M_2}$.*

The projections $\pi_1 : [\![R]\!]_{M_3} \to [\![R]\!]_{M_1}$ and $\pi_2 : [\![R]\!]_{M_3} \to [\![R]\!]_{M_2}$ are based on the monoid homomorphisms $\Pi(T_1, T_1 \cup T_2)$ and $\Pi(T_2, T_1 \cup T_2)$. Given a graph $[\![R]\!]_{M(T)}$ and graph homomorphisms $f : [\![R]\!]_{M(T)} \to [\![R]\!]_{M_1}$ and $g : [\![R]\!]_{M(T)} \to [\![R]\!]_{M_2}$ then there exists a unique graph homomorphism $u : [\![R]\!]_{M(T)} \to [\![R]\!]_{M_3}$ based on the unique monoid homomorphism $\Pi(T_1 \cup T_2, T)$. $\square$

The definitions in this section give a precise meaning to the development process of simple rule based systems. Quality Control is then defined as the activity that checks at each stage of the development process that the properties of consistency and completeness are maintained and that each step in the development is a refinement. The definitions are quite simple and are specifically tailored to address the example applications in the next section. In order to address more sophisticated applications, the definitions will undoubtedly have to be modified and extended, for example to incorporate real-time information.

3 Examples of rule based systems

This section uses the mechanisms developed in the previous section to construct specifications and implementations of three example applications. In each case the specifications are shown to be complete. During the development process of rule based software, Quality Control is responsible for ensuring that each modification to the current implementation preserves the desired properties of consistency and completeness and achieves a refinement. In each of the three example applications, a collection of representative modifications is presented along with proofs of various properties. Modifications that fail are shown in addition to modifications that succeed.

This section is structured as follows. Section 3.1 specifies and refines the Blocks World. Section 3.2 specifies and refines the game of Mastermind. Section 3.3 specifies and refines the Towers of Hanoi. The Blocks World system is treated in depth in order to show how a complete refinement is performed. The subsequent systems are sketched in order to give examples of different types of application.

3.1 Blocks World

The Blocks World system (Nilsson, 1980) involves stacks of labelled blocks on a table top. A legal move is to remove a block from the top of one of the available stacks and to place it on the top of any available stack. The goal of the Blocks World is to arrive at a configuration where a particular block is directly on top of another particular block.

3.1.1 Specification and completeness

The specification for the Blocks World is (T, I, O, M, S) where T is a many sorted algebra whose distinguished sort is a set of lists of blocks, where a block is to be represented as an integer. A typical state of the Blocks World is $\{[1] + [2]\} \cup \{[3] + [4]\}$ This describes two towers of blocks on a table top, block 1 is on block 2 and block 3 is on block 4. A state is legal when it contains all possible blocks exactly once. The set I of initial states is the same as T. The set O of terminal states contains only those configurations of blocks where block 1 is directly on top of block 2 (these blocks have been arbitrarily chosen without any loss of generality). The set M of legal moves is $\{r_1, I\}$ where I is the identity move that corresponds to removing a block and placing it back on the same tower and:

$$r_1 = \{[v_1] + v_2\} \cup \{v_3\} \cup v_4 \Rightarrow \{v_2\} \cup \{[v_1] + v_3\} \cup v_4$$

takes a block off a tower and places it on a different tower (or the table). The set S contains pairs of initial and terminal states and is defined as $S = I \times O$.

Given such a specification, it must be shown that the available moves are complete with respect to the specified input/output pairs. One way to do this is to show that any state can be transformed into any other state using the moves defined by M. This is proved using the following collection of lemmas and theorems. For any Blocks World specification there is a sub-specification that contains the blocks 1 and 2 but contains a subset of the original blocks. For any Blocks World specification, there is a unique state where all the available blocks exist as towers by themselves, this is referred to as an initial object of the Blocks World. If t_1 and t_2 are initial Blocks World objects from different specifications then the state $t_1 \cup t_2$ is an initial object from some super-specification.

Lemma 1 *Given two calculations $t_1 \mapsto^* t'_1$ and $t_2 \mapsto^* t'_2$ then there exists a calculation $t_1 \cup t_2 \mapsto^* t'_1 \cup t'_2$.*

The proof is by contradiction. Assume the converse is true, *i.e.* for some step in a calculation $t_1 \mapsto t'_1$ it is not possible to derive the following calculation $t_1 \cup t_2 \mapsto t'_1 \cup t_2$. This is false for the identity rule I. It is also false for r_1 since the variable v_4 can stand for any set and does not change over the transition. Therefore any two calculations may be merged by performing successive transitions randomly from each. □

Lemma 2 *Given an initial object for a Blocks World system t_0, there exists a finite sequence of moves from any state t to t_0, i.e. $t \mapsto^* t_0$.*

The proof is by induction on the structure of a finite Blocks World state:

1. If the state is $t_1 \cup t_2$ then we assume by induction that $t_1 \mapsto^* t'_1$ and $t_2 \mapsto^* t'_2$ where $t'_1 \cup t'_2 = t_0$. Now by lemma 1 if $t_1 \mapsto^* t'_1$ and $t_2 \mapsto^* t'_2$ then there exists a finite sequence of steps $t_1 \cup t_2 \mapsto t_0$.

2. If the state is $\emptyset$ then since $\emptyset$ is the initial object of the empty Blocks World specification, the single move is $I(\emptyset)$.

3. If the state is $\{t\}$ then there are three cases:

   (a) If the state is $\{[]\}$ then the move is $I(\{[]\})$.

   (b) If the state is $\{[t']\}$ then the move is $I(\{[t']\})$.

   (c) If the state is $\{[t_1] + t_2\}$ then assume by induction that $\{t_2\} \mapsto^* t'_2$ such that t'_2 is a terminal object. Now combine the single move $\{[t_1] + t_2\} \mapsto^* \{[t_1]\} \cup \{t'_2\}$ to produce a finite sequence of moves.

□

Lemma 3 *Given any sequence of moves $t \mapsto^* t'$ then there exists a sequence of moves $t' \mapsto^* t$.*

Proof is by contradiction. Assume the contrary, *i.e.* that no move is reversible. This is false for the identity rule I. It is also false for r_1 since $[\![\{I\}]\!] \subseteq [\![\{r_1; r_1\}]\!]$. $\square$

Corollary 1 *Given any sequence of moves $t \mapsto^* t_0$ there exists a sequence of moves $t_0 \mapsto^* t$. This follows directly from lemma 3.*

Theorem 2 *Given any two Blocks World states t and t' there exists a sequence of moves $t \mapsto^* t'$.*

Starting with t and t' there exists a sequence of moves $t \mapsto^* t_0$ and $t' \mapsto^* t_0$ by lemma 2. Given $t' \mapsto^* t_0$ there exists a sequence of moves $t_0 \mapsto^* t'$ by corollary 1. Given two sequences of moves $t \mapsto^* t_0$ and $t_0 \mapsto^* t'$ then there exists a sequence of moves $t \mapsto t'$. $\square$

Theorem 3 *The Blocks World specification is complete.*

The Blocks World specification is complete when for any pair $(t_1, t_2) \in S$ there exists a sequence of moves $t_1 \mapsto^* t_2$. Since there exists a sequence of moves between any pair of Blocks World states by theorem 2, the specification is complete. $\square$

3.1.2 Implementation and modification

Now consider an implementation of Blocks World and a modification to the implementation. We wish to ensure that the modification leads to an implementation consistent with the original specification and that the modification is a refinement, *i.e.* it leads to an implementation that computes the desired function (potentially) more efficiently.

We will address three types of modification: addition, deletion and replacement. In each case we present a modification that could occur during the development of a Blocks World implementation. The modification will be related to the definitions of completeness and refinement in order to determine some guidelines as to modifications which both preserve completeness and refine the implementation.

Firstly, consider starting with an implementation containing the two moves given in the specification. It is noticed that given a state of the form:

$$\{[1, \ldots], [2, \ldots], \ldots\}$$

it is always possible to get to the desired result in a single deterministic move. Let

$$r_2 = \{[1] \mathbin{+\!\!+} v_1, [2] \mathbin{+\!\!+} v_2\} \cup v_3 \Rightarrow \{[1] \mathbin{+\!\!+} [2] \mathbin{+\!\!+} v_2, v_1\} \cup v_3$$

then we wish to show that the following modification $\{r_1, I\} \longmapsto \{r_1, r_2, I\}$ is complete and a refinement.

Theorem 4 $\{r_1, I\} \longmapsto \{r_1, r_2, I\}$ *is complete and a refinement.*

The proof follows directly from the definition of the rule r_2 since $[\![r_2]\!] \subseteq [\![r_1]\!]$. It follows that $\{r_1, r_2, I\}$ is complete when $\{r_1, I\}$ is complete and the identity mapping can be used as a graph homomorphism which embeds $\{r_1, r_2, I\}$ viewed as a graph into $\{r_1, I\}$ viewed as a graph. □

Suppose that we want to add a rule. We must also show that it is consistent with the original specification. The following is an example of a modification leading to an inconsistency. Let

$$r_3 = \{[v_1] \mathbin{+\!\!+} v_2\} \cup v_3 \Rightarrow \{v_2\} \cup v_3$$

then the modification $\{r_1, I\} \longmapsto \{r_1, r_3, I\}$ is inconsistent with the original specification.

Theorem 5 $\{r_1, r_3, I\}$ *is inconsistent.*

The proof shows that a calculation step possible using $\{r_1, r_3, I\}$ is not allowed by the original specification. Consider, $[(\{[x_1] \mathbin{+\!\!+} x_2\} \cup x_3, \{x_2\} \cup x_3)]$ that is a step possible by the rule set $\{r_1, r_3, I\}$. Notice that the ground term x_1 is removed from the state. Given the initial specification $M = \{r_1, I\}$, there is no sequence of rules $m \in M^*$ that produce this calculation step. We therefore conclude that $\{r_1, r_3, I\}$ is inconsistent with respect to the original specification. □

Deleting a rule will only lead to an implementation that is complete with respect to the specification when the rule's behaviour is duplicated elsewhere in the implementation. For example, given the initial implementation of the Blocks World based on the specification, the following modification $\{r_1, I\} \longmapsto \{I\}$ is trivially incomplete since it leaves only the identity move.

Rules are replaced in order to make them more specific, *i.e.* to ignore calculations that do not lead to a solution, that never terminate or are not optimal. The replacements must be shown to produce an implementation consistent and complete with respect to the specification and more specific than the implementation before the modification. For example, let

$$\begin{aligned} r_4 = &\{[v_1] \mathbin{+\!\!+} v_2, v_3\} \cup v_4 \Rightarrow \\ &\quad \{v_2, [v_1] \mathbin{+\!\!+} v_3\} \cup v_4 \\ &\quad\quad \textbf{when} \ \ (\mathit{covered}(1, \{[v_1] \mathbin{+\!\!+} v_2, v_3\} \cup v_4))| \\ &\quad\quad\quad\quad (\mathit{covered}(2, \{[v_1] \mathbin{+\!\!+} v_2, v_3\} \cup v_4)) \end{aligned}$$

where the boolean expression, *covered*(t_1, t_2), is true when the state t_2 contains a stack of blocks where t_1 is covered.

The implementation $\{r_1, r_2, I\}$ contains two duplicate moves: when r_2 is applicable, there is an application of r_1 one of whose outcomes will produce the same result. The problem is that the application of r_1 will lead to many other outcomes that are not optimal. The following modification $\{r_1, r_2, I\} \longmapsto \{r_2, r_4, I\}$ replaces rule r_1 with one that performs the same moves except for the move to be performed by r_2.

Theorem 6 *$\{r_2, r_4, I\}$ is complete.*

The proof is by induction on the length of the calculation. Suppose that we have a calculation $x_1 \mapsto^* x_2$ such that $x_2 \in O$. We must show that the given rule set produces the calculation. Consider x_1:

1. if blocks 1 and 2 are both uncovered then rule r_2 is used to produce a state in O.
2. if block 1 is covered then let $x_1 \mapsto x_1'$ be the step produced by rule r_4. By induction we assume that $x_1' \mapsto x_2$ is produced by the rule set.
3. if block 2 is covered then let $x_1 \mapsto x_1'$ be the step produced by rule r_4. By induction we assume that $x_1' \mapsto x_2$ is produced by the rule set.

□

Theorem 7 *$\{r_1, r_2, I\} \longmapsto \{r_2, r_4, I\}$ is a refinement.*

The proof follows from observing that $\{r_2, r_4, I\}$ is complete and therefore there exists a mapping from the graph of $\{r_2, r_4, I\}$ to the graph of $\{r_1, r_2, I\}$. □

Theorem 8 *$\{r_2, r_4, I\}$ is consistent.*

The consistency of the rule set follows from its completeness since there is no move that can be made by $\{r_2, r_4, I\}$ that cannot be made by $\{r_1, r_2, I\}$. □

3.1.3 Modularity

Although the Blocks World system is very simple, the development process from specification to satisfactory implementation is less structured that it could be. This section follows the method defined in §2.2.3. The method can be used in general to break down a specification into simpler components. Each component may be developed individually and then re-composed to give an implementation for the overall specification. The ability to decompose a specification into smaller modules is important for a Quality Control point of view. Such a decomposition into atomic units provides a basis for many analyses such as resource estimation, risk analysis, size estimation and reuse. In addition, it is claimed that the development of smaller modules is less complex in most cases than the development of larger modules. The framework for decomposition is as follows:

1. A specification is decomposed into two sub-specifications using two systematic and independent rules that "forget" information. It is important that the rules allow the resulting sub-specifications to be re-composed in such a way as to produce the original specification.

2. Step (1) is repeated until each specification component is at the required level of detail.

3. Each specification is then treated separately. For example, each specification may be refined or compared against a library of specifications containing cost information.

4. The results from each component are then composed. The rules of composition should be shown to be complete and consistent. The re-composed rule set must be a refinement of the decomposed rule set.

The rules of decomposition for the Blocks World scenario allow blocks to be forgotten from a specification. For example, if the specification of Blocks World involves blocks 1, 2, 3, 4 and 5 (with the aim of placing block 1 on block 2) then one decomposition rule is "forget block 3". In general we are interested in reducing the specification to atomic sub-specifications containing blocks 1, 2 and one other block.

Given the rules of decomposition above, the rules of composition must take two sub-specifications and produce the specification that was "projected" to produce them. In general this is not a one-to-one mapping because more than one specification state can be projected to produce the same pair of sub-specification states. For example, given tables $\{[1],[2],[3]\}$ and $\{[1],[2],[4]\}$ then the following are possible mergings: $\{[1],[2],[3],[4]\}$, $\{[1],[2],[3,4]\}$ and $\{[1],[2],[4,3]\}$. Notice also that some pairs of sub-specification states do not have a corresponding merging: $\{[1],[2],[3]\}$ and $\{[1,2],[4]\}$ since there is no state that can be projected onto both.

Given rules of composition and decomposition for the states in a specification (or implementation), we must also have rules for composing rule sets. Consider the refinement of two sub-specifications. Each refinement changes the calculations performed by each of the two corresponding implementations by forgetting some of the steps and merging other steps together. The calculations are systematically merged using the rules of state composition. Rule sets must be merged in order to give rise to the corresponding merged calculations.

The decomposition of the Blocks World scenario for five blocks is shown in figure 1 where the specifications are as follows:

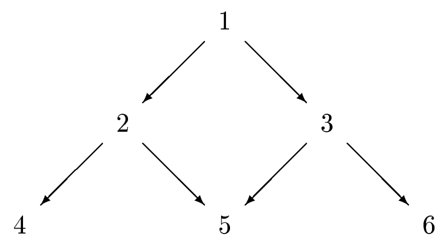

Figure 1: Blocks World Decomposition

1. The specification of a Blocks World with the following blocks: 1, 2, 3, 4 and 5.
2. The specification of a Blocks World with the following blocks: 1, 2, 3 and 4, *i.e.* the same as (1) except block 5 has been forgotten.
3. The specification of a Blocks World with the following blocks: 1, 2, 3 and 5, *i.e.* the same as (1) except block 4 has been forgotten.
4. The specification of a Blocks World with the following blocks: 1, 2, 4, *i.e.* the same as (2) except block 3 has been forgotten.
5. The specification of a Blocks World with the following blocks: 1, 2, 3, *i.e.* the same as (2) except block 4 has been forgotten and the same as (3) except block 5 has been forgotten.
6. The specification of a Blocks World with the following blocks: 1, 2, 5, *i.e.* the same as (3) except block 3 has been forgotten.

Each specification occurring at the leaves of the decomposition tree is now refined. Consider the specification 5 containing the blocks 1, 2 and 3. Since there are so few blocks, all the possible configurations can be enumerated and refinement will involve modifying the rule set to handle a subset of the possible configurations. The possible configurations are as follows:

(C_1)	$\{[1,2,3]\}$	(C_7)	$\{[1],[2,3]\}$	(C_{13})	$\{[1],[2],[3]\}$
(C_2)	$\{[1,3,2]\}$	(C_8)	$\{[1],[3,2]\}$		
(C_3)	$\{[2,1,3]\}$	(C_9)	$\{[2],[1,3]\}$		
(C_4)	$\{[2,3,1]\}$	(C_{10})	$\{[2],[3,1]\}$		
(C_5)	$\{[3,2,1]\}$	(C_{11})	$\{[3],[1,2]\}$		
(C_6)	$\{[3,1,2]\}$	(C_{12})	$\{[3],[2,1]\}$		

Each of the configurations are inspected in turn and a rule is devised as to the best move (or moves) in each case. This produces a collection of rules, each one of which is specific to a particular configuration:

$$\begin{array}{ll}
(r_1) & \{[1],[2],[3]\} \Rightarrow \{[1,2],[3]\} \\
(r_2) & \{[1],[2,3]\} \Rightarrow \{[1,2,3]\} \\
(r_3) & \{[2],[1,3]\} \Rightarrow \{[1,2],[3]\} \\
(r_4) & \{[2],[3,1]\} \Rightarrow \{[1],[2],[3]\} \\
(r_5) & \{[3],[2,1]\} \Rightarrow \{[1],[2],[3]\} \\
(r_6) & \{[2,1,3]\} \Rightarrow \{[2],[1,3]\} \\
(r_7) & \{[3,1,2]\} \Rightarrow \{[3],[1,2]\} \\
(r_8) & \{[2,3,1]\} \Rightarrow \{[2],[3,1]\} \\
(r_9) & \{[3,2,1]\} \Rightarrow \{[3],[2,1]\} \\
(r_{10}) & \{[1],[3,2]\} \Rightarrow \{[1],[2],[3]\} \\
(r_{11}) & \{[1,3,2]\} \Rightarrow \{[1],[3,2]\}
\end{array}$$

Now the rules can be merged in order to take account of similarities in the moves. It should be stressed that although the rules given here are complete with respect to all of the configurations that are possible in the specification, this need not be the case. In many Knowledge Based Systems applications, complete knowledge will not be available or will be infeasible for controlling an atomic specification component. For example, in a plant controller, the atomic unit of specification may be a physical component such as a valve with a number of possible states depending on whether it is open or closed, *why* it is open and closed and the recent alarm history. Rules govern whether or not to change the state of the valve given the current state *e.g.* alarm 12 is ringing, alarm 43 has just stopped and tank 15 is being drained. Given the number of possible configurations for the valve, it is unlikely that a designer can be complete with respect to the atomic rules governing the component.

Rules are merged to produce new rules by introducing variables where the patterns in the rules differ. The constraint on the result of the merge is that it must be complete and consistent with respect to the rest of the specification, *i.e.* after a merge there may be no new moves and any moves which could be performed before the merge can also be performed after the merge. Consider rules r_1 and r_2 that both move block 1 onto block 2. These rules are merged to produce the following rule:

$$r_{12} = \{[1],[2] +\!\!+ v_1\} \cup v_2 \Rightarrow \{[1,2] +\!\!+ v_1\} \cup v_2$$

where the variable v_1 corresponds to the constant $[]$ in r_1 and $[3]$ in r_2 and the variable v_2 corresponds to $\{[3]\}$ in r_1 and $\emptyset$ in r_2. Merging r_{12} with rule r_3 produces the following rule:

$$r_{13} = \{[1] +\!\!+ v_1, [2] +\!\!+ v_2\} \cup v_3 \Rightarrow \{[1,2] +\!\!+ v_2, v_1\} \cup v_3$$

that places block 1 on block 2 when they are both free in any configuration. Rules r_4 and r_5 both clear block 1 and are merged to produce the following rule:

$$r_{14} = \{[v_1,1],[v_2]\} \Rightarrow \{[v_1],[1],[v_2]\}$$

and merges with r_6 to produce:

$$r_{15} = \{[v_1, 1] + v_2\} \cup v_3 \Rightarrow \{[v_1], [1] + v_1\} \cup v_3$$

The rule r_7 merges with r_{15} to produce r_{15} since the move r_7 can make is already covered by r_{15}. The rule r_8 merges with r_{15} to produce the following:

$$r_{16} = \{[v_1] + v_2 + [1] + v_3\} \cup v_4 \Rightarrow \{[v_1], v_2 + [1] + v_3\} \cup v_4$$

clearing block 1 in any configuration. Similarly rules r_7, r_9, r_{10} and r_{11} merge to produce the following rule:

$$r_{17} = \{[v_1] + v_2 + [2] + v_3\} \cup v_4 \Rightarrow \{[v_1], v_2 + [2] + v_3\} \cup v_4$$

clearing block 2 in any configuration. The following table ensures that the rules are complete with respect to performing the best move in all possible configurations:

Configuration	Rule
C_1	Terminal
C_2	r_{17}
C_3	r_{16}
C_4	r_{16}
C_5	r_{16}, r_{17}
C_6	r_{16}, r_{17}
C_7	r_{13}
C_8	r_{17}
C_9	r_{13}
C_{10}	r_{16}
C_{11}	Terminal
C_{12}	r_{16}
C_{13}	r_{13}

The table is correct providing that the rules are shown to be complete, consistent and a refinement. The proof of these properties follows from the discussion in §3.1.1. Since all of the configurations are catered for, the original non-deterministic move can be deleted from the rule set.

Each of the leaf-specifications 4, 5 and 6 shown in figure 1 will produce a rule set implementation using the analysis shown above. The rule set $\{r_{13}, r_{16}, r_{17}\}$ will, in each case, refer to the blocks which are available in the specification. It remains to merge the rule set implementations in order to produce a refinement of the overall specification with five blocks.

In order to merge two rule sets, we take the product of all the elements in the rule set. Consider merging rule sets from implementations for blocks 1, 2 and 3 and blocks 1, 2 and 4. This will produce the following rule set:

$$\{r_{13} \times r_{13}, r_{13} \times r_{16}, r_{13} \times r_{17}, r_{16} \times r_{13}, r_{16} \times r_{16}, r_{16} \times r_{17}, r_{17} \times r_{13}, r_{17} \times r_{16}, r_{17} \times r_{17}\}$$

The criteria for merging two rules is defined as follows: let r_1 and r_2 be two rules, and let P and Q be the semantics of the rules, *i.e.* sets of pairs of ground terms. For any pair of ground terms t_1 and t_2, there is a set of terms T such that t_1 is the left projection of every term $t \in T$ and t_2 is the right projection of every term $t \in T$. The product of a rule r is a rule set R such that the semantics of the rules in R correspond to the products:

$$\{(t_x, t_y) \mid t_x \in t_1 \times t_3 \wedge t_y \in t_2 \times t_4 \wedge (t_1, t_2) \in P \wedge (t_3, t_4) \in Q\}$$

In the Blocks World, the product of a pair of terms is formed by replacing the forgotten blocks (3 and 4). The product of the rules $r_{13} \times r_{13}$ is simply the rule set $\{r_{13}\}$ since the positions of the blocks 1 and 2 are fixed and the rest of the positions of any blocks are represented by variables. The products $r_{13} \times r_{16}$ and $r_{13} \times r_{17}$ are the empty rule sets since both products require blocks 1 and 2 to be in two positions at once. Note that since the blocks 3 and 4 are not required to be in any positions by the rules r_{13}, r_{14} and r_{17} then the product is a commutative operator: $r_i \times r_j = r_j \times r_i$. The product $r_{16} \times r_{16}$ includes the rule:

$$\begin{array}{l}\{([v_1] \times [v_5]) + (v_2 \times v_6) + [1] + (v_3 \times v_7)\} \cup (v_4 \times v_8) \Rightarrow \\ \qquad (\{[v_1]\} \times \{[v_5]\}) \cup \{(v_2 \times v_6) + [1] + (v_3 \times v_7)\} \cup (v_4 \times v_8)\end{array}$$

The rule is of interest when the variables v_1 and v_5 correspond to the missing blocks 3 and 4. When this is the case there is ambiguity as to the order in which the blocks are "popped" off the head of the stack containing block 1 and where the blocks end up, *i.e.* $[3] \times [4]$ could be $[3, 4]$ or $[4, 3]$ and $\{[3]\} \times \{[4]\}$ could be $\{[3, 4]\}$, $\{[4, 3]\}$ or $\{[3], [4]\}$. At this point we can perform a slight amount of extra refinement by outlawing the rules that produce $\{[3, 4]\}$ and $\{[4, 3]\}$, this conveniently leaves $r_{16} \times r_{16} = r_{16}$. Similarly $r_{17} \times r_{17} = r_{17}$. The product of r_{16} and r_{17} is a little more complex and leads to the following rules:

$$\begin{array}{l}r_{18} = \{[v_1] + v_2 + [1] + v_3 + [2] + v_4\} \cup v_5 \Rightarrow \{[v_1], v_2 + [1] + v_3 + [2] + v_4\} \cup v_5 \\ r_{19} = \{[v_1] + v_2 + [2] + v_3 + [1] + v_4\} \cup v_5 \Rightarrow \{[v_1], v_2 + [2] + v_3 + [1] + v_4\} \cup v_5 \\ r_{20} = \begin{array}{l}\{[v_1] + v_2 + [1] + v_3\} \cup \{[v_4] + v_5 + [2] + v_6\} \Rightarrow \\ \quad \{[v_1], [v_4], v_2 + [1] + v_3, v_5 + [2] + v_6\}\end{array} \\ r_{21} = \{[3, 4] + v_1 + [1] + v_2 + [2] + v_3\} \Rightarrow \{[3, 4], v_1 + [1] + v_1 + [2] + v_3\} \\ \vdots\end{array}$$

When this list is complete it can be viewed as a refinement of the rule set $\{r_{16}, r_{17}\}$. Conveniently, the product of the rule set $\{r_{13}, r_{16}, r_{17}\}$ with itself is a refinement of the same rule set: $\{r_{13}, r_{16}, r_{17}\}$. Notice that this will not always be the case for different applications. Also notice that although the product operator would appear to have produced the same rule set, the types of the variables in the rules are different, *i.e.* they stand for values containing the blocks 1, 2, 3 and 4. Exactly the same analysis will lead to the rule set for implementation 1 in figure 1 being $\{r_{13}, r_{16}, r_{17}\}$.

This section has described the decomposition, refinement and re-composition of a simple rule based system. Each step of the development is based on formally defined transformations and can be justified with respect to desirable system properties such as completeness and consistency. The proposed method contributes to the Quality Control process of rule based system development by allowing each step of the process to be checked.

3.2 Mastermind

The game of Mastermind involves two players. The first player chooses a sequence of colours and the second player has to guess the correct sequence. The first player knows how many colours have been chosen but not the colours or the sequence. The game proceeds by turns: the first player (the codebreaker) selects a sequence of colours and the second player (the codemaker) informs the first as to how many correct colours have been chosen and whether or not they are in the correct places. The codemaker does this using a scoring system involving two colours: black and white. The number of blacks indicates how many colours guessed correctly and in the correct position in the sequence. The number of whites indicates how many colours guessed correctly but in the wrong position in the sequence.

3.2.1 Specification and completeness

The specification of Mastermind is (T, I, O, M, S) where T is a many-sorted term algebra, the distinguished sort is the set of triples (t_1, t_2, t_3) where t_1 is a set of guesses, t_2 is a set containing the outcomes for all possible guesses, and t_3 is a sequence of outcomes. A guess is a sequence of colours, *e.g.* [*green*, *red*, *pink*]. All guesses are sequences of the same length and repetition of colours is allowed in a guess. An outcome is a pair (t_1, t_2) where t_1 is a guess and t_2 is a bag of scores. A score may be black $\mathbf{b}$, meaning that one of the colours was guessed correctly in the right position in the sequence, or white $\mathbf{w}$, meaning that one of the colours was guessed correctly but in the wrong position.

The set I is a singleton $\{(t_1, t_2, [])\}$ where t_1 is the set of all possible guesses that can be made and t_2 is used as a function to determine the outcome of a guess when it is made. The set O contains states of the form $(t_1, t_2, [(t_3, t_4)] +\!\!\!+ t_5)$ where t_1 is the set of all possible guesses, t_2 is the outcome function, t_3 is a guess, t_4 is a bag whose cardinality is the length of t_3 and all of whose elements are $\mathbf{b}$. The sequence t_5 is a sequence of previous guesses and outcomes, (t_6, t_7) for which either the cardinality of t_7 is not equal to the length of t_6 or t_7 contains $\mathbf{w}$. The set M is $\{r_1\}$ where:

$$r_1 = (\{v_1\} \cup v_2, \{(v_1, v_3)\} \cup v_4, v_5) \Rightarrow (\{v_1\} \cup v_2, \{(v_1, v_3)\} \cup v_4, [(v_1, v_3)] +\!\!\!+ v_5)$$

The rule r_1 non-deterministically selects a guess v_1 from the available set of guesses and adds the outcome (v_1, v_3) to the list of outcomes.

The set S contains all of the pairs that must be computed by any implementation of the Mastermind specification. It simply associates all of the possible inputs with all of the possible outputs: $S = I \times O$.

Theorem 9 *The specification of Mastermind is complete.*

Starting with the single initial state $(t_1, t_2, [])$ the outcome of $r_1(t_1, t_2, [])$ is a set of states; either:

1. $(t_1, t_2, [(t_3, < \mathbf{b}, \ldots, \mathbf{b} >)]) \in r_1(t_1, t_2, [])$; or
2. $(t_1, t_2, [(t_3, t_4)]) \in r_1(t_1, t_2, [])$ where $|t_4| \neq \# t_3$ or $\mathbf{w} \in t_4$

If the outcome is the former then $(t_1, t_2, < \mathbf{b}, \ldots, \mathbf{b} >) \in O$. Now suppose that there is a list of outcomes that are not correct and not produced by r_1. This would mean that there is no sequence of r_1 applications that selects a given sequence of guesses before hitting on the correct one. This must mean there are some guesses that cannot be selected after a given number of applications of r_1. But, from the definition of r_1, this is not the case: an application may select any guess and all guesses are preserved over an application. Therefore the specification is complete. □

3.2.2 Implementation and modification

Now consider an implementation of Mastermind and some modifications to the implementation. We start with the initial implementation derived from the specification and investigate whether various modifications are complete, consistent and are refinements.

By inspecting the initial implementation, it is detected that if a guess is made containing all the same colour and the outcome from the guess contains at least one black result then the colour should be chosen again in subsequent guesses. This is expressed by the following rule:

$$r_2 = (\{v_1 : \_ + \!\!\! + [v_2] + \!\!\! + \_\} \cup v_3, \{(v_1, v_4)\} \cup v_5, v_6 : (\_ + \!\!\! + [(v_2^*, < \mathbf{b} > \frown \_)] + \!\!\! + \_)) \Rightarrow$$

$$(\{v_1\} \cup v_3, \{(v_1, v_4)\} \cup v_5, [(v_1, v_4)] + \!\!\! + v_6)$$

Theorem 10 $\{r_1, r_2\}$ *is complete.*

If we view the two rules as deterministic functions then it is easy to see that whenever $r_2(t) = P$ for some state t and set of states P, $r_1(t) = Q$ such that $P \subseteq Q$. Since the implementation $\{r_1\}$ is complete and adding the rule r_2 does not include any new moves, $\{r_1, r_2\}$ is also complete. □

Also from the initial implementation, it is detected that if a guess is made producing a null outcome, then none of the colours included in the guess will be in the correct code. Once a guess is made producing the null outcome, then no subsequent guess should contain any of the colours in the code that failed. This is expressed by the following rule:

$$r_3 = (\{v_1 : (\_ \mathbin{+\!\!+} [v_2] \mathbin{+\!\!+} \_)\} \cup v_3, \{(v_1, v_4)\} \cup v_5, v_6@(\_ \mathbin{+\!\!+} [(\_ \mathbin{+\!\!+} [v_2] \mathbin{+\!\!+} \_, <>)] \mathbin{+\!\!+} \_)) \Rightarrow$$

$$(\{v_1\} \cup v_3, \{(v_1, v_4)\} \cup v_5, [(v_1, v_4)] \mathbin{+\!\!+} v_6)$$

Theorem 11 $\{r_1, r_3\}$ *is complete.*

Viewing the rules as deterministic functions: whenever $r_3(t) = P$, $r_1(t) = Q$ such that $P \subseteq Q$. This is because r_3 only picks guesses containing colours in the code, whereas r_1 selects guesses indiscriminately. Notice that when a state s does not contain failures, such as initial states in I, then r_1 and r_3 will be equivalent. □

Now suppose that the implementation $\{r_1, r_3\}$ is modified by deleting the original rule r_1. This is justified on the basis that r_3 is a better rule and will arrive at a terminal state in a potentially shorter number of steps.

Theorem 12 $\{r_3\}$ *is complete.*

The proof relies on showing that the calculations arriving at a terminal state do not rely on any of the moves that are deleted. Consider the possible moves using r_1 but which are not possible using the rule r_3. These moves test colours already shown not to be part of the code. If such a move is part of a calculation contributing to the completeness of $\{r_1, r_3\}$ then there is always a move defined by r_3 contributing more information, and therefore must contribute to the completeness of $\{r_3\}$. Therefore, since no critical moves have been deleted and since $\{r_1, r_3\}$ is known to be complete, the given modification is complete. □

Consider the following rule:

$$r_4 = (\{v_1\} \cup v_2, \{(v_1, v_3)\} \cup v_4, v_5@(\_ \mathbin{+\!\!+} [(v_1, \_)] \mathbin{+\!\!+} \_)) \Rightarrow$$

$$(\{v_1\} \cup v_2, \{(v_1, v_3\} \cup v_4, [(v_1, v_3)] \mathbin{+\!\!+} v_5)$$

The rule r_4 states that a guess is chosen only if it has not been tried before. The modification of the initial implementation $\{r_1\} \longmapsto \{r_1, r_4\}$ is complete by a similar argument to that for the addition of the rule r_3. Similarly, the modification $\{r_1, r_4\} \longmapsto \{r_4\}$ is complete by a similar argument to the deletion of the rule r_3: the complete calculations do not depend upon deleted moves. Consider the following two rules:

$$r_5 = (\{v_1 : \_ +\!\!+ [v_2] +\!\!+ \_\} \cup v_3, \{(v_1, v_4)\} \cup v_5, v_6 : \_ +\!\!+ [(\_ +\!\!+ [v_2] +\!\!+ \_, \_)] +\!\!+ \_) \Rightarrow$$

$$(\{v_1\} \cup v_3, \{(v_1, v_4)\} \cup v_5, [(v_1, v_4)] +\!\!+ v_6)$$

$$r_6 = (\{v_1 : \_ +\!\!+ [v_2] +\!\!+ \_\} \cup v_3, \{(v_1, v_4)\} \cup v_5, v_6 @ \_ +\!\!+ [(\_ +\!\!+ [v_2] +\!\!+ \_, \_)] +\!\!+ \_) \Rightarrow$$

$$(\{v_1\} \cup v_3, \{(v_1, v_4)\} \cup v_5, [(v_1, v_4)] +\!\!+ v_6)$$

Rule r_5 only selects guesses containing colours tried before. Rule r_6 only selects guesses containing colours not tried before.

Starting with the initial implementation $\{r_1\}$, it is argued that for any move performed by the implementation, each colour in the guess will either have been selected before or it will not have been selected before. This means that the implementation $\{r_5, r_6\}$ is complete.

The rule set $\{r_5\}$ is not complete because it cannot be applied to any initial state. The rule r_5 can only be applied to states containing previous guesses, whereas all initial states contain [] as the guess history.

The rule set $\{r_6\}$ is not complete, but in a weaker sense. Although it is capable of eventually finding a correct guess, this is not guaranteed. One of the possible guesses for each application of r_6 will be correct, but if the incorrect path is chosen then the inability to repeat a colour could prevent the correct code from ever being guessed.

3.2.3 Modularity

In order to decompose Mastermind we must choose how individual states and their components are to be projected. There are two orthogonal attributes that may be projected: the colours and their positions.

Consider two projections applied to a system X with two positions and two colours. The first projection reduces two positions two a single position. The resulting sub-systems, X_1 and X_2 each have two colours but only one position.

The second projection reduces two colours to a single colour and is applied to both X_1 and X_2 producing sub-systems X_{11}, X_{12}, X_{21} and X_{22}. States of the final sub-systems all have one of the following forms:

$$(\{[c]\}, \{([c], <\mathbf{b}>)\}, t_1)$$
$$(\{[c]\}, \{([c], <\mathbf{w}>)\}, t_2)$$
$$(\{[c]\}, \{([c], <>)\}, t_3)$$

where c is the single colour and $t_1 - t_3$ are sequences of choices and outcomes. The rule r_1 will achieve completeness and consistency for each sub-system.

Consider finding the product of X_{11} and X_{12}. Let c_1 and c_2 be the colours in each sub-system respectively. The original system contained both colours but only one position, its states are therefore of the form:

$$\begin{array}{l}
(\{[c_1],[c_2]\},\{([c_1],<\mathbf{b}>),([c_2],<\mathbf{w}>)\},\ldots) \\
(\{[c_1],[c_2]\},\{([c_1],<\mathbf{b}>),([c_2],<>)\},\ldots) \\
(\{[c_1],[c_2]\},\{([c_1],<>),([c_2],<>)\},\ldots) \\
\ldots
\end{array}$$

where each of the components are formed by composing the corresponding elements from the states of X_{11} and X_{12}.

Finally, consider finding the product of $X_{11} \times X_{12}$ and $X_{21} \times X_{22}$. Each system has states as described above. The product has states of the following form:

$$\begin{array}{l}
(\{[c_1,c_1],[c_1,c_2],[c_2,c_1],[c_2,c_2]\},\{([c_1,c_1],<>),\ldots\},\ldots) \\
\ldots
\end{array}$$

Notice that in each case $r_1 = r_1 \times r_1$, *i.e.* each sub-system is complete and consistent with respect to the single rule r_1. This section has described a method of decomposition for Mastermind. This method can be used as a basis for the structured refinement of the specification.

3.3 Towers of Hanoi

The Towers of Hanoi (Grogono & Nelson, 1982) is a system involving three pegs and a collection of rings all of different sizes and which fit over the pegs. Initially, all of the rings are on a single peg and they are ordered with the smallest ring at the top of the peg and the largest ring at the bottom. The aim is to move rings from one peg to another in order to finish with all of the rings on a different peg from the start. Rings may only be moves one at a time and only smaller rings may be placed on top of larger rings.

3.3.1 Specification and completeness

A specification for the Towers of Hanoi is (T, I, O, M, S) where T is a set of ground terms of the form:

$$\{(x_1,x_2),(x_3,x_4),(x_5,x_6)\}$$

where x_1, x_3 and x_5 are all sequences of rings and x_2, x_4 and x_6 are all unique peg identifiers. Rings are totally ordered and each sequence of rings in a state must contain rings in decreasing order. The set of initial states is:

$$I = \{\{(x_1,x_2),([],x_3),([],x_4)\} | \mathit{ordered}(x_1) \wedge x_2 \neq x_3 \neq x_4\}$$

The set of terminal states O contains the same set of states, $O = I$. The set M contains two possible moves $M = \{r_1, I\}$ where I is the identity move and r_1 is defined:

$$\begin{aligned} r_1 = &\{([v_1] +\!\!\!+\, v_2, v_3), (v_4, v_5)\} \cup v_6 \Rightarrow \\ &\quad \{(v_2, v_3), ([v_1] +\!\!\!+\, v_4, v_5)\} \cup v_6 \\ &\qquad \textbf{when } \mathit{ordered}([v_1] +\!\!\!+\, v_4) \end{aligned}$$

The set M contains the identity move I since it is possible to remove a ring from a peg and to place it back onto the same peg.

The set S contains all the pairs that must be computed by any implementation of the Towers of Hanoi specification. The set is defined as follows:

$$S = \{(t_1, t_2) \mid t_1 \in I \wedge t_2 \in O \wedge t_1 \neq t_2\}$$

specifying that any implementation must be able to move a tower of rings from any peg to any other peg.

Theorem 13 *The Towers of Hanoi specification is complete.*

We show by induction than it is always possible to reach a terminal state from a starting state. Suppose that we are in a situation where the source peg is p_1, the target peg is p_2 and p_3 is free. Let all the rings on p_2 be smaller than those on p_1. By induction we assume that we can move the rings on peg p_2 to peg p_3. The top ring from p_1 is moved to p_2 and then by induction we assume that rings from p_3 are moved back to p_2. This leaves us one step closer to a terminal state and we can assume that by repeated application of this process we can move all the rings from p_1 to p_2. Since we have used only simple steps that are possible using r_1 then we conclude that the specification is complete. □

3.3.2 Implementation and modification

Now consider an implementation of the Towers of Hanoi and some modifications to the implementation. We start with the implementation based on the specification and investigate whether various modifications are complete, consistent and are refinements.

The initial implementation will not identify final moves as special. A final move occurs when all but the smallest ring has been moved on to the target peg. When a final move is possible, the initial implementation will move the smallest ring either to the target peg or to the empty peg. The following rule identifies the final move in which the target peg is on the right and the smallest ring is in the middle as being special:

$$r_2 = \{([], v_1), ([v_2], v_3), (v_4, v_5)\} \Rightarrow \{([], v_1), ([], v_3, [v_2] +\!\!\!+\, v_4, v_5)\}$$

Given such a rule, it can be added to the initial implementation $\{r_1, I\} \longmapsto \{r_1, r_2, I\}$

Theorem 14 $\{r_1, r_2, I\}$ *is complete.*

The rule set is complete since $[\![r_2]\!] \subseteq [\![r_1]\!]$. □

Consider the following rule that leads to an illegal modification $\{r_1, I\} \longmapsto \{r_1, r_3, I\}$:

$$r_3 = \{v_1, ([v_2] + v_3, v_4), v_5\} \Rightarrow \{v_1, ([v_2, v_2] + v_3, v_4), v_5\}$$

The modification is illegal because the rule r_4 is not consistent with respect to the specification.

Theorem 15 $\{r_1, r_3, I\}$ *is inconsistent.*

Clearly, there are no rules in the original specification allowing a ring to be duplicated. Therefore, rule r_3 performs an illegal step. □

Suppose that it is postulated that being able to swap rings at the top of pegs will lead to a refinement. The following rule:

$$r_4 = \{v_1, ([v_2, v_3] + v_4, v_5), v_6\} \Rightarrow \{v_1, ([v_3, v_2] + v_4, v_5), v_6\}$$

is added to the initial implementation $\{r_1, I\} \longmapsto \{r_1, r_4, I\}$. The addition is illegal because the modification is inconsistent.

Theorem 16 $\{r_1, r_4, I\}$ *is inconsistent.*

From the initial specification of the Towers of Hanoi, a state contains no duplicate rings and rings are totally ordered on pegs. So, if a peg matches the pattern $[v_2, v_3] + v_4$ then $v_2 \neq v_3$ therefore $v_2 < v_3$. When the order is reversed, the peg $[v_3, v_2] + v_4$ is no longer ordered. The rule may be applied to a valid state and produce an invalid state: therefore it is inconsistent. □

Suppose that it is postulated that being able to extract a ring from under the top ring of a peg will lead to a refinement. The following rule:

$$\begin{aligned} r_5 = &\{v_1, ([v_2, v_3] + v_4, v_5), (v_6, v_7)\} \Rightarrow \\ &\{v_1, (v_2 + v_4, v_5), ([v_3] + v_6, v_7)\} \\ &\textbf{when } \textit{ordered}([v_3] + v_6) \end{aligned}$$

A proof of this would need to show that there is a sequence of applications of rule r_1 equivalent to a single application of r_5 for any values $v_1 - v_7$.

3.3.3 Modularity

The Towers of Hanoi can be decomposed as follows. Given a system containing n disks we can produce a pair of sub-systems each containing 1 and $n-1$ disks respectively where the single disk sub-system contains the largest disk. Call these sub-systems X_1 and X_2 respectively. The rule r_1 may be used in both X_1 and X_2 to achieve completeness and consistency.

The product of X_1 and X_2 is a system containing all of the rings from X_2 with an extra, larger ring added in all the positions satisfying the ordering constraint. The product $r_1 \times r_1 = r_1$ and therefore the sub-system decomposition method described in this section can be used as the basis for refinement of the Towers of Hanoi.

4 Conclusion

This paper has described a view of software development involving the translation of a specification into an implementation using formal reasoning. The components of the process have been described in general and then made specific for Knowledge Based Systems. The characteristic feature of knowledge based systems is non-determinism, included at the outset of the process in the specification.

The aims of this work are to identify Quality Control procedures with respect to formal development of KBS. This has been achieved by giving a formal definition for the components of the KBS development process and defining a collection of properties to be preserved and achieved by the steps in the process. These properties are *completeness*, *consistency* and *refinement*.

4.1 Elements of quality control

One of the aspects of Quality Control is to increase the likelihood that the development process will achieve an implementation that performs the required job. The work presented in this paper suggests that the following mutually dependent components will contribute towards the Quality Control process:

- The components of the process must be defined clearly. For example, §2 defines what it means by the terms *specification*, *implementation*, *rule set*, *etc.* Although the definition of the components alone will not guarantee the desired level of quality, a clear statement of their information content provides an important check during development. Note that the definitions given in this paper were tailored for the example applications. A different application area may use its own definitions for specification, for example by including timing information or memory limits.

- There must be a clearly defined method for decomposing complex specifications and implementations. Where possible the method must allow the decomposition steps to be atomic and composable *i.e.* the developer can control the decomposition of a system description to the smallest level of detail. Once decomposed, the steps in the development method must apply to all the leaves in the decomposition tree. There must be a clearly defined method for re-composing the leaves of the tree so that development of the entire system can be viewed as the development of many small atomic system components.

- The steps in the process must be defined clearly. For example, §2 defines what it means by the term *modification* as applied to a rule set. Together with the components of the process described above, the steps provide a clear picture of what is allowed to happen during software development, and as such provide a useful check for each stage during the process.

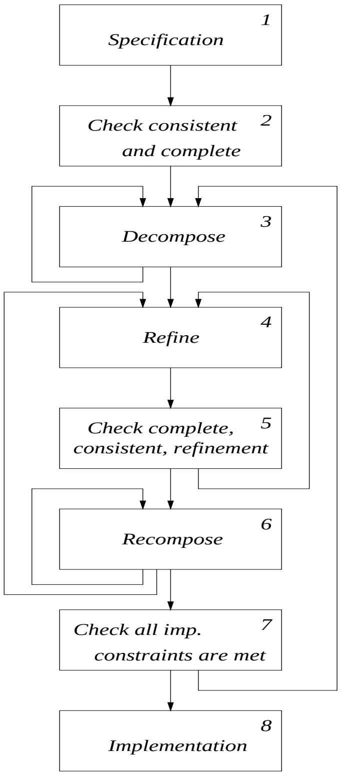

Figure 2: A Quality Control Process

- The properties to be achieved or preserved by the steps in the development process must be defined clearly. Together with the previous two definitions, this is the final element necessary to perform Quality Control in the manner described above. Using definitions of legal development components, the properties define the development steps needed to achieve the required results. For example, §2 defines *completeness*, *consistency* and *refinement* as desirable properties and then gives modifications that are both legal and illegal with respect to these properties and the given example applications.

Figure 2 shows a development process based on the results of this work. The process involves the basic ideas of describing KBS programs as rule sets interpreted by non-deterministic state transition machines. The development of such a rule set from a precise specification to a precise implementation is described in detail as follows:

1. The process starts with a specification. The specification must at least define the inputs and outputs of the required system. A specification need not state how the process is to be achieved but it must be precise about what must be achieved. Any computational requirements may be defined at this stage including resource requirements.

2. The specification is checked for consistency and completeness. If either of these tests fails then it may be impossible to produce a correct implementation.

3. The specification is repeatedly decomposed in order to produce manageable components. The rules by which the decomposition is performed will differ from application to application.

4. Each system component is refined by replacing rules with new, more efficient, rules.

5. The refinement steps must be checked at each stage to ensure that they are consistent and complete with respect to the specification. The steps must also be checked to ensure that the modifications are true refinements, *i.e.* the step has not introduced any new steps or has increased the amount of non-deterministic behaviour. Refinement and checking is performed repeatedly.

6. Once refined to a required level, the system components are re-composed. The rules of recomposition must be defined in such a way as to ensure consistency and completeness. Recomposition may be applied repeatedly and may also lead to further refinement steps of the resulting system components.

7. After all the system components have been re-composed, a check is made to ensure that the computational constraints are met by the resulting implementation. If they are not then it may be necessary to start the decomposition and refinement process again.

8. The result of the process is a formally developed implementation. The Quality Control aspects of the development process have increased the likelihood that:

    - the implementation is correct with respect to the implementation; and
    - the process can be controlled and measured at each stage; and
    - the development can be planned in terms of decomposition and subsequent development of manageable sub-components.

4.2 Related work

There exists a body of work reported in the literature describing techniques for specifying and verifying KBS. For example: (Roman, 1993), (Krause, 1993), (International Journal of Expert Systems: Research and Applications, 6(2), 1993) and (International Journal of Expert Systems: Research and Applications, 6(3), 1993). An overview of the field is given in (Nazereth, 1993). Many of these systems address particular types of rule interpreters, often based on OPS5, and do not address the development process. The rule language that is used in this paper has the merit of being very simple, universal and suitable as the basis of more sophisticated languages. In addition, we have addressed the development process of KBS from specification to implementation.

We have focussed on the terms used as system states and the ways in which they are manipulated by rules. This is essentially the same approach as is taken in (Meseguer & Verdaguer, 1993):

> we believe ... that verification cannot be complete without following an accurate model of the procedural semantics of KB objects. In other words verification must analyse KB objects taking into account the way they will be used during ES execution.

although the details of their research involves the use of an ATMS. We have defined the semantics of KBS development in terms of a graph semantics for the execution traces. Graphs are also used in (Valiente, 1993) where they are used to define redundancy and subsumption within a single rule base. The rule execution semantics has relied heavily on non-determinism; this also occurs in (Gamble, Roman, Ball & Cunningham, 1994) due to concurrency although their rule system seems less expressive and a formal logic is used to express system properties. We have used the term *refinement* to describe a step-wise process of increasing the efficiency of a rule base; (Ma & Wilkins, 1995) also use this term to describe the process of finding a *sociopathic* subset of a rule base that is more efficient without modifying the overall semantics.

This work is closely related to the area of rewrite systems where a collection of directed rules are successively applied to terms in order to transform them into a normal form. A introduction to rewrite systems and a survey of the available literature is found in Jounanaud & Dershowitz (1990). In particular, the system described in this paper is related to conditional ground rewrite systems involving matching rather than general unification and allow guards on the rewrite rules. The system is also intimately related to rewrite systems that incorporate equations into the rewriting machinery, such as the R/S systems described in Jounanaud & Dershowitz (1990), where the equations define associativity, commutativity and identity. The following are important differences between rewrite systems and the mechanism described in this paper:

- Rewrite systems are primarily concerned with transforming terms into a normal form from which there are no applicable rewrites. In general, this is not true of the rules that are used in Knowledge Based Systems, for example the Blocks World rules need not terminate when the goal state is achieved.

- Rewrite systems are often concerned with proving termination. We have not addressed this issue with respect to proving that goal states are reachable. The issue of specifying Knowledge Based Systems and proving that the implementation in terms of a transition system will reach a goal in a finite number of moves is an area for further work.
- Rewrite systems use the full power of unification to perform transformations. The transition system for the applications in this paper requires only simple matching.

Embedding equations into a matching framework is also closely related to E-unification (Baader & Siekmann, 1994), where unification is extended to include equations defining, for example, associativity, commutativity and identity. As described above, the application areas addressed by this paper do not require unification, but it is an area for further work to investigate the requirements of more sophisticated applications.

This work is broadly related to the area of program transformation where Partsch (1983) and Luqi (1993) cover the general issues and Bird (1987) and Burstall and Darlington (1977) are more in the spirit of the transformations described here. We have viewed the specification of rule based transition systems as computational processes from the start whereas Morgan (1990) gives a different perspective of specification and refinement.

Nilsson (1980) and Charniak *et al.* (1987) describe various search related applications for computer programs, in particular Charniak *et al.* (1987) describes the connection between search problems and non-deterministic computational processes. Non-deterministic (functional) programs are discussed in Sondegaard (1992) and Clark (1994) gives an operational description of a system with builtin non-determinism.

In general, systems that provide support for the development of KBS software employ complex control strategies and provide expressive rule languages which include packaging rules into modules and meta-rule facilities. The languages Prolog (Clocksin & Mellish, 1984) and OPS5 (Brownston *et al*, 1985) are examples of rule languages with complex control strategies; the strategy of the former is often referred to as *backward chaining* and the strategy of the latter is *forward chaining*. The system described in this paper is very simple and does not provide such facilities, however the simplicity has allowed the system to be given a formal semantics and even without sophisticated features the rule language is capable of expressing a large class of interesting systems.

The language used for expressing rule based systems defined in this paper is an extension of that described in (Clark, 1996). The extensions allow variables to match repeated patterns (t^*) and allow patterns to exclude terms ($v@t$).

In (Clark, 1996) we address the issues of implementing the rule language as a pattern compiler for a functional programming language and also give an analysis of the refinement technique.

The results of the previous paper have been extended with a clear definition of the KBS development process in terms of specifications, implementations, modifications, refinements and modularity. Together, these results represent a firm theoretical and practical foundation for the development of rule based systems.

4.3 Further work

The results of this paper together with those of (Clark, 1996) form a firm theoretical and practical foundation for the development of rule based systems. The obvious next step is to use these results to address real-world applications. In doing so, we envisage the following developments:

- The expressivity of the rule based language can easily be extended by adding new types of patterns. It may be useful to add types so that rules can be checked.
- The implementation mechanisms described in (Clark, 1996) for the rule based language briefly address efficiency issues. It is likely that efficient implementation techniques will be required for large applications.
- The rule based language described in this paper uses equational theories for lists, sets and bags. In order to address large applications new techniques must be developed to allow user defined equational theories to be incorporated into the language.
- The definition of modularity given in this paper uses projections based upon monoids. This is convenient for applications that use lists, sets and bags but will not be appropriate for applications with user defined equational theories. A general definition of modularity must be developed that incorporates arbitrary equational theories.
- In order to address large scale applications it is likely that some form of computer assistance will be required. This could take the form of a tool that allows systems to be defined and will perform (or assist in performing) completeness, consistency and modularity checks.

5 Acknowledgements

The author would like to thank those involved in the reviewing process for their generous constructive criticism of an earlier draft of this paper.

References

Baader F. & Siekmann J. (1994) Unification Theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming* ed. Gabbay Dov M., Hogger C. J. & Robinson J. A. Oxford University Press.

Beynon-Davies, P. (1993) *Knowledge Engineering for Information Systems.* The McGraw-Hill International Series in Software Engineering.

Bird, R. S. (1987) *A Calculus of Functions for Program Derivation.* Oxford University Programming Research Group Monograph.

Brownston L., Farrell R., Kant E. & Martin N. (1985) *Programming Expert Systems in OPS5: An introduction to Rule-Based Programming.* Addison-Wesley.

Burstall, R. M. & Darlington, J. (1977) A Transformation System for Developing Recursive Programs. *J. ACM* 24 (1).

Charniak, E. *et al.* (1987) *Artificial Intelligence Programming.* Lawrence Erlbaum Associates.

Clark, A. N. (1996) Formal basis for the refinement of rule based transition systems. *Journal of Functional Programming* 6(2).

Clark, A. N. (1994) Pattern Recognition of Noisy Sequences of Behavioural Events Using Functional Combinators. *Computer Journal* 37(5).

Clocksin W. F. & Mellish C. S. (1984) *Programming in Prolog.* Springer-Verlag, Second ed.

Culbert, C., Riley, G. & Savely, R. T. (1987) Approaches to the Verification of Rule-Based Expert Systems. in *First Annual Workshop on Space Operations Automation and Robotics (SOAR 87), NASA Conf. Pub. 2491.*

Gamble, R. F., Roman, G., Ball, W. E. & Cunningham, H. C. (1994) Applying Formal Verification Methods to Rule-Based Programs. *International Journal of Expert Systems: Research and Applications,* 7(3).

Goguen, J. A., Thatcher, J. W. & Wagner E. G. (1976) An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. in Yeh, R. (editor), *Current Trends in Programming Methodology IV.* Prentice Hall.

Grogono, P., Nelson, S. H. (1982) *Problem Solving and Computer Programming.* Addison-Wesley Publishing Company.

International Journal of Expert Systems: Research and Applications. Special Issue on Verification and Validation, Part I, 6(2), 1993.

International Journal of Expert Systems: Research and Applications. Special Issue on Verification and Validation, Part II, 6(3), 1993.

Jounanaud J-P. & Dershowitz N. (1990) Rewrite Systems. in *Handbook of Theoretical Computer Science* ed. J. Van Leeuwen. Elsevier.

Krause P. *et al.* (1993) Can we formally specify a medical decision support system? *IEEE Expert June.*

Lucas P. & Van Der Gaag L. (1991) *Principles of Expert Systems.* Addison-Wesley. International Computer Science Series.

Luger G. F. & Stubblefield W. A. (1989) *Artificial Intelligence and the Design of Expert Systems.* The Benjamin Cummings Series in Artificial Intelligence.

Luqi, V. B. & Yehudai, A. (1993) Using Transformations in Specification-Based Prototyping. *IEEE Transactions on Software Engineering* 19(5).

Lopez, B., Meseguer, P. & Plaza, E. (1990) Knowledge Based Systems Validation: A State of the Art. *AI Communications* 3(2).

Ma, Y. & Wilkins, D. C. (1995) The Refinement of Probabilistic Rule Sets for Classification Expert Systems. *Internation Journal of Expert Systems,* 8(1).

Major, F. Lapalme, G. and Cedergren, R. (1991) Domain generating functions for solving constraint satisfaction problems. *Journal of Functional Programming,* 1(2).

Meseguer P. & Verdaguer A. (1993) Verification of Muli-Level Rule-Based Expert Systems: Theory and Practice. *International Journal of Expert Systems: Research and Applications,* 6(2).

Morgan, C. (1990) *Programming from Specifications.* Prentice Hall International Series in Computer Science.

Nazereth, D. L. & Kennedy M. H. (1993) Knowledge-Based System Verification, Validation and Testing: The evolution of a Discipline. *International Journal of Expert Systems: Research and Applications,* 6(2).

Nilsson, N. J. (1980) *Principles of Artificial Intelligence.* Springer-Verlag.

O'Keefe, R. M., Balci, O. & Smith, E. P. (1987) Validating Expert System Performance. *IEEE Expert Winter 1987.*

Partsch, H. & Steinbruggen, R. (1983) Program Transformation Systems *ACM Computing Surveys* 15(3).

Roman G., Gamble R. F. & Ball W. E. (1993) Formal Derivation of Rule-Based Programs. *IEEE Transactions on Software Engineering* 19 (3).

Rushby, J. (1988) Quality Measures and Assurance for AI Software. NASA Contractor Report CR-4187.

Sondegaard, H. and Setsoft, P. (1992) Non-Determinism in Functional Languages. *The Computer Journal,* 35(5).

Valiente, G. (1993) Verification of Knowledge Base Redundancy and Subsumption using Graph Transformations. *International Journal of Expert Systems: Research and Applications,* 6(3).