18th IEEE Real-Time Systems Symposium: Work in Progress Sessions

Jones, Michael
Boston University Computer Science Department


http://hdl.handle.net/2144/3748

Boston University
Optimization of Real-Time MRL Rule-Based Systems with the EQL Optimizer

Albert Mo Kim Cheng
Real-Time Systems Laboratory
Department of Computer Science
University of Houston—University Park
Houston, Texas 77204-3475, U.S.A.

Abstract

In [10], we developed an efficient algorithm for optimizing a class of EQL rule-based systems so that they can meet specified response time constraints. In this paper, we show that this EQL optimizer with minor modifications can be used to optimize a class of real-time MRL rule-based systems. As a more expressive superset of EQL, MRL allows existentially quantified as well as universally quantified variables (simple or macro), making it comparable in expressive power to that of OPS5 and CLIPS (two of the most popular commercially available rule-based system languages) while maintaining predictable response time behavior.

1 Introduction

Real-time rule-based monitoring/control systems are embedded artificial intelligence (AI) systems increasingly used in different industrial applications, such as airplane avionics, smart robots, space vehicles and other safety critical applications. Apart from functional correctness, these systems must also satisfy stringent timing constraints. The result of missing a deadline in these systems may be fatal. The verification task is to prove that a given system can deliver an adequate performance in bounded time. If this is not the case or if the real-time expert system is too complex to analyze, the system has to be resynthesized or optimized.

In [10], we developed an efficient algorithm for optimizing a class of EQL rule-based systems so that they can meet specified response time constraints. A toolset implementing this EQL optimizer is described in [11].

Our approach is based on the generation of the state space graph for each independent set of rules in the real-time rule-based program. In contrast with the algorithms that generate the state space graph starting from the initial states [7],[8], [9], we use a bottom-up approach. The generation starts with the identification of the final states (fixed points) of the system, and gradually expands the state space graph until all of the states with a reachable fixed point are found. A new and optimized rule based system is then generated directly from the constructed state space graph. Stability, determinism, and confluence are also achieved [11]. Here, we show how the EQL optimizer can be used to optimize real-time MRL programs.

It has been shown [1] that even for rule-based systems using variables with finite domains, the derivation of the state space graph in the worst case requires exponential computation time as a function of the number of variables in the program. We have identified the techniques that allow us to reduce this complexity by grouping the equivalent states into a single vertex of a state space graph and exploiting the concurrency by labeling a single edge of a state space graph with a set of rules fired in parallel.

The remainder of this paper is organized as follows.

Section 2 introduces the EQL rule-based language, its execution model and its state space representation. The EQL optimization algorithm is presented in Section 3. Section 4 describes the more expressive MRL rule-based language. Section 5 shows how the EQL optimizer can be applied to MRL programs and concludes the paper.

2 EQL Programs and Basic Definitions

An EQL program is given in the form of an n rules (r1, ..., r_{n}) that operate over a set of m variables (x_1, ..., x_m). Each rule has an action and a condition part. Formally,

\[ F_k (s) \text{ IF } EC_k (s) \]

where \( k \in \{1, \ldots, n\} \), \( EC_k (s) \) is the enabling condition of rule \( k \), and \( F_k (s) \) is the action. Both the enabling condition and the action are defined over the state \( s \) of a system. Each state \( s \) is expressed as a tuple \( s = (x_1, \ldots, x_m) \), where \( x_i, i \in \{1, \ldots, m\} \) represents a value of the \( i \)-th variable. An action \( F_k \) is given as a series of \( n_k \geq 1 \) subactions separated by **"**:

\[ L_{k,1} := R_{k,1}(x_1, \ldots, x_m) \ldots \]

\[ L_{k,n_k} := R_{k,n_k}(x_1, \ldots, x_m) \]
The subactions are interpreted from left to right. Each subaction sets the value of variable $L_{k,i} \in \{x_1, \ldots, x_m\}$ to the value returned by the function $R_{k,i}$, $i \in \{1, \ldots, n_k\}$.

The enabling condition is a two-valued function and for each state of the system it determines if the rule is enabled to fire.

We first constrain the EQL program to use only constant assignments in the subactions of rules, i.e., $R_{k,i} \in \{0, 1\}$. This can potentially reduce the complexity of algorithms for optimization. Then we will extend these algorithms to cover more general classes of programs.

Example 1. An EQL rule-based expert system with boolean variables.

```
PROGRAM an_eql_b_program;
VAR a, b, c, d : BOOLEAN;
RULES
  (* 1 *) c := 1 IF a = 0 AND b = 0 AND d = 0
  (* 2 *) b := 1 IF d = 0 AND (a = 1 OR c = 1)
  (* 3 *) a := 1 ! c := 1
          IF a = 0 AND d = 0 AND (b = 1 OR c = 1)
  (* 4 *) b := 0 ! c := 0 IF d = 0 AND b = 1 AND
          (a = 0 AND c = 1 OR a = 1 AND c = 0)
  (* 5 *) d := 0 ! a := 1 IF a = 1 AND c = 1 AND d = 1
END.
```

The above example program will be used throughout the paper. For clarity, the example program is intentionally kept simple. In practice, our method can be used for systems of much higher complexity, possibly consisting of several thousands of rules.

Execution Model of a Real-Time Decision System Based On an EQL Program Paradigm

Real-time decision systems based on the EQL program paradigm interact with the environment through sensor readings. Sensor readings are then represented as the values of the variables used in the EQL program that implements the decision system.

We define the response time as the time the EQL system spends to reach a fixed point. This can be denoted by a maximum number of rules to be fired to reach a fixed point. A real time decision system is said to satisfy the timing constraints if the response time is smaller or equal to the smallest time interval in-between two sensor readings.

State Space Representation

To develop an optimization method we view an EQL(B) system as a transition system $T$ with a finite set of states. $T$ is a triple $(S, R, \rightarrow)$ where

1. $S$ is a finite set of states. Assuming a finite set $V$ of two-valued variables $x_1, x_2, \ldots, x_m$ and an ordering of $V$, $S$ is the set of all $2^m$ possible Cartesian products of the values of variables;
2. $R$ is a set of rules $r_1, r_2, \ldots, r_n$ in the rule base of the expert system;
3. $\rightarrow$ is a mapping associated with each $r_k \in R$, i.e., a transition relation $\rightarrow \subseteq S \times S$, so that if $r_k$ is enabled at $s_1 \in S$ and firing of $r_k$ at that state $s_1$ would result in the new state $s_2 \in S$, we can write $s_1 \xrightarrow{r_k} s_2$, or shorter, $s_1 \xrightarrow{k} s_2$.

A graphical representation of a transition system $T$ is a labeled finite directed transition or state space graph $G = (V, E)$. $V$ is a set of vertices each labeled with the states $s \in S$. $E$ is a set of edges labeled with $r_k \in R$. Edge $r_k$ connects vertex $s_1$ to $s_0$ if and only if $s_1 \xrightarrow{r_k} s_2$.

A path in a transition graph is a sequence of vertices such that for each consecutive pair $s_i, s_j$ in a sequence, there exists a rule $r_k \in R$ such that $s_i \xrightarrow{r_k} s_j$. If there is a path from $s_i$ to $s_j$, $s_j$ is said to be reachable from $s_i$. A cycle is a path from a vertex to itself.

3 Optimization Algorithm

After deriving an assertion for the fixed points of the program, we apply the optimization algorithm to the program. Our optimization method consists of two main steps: construction of an optimized finite state space graph and synthesis of a new EQL rule-based expert system from it. The potentially exponential complexity of these two phases is reduced by optimizing only one independent rule set at a time. The optimization schema is outlined below.

```
Procedure Optimize
  Begin
    Read in the original EQL program $P$;
    Construct high level dependency (HLD) graph;
    Using HLD graph, identify independent rule sets in $P$;
    For all independent rule sets in $P$ Do
      Construct optimized state space graph $T$;
      Synthesize optimized EQL program $O$ from $T$;
    End For all
  End
```

Decomposition of an EQL Program

We use a decomposition algorithm for EQL as given in [2]. The algorithm is based on the notion of rule independency. The decomposition algorithm uses the set $L_k$ of variables appearing in the left-hand-side of the multiple assignment statement of rule $k$ (e.g., for the EQL program in Example 1, $L_5 = \{a, d\}$).

If the optimization technique maintains the assertion about fixed-point reachability for every independent rule set, each rule set can be optimized independently. The above decomposition method was evaluated in [2] and the results encourage us to use it to substantially reduce the complexity of the optimization process.

Derivation of an Optimized State Space Graph

The core of EQL optimization is the construction of a corresponding state space graph. We use a bottom-up approach and start the derivation from the fixed points. We show that the main advantage of this approach is its simplicity in removing the cycles and in identifying the
paths with the minimal number of rules to fire to reach the fixed points [11].

Synthesis of an Optimized EQL Program

A new EQL program is synthesized from the constructed optimized transition system. For each rule in the independent rule set, the new enabling condition is determined by scanning through the state space graph so that for rule \( r_i \), the new enabling condition is:

\[
EC_i^{new} = \left( \bigvee_{S \in S, r_i \in R} S \right) \land EC_i
\]

where \( S \) and \( S' \) are labels of two vertices in the state space graph that are connected by the edge labeled with \( r_i \) or with the set that includes \( r_i \). If all edges in the state space graph are labeled with a single rule, the conjunctive term \( EC_i \) can be omitted in the above expression.

The new rules are then formed with the assignment parts in the original rules and the new enabling conditions. Rules not included in any of constructed state space graph are redundant and are not added to the new rule base.

The optimized EQL program constructed using the optimization algorithm BU is shown below (single-rule transitions with or without equivalent states). Note that rule \( r_i \) was found to be redundant and does not appear in the resulting program.

Example 2. An optimized EQL program derived from the program in Example 1.

```eql
PROGRAM an_optimized_eql_b_program_1;
VAR a, b, c, d : BOOLEAN;
RULES
(* 1 *) c:=1 IF a=0 AND b=0 AND c=0 AND d=0
(* 2 *) b:=1 IF a=1 AND b=0 AND c=1 AND d=0
(* 3 *) a:=1 AND c:=1 IF a=0 AND c=1 AND d=0 OR a=0 AND b=1 AND d=0
(* 5 *) d:=0 ; a:=1 IF a=1 AND c=1 AND d=1
END;
```

4 MRL Production Systems

Syntactically, MRL [6] closely resembles EQL in that the enabling conditions of its rules are based on logic expressions. Semantically, it is a production system language which can be used for practical real-time applications. As a more expressive superset of EQL, MRL allows existentially quantified as well as universally quantified variables (primitive or macro), making it comparable in expressive power as that of OPS5 and CLIPS, two of the most popular commercially available rule-based system languages. The development of an OPS5-to-MRL translator [3] further extends the practicality of our proposed approach.

Each macro variable represents a set of primitive variables called expanded variables or working memory elements (wme). Rules that contain macro variables are macro rules, in contrast to plain rules in EQL programs. Conceptually, we model the execution of MRL programs as follows. Before execution, a macro rule is expanded to a set of plain rules each of which contains one of the possible combinations of macro variable expansions, i.e., each macro variable is instantiated to one of the corresponding expanded variables. The result of the expansion is an equivalent EQL program which has the same behavior as the MRL program. Macro variables make it more convenient to assemble related input data into data structures, and also allow us to introduce logical quantifiers, either existential quantifiers or universal quantifiers, into MRL. The combination of macro variables and logical quantifiers makes MRL a sufficiently powerful production language for writing OPS5-type [5] real-time expert systems. We shall introduce the MRL language by an example.

Example 3a.

```eql
PROGRAM example_2;
TYPE
new_type = RECORD a : INTEGER;
END;
MACRO_VAR new_rec : new_type;
VAR c : INTEGER;
PATTERN_VAR x, y : new_rec;
INPUT
READ_RECORD new_rec;
READ c;
RULES
?x.b := a IF ?y.a <> 0
[] ?x.b := 1 IF ?y.a := 1 IF c <> 0 AND ?y.a = 0
END.
```

The type definition is similar to the type definition in procedural languages. The MACRO_VAR section declares a macro variable new_rec of type new_type. This statement associates a set of wmes (each is a record) to new_rec. Macro variables can be referred to only by special symbols called pattern variables, which are not real variables since they do not occupy memory space. They are used solely as symbols that refer to macro variables. Two occurrences of the same wme in a rule have to be instantiated to the same expanded variable if they are existentially quantified. The scopes of pattern variables are restricted to one rule only. In the above example, the PATTERN_VAR section declares two pattern variables x and y for the macro variable new_rec.

When used in MRL rules, pattern variables must be quantified. Pattern variables with the prefix of a question mark “?” are existentially quantified variables. Pattern variables with the prefix of an asterisk “*” are universally quantified variables. Existentially quantified variables may appear anywhere in a rule. Each existentially quantified pattern variable must be instantiated to one of the expanded variables (wmes) of the corresponding macro variable when the rule is fired. In contrast, universally quantified pattern variables can only appear in the enabling conditions. A rule is enabled if its enabling condition is always true for all possible instantiations of the universally quantified variables.
Let the size of the working memory associated with the macro variable new_rec in the above program be two, i.e., two expanded variables.

**Example 3b. EQL Program.**

```
PROGRAM expanded_example_2;
VAR c: BOOLEAN;
   r1.a, r2.a: INTEGER;
   r1.b, r2.b: 0..9;
RULES
(*1.1*) r1.b := 0 IF r1.a <> 0 AND r2.a <> 0
(*1.2*) r2.b := 0 IF r1.a <> 0 AND r2.a <> 0
(*2.1*) r1.b := 1 ! r1.a := 1
   IF c <> 0 AND r1.a = 0
(*2.2*) r2.b := 1 ! r2.a := 1
   IF c <> 0 AND r2.a = 0
(*3.1*) r2.b := 1 ! r1.a := 1
   IF c <> 0 AND r1.a = 0
(*4.1*) r2.b := 1 ! r2.a := 1
   IF c <> 0 AND r2.a = 0
END.
```

Variables r1.a and r2.a are expanded variables of macro variables new_rec.a, and variables r1.b and r2.b are expanded variables of macro variables new_rec.b. Rules 1.1 and 1.2 in the EQL program are expanded rules of rule 1 in the MRL program. Rules 2.1 to 2.4 in the EQL program are expanded rules of rule 2 in the MRL program.

The firing of a macro rule in an MRL program is in fact the firing of one of the expanded rules of this macro rule. An MRL program reaches a fixed point if none of the macro rules in the program is enabled or firing any rules in this program does not change the contents of the working memory.

The existential quantifier and universal quantifier of predicate logic are used to quantify the elements of the working memory. For example, the macro rule \(?x := 0 IF \?x = \?y\) can be interpreted as the following formula, where \(x\)' is the value of \(x\) after the executing the assignment if the rule is selected for firing:

\[ \exists x \forall y : (x = y) \rightarrow x' := 0.\]

In terms of rule-based programming, this formula means that if there is an expanded rule with an instantiation for \(x\) such that for all the possible instantiations of \(y\), \(x = y\), then this rule can be fired by assigning \(0\) to \(x\). Note that according to the MRL language definition, the \(\exists\) quantifiers always appear before the \(\forall\) quantifiers in the logic formula.

## 5 MRL Optimizer and Concluding Remarks

To apply our EQL optimizer to MRL programs without expanding the MRL rules into corresponding EQL rules, we have to redefine the notion of a state and hence the fixed-point assertion. EQL properties can also be redefined for MRL rules. For instance, two MRL rules are defined to be compatible if all the expanded rules of these rules are pairwise compatible. However, the compatibility property of MRL rules can be detected without actually expanding the MRL rules into EQL rules. Our preliminary research shows that these redefinitions can be readily handled by the EQL optimizer without major modifications. More precisely, a state graph corresponding to the optimized MRL program can be constructed and hence the optimized MRL program can be synthesized from the graph. A full paper reporting these results will be presented later.

### References


