As we have seen in chapters 10 and 11, embedded rule-based expert systems must satisfy stringent timing constraints when applied to real-time environments. We now describe a novel approach to reduce the response time of rule-based expert systems. This optimization is needed when a rule-based system does not meet the specified response-time constraints. Our optimization method is based on a construction of the reduced cycle-free finite-state space-graph. In contrast with traditional state-space graph derivation algorithms, our optimization algorithm starts from the final states (fixed points) and gradually expands the state-space graph until all of the states with a reachable fixed point are found. The new and optimized rule-based system is then synthesized from the constructed state-space graph. We present several algorithms implementing the optimization method. They vary in complexity as well as in the usage of concurrency and state-equivalence, both targeting to minimize the size of the optimized state-space graph.

The optimized rule based systems generally (1) have better response time, that is, require fewer number of rule firings to reach the fixed point, (2) are stable, that is, have no cycles that would result in the instability of execution, and (3) include no redundant rules. The actual results of the optimization depend on the algorithm used. We also address the issue of deterministic execution and propose optimization algorithms that generate the rule bases with a single corresponding fixed point for every initial state.

The synthesis method also determines a tight response-time bound of the new system and can identify unstable states in the original rule base. No information other than the rule-based real-time decision program itself is given to the optimization method. The optimized system is guaranteed to compute correct results independent of the scheduling strategy and execution environment.
12.1 INTRODUCTION

Embedded rule-based systems must also satisfy stringent environmental timing constraints which impose a deadline on the decision/reaction time of the rule base. The result of missing a deadline in these systems may be harmful. The task of verification is to prove that the system can deliver an adequate performance in bounded time [Browne, Cheng, and Mok, 1988]. If this is not the case or if the real-time expert system is too complex to analyze, the system has to be resynthesized.

We present a novel optimization method for rule-based real-time systems. The optimization is based on the derivation of a reduced, optimized, and cycle-free state-space graph for each independent set of rules in the rule-based program. Once the state space graph is derived, no further reduction and/or optimization is required, and it can then be directly used for resynthesis of the new and optimized rule-based program.

The optimization makes use of several approaches and techniques previously used for the analysis and parallel execution of real-time rule-based systems. It also employs several known techniques originated from protocol validation to minimize the number of states in state-space graphs. In particular:

- The complexity of the optimization is reduced by optimizing each independent set of rules separately. This technique originates from [Cheng et al., 1993], where the same approach was used to lower the complexity of analysis of real-time rule-based systems.
- The state-space graph representation of the execution of real-time rule-based system was introduced in [Browne, Cheng, and Mok, 1988]. It was used for the analysis of rule-based systems, but, because of the possible state explosion, the approach may be used solely for systems with few variables. We show how this representation also may be used for larger systems if the reduced state-space graphs are used instead. To reduce the number of states, known methods from protocol analysis (representation of a set of equivalent states with a single vertex of the state-space graph) and from rule-based system analysis (parallel firing of rules within independent rule sets) are employed.

Specific to the optimization method proposed here are reduction and optimization of the state-space graph while it is derived from a set of rules. Also specific are bottom-up derivation and resynthesis of a new and optimized rule-based system. In particular:

- The derivation of the state-space graph 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. This bottom-up approach combined with a breadth-first search finds only the minimal-length paths to the fixed points.
- Rather than first deriving the state-space graph and then reducing the number of states, the reduced state-space graph is built directly. We identify the techniques that, while building the state space graph, allow us to group the equivalent states
into a single vertex of a graph and exploit the concurrency by labeling a single edge of a graph with a set of rules fired in parallel.

- The derivation of the state-space graph is constrained so that it does not introduce any cycles. The new rule-based system constructed from such a graph is cycle-free (stable).
- The derived state-space graph does not require any further reduction of states and/or optimization, and it can be directly used to resynthesize an optimized real-time rule-based program.

In this chapter, several state space derivation techniques are described, addressing:

- **response-time optimization**: the optimized programs require the same or fewer numbers of rules to fire from some state to reach a final state;
- **response-time estimation**: it is of crucial importance for real-time rule-based systems not only to speed up the execution time but also to at least estimate its upper bounds [Browne, Cheng, and Mok, 1988; Chen and Cheng, 1995b];
- **stability**: all cycles of the original rule-based systems that make the system unstable are removed;
- **determinism and confluence**: if more than one rule is enabled to be fired at a certain stage of the execution, the final state is independent of the execution order.

The algorithms presented here were developed for a two-valued version of the equational rule-based language EQL described in chapter 10. The language was initially developed to study the rule-based systems in a real-time environment. In contrast with popular expert systems languages such as OPS5, where the interpretation of the language is defined by the recognize–act cycle [Forgy, 1981], EQL’s interpretation is defined as fixed point convergence. For EQL programs, a number of tools for analysis exist and are described in chapter 10.

### 12.2 BACKGROUND

Validation and verification is an important phase in the life cycle of every rule-based system [Eliot, 1992]. For real-time rule-based systems, we define the validation and verification as an **analysis problem**, which is to decide if a given rule-based system meets the specified integrity and timing constraints. Here, we focus on the latter.

To determine if a system satisfies the specified timing constraints, one has to have an adequate performance measure and a method to estimate it. We define the **response time** of a rule-based program in terms of the computation paths leading to fixed points. These paths can be obtained from a **state-space representation**, where a vertex uniquely defines a state of the real-time system and a transition identifies a single firing of a rule. An upper bound on the response time is then assessed by the maximum length of a path from an initial (launch) state to a fixed point. We show that even for the rule-based systems that use variables with finite domains, such an
approach in the worst case requires exponential computation time as a function of the number of variables in the program [Browne, Cheng, and Mok, 1988].

We have implemented the methods on a class of real-time decision systems where decisions are computed by an *equational rule-based (EQL)* program. Corresponding analysis tools were developed to estimate the time responses for programs written in other production languages, for example, MRL [Wang and Mok, 1993] and OPS5 [Chen and Cheng, 1995a].

Within the deductive database research, similar concepts are presented by [Abiteboul and Simon, 1991]. They discuss the *totalness* and *loop-freeness* of a deductive system, which, in the terminology of rule-based systems, describes the stability of the systems in terms of the initial points reaching their fixed points in finite time.

If the analysis finds that the given real-time rule-based program meets the integrity but not the timing constraints, the program has to be optimized. We define this as the *synthesis problem*, which has to determine whether an extension of the original real-time system exists that would meet both timing and integrity constraints. The solution may be achieved by either (1) transforming the given equational rule-based program or (2) optimizing the scheduler to select the rules to fire such that some fixed point is always reached within the response-time constraint. The latter assumes that there is at least one sufficiently short path from a launch state to every one of its end points. In chapter 10, we gave an example for both solutions, but did not propose a corresponding algorithm.

In our definition of the synthesis problem, the original program is supposed to satisfy the integrity constraints. To obtain the optimized program satisfying the same constraints, we require that each launch state of the optimized program has the same set of corresponding fixed points as the original program. We believe that the optimization can benefit highly by easing this constraint, so that for each launch state the optimized program would have only a single corresponding fixed point taken from the set of fixed points of the unoptimized system. Such system has a *deterministic* behavior. [Aiken, Widom, and Hellerstein, 1992] formalize this concept for database production rules and discuss the *observable determinism of a rule set*. The rule set is observably deterministic if the execution order does not make any difference in the order of appearance of the observable actions. Similar concepts can be found in the process-algebraic approach described in chapter 9.

The chapter is organized as follows. We first review the necessary background and discuss in more detail the analysis and synthesis problem of real-time rule-based systems. Next we review the EQL rule-based language, its execution model and its state-space representation. Several optimization algorithms are then presented. We next experimentally evaluate these optimization algorithms. The methods—their limitations and possible extensions—are finally discussed.

### 12.3 Basic Definitions

The real-time programs considered here belong to the class of EQL programs. Here we define the syntax of EQL programs and its execution model, define the measure
for the response time of the EQL system, and formally introduce their state-space graphs.

12.3.1 EQL Program

An EQL program is given in the form of \( n \) rules \((r_1, \ldots, r_n)\) that operate over the set of \( m \) variables \((x_1, \ldots, x_m)\). Each rule has action and condition parts. Formally,

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

where \( k \in \{1, \ldots, n\} \), \( EC_k(s) \) is an enabling condition of rule \( k \), and \( F_k(s) \) is an 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 \( x_i \) represents a value of \( i \)th variable. An action \( F_k \) is given as a series of \( n_k \geq 1 \) subactions separated by “!”:

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

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 \( EC_k(s) \) is a two-valued function that evaluates to TRUE for the states where rules can fire.

Throughout the chapter we will use two-valued variables only, that is, \( x_i \in \{0, 1\} \).

We will identify this subset of EQL by EQL(B). Any EQL program with a predefined set of possible values of the variables can be converted into the EQL(B). Due to the simplicity of such conversion, here we show only an example. Consider the following rules:

\[
i := 2 \quad \text{IF} \quad j < 2 \quad \text{AND} \quad i = 3
\]

\[
\square \quad j := i \quad \text{IF} \quad i = 2
\]

where \( i \) and \( j \) are four-valued variables with their values \( i, j \in \{0, 1, 2, 3\} \). The corresponding EQL(B) rules using two-valued variables \( i_1, i_2, j_1, \) and \( j_2 \) are:

\[
i_0 := \text{TRUE} \quad ! \quad i_1 := \text{FALSE}
\]

\[
\quad \text{IF} \quad \text{(NOT} \quad j_0 \quad \text{AND} \quad \text{NOT} \quad j_1) \quad \text{OR} \\
\quad \quad \text{(NOT} \quad j_0 \quad \text{AND} \quad j_1) \quad \text{AND} \quad (i_0 \quad \text{AND} \quad i_1)
\]

\[
\square \quad j_0 := i_0 \quad ! \quad j_1 := i_1
\]

\[
\quad \text{IF} \quad (i_0 \quad \text{AND} \quad \text{NOT} \quad i_1)
\]

Furthermore, we constrain EQL(B) to use only constant assignments in the subactions of rules, that is, \( R_{i, j} \in \{0, 1\} \). This can potentially reduce the complexity of optimization algorithms (see Section 12.3.4).

An example of an EQL(B) program is given in Figure 12.1. This program will be used in the following sections to demonstrate the various effects of the optimization. For clarity, the example program is intentionally kept simple. In practice, our method can be used for the systems of much higher complexity, possibly consisting of several hundred rules.
12.3.2 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. Readings are then represented as the values of variables used in the EQL program that implements the decision system. The variables derived directly from sensor readings are called input variables. All other variables are called system variables. After the input variables are set the EQL program is invoked. Repeatedly, among the enabled rules a conflict resolution strategy is used to select a rule to fire. This (possibly) changes the state of the system, and

![Diagram of EQL program-based decision system](image.png)

**Figure 12.2** An EQL program-based decision system.
the whole process of firing is repeated until either no more rules are enabled or rule firing does not change the system’s state. The resulting state is called a fixed point. The values of the variables are then communicated back to the environment.

The process of reading the sensor variables, invoking the EQL program, and communicating back the values of the fixed point is called the monitor cycle. A fixed point is determined in a repetitive EQL invocation named the decide cycle (Figure 12.2).

As described in chapter 10, EQL’s response time is the time an EQL system spends to reach a fixed point, or, equivalently, the time spent in the decide cycle. 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 between two sensor readings. A response time can be assessed by a maximum number of rules to be fired to reach a fixed point. The conversion from number of rule firings to response time can be done if one knows the time spent for identification of a rule to fire and the time spent for firing the rule itself. These times depend on the specific architecture of an implementation and will not be discussed here.

### 12.3.3 State-Space Representation

In order to develop an optimization method, we view an EQL(B) system as a transition system $\mathcal{T}$, which is a triple, $(\mathcal{S}, \mathcal{R}, \rightarrow)$, where

1. $\mathcal{S}$ is a finite set of states. Assuming a finite set $\mathcal{V}$ of two-valued variables $x_1, x_2, \ldots, x_m$ and an ordering of $\mathcal{V}$, $\mathcal{S}$ is the set of all $2^m$ possible Cartesian products of the values of variables;
2. $\mathcal{R}$ is a set of rules $r_1, r_2, \ldots, r_n$ in the system’s rule base;
3. $\rightarrow$ is a mapping associated with each $r_k \in \mathcal{R}$, that is, a transition relation $r_k \subseteq \mathcal{S} \times \mathcal{S}$. If $r_k$ is enabled at $s_1 \in \mathcal{S}$ and firing of $r_k$ at that state $s_1$ results in the new state $s_2 \in \mathcal{S}$, 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 $\mathcal{T}$ is a labeled finite directed transition or state-space graph $G = (V, E)$. $V$ is a set of vertices, each labeled with the states $s \in \mathcal{S}$. $E$ is a set of edges labeled with $r_k \in \mathcal{R}$. An edge $r_k$ connects vertex $s_1$ to $s_2$ 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 $r_k \in \mathcal{R}$ such that $s_i \xrightarrow{r_k} s_j$. If a path exists 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.

A state-space graph for the program from Figure 12.1 is shown in Figure 12.3. Throughout the chapter we use the following labeling scheme: edges are labeled with the rule responsible for the transition, and vertices are labeled with the two-valued expression, which evaluates to TRUE for a state represented by a vertex. After the introduction of the equivalent states (section 12.4.2), we will also allow the vertices to be labeled with a two-valued expression denoting a set of states and allow edges to be labeled with a set of rules. We use a compact notation for two-valued expressions. For example, $ab$ represents a conjunction of $a$ and $b$, $a+b$ denotes a disjunction of $a$ and $b$, and $\neg a$ represents a negation of $a$. 
Each state may belong to one or more categories of states, which are:

- **fixed point**: A state is said to be a fixed-point state if it does not have any out-edges or if all of the out-edges are self-loops. For the state-space graph in Figure 12.3, an example of fixed points are \( \overline{abcd} \), \( \overline{abc}d \), and \( \overline{a}bcd \).

- **unstable**: A state is unstable if no fixed point is reachable from it. Because such states have to have out-edges (or else it would be a fixed point state), it either has to be involved in a cycle or has to have reachable a state involved in a cycle. States \( \overline{a}bcd \) and \( \overline{abc}d \) from Figure 12.3 are unstable.

- **potentially unstable**: A potentially unstable state is either an unstable state or a state with a reachable fixed point, and either is involved in a cycle or has a path to the state involved in a cycle. States \( \overline{abc}d \), \( \overline{bcd} \), and \( \overline{abcd} \) from Figure 12.3 are potentially unstable.

- **stable**: A stable state is either a fixed point or a state from which no unstable or potentially unstable state is reachable. For example, states \( \overline{abcd} \) and \( \overline{abc}d \) from Figure 12.3 are stable, while \( \overline{abcd} \) is not.

- **potentially stable**: A potentially stable state is one with a reachable fixed point or is a fixed point itself. For example, states \( \overline{abc}d \) and \( \overline{abcd} \) from Figure 12.3 are potentially stable.

- **launch**: The state in which the program is invoked is called the launch state.

Stable states are a subset of potentially stable states. Unstable states are a subset of potentially unstable states. No unstable state is potentially stable. Any valid state is either potentially stable or unstable.
Using the above terminology, the response time of the system as defined in the previous section is given by a maximum number of vertices between any launch state and the corresponding fixed points. This definition is clear in the case of stable launch states. The response time is infinite if the system includes unstable launch states. For a system that has potentially unstable launch states, the response time cannot be determined without knowing the conflict resolution strategy.

We will treat the EQL system as generally as possible and will not distinguish between input and system variables. As a consequence, all states in the system are considered to be potential launch states. Also, our optimization strategy will not use any knowledge about the conflict resolution strategy that the system might use. The sole information given to the optimizer is the EQL program itself.

### 12.3.4 Derivation of Fixed Points

A state $s$ is a fixed point if:

- F1. no rule is enabled at $s$, or
- F2. for all the rules that are enabled at $s$, firing each of the rules will again result in the same state $s$.

We can consider the enabling condition to be an assertion over states, deriving TRUE if the rule is enabled or FALSE if it is disabled at a certain state. Our method does not find the fixed points explicitly (this would require an exhaustive search over all $2^m$ legal states), but rather constructs an assertion that would derive TRUE for fixed points and FALSE for all other states.

The assertion for fixed points for (F1) is defined as

$$FP_1 := \bigwedge_{i=1}^{n} EC_i$$

To derive an assertion for (F2) (see Figure 12.4), we have to use an important property of the EQL(B) programs: a rule with constant assignments cannot be consecutively fired more than once so as to derive different states. That is, $a \rightarrow b \rightarrow c$ for $a \neq b \neq c$ does not exist. Furthermore, for each rule $r_i$ we define a destination assertion $D_i$. $D_i$ evaluates to TRUE for all states that may result from firing rule $r_i$, and evaluates to FALSE otherwise. In other words, $D_i$ is TRUE for state $s$ if and only if there exists $s'$, so that $s' \overset{r_i}{\rightarrow} s$. If such $s'$ exists, $s$ is called a destination state of rule $r_i$.

The assertion $FP_2$ is initially FALSE, that is, initially the set of states of type (F2) is empty. Then, for every rule $r_i$, an assertion $S$ is constructed that is TRUE only for the states that both are destination states of that rule and enable the same rule (outmost For loop in Figure 12.4). Next, this assertion is checked against all other rules $r_j$ (inmost For loop): the algorithm specializes the assertion $S$ to exclude all the states that are not $r_j$’s destination states and that enable $r_j$. For every rule, the
Procedure Derive_F\textit{P}_2

\textbf{Begin}
\begin{align*}
\text{FP}_2 & := \text{FALSE} \\
\text{For } i & := 1 \text{ To } n \text{ Do} \\
S & = D_i \land EC_i \\
\text{If } S & \neq \text{FALSE} \text{ Then} \\
\text{For } j & := 1 \text{ To } n \text{ Do} \\
\text{If } i & \neq j \text{ and } EC_j \land S \text{ Then} \\
S & := (S \land EC_j) \lor (S \land EC_j \land D_j) \\
\text{End If}
\end{align*}
\text{End If}
\text{End For}
FP_2 := S \lor FP_2
\text{End For}
\text{End}

\textbf{Figure 12.4} Derivation of fixed-point assertion \textit{FP}_2.

assertion \textit{S} is disjuncted with current \textit{FP}_2 to form a new \textit{FP}_2. In other words, the states of type (F2) found for rule \textit{r}_i are added to the set of fixed points.

Finally, an assertion for the fixed points is a disjunction, \textit{FP} = \textit{FP}_1 \lor \textit{FP}_2. In the following discussions we use this assertion implicitly, meaning that when we assign a vertex to include all the fixed points, the vertex actually stores the assertion rather than the set of states. Due to the substantial number of details involved, here we omit the associated proofs and algorithms, which can be found in [Zupan, 1993].

\section{12.4 Optimization Algorithm}

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 potential exponential complexity of these two phases [Cheng, 1993b] is reduced by optimizing only one independent rule-set at a time. The optimization schema is depicted in Figure 12.5.

In this section we first present the EQL(B) rule-base decomposition technique. We then propose different optimization methods, all of which have in common the idea of generating the transition system from fixed points up and vary in the complexity of vertices and edges in the generated state-space graphs. Methods that are simpler in the implementation but potentially more complex in the execution are presented first. The section concludes with the algorithm that uses the generated state-space graph to synthesize the optimized EQL(B) program.

\subsection{12.4.1 Decomposition of an EQL(B) Program}

We use a decomposition algorithm for EQL as given in [Cheng, 1993b] and modify it for the EQL(B) case. The algorithm is based on the notion of \textit{rule independence}. 
Procedure Optimise

Begin

Read in the original EQL(B) program \( P \)

Construct high level dependency (HLD) graph

Using HLD graph, identify independent rule-sets in \( P \)

Forall independent rule-sets in \( P \) Do

Construct optimized state-space graph \( T \)

Synthesize optimized EQL(B) program \( O \) from \( T \)

Output \( O \)

End Forall

End

Figure 12.5 General optimization schema.

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(B) program in Figure 12.1, \( L_5 = \{a, d\} \)). Rule \( a \) is said to be independent from rule \( b \) if

\[
D1a. \quad L_a \cap L_b = \emptyset,
D1b. \quad L_a \cap L_b \neq \emptyset \text{ and for every variable } v \in L_a \cap L_b, \text{ the same expression must be assigned to } v \text{ in both rules } a \text{ and } b, \text{ and}
D2. \quad \text{rule } a \text{ does not potentially enable rule } b, \text{ that is, a state does not exist where } a \text{ is enabled and } b \text{ is disabled, and firing } a \text{ enables } b.
\]

The algorithm first constructs the rule-dependency graph. This consists of vertices (one for every rule) and directed edges. A directed edge connects a vertex \( a \) to \( b \) if rule \( a \) is not independent from rule \( b \). All vertices that belong to the same strongly connected component are then grouped into a single vertex. The derived graph is called a high-level dependency graph and each vertex stores the forward-independent rule-set. Figure 12.6 shows an example of a HLD graph for the EQL(B) program given in Figure 12.1.

Rules can now be fired by following the topological ordering of the vertices (rule sets). For each vertex the corresponding rules are fired until a fixed point is reached. If the EQL program is guaranteed to reach the fixed point from every launch state, the above rule schedule will guarantee the program will reach a fixed point as well [Cheng, 1993b].

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 [Cheng, 1993b] and the results encourage us to use it to substantially reduce the complexity of the optimization process.
12.4.2 Derivation of an Optimized State-Space Graph

The core of EQL(B) optimization is a 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 to remove the cycles and to identify the paths with the minimal number of rules to fire to reach the fixed points.

Here, no notion of conflict resolution strategy is used. For each stable or potentially unstable state, all corresponding fixed points are treated as equivalent. In other words, the EQL(B) execution is valid if for each launch state the system converges to a fixed point arbitrarily selected from a set of corresponding fixed points.

**Bottom-Up Derivation** The optimized transition system is derived directly from the set of EQL(B) rules. The derivation algorithm combines the bottom-up and breadth-first search strategies. It starts at the fixed points and gradually expands each fixed point until all stable and potentially unstable states are found. Note that the stable and potentially unstable states constitute the set of all states that have one or more reachable fixed points.

We will refer to the algorithm step of adding a new vertex $s'$ and a new edge $r$ to a state-space graph as an expansion. The state $s$ for which $s' \xrightarrow{r} s$ is referred to as an expanded state.

The optimization algorithm BU (Figure 12.7) uses the variables $V$ and $E$ to store the vertices and edges of the current state-space graph. The fixed points are determined by using the fixed point assertion (section 12.3.4). Rather than scanning the whole state space ($2^m$ states) to find the fixed points, the algorithm examines the fixed-point assertion and directly determines the corresponding states. For example, rule-set $R_1 = \{r_1, r_2, r_3, r_4\}$ by itself has $FP = a \ AND \ b \ AND \ c \ OR \ d$. In the first term the variable $d$ is free, so the fixed points are $abcd$ and $abc\overline{d}$. The fixed points

![Diagram](image-url)
Procedure BU

Begin
Let $V$ be a set of fixed points
Let $E$ be an empty set
Let $X$ be $V$
Repeat
Let $X^*$ be an empty set
Forall rules $r \in R$ such that $s' \xrightarrow{r} s$, where $s \in X$, $s' \in S$, and $s' \notin V$ Do
Add $s'$ to $V$
Add $s' \xrightarrow{r} s$ to $E$
Add $s'$ to $X^*$
End Forall
Let $X$ be $X^*$
Until $X$ is an empty set
End

Figure 12.7  Bottom-up generation of an optimized transition system.

derived from the second term are composed of value 1 for $d$ combined with all combinations of values for variables $a$, $b$, and $c$, yielding $2^3 = 8$ different fixed points.

The optimized state-space graphs have no cycles. This is a result of constraining the states in the system to have at most one out-transition; that is, no two rules $r_1, r_2 \in R$ exist such that $s' \xrightarrow{r_1} s_1$ and $s' \xrightarrow{r_2} s_2$. Consequently, each state in the resulting system will have exactly one reachable fixed point.

The breadth-first search uses two sets, $X$ and $X^*$. $X$ stores the states that are potential candidates for expansion. The states used in the expansion of states in $X$ are added to $X^*$. After the states in $X$ are exhausted, the expansion is continued for the states that have been stored in $X^*$. Note that at any time instant each set stores the states that are equally distant from the fixed point; that is, a fixed point can be reached by firing the same number of rules.

A breadth-first search guarantees that all the fixed points in the resulting system are reached with a minimal number of rules to be fired. In other words, for each state that is not unstable in the original system, the only reachable fixed point in the new system will be the closest one with respect to the number of rules to fire.

The bottom-up approach discovers only the states that are either stable or potentially unstable. All unstable states, as well as cycles that they are a part of, are removed from the system.

Figure 12.8 shows a possible optimized transition system derived for our EQL(B) example program. Comparison with Figure 12.3 reveals that the optimization eliminates the cycles $\overline{abcd} \overset{2}{\rightarrow} \overline{abcd} \overset{4}{\rightarrow} \overline{abc}d \overset{1}{\rightarrow} \overline{abc}d$ and $\overline{abcd} \overset{2}{\rightarrow} \overline{ab}cd \overset{4}{\rightarrow} \overline{ab}cd$ and removes unstable states $\overline{abc}d$ and $\overline{ab}cd$. The optimization arbitrarily breaks the tie of which rule to use for expansion, and thus an alternative system with equivalent response time could have a transition $\overline{abc}d \overset{3}{\rightarrow} \overline{abc}d$ instead of $\overline{abc}d \overset{2}{\rightarrow} \overline{abc}d$. 
Equivalence of States. Although the plain bottom-up derivation of the state-space graph approach outlined above reduces the number of examined states by excluding the unstable states, the number of vertices in the state space graph remains potentially high and further reductions are necessary. The idea is to join the equivalent states. The new optimization algorithm derives a state-space graph with vertices representing one or more equivalent states. These vertices are labeled with the expression that identifies the equivalent states of the vertex.

To distinguish the labeling of a vertex with a single state and with the set of states, we will use the symbols $s$ and $S$, respectively. Thus, for a vertex $S$, $S = \{s : s \in S\}$, all $s$ in $S$ are equivalent. Also, the transition $r$ from a set $S_i$ to a set $S_j$ would mean that for any state in $S_i$ there is a transition $s_i \xrightarrow{r} s_j$ such that $s_i \in S_i$ and $s_j \in S_j$.

Figure 12.9 shows the recursive algorithm that transforms a state-space graph as derived in section 12.4.2 to a graph with equivalent states. Note that the transform-
Figure 12.10  Bottom-up generation of an optimized transition system with derivation of equivalent states.

A bottom-up optimization process uses the operator $s' \rightarrow S$, which means that a transition $s' \rightarrow s$ exists for $s \in S$.

Rather than transforming the optimized state-space graph, we use the ES algorithm that directly derives the system with equivalent states (Figure 12.10). Again, the algorithm uses the bottom-up approach starting from the fixed-point states and the breadth-first search approach to derive the optimal system with respect to the number of rules to fire. Each time the algorithm expands the system with a new vertex, all possible expansions are considered and the vertex with the biggest number of equivalent states is chosen. This greedy approach may contribute to further minimization of the size of the state-space graph.

For example, suppose there are two states, $S_1$ and $S_2$, that are considered for expansion. Let there be two states, $s_a$ and $s_b$, that are not yet included in the state-space graph such that $s_a \rightarrow s_1$, $s_b \rightarrow s_1$, $s_b \rightarrow s_2$, where $s_1 \in S_1$ and $s_2 \in S_2$. The greedy algorithm will generate a set $S_3 = \{s_a, s_b\}$ and establish a transition $S_3 \rightarrow S_1$ instead of using the expansion of $S_2$ with $S_3 = \{s_b\}$.

Figure 12.11 shows a state-space graph with equivalent states constructed using the ES optimization algorithm. Note that instead of the states $\overline{\text{ab}} \overline{\text{cd}}$ and $\overline{\text{ab}} \overline{\text{cd}}$, there is a new state $\overline{\text{ab}} \overline{\text{d}}$ (besides joining the fixed points into a single vertex, two equivalent
states have been found) and that the number of states for \( R_2 \) has halved (two pairs of equivalent states have been identified).

**Multiple-Rule Transitions** To further reduce the number of vertices in the state-space graphs, we use the technique that originates from the notion of *intra-rule-set parallelism* [Cheng, 1993b] and from the idea of utilizing the concurrency for preventing the state explosion [Godefroid, Holzmann, and Pirottin, 1992]. In contrast with the BU and ES algorithms, here we allow transitions to be labeled with a set of rules, \( R \), rather than with a single rule, \( r \).

For every vertex \( S \) in the state-space graph considered for expansion, we find all possible rule-sets so that for a particular set \( R \), each of the rules \( r \in R \) can be used to expand \( S \). In other words, for all rules \( r \in R \) there should exist \( S' \) such that \( S' \xrightarrow{R} S \) and none of the states in \( S' \) is yet included in the transition system.

Furthermore, for every pair of rules \( r_i, r_j \in R, r_i \neq r_j \), the following conditions should hold:

1. **M1.** rules \( r_i \) and \( r_j \) do not potentially disable each other.
2. **M2.** \( L_{r_i} \cap L_{r_j} = \emptyset \), or \( L_{r_i} \cap L_{r_j} \neq \emptyset \) and rule \( r_i \) and \( r_j \) assign the same values to the variables of the subset \( L_{r_i} \cap L_{r_j} \).

(M1) follows the idea of intra-rule-set parallelism [Cheng, 1993b]. Rule \( r_i \) potentially disables \( r_j \) if a state exists where both rules are enabled and firing \( r_i \) results in a state where \( r_j \) is disabled. (M2) guarantees the cycle-free firing of rules in \( R \) for the states in \( S' \), where \( S' \xrightarrow{R} S \). (For a detailed proof, see [Zupan, 1993].)

The algorithm (Figure 12.12) exploits both equivalency of states and allows multiple-rule transitions. It uses the bottom-up approach and breadth-first search. The method is greedy and at each step adds the biggest set of equivalent states to the evolving graph.

For our example EQL(B) program, an optimized state-space graph using the ESM algorithm is shown in Figure 12.13. Note that for rule-set \( R_1 \) the number of vertices is reduced to three, and the rules \( r_2 \) and \( r_3 \) can fire in parallel.
Procedure ESM
Begin
Let $V$ be a set of fixed points
Let $E$ be an empty set
Let $X$ be $V$
Repeat
Let $X^*$ be an empty set
Repeat
Construct a set $T$ of all possible expansions $t_{S,R,S^*}$, such that for every $t_{S,R,S^*}$ and every $s \in S$
- $s \not\in S'$ if $S' \in V$,
- rules in $R$ can fire in parallel, and
- for all $r \in R$, $s \xrightarrow{r} s^*$, where $s^* \in S^*$ and $S^* \in X$.
If set $T$ is not empty Then
Choose $t_{S,R,S^*}$ from $T$ such that $S$ includes the biggest number of states
Add $S$ to $V$
Add $S \xrightarrow{R} S^*$ to $E$
Add $S$ to $X^*$
End If
Until $T$ is an empty set
Let $X$ be $X^*$
Until $X$ is an empty set
End

Figure 12.12 Bottom-up generation of an optimized transition system with generation of equivalent states and multiple-rule transitions.

Figure 12.13 State-space graphs for independent rule-sets $R_1$ and $R_2$ as generated from the EQL(B) program in Figure 12.1, using the ESM algorithm.
While the introduction of multiple-rule transitions minimizes the number of states in the transition system, such a system loses its determinism. A two-vertex section of an imaginary state-space graph shown in Figure 12.14(a) explains the point. Vertex $S_2$ combines six states, and uses three rules $R = \{a, b, c\}$ that can fire in parallel such that $S' \xrightarrow{R} S$. Both rules a and b are enabled at states 2 and 3 and, for example, from launch state 2, two different states 8 and 9 are reachable and can potentially lead to two different stable states (fixed points).

The ESM algorithm does not guarantee the minimal response time for the state-space graphs it generates. This is because ESM does not optimize the rule firings for the states in the same vertex. This shortcoming can be overcome by additionally optimizing each vertex with the ES method. The optimized graph would then be equivalent to the one using the ES method alone. This shows a potential use of the combination of these two methods, when the approach using the ES method is prohibitively expensive because of the large size of the state-space graph.

**Deterministic State Graphs With Multiple-Rule Transitions** To preserve the determinism in the state graphs with multiple rules labeling a single edge, additional constraints have to be imposed. The constraints should enforce the mutual exclusivity of the enabling conditions of the resulting rules; that is, in each state of the optimized transition system no more than one rule can fire. For example, a system from Figure 12.14(a) can have a corresponding deterministic system shown in Figure 12.14(b). The tradeoff is the increased complexity of the state-space graph.
in our example, the deterministic system includes two more vertices and one more transition.

The ESM algorithm can be modified for the purpose of deriving the deterministic system. We propose a set of constraints over the set of rules to be fired in parallel. If $S$ is a vertex to be expanded, then rules $a$ and $b$ can fire in parallel and together appear in a label of an edge incident on vertex $S$ if and only if:

1. states $s_a$ and $s_b$ do not exist such that $s_a \xrightarrow{a} s$ and $s_b \xrightarrow{b} s$, where $s$ is the state in $S$,
2. state $s'$ does not exist such that $s' \xrightarrow{a} s_a$ and $s' \xrightarrow{b} s_b$, where $s_a$ and $s_b$ are the states in $S$, and
3. rules $a$ and $b$ do not enable each other; that is, if rule $a$ is disabled, firing rule $b$ does not enable rule $a$, and vice versa.

(I1) guarantees that the set of reachable states for these two rules are disjoint. Furthermore, the two rules cannot be enabled at the same primitive states of $S'$, $S' \xrightarrow{R} S$ and $a, b \in R$, because of the constraints (I2) and (I3). We will refer to this modified algorithm as ESMD.

We believe that in addition to the set of constraints mentioned above, other similar sets exist that could possibly lead to similar or better results. Future research work dealing with the determinism of the transition system should address this issue.

12.4.3 Synthesis of an Optimized EQL(B) Program

A new EQL(B) 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:

$$EC_i^{New} = \left( \bigvee_{S \xrightarrow{r_i} S', r_i \in R} S \right) \land EC_i.$$

$S$ and $S'$ are labels of two vertices in the state-space graph that are connected with the edge labeled with $r_i$ or with the set that includes $r_i$. In the case of the state-space graph having all edges 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 same assignment parts as the original rules and the new enabling conditions. Rules not included in any constructed state-space graph are redundant and are not added to the new rule base.

The optimized EQL(B) program constructed using the optimization algorithm BU or ES is shown in Figure 12.15. Because of the inclusion of state $aba\bar{c}\bar{a}$ to the enabling condition of rule $r_3$, the optimization algorithm ESM derives a slightly different re-
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 b=0 AND c=1 AND d=0
  (* 3 *) [] a:=1 ! c:=1 IF a=0 AND b=1 AND d=0
  (* 5 *) [] d:=0 ! a:=1 IF a=1 AND c=1 AND d=1
END.

Figure 12.15  An optimized EQL(B) program derived from the program in Figure 12.1, using either the BU or ES algorithm.

PROGRAM an_optimized_eql_b_program_2;
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 b=0 AND c=1 AND d=0
  (* 3 *) [] a:=1 ! c:=1 IF a=0 AND b=1 AND d=0 OR
            a=0 AND c=1 AND d=0
  (* 5 *) [] d:=0 ! a:=1 IF a=1 AND c=1 AND d=1
END.

Figure 12.16  An optimized EQL(B) program derived from the program in Figure 12.1, using the ESM algorithm.

sult (Figure 12.16). Note that rule r₄ was found to be redundant and does not appear in any of the resulting programs.

12.5 EXPERIMENTAL EVALUATION

We have implemented three of the four optimization algorithms presented here, and intentionally bypass the implementation of the BU algorithm due to the possible combinatorial complexity of state-space graph generation. This section attempts to experimentally evaluate the performance of these optimization algorithms on two sets of computer-generated EQL programs and on a commercial rule-based expert system.

Because of the general unavailability of analysis tools for real-time rule-based systems, it is hard to assess the performance of optimization methods proposed here. Such analysis tools should derive the number of rules to be fired to reach a fixed point for both original and optimized rule-based programs and should analyze the stability of the programs. For EQL rule-based programs, the Estella analysis tool (described in chapter 10) has the potential to estimate both. Estella can discover potential cycles, and, when no cycles exist, Estella can estimate an upper bound of the number of
rules to fire to reach a fixed point. Unfortunately, Estella imposes several constraints to rules in EQL programs it can analyze, and even if these constraints are met, it might discover a potential cycle even if one does not exist. Estella’s assessment of stability is sufficient, but not necessary. That is, if no potential cycles are found, the program is guaranteed to be stable, but the discovery of a potential cycle might not ensure that one really exists.

To clearly state the difference between the original and optimized programs, a precise upper bound of the number of rules to fire to reach a fixed point is required. For this purpose, we have built a system like the one described in [Browne, Cheng, and Mok, 1988] that takes an EQL program, converts it to a C-language routine, and then uses it to build an exact state-space graph. Next, the system performs an exhaustive search to find if the EQL program is stable and to derive a precise upper bound of the number of rules to fire to reach a fixed point. In case of cycles, such a bound is given as a maximum number of rules to fire to reach a fixed point from some state without going into a firing cycle. On one side, such estimation of the bound is pessimistic, because it assumes a rule-firing scheduler to select the rules to fire so that it does not find an optimal and shortest path to a fixed point. On the other side, such estimation may be very optimistic, since it assumes the scheduler would avoid any cycle.

Although the analysis method described above is general and could be used with any EQL(B) program, due to substantial computer resources required and high time complexity, it works only with EQL(B) programs that use only a few variables. For this reason, two sets of relatively simple EQL(B) programs that allow for such analysis but that are still complex enough to show the advantages and disadvantages of the proposed optimization methods were computer-generated.

12.5.1 EQL(B) Programs without Cycles

A set of five cycle-free EQL(B) programs were computer-generated. The generator is given the number of rules and variables in the program, and the desired average complexity of the action and condition parts. The action part complexity is proportional to the number of assignments and the enabling condition complexity is proportional to the number of Boolean operations in the rules’ enabling conditions. All test programs were generated to consist of only one independent rule-set. Additional constraints were imposed for the programs to meet one of the special form requirements and therefore to be acceptable for analysis with the Estella tool. Both Estella and the exact state-space graph–based analysis described above were then used to, respectively, estimate and derive the precise upper bound of the number of rules to fire.

Table 12.1 presents the results of the analysis of generated EQL(B) programs and their corresponding optimized versions. Algorithms ES and ESM were used for the optimization. We observed the following:

- Optimization always resulted in a program with the same (P2 and P4) or lower number (P1, P3, and P5) of rule firings to reach a fixed point.
TABLE 12.1 Analysis results for unoptimized (U) programs P1 . . . P5 and their corresponding optimized programs using ES and ESM optimization algorithms

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>ES</th>
<th>ESM</th>
<th></th>
<th>U</th>
<th>ES</th>
<th>ESM</th>
<th></th>
<th>U</th>
<th>ES</th>
<th>ESM</th>
<th></th>
<th>U</th>
<th>ES</th>
<th>ESM</th>
</tr>
</thead>
<tbody>
<tr>
<td>#R</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>#V</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>#rules to fire</td>
<td>3(5)</td>
<td>2(4)</td>
<td>2(5)</td>
<td>1.8</td>
<td>10.0</td>
<td>13.0</td>
<td>1.8</td>
<td>4.7</td>
<td>4.3</td>
<td>6</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#oper/rule</td>
<td>7.2</td>
<td>7.8</td>
<td>10.0</td>
<td>3.8</td>
<td>3.8</td>
<td>4.0</td>
<td>12</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#vars/rule</td>
<td>1.5</td>
<td>11.8</td>
<td>8.6</td>
<td>1.6</td>
<td>4.7</td>
<td>4.8</td>
<td>9</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#vertices</td>
<td>2.3</td>
<td>11.7</td>
<td>8.0</td>
<td>2.1</td>
<td>5.0</td>
<td>4.0</td>
<td>18</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>10</td>
<td>9</td>
<td>3</td>
<td>10</td>
<td>6(7)</td>
<td>5(7)</td>
<td>4(7)</td>
<td>7.2</td>
<td>114.4</td>
<td>143.7</td>
<td>4.5</td>
<td>9.6</td>
<td>9.9</td>
<td>37</td>
<td>6</td>
</tr>
</tbody>
</table>

#R = number of rules in a program. #V = number of variables used in a program. #rules to fire = maximum number of rules to fire to reach a fixed point (a precise number of rules to fire is given, a number estimated by Estella is given in parentheses). #oper/rule = average number of logical AND, OR, and NOT operations required to evaluate an enabling condition of a rule. #vars/rule = average number of variables used in enabling condition of a rule. #vertices = number of vertices in the state-space graph derived by optimization algorithm.

- For every example program, the optimization found some rules to be redundant.
- The complexity metrics for enabling conditions of the rules (#oper/rule and #vars/rule) indicate that the optimized programs have more complex enabling conditions. The reason for this is that the optimization is actually a process of removing the unstable states with transitions that lead into cycles. After optimization, the enabling conditions are satisfied for equal or usually fewer states than in the original program. Even though we have applied Boolean minimization to the resulting enabling conditions, the increased specialization usually leads to increased complexity of enabling conditions.
- As expected, compared to algorithm ES, algorithm ESM always found a more compact state-space graph with fewer vertices.
- All optimized programs satisfy Estella constraints and enable Estella to be used for analysis. This is a rather special case, and although cycle-free, the optimized programs do not always satisfy Estella constraints. Furthermore, because of the approximate nature of Estella analysis, this may not be sufficient to judge the optimization of an upper bound of the number of rules to fire to reach a fixed state. For example, based on the sole analysis of Estella, one could conclude that algorithm ES optimized program P2 but did not optimize P5, which is in contradiction with a finding based on derivation of the precise upper bound of the number of rules to fire. In all cases, however, Estella found the optimized programs to be stable—the result expected due to the nature of our optimization method.

12.5.2 EQL(B) Programs with Cycles

Similar to programs from Table 12.1, a set of six unstable EQL(B) programs were generated (Table 12.2). These programs have state-space graphs that include unstable and potentially unstable states. An upper bound of the number of rules to fire to reach
TABLE 12.2 Analysis results for unoptimized (U) programs S1 . . . S6 and their corresponding optimized programs using ES and ESM optimization algorithms

<table>
<thead>
<tr>
<th></th>
<th>#R</th>
<th>#V</th>
<th>#rules to fire</th>
<th>#oper/rule</th>
<th>#vars/rule</th>
<th>#vertices</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>ES</td>
<td>ESM</td>
<td>U</td>
<td>ES</td>
<td>ESM</td>
<td>U</td>
<td>ES</td>
<td>ESM</td>
<td>ES</td>
</tr>
<tr>
<td>S1</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1(1)</td>
<td>6.2</td>
<td>5.7</td>
<td>6.5</td>
<td>2.8</td>
</tr>
<tr>
<td>S2</td>
<td>9</td>
<td>9</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>3(4)</td>
<td>5.7</td>
<td>5.7</td>
<td>8.2</td>
</tr>
<tr>
<td>S3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>1.8</td>
<td>11.2</td>
<td>10.5</td>
</tr>
<tr>
<td>S4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1.8</td>
<td>10.7</td>
<td>10.7</td>
</tr>
<tr>
<td>S5</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>4</td>
<td>4</td>
<td>7.2</td>
<td>97.6</td>
<td>58.7</td>
</tr>
<tr>
<td>S6</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>4(5)</td>
<td>2.3</td>
<td>11.0</td>
<td>9.2</td>
</tr>
</tbody>
</table>

%pus = number of potentially unstable states as a percent of all states in unoptimized program. For an explanation of other abbreviations used, see Table 12.1.

a fixed point was derived for original and optimized programs as described in the previous section.

From experimental results shown in Table 12.2, similar conclusions to those in the previous section can be drawn. Most important, optimization always reduced the number of rules to fire to reach a fixed point. Again, compared to ES, algorithm ESM derived the state-space graphs with considerably fewer numbers of vertices. Also, due to the specialization of rules, optimized rules have more complex enabling conditions than unoptimized rules.

A number of the optimized programs (S1, S2, and S6) satisfied Estella’s constraints and enabled Estella to estimate the upper bound of the number of rules to be fired. The effect of optimization to enable this particular use of the Estella tool is not general, but one might find it useful when it takes place.

The programs in both Tables 12.1 and 12.2 were also optimized using algorithm ESMD. This gave similar results as when algorithm ES was used, resulting in state-space graphs with much higher numbers of vertices than in those generated by algorithm ESM. This is due to the high specificity of constraints that algorithm ESMD imposes over the rules to fire in parallel in order to derive a deterministic state-space graph.

12.5.3 Industrial Example: Optimization of the Integrated Status Assessment Expert System

To demonstrate the applicability of the proposed optimization techniques, we used it to optimize the Integrated Status Assessment Expert System (ISA) [Marsh, 1988]. This real-time expert system, originally written by Mitre in OPS5, identifies the faulty components in the planned NASA Space Station. Its EQL equivalent is described in [Cheng et al., 1993], and for the purpose here we have converted it to an EQL(B) program that consists of 35 rules and 46 variables.

In the first optimization step, ISA is decomposed into 17 independent rule-sets. The analysis with the Estella tool shows that for only four rule-sets, either the potential number of firings to reach the fixed point is greater than one, the rule-set has
TABLE 12.3 Independent rule-sets of ISA with cycle (ISA1), with more than one rule to fire to reach a fixed point (ISA2 and ISA3), or not conforming to constraints to be analyzable by Estella (ISA4)

<table>
<thead>
<tr>
<th>ID of rules in a subset</th>
<th>#vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA1</td>
<td>#3, #6, #18, #34, #35</td>
</tr>
<tr>
<td>ISA2</td>
<td>#11, #19</td>
</tr>
<tr>
<td>ISA3</td>
<td>#9, #17</td>
</tr>
<tr>
<td>ISA4</td>
<td>#8, #16, #32, #33</td>
</tr>
</tbody>
</table>

#vars = number of variables used in a rule set.

cycle, or the rules in the set are not analyzable with Estella. Table 12.3 lists the IDs of rules in these sets and gives the number of variables used within each set.

For rule set ISA1, the Estella analysis tool identifies a possible cycle involving rules #34, #10, and #18 (see [Cheng et al., 1993] for the similar analysis that discovered a potential cycle in ISA). For ISA2 and ISA3, Estella estimated the upper bound of rule firings to reach a fixed point to be 2. Rules in ISA4 do not meet Estella’s constraints and could not be further analyzed with Estella.

As for EQL(B) computer-generated programs that were presented in this section, the derivation of the precise upper bound of the number of rules to fire to reach a fixed point is needed to clearly evaluate the effects of the optimization. Due to the relatively low number of variables used, the analysis for ISA2, ISA3, and ISA4 that uses the exact derivation of the complete state-space graph as described in the beginning of this section is possible. To make such exact analysis possible for ISA1, we have changed its rules to obtain an equivalent but analyzable system. Namely, we have observed that several variables appear in ISA1 only in the enabling conditions, and we have found that some pairs of such variables could be replaced by a single variable. For example, if the expressions \( a=\text{TRUE AND } b=\text{TRUE} \) and \( a=\text{FALSE AND } b=\text{FALSE} \) are the only ones that involve \( a \) and \( b \) and are found among rules in a specific subset, a new variable \( c \) may be introduced and used instead of \( a \) and \( b \), so that \( c=\text{FALSE} \) indicates that both \( a \) and \( b \) are \( \text{FALSE} \) and \( c=\text{TRUE} \) indicates that both \( a \) and \( b \) are \( \text{TRUE} \). Two expressions involving \( a \) and \( b \), respectively, are then changed to \( c=\text{TRUE} \) and \( c=\text{FALSE} \). Such manual alteration of rules reduced the number of variables used in ISA1 to 14.

The optimization algorithms ES and ESM were then used on the four ISA rule-sets mentioned. The exact analysis with the use of the complete state-space graph shows that:

- For ISA1, as Estella analysis suggested, the unoptimized set of rules really is unstable. The longest path from some state to a fixed point that does not involve a cycle requires firing four rules. A corresponding optimized rule set is cycle-free and takes a maximum of three rules to be fired to reach a fixed point. The average number of logical AND, OR, and NOT operations to evaluate an
enabling condition of a rule increased from 15.6 to 58.8. Algorithms ES and ESM derived the graph with five vertices.

- For ISA4, the unoptimized rule-set is cycle-free and requires a maximum of five rules to be fired to reach a fixed point. A corresponding optimized set required a maximum of four rules to be fired. The average number of operations to evaluate an enabling condition increased from four to 24. While algorithm ESM derived a state space consisting of nine vertices, the one derived by ES used 14 vertices.
- In the case of ISA2 and ISA3, an optimal rule-set would still need a maximum of two rule firings to reach a fixed state, so no optimization of the original rule-set is required.

12.6 COMMENTS ON OPTIMIZATION METHODS

This section provides a qualitative comparison of the optimization methods, comments on the constraints over EQL language required by optimization algorithms, and considers the optimization of other real-time rule-based systems.

12.6.1 Qualitative Comparison of Optimization Methods

In the previous section we presented several techniques for the optimization of rule-based systems. They were all based on a bottom-up approach for derivation of an optimized state-space graph. All remove the unstable states and generate a state graph with only stable states. Potentially unstable states of the unoptimized program are transformed into stable states in the optimized system by removing the transitions that were responsible for cycles. Optimization alters the enabling condition part of rules, leaving the action part unchanged. Although these methods exhibit many similarities, they vary in complexity (execution time and required memory space) and in the optimization results they achieve.

Table 12.4 qualitatively compares the optimization methods. It shows the trade-off between minimizing the number of rules to fire and minimizing the size of the state-space graphs. While BU and ES can guarantee minimization of rule firings, the number of vertices in the state-space graph generated by ESM is expected to be

<table>
<thead>
<tr>
<th>TABLE 12.4 Optimization algorithms and their features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
</tr>
<tr>
<td>Exploits equivalence of states</td>
</tr>
<tr>
<td>Exploits rule-firing concurrency</td>
</tr>
<tr>
<td>Removes cycles and unstable states</td>
</tr>
<tr>
<td>Stabilizes potentially unstable states</td>
</tr>
<tr>
<td>Minimizes number of rules to fire</td>
</tr>
<tr>
<td>Generates deterministic transition system</td>
</tr>
</tbody>
</table>
lower. ESMD minimizes the number of the rule firings, but as shown in the following discussion, does not perform well with respect to the state-space graph complexity minimization.

12.6.2 On Constraints over EQL Language Required by Optimization Algorithms

To enable the use of the proposed optimization techniques, several constraints were imposed on EQL rule-based programs. First, a two-valued (Boolean) EQL(B) variant of the EQL language is used. In section 12.3.1 we showed that the conversion from multi-valued EQL program to corresponding two-valued EQL(B) is straightforward. EQL(B) rules are further constrained to use only constant assignments in the sub-actions (section 12.3.1). This simplifies the derivation of fixed points and is then implicitly used also in the conditions for rule-based decomposition (D1a), (D1b), and (D2), section 12.4.1) and for parallel rule firing (M1) and (M2), section 12.4.2).

To use the proposed optimization technique also for EQL(B) programs with nonconstant assignments, a possible solution is to convert them to corresponding programs with constant assignments. Such a conversion is straightforward and we illustrate it through an example. Consider the following rule:

\[ v_1 := \text{Exp1} \land v_2 := \text{Exp2} \land \text{EC} \]

where Exp1 and Exp2 are nonconstant expressions and EC is the enabling condition.

This rule can be converted to its constant-assignment equivalent:

\[ v_1 := \text{TRUE} \land v_2 := \text{TRUE} \land \text{EC} \]

\[ \neg v_1 := \text{TRUE} \land v_2 := \text{FALSE} \land \text{EC} \]

\[ \neg v_1 := \text{FALSE} \land v_2 := \text{FALSE} \land \text{EC} \]

\[ \neg v_1 := \text{FALSE} \land v_2 := \text{TRUE} \land \text{EC} \]

In general, for a rule with \( k \) nonconstant assignments, such a conversion would replace this rule with \( 2^k \) new rules with constant assignments.

A different approach would be to allow EQL(B) programs to have nonconstant assignments, but to change the optimization algorithm to handle such cases as well. This would require a more sophisticated fixed point derivation algorithm, Derive\( _F \)\( F \)\( P \)\( _2 \) (section 12.3.4). New constraints that would handle rules with nonconstant assignments for decomposition and parallel rule firing could then be adapted from existing ones used for analysis and parallelization of such rule-bases as described in [Cheng et al., 1993] and [Cheng, 1993b].

Several restrictions were imposed that influenced the generation of state-space graphs and defined which state may be merged in a single vertex and thus reduce the number of vertices in state-space graphs. Through experimental evaluation (section 12.5), we have shown that this restriction may still allow the construction of a reduced state-space graph of relatively low complexity.
12.6.3 Optimization of Other Real-Time Rule-Based Systems

The equational rule-based EQL language was initially developed to study the rule-based systems in a real-time environment. EQL’s simplicity is due to its use of zero-order logic. To increase the expressive power of such production language, a similar fixed-point interpretable Macro Rule-based language (MRL) that uses first-order logic was derived from EQL [Wang, Mok, and Cheng, 1990]. Consequently, many algorithms for decomposition, response-time analysis, and parallelization of EQL (see [Cheng et al., 1993; Chen and Cheng, 1995b; Cheng, 1993b]) were then reimplemented for MRL [Wang and Mok, 1993]. It was shown that the expressive power of MRL and a popular production language, OPS5, is the same, and to enable the use of response-time analysis tools for OPS5, translation methods from OPS5 to MRL and vice versa were proposed [Wang, 1990c]. Furthermore, the analysis tools that originated from those of EQL and consequently from those of MRL were recently developed for OPS5 as well (see chapter 11).

The importance of the above-mentioned work for the generality of the methods proposed here is that the concepts used for decomposition, response-time analysis, and parallelization of EQL, most of which were adopted and used here, were then defined and used for MRL and OPS5 production systems. For example, the optimization of MRL might as well use state-space graphs that would be derived starting from the fixed points and would gradually be expanded to exclude the cycles. Again, a breadth-first search would optimize the response time of MRL assessed through the number of rules to fire to reach a fixed point. The main difficulty in implementing such optimization is due to different logics used in EQL and MRL: while EQL is zero-order-logic-based, MRL uses first-order logic. In other words, the major task of adapting the optimization methods used for EQL to those for MRL would be the reimplementations of symbolic manipulation routines that handle expressions (logical operators, minimization). This may be a difficult task, but once accomplished it would enable the development of proposed optimization methodology for MRL and for similar first-order-logic-based production languages.

12.7 HISTORICAL PERSPECTIVE AND RELATED WORK

The timing analysis problem is similar to that of protocol reachability analysis. Reachability analysis tries to generate and inspect all the states of a system that are reachable from a given set of initial states [Holzmann, 1991]. This top-down approach requires that either a set of initial states is given, or, as advocated by [Valmari and Jokela, 1989], the model of the environment is known from which the initial states are derived. Reachability analysis tools use algorithms such as hashing without collision, state-space hashing, and similar techniques to reduce the number of the states stored in the working memory. These methods are used only for analysis and limit their usability in synthesis and optimization because they generate only a part of the state-space graph. [Bouajjani, Fernandez, and Halbwachs, 1991; Bouajjani et al., 1992] presented a useful method to reduce the complexity of state-space graphs that is based on the identification of equivalent states. [Godefroid, Holzmann,
and Pirottin, 1992] showed that most of the state explosion due to the modeling of concurrency can be avoided. They annotated the vertices of a state-space graph with sleep sets. If the rule that is enabled in a certain state is also in the sleep set of the same state, its firing would result in a state that was already checked in the earlier stage of the reachability analysis.

To overcome the prohibitive complexity of state-space graphs in the analysis problem, [Cheng et al., 1993] presented the static analysis method which determines if a program has a finite response time. They have identified several special forms of rules: if all rules belong to one of such forms, the program’s response time is finite. Recent work of [Chen and Cheng, 1995b] focused on the methods that extend this approach by further estimating the program’s response time in terms of the number of rules to fire to reach a fixed point.

[Cheng, 1993b] proposed an approach to decompose the rule-based system into independent rule-sets. This allows for independent analysis of each such rule-set. The overall response time is then derived from the response times of the independent rule-sets. A similar modular approach in the analysis and synthesis of rule-based systems is also advocated in [Browne et al., 1994].

A different approach to response-time optimization is to speed up the rule-based system by a parallel-rule execution. For example, [Kuo and Moldovan, 1991] proposed a parallel OPS5 rule firing model. They used a notion of context, which groups the rules that interfere with each other. Several contexts can be executed in parallel if they are mutually independent. [Cheng, 1993b] proposed a similar rule-firing model for EQL programs and further investigated the possibility of parallel firing of the rules that belong to the same context. In terms of the synthesis problem, a rule-base can be rewritten to increase the parallelism. [Pasik, 1992] introduced the constraint copies of so-called culprit rules, thus balancing the work performed in parallel and reducing the amount of work done sequentially. Culprit rules are rules that require substantially more computation time, which is due to their condition elements that require comparisons to many more working elements than other rules. While the culprit rules have a modified condition part, their action part is the same as that of the rules they are derived from. Another example of speeding up the execution of the production system is Ishida’s algorithm [Ishida, 1994], which enumerates possible joint structures for OPS-like systems and selects the best one.

Although Pasik and Ishida both addressed the execution time, their methods do not explicitly address problem of fulfillment of the timing constraints and do not estimate the upper bound of execution time. Increased parallelization and execution speedup as proposed above can thus be used to reduce the execution time, but does not give an adequate solution to synthesis and optimization problems for real-time systems as stated in [Browne, Cheng, and Mok, 1988]. [Zupan and Cheng, 1994a; Zupan and Cheng, 1994b] first propose the optimization techniques described here, which are refined and extended in [Zupan and Cheng, 1998].
12.8 SUMMARY

We have presented a novel approach to the optimization of rule-based expert systems. We proposed several methods, all based on a construction of the reduced state-space graph corresponding to the input rule-based system. The optimization methods focus on changing the rules’ enabling conditions while leaving their assignment parts unchanged.

The new and optimized rule-based expert systems are synthesized from the derived state graph. The vertices of the state graph are mutually exclusive. This, together with the cycle-free nature of the state graph, contributes to the special properties of the rule-based system constructed from it. In comparison with the original system, the optimized rule-based systems have the same or fewer numbers of rule firings to reach the fixed point. They contain no cycles and are thus inherently stable. Redundant rules present in the original systems are removed. Three of the four optimization methods proposed derive a deterministic system; that is, each launch state in a system will always have a single corresponding fixed point. This is obtained by enforcing only a single rule to be enabled at each state. For the same reason, the use of the conflict resolution for such systems becomes obsolete.

The proposed optimization strategies can also be used for analysis purposes. Namely, they all implicitly reveal the unstable states in the system. All stable and originally potentially unstable states are included in the enabling conditions of optimized rules. Subtracting these from the states included in the enabling conditions of the unoptimized rules identifies the unstable states.

In this chapter, we have constrained the class of EQL programs to have constant assignments only. For unconstrained programs, the same approach can be used. The only major difference is the identification of fixed points. To avoid an exhaustive search of the state space, we show in [Zupan, 1993] that a low-complexity fixed-point derivation algorithm exists if the rule-sets belong to a specific special form. This is a much lesser constraint than in the constant assignments case. Ongoing work is being performed to identify different algorithms to find fixed points for the rules belonging to different special forms.

No information other than the EQL program itself is given to the optimization method. The methods would possibly benefit from environmental and execution constraints, for example, the knowledge about impossible or prohibited states or the knowledge of prohibited sequences of rule firings. Such constraints could be effective in reducing both the execution complexity of optimization and the complexity of the resulting state-space graphs.

The optimization methods were evaluated on several randomly generated rule-based programs and on the real-time rule-based expert system developed by Mitre [Marsh, 1988]. The experiments confirm that optimization techniques proposed may indeed reduce the number of rules needed to fire and stabilize the rule-based system. The experiments also suggest that proposed optimization methods should not be used alone, but rather in combination with the analysis tools. Optimization of rule-based systems should then be an iterative process of discovering which rule-set (or even
which set of rules in a rule-set) to optimize, and, depending on the desired properties or the specific bias in the optimization, which optimization method to use.

**EXERCISES**

1. Describe two approaches for optimizing rule-based systems.
2. How does the high-level dependency (HLD) graph group related rules together into vertices consisting of a set of rules?
3. Transform the following EQL programs into equivalent EQL(B) programs. Let all the variables be four-valued variables with their values $\in \{-1, 0, 1, 2\}$.

**PROGRAM EQL1**

```
INPUTVAR
    a, b : INTEGER;
VAR
    c : INTEGER;
INIT
    c := 0
RULES
    (*r1*) c := 1 IF a > 0 and b > 0
    (*r2*) c := 2 IF a > 0 and b <= 0
END.
```

**PROGRAM EQL2**

```
INPUTVAR
    a : INTEGER;
VAR
    d, g, h : INTEGER;
INIT
    d := 0, g := 0, h := 0
RULES
    (*r1*) d := 2 IF a <= 0
    (*r2*) g := 1!h := 1 IF a > 1 and b > 1
    (*r3*) g := 2!h := 2 IF a <= 1
END.
```

4. Construct the state-space graph of the EQL(B) programs obtained in exercise 2.
5. Construct the rule-dependency graph and the HLD graph corresponding to the EQL(B) programs obtained in exercise 2.
6. Describe extensions to the optimization algorithms or propose new algorithms for EQL(B) programs with nonconstant assignments.