DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Message Based Event Specification For Debugging Nondeterministic Parallel Programs

Suresh K. Damodaran-Kamal
Distributed Computing Group
M/S B272, CIC-8,
Los Alamos National Laboratory,
Los Alamos, NM-87545
email: suresh@lanl.gov

Joan M. Francioni
Computer Science Department
P.O. Box 41771,
University of Southwestern Louisiana,
Lafayette, LA-70504
jf@cacs.usl.edu

Abstract

Portability and reliability of parallel programs can be severely impaired by their nondeterministic behavior. Therefore, an effective means to precisely and accurately specify unacceptable nondeterministic behavior is necessary for testing and debugging parallel programs. In this paper we describe a class of expressions, called Message Expressions that can be used to specify nondeterministic behavior of message passing parallel programs. Specification of program behavior with Message Expressions is easier than pattern based specification techniques in that the former does not require knowledge of run-time event order, whereas that latter depends on the user's knowledge of the run-time event order for correct specification. We also discuss our adaptation of Message Expressions for use in a dynamic distributed testing and debugging tool, called mdb, for programs written for PVM (Parallel Virtual Machine).

Keywords: Event Specification, Debugging, Testing, Message Passing, Distributed Computing.

1 Introduction

One difficulty with implementing a parallel program is that a parallel program can display nondeterministic behavior [9] that can severely impair its portability and reliability. In order to detect and correct unacceptable nondeterministic behavior, a simple means to specify such
behavior is necessary. Therefore, an effective means to precisely and accurately specify unacceptable nondeterministic behavior is an important requirement for testing and debugging parallel programs.

Message passing parallel programs use send and receive statements to communicate among the processes. Nondeterministic behavior of message passing parallel programs results in the receipt of different messages in different executions (with the same input) at the same receive statement in a program. Therefore, nondeterministic behavior is an important event of interest that needs to be specified. However, such a specification is made more difficult by the fact that a nondeterministic program is not necessarily erroneous. On the contrary, many scientific parallel programs exhibit nondeterminism, and yet work correctly. For example, a process that sums up the contents of whatever messages it receives at a receive statement will produce the correct result irrespective of the order in which the messages arrive at the receive. Thus, we need a technique to specify what exactly constitutes an unacceptable nondeterministic behavior, at the same time allowing for specification of acceptable nondeterministic behavior.

The specification languages that exist today can be broadly classified as pattern oriented specification languages. These languages specify patterns in the run-time behavior of programs. The patterns of behavior may consist of either control or data paths. Bruegge and Hibbard discuss the use of generalized path expressions for verification of execution paths [3]. Hseush and Kaiser extend path expressions to support expression of data paths in parallel programs [13]. Baiardi et. al. discuss the use of Behavioral Expressions (BE) that are capable of defining predicates on the sequences of primitive events [2]. A sophisticated assertion language aimed at occam language is discussed by Auguston and Fritzson in [1].

Event based abstraction [12] allows the viewing of events at a higher level of abstraction. Scalability of this approach is demonstrated in the design of Ariadne debugger [4]. The pattern oriented techniques are general purpose techniques. The events can range from creation of a process to the presence of a specific type of message. The pattern oriented specification is most useful when the pattern of execution is known a priori. However, nondeterministic behavior is difficult to specify a priori as a run-time behavioral pattern because a program with nondeterministic behavior can execute in many ways for the same input. Enumerating all patterns of run-time events corresponding to the execution of a nondeterministic program for nontrivial programs can prove to be very cumbersome. The use of time and causal relations among run-time events may be unavoidable in real-time systems, but is tedious and requires an intimate knowledge of the run-time characteristics of the program. Most of the scientific parallel programs are not real-time programs, and therefore, we look for other methods of specifying run-time behavior.

In this paper, we describe a significantly different approach to run-time event specifi-
cation. In our approach, we use Message Expressions. Message Expressions are similar to expressions in common programming languages such as C. The main difference is that Message Expressions make use of an additional operator, called the prime operator. Use of the prime operator permits the user to express nondeterministic behavior based simply on the variables in a message, without any need to refer to the relative order of run-time events. This aspect of Message Expressions frees the user from the need to think in terms of run-time events. Instead, the user can think directly in terms of the source code in the program. To write a Message Expression one needs to know only the syntax and semantics of the prime operator in addition to the syntax and semantics of common program expressions. Since Message Expressions can be evaluated based on messages alone, there is no need to access program data for such evaluation. Message Expressions may be used to either to detect events at run-time, or to detect events from traces.

To implement Message Expressions we preprocess the source of the parallel program. This preprocessing generates the structures that are necessary for the interpretation of the messages at run-time. Evaluation of Message Expressions is also possible based on trace files. Such an approach may be useful for large applications where run-time evaluation of Message Expressions is infeasible. Moreover, Message Expressions can complement other forms of specification that may exist in any specific debugger, or performance analyzer. The detection of a message sent to a process with a specific set of attributes can be accomplished by Message Expressions. Message Expressions can also be used to continuously monitor (or display) the value of some field of a message.

We also demonstrate using examples how to specify nondeterministic behavior based only on the properties of messages. Message Expressions are particularly useful in message based debugging, a technique for introducing break points or watch points in programs based on the properties of messages [5]. We have implemented Message Expressions successfully in a dynamic distributed debugger, called mdb[7], for Parallel Virtual Machine (PVM)[11] programs.

We show that nondeterministic behavior in a message passing program can be specified based on message contents (Section 2). What constitutes acceptable (or unacceptable) nondeterministic behavior can be specified precisely using Message Expressions in two ways (Section 3). Our implementation of Message Expressions in a dynamic distributed debugger is discussed in Section 4. We also discuss the use of Message Expressions for specifying other events of interest in Section 4. We conclude in Section 5.
2 Characterizing Nondeterministic Behavior

From a theoretical point of view, one can term a program as determinate, or nondeterminate, depending on whether a program always gives the same result, or whether we can show that a program can possibly have different results [10]. However, from the point of view of programmers, Emrath and Padua in [10] find it useful to further refine the notion of determinacy. This refinement is used to derive a nomenclature for various kinds of determinacy and nondeterminacy for shared memory parallel programs. We will do a similar refinement of the notions of determinacy and nondeterminacy for message passing parallel programs as a prelude to describing mechanism to specify determinacy and nondeterminacy based on messages in Section 2.2.

This characterization of nondeterministic behavior will be based on a model of message passing computation that we shall describe here. A message passing parallel program uses send and receive operations to communicate through messages among its processes. These messages are delivered to the individual processes by an underlying mail system. Each message has a message type (or msgid). A receive operation specifies one or more msgids, indicating that the receive will accept only messages that belong to the msgids specified. A receive operation always differentiates messages with differing msgids after their receipt. The send operations are nonblocking, and receive operations are blocking. A blocking operation will cause the process executing the operation to block until the operation is completely carried out. For simplicity, we assume that the number of processes in a program is fixed. A fail safe mail system is assumed to exist that is responsible for the delivery of messages to processes.

A convenient representation of a program’s execution is in the form of a space-time diagram. In a space time diagram, time flows from top to bottom, the vertical lines represent different processes, and the diagonal arrows represent messages (see Figure 1). Each receive operation specifies the message type(s) of messages it can accept.

In this section, we first characterize nondeterministic (or deterministic) behavior of a program based on the properties of the program variables. Subsequently, we show how a message based specification can be used to do the same thing.

2.1 Variable Based Characterization

Specification of the nondeterministic(deterministic) behavior of a program based on the property of one or more program variables is called a variable based specification. Karp and Miller [14] were the first to define determinacy of a program based on the attributes of program variables more than a quarter century ago. Though Definition 1.9 in [14] is based on the interpretation of a schema, we restate their definition in the context of parallel pro-
grams in Definition 2.1. However, similar to Emrath and Padua [10] (also see the beginning paragraph of Section 2), we find it more appropriate to call the notion of determinacy of Karp and Miller as internal determinacy. For the sake of simplicity, we assume that the output(s) produced by a program are stored in one or more program variables before written elsewhere.

**Definition 2.1 Internally Determinate Programs:** A program is **internally determinate** iff the entire sequence of values written into each program variable remains unchanged in all possible executions of the program with the same input. A program variable is an **internally determinate** program variable, if in all possible executions with the same input the sequence of values written into this variable is the same.

Note that in an internally determinate program all program variables are internally determinate.

In practice, it is not necessary for all parallel program variables to have the same sequence of values written into each of them for the program to be correct. In particular, it is possible that a programmer is concerned with only whether some specific variables of interest always have the same sequence of values written into them for all executions with the same input.

For example, consider the message deliveries involved in copying an array \( \mathbf{x} \) in two processes (\( P_1 \) and \( P_3 \) in Figure 1) to an array \( \mathbf{Y} \) in another process (\( P_2 \) in Figure 1). The source code of \( P_2 \) at the receive is in Figure 1. \( P_2 \) receives messages that contain the value of \( \mathbf{x} \) (\( \text{x\_value} \)) and \( I \) (\( \text{index} \)). Consequent to the receipt of such a message, the value of \( \mathbf{x} \) is read into a buffer \( \text{BUF} \) (now \( \text{BUF} \) contains \( \mathbf{x}[I] \)). Then the value of \( I \) is read from the message into a program variable \( I \). Subsequently, \( \mathbf{Y}[I] = \text{BUF} \) is done. If we assume that at no time will there be two messages with the same \( I \) available at a receive, then each cell of the array \( \mathbf{Y} \) will be written by the correct value (\( \mathbf{Y}[I] = \mathbf{x}[I] \) in this case) in all such executions. In this example, \( \text{BUF} \) is not the variable of interest; the array \( \mathbf{Y} \) is. Consequently, it is possible to specify the nondeterministic behavior a program with respect to the order in which values are written into each cell of the array \( \mathbf{Y} \). The variables that are written with the same sequence of values in every execution are called **internally determinate** variables (Definition 2.1). In this example, \( \mathbf{Y}[1] \) and \( \mathbf{Y}[2] \) in process \( P_2 \) are internally determinate variables, whereas \( \text{BUF} \) is not. Thus, the apparent nondeterministic (yet correct) behavior of this segment of the program can be specified as "\( \mathbf{Y}[1] \) and \( \mathbf{Y}[2] \) are required to be internally determinate in all executions of the program."

Another scenario is that a programmer is not concerned with the intermediate values of a variable, rather only with the final value. For example, in Figure 2, it does not matter which order the messages are received, since \( \text{SUM} \) will eventually contain the sum of the values of \( \mathbf{x}[I] \). An **externally determinate** variable has the same final value in all executions with the same input.
$P_2$

Recv(1) to BUF;
if (BUF.index == 1) \[ Y[1] = BUF.x\_value; \]
else if (BUF.index == 2) \[ Y[2] = BUF.x\_value; \]

$P_2$

Recv(1) to BUF;
SUM = SUM + BUF.x\_value;

Message Structure: \(<msgid, x\_value, index>\>
Msg1 = <1,6,2>  \quad Msg2 = <1,5,1>

Figure 1: Internally Determinate Variable

$P_2$

Send(1)

Message Structure: \(<msgid, x\_value, index>\>
Msg1 = <1,6,2>  \quad Msg2 = <1,5,1>

Figure 2: Externally Determinate Variable
Definition 2.2 *Externally Determine Variable*: A program variable is *externally determine* iff in all possible executions with the same input, the final value written into it is the same.

Note that an internally determinate variable is also an externally determinate variable.

For the sake of completion, we also define a *nondeterminate variable*.

**Definition 2.3 Non-determinate Variable**: A program variable is *nondeterminate*, if there is at least one execution among executions with the same input where the final value written into this variable is not the same.

Thus, the deterministic behavior of a program may be fully specified in either of the following ways:

1. Specify that a program is internally determinate, or

2. Specify for each program variable, whether it is internally determinate and if not, whether externally determinate, or nondeterminate.

However, for Case (2) above, it may be more practical to specify a subset of program variables, $S$, and specify whether each variable in $S$ is internally determinate and if not, whether externally determinate. It can be seen that in a program, the set of internally determinate variables is a subset of the set of externally determinate variables, which in turn is a subset of the set of all program variables.

However, testing whether a program variable shows the expected behavior based on specification of program variables require intrusive access to program data space at runtime. Since a program can have large number of variables, specifying and verifying each variable for nondeterministic behavior can be very costly. In the next subsection, we show that a message based characterization can achieve the same purpose as a variable based specification.

### 2.2 Message Based Characterization

In this subsection, we informally discuss how nondeterministic behavior can be characterized in terms of messages. One way to express nondeterministic behavior in terms of messages is through *message races* [6]. A message race occurs when it is possible that two messages with the same *msgid* are received at a receive. However, a race condition may occur either when a message race occurs, or when the receive operation does not differentiate the messages received based on their *msgids* after their receipt [5, 8, 16]. Our goal in this paper is to devise means to specify unintentional message races. We find that specifying races based on message races is useful, since in most cases we observed programmers process messages with
differing msgids differently after the receipt of the messages. We informally discuss message races in this section.

Consider the example in Figure 1. The values of x[1] and x[2] are sent to process P2, from processes P3 and P1, respectively. Let the messages Msg1 and Msg2 have the structure: `<msgid, x.value, index>`. Here msgid is the message type of the message, x.value is the value of x, and index is the index of the value sent, i.e., when x[1] is sent, index is 1.

There is a message race at the first receive in P2 since either of Msg1 or Msg2 can be accepted at the first receive depending on the relative speeds of processes (P1,P3) and/or message delays. The fact that “if there are at least one pair of messages acceptable at a receive with the same value of msgid then it is a race” can be expressed as `msgid == msgid'` . Here msgid indicates the value of msgid in one message, and msgid' indicates the value of msgid in any other message.

This program differentiates between Msg1 and Msg2 based on index, and therefore the values of Y[1] and Y[2] in P2 will be, respectively, 5 and 6, irrespective of the order in which the messages is received at the first receive in P2. Therefore, the race between Msg1 and Msg2 is not an error. Therefore, we term such message races as **intentional races**.

We can express the fact that “if there are at least one pair of messages with the msgid value of 1 and identical values for index, then there is an unintentional race,” as follows: `msgid == 1 && index == index'` When this expression evaluates to true on a pair of messages, there is an unintentional race at that receive. Here, index denotes the value of index in one message, and index' denotes the value of index in any other message. Observe that this expression is equivalent to stating that “Y[1] and Y[2] are required to be internally determinate in all executions of the program.” This expression is an example of a **strict** Race Expression (Section 4.3). The behavior of the externally determinate variable in example in Figure 2 can be expressed using a **nonstrict** Race Expression (Section 4.3).

From the above discussions it follows that evaluation of a Message Expression to verify race conditions is different from normal expressions encountered in programming languages in that a Message Expression requires the evaluation of the same variables (here message variables) over two (or more) messages. Such Message Expressions are termed Race Expressions. Some of the Message Expressions may be evaluated over a single message. However, other kinds of Message Expressions can be used to determine the existence of a message with a specific set of attributes at run-time, or to evaluate messages at run-time.
3 Race Expressions

In Section 2, we have shown informally how a specific kind of Message Expression, called Race Expression, is used to evaluate race conditions. In this section, we attempt a more formal approach at describing Race Expressions.

Race Expressions are expressions consisting of message variables, i.e., the variables whose values are transmitted in messages. In the context of programming languages, an expression is evaluated with respect to a store, which is a component of a computational state [17]. Since a Race Expression is evaluated over two messages, evaluation of a Race Expression needs to be defined over two stores, each store containing a possibly different set of values for the same set of variables. Also, a Race Expression must return a boolean value – a return value of true indicating the presence of an unintentional race, and false indicating either an intentional race, or no race. Each active variable has its value stored in a store. Thus, a variable in a Race Expression may have multiple values, one in each store.

For the convenience of exposition, we assume that a store contains an ordered tuple of values, e.g., <2,5,1.34>. From here onwards, we make this assumption and call this ordered tuple of values a value-tuple. Each component of a value-tuple corresponds to a typed variable of a program (a message variable). Again, for convenience, we refer to these typed variables as fields.

Evaluation of a Race Expression over a single value-tuple is similar to the evaluation of expressions in programming languages. We discuss below the evaluation of Race Expressions over two value-tuples.

3.1 Evaluation Over Two Value-Tuples

During the evaluation of a Race Expression over two value-tuples, we arbitrarily choose one of these tuples as the left value-tuple, and the other as the right value-tuple. A Race Expression can be written in any programming language that is augmented with a special operator, ’, called the prime operator.

Definition 3.1 Prime Operator: Consider two value-tuples, each containing a value of a variable, v. In a Race Expression, v returns the value of v in the left value-tuple and v’ returns the value of v in the right value-tuple. The operator, ’, is called the prime operator, and v’ is called a prime field.

The semantics of a Race Expression will be that of the programming language in which it is expressed, augmented with the semantics of the prime operator. Though Race Expressions can be defined in any language, for the sake of exposition, we use the C language in the following discussion. The prime operator is similar to the address referencing operator & in
the C language [15]. Though the & operator does not change the values of any variables, it
returns the address of a variable on which the operator is applied.

We also require that the evaluation of a Race Expression must not cause any side effect.
This is to prevent the variables of a message, over which a Race Expression is defined, from
being overwritten during evaluation.

Examples

Consider two value-tuples, < 4, 2, 1 > (left value-tuple) and < 4, 8, 16 > (right value-tuple),
that correspond to three integer fields named field1, field2 and field3.
1. field1==field1'; will evaluate to true since the value is 4 in both cases.
2. field1==field1' && field3' <= (field1 * field2); will evaluate to false.

3.2 Evaluation Over More Than Two Value-Tuples

Checking for races among messages frequently requires comparison of more than two mes-
sages. To support this, we require that Race Expressions return meaningful values when
evaluated over more than two value-tuples.

Evaluation of a Race Expression over three or more value-tuples is accomplished as
follows. From a set of n value-tuples, all combination of pairs of value-tuples are chosen and
evaluated as explained in the Section 3.1. The result of a Race Expression evaluated over
a set of value-tuples is the logical OR of the evaluation results of all combinations of value-
tuples pairs. As each pair of value-tuples may be chosen in any order, for consistency, the
result of evaluation must be independent of the order of evaluation. An expression1 having at
least one prime field and no logical operators && or || is called a prime expression. A prime
expression can be combined with other expressions to form a Race Expression if and only if
the dyadic operators used in a prime expression are commutative, i.e., == and != are the
only permitted operators. We show below with an example that the use of noncommutative
dyadic operators may give ambiguous results when used to evaluate a prime expression with
prime and nonprime fields.

Consider two value-tuples, < 1,8 > and < 4,8 >. Let the field names be field1 and
field2 respectively. Consider the evaluation of field1 > field1'; If the first and second
tuples are the left and right tuples, respectively, then this expression evaluates to false.
However, if the first and second tuples are the right and left tuples, respectively, then this
expression evaluates to true.

Briefly, a Race Expression has the following attributes:

- Contains at least one prime expression,
• All dyadic operators that operate on a prime field and a nonprime field are commutative,
• Permits side effect free evaluation, and
• Returns a boolean value.

4 Implementation

In this section, we describe our adaptation of Message Expressions for the PVM 3.x system [11]. The PVM 3.x system is a distributed heterogeneous programming environment. To make Message Expressions applicable to testing and debugging situations for this environment, we need to extend the Message Expressions in certain ways. We discuss how semantics may be added to an Message Expression in the general context, and then describe our adaptation of Message Expressions specifically for PVM 3.x.

4.1 Adding Semantics and Concrete Syntax

It is necessary to add more semantics to Message Expressions to adapt them to a specific programming environment. We can also specify the concrete syntax of Message Expressions. In particular, the following can be done.

1. The operators and constructs permissible in an Message Expression can be defined.

2. Mandatory fields in Message Expressions can be specified.

3. Message Expressions can be qualified with key words.

4. Evaluation conventions based on (1), (2) and (3) above can be stipulated.

We describe here the adaptation of Message Expressions that we have used in our implementation of a debugging tool, mdb[7] for PVM 3.x. The adapted Message Expressions are called PVM Message Expressions (pvmME).

The version of pvmME we have implemented uses only a subset of the C language. This subset of C does not allow loops, assignments, pointers, structures, or functions. It may be noted that since the major purpose of a Race Expression is pairwise comparison of a set of messages, none of the previously mentioned language constructs considerably enhance the expressiveness or utility of a Race Expression.

Our adaptation of Message Expressions has come about from the structure of PVM 3.x messages. Each message has a \texttt{msgid} and a \texttt{tid}, apart from a number of typed fields that contain values. Every pvmME contains two default fields, \texttt{msgid} and \texttt{processID}, corresponding
to the \textit{msgid} and \textit{tid} of PVM 3.x messages, respectively. We associate a \textit{pvmME} with every \textit{msgid}, i.e., if no \textit{pvmME} is specified for a particular \textit{msgid}, say 3, a default \textit{pvmME} of \texttt{msgid == 3 && msgid' == 3}; is assumed. This \textit{pvmME} will report any message race involving messages with \texttt{msgid == 3} as an unintentional race. The decision to attach a \textit{pvmME} with each \textit{msgid} came about due to efficiency reasons. For early detection of a race, it is more efficient to simply compare messages with the same \textit{msgid}, rather than evaluating the \textit{pvmME} corresponding to every future receive event. This approach also avoids storing one \textit{Race Expression} for every receive statement in the program. Also note that comparison of messages with different \textit{msgids} may not yield meaningful values under this convention, and therefore is not supported. An alternate implementation approach of storing a \textit{Race Expression} with every receive statement may not be efficient for early race detection, but can give more flexibility in programming. The next section contains a description of our adaptation of Message Expressions, called \textit{pvmM-Expressions}, for PVM 3.x.

4.2 \textit{pvmM-Expressions}

\textit{pvmME}s are used to specify races, to specify the existence of a message with a particular set of attributes, and to evaluate a set of messages. The \textit{pvmME}s for race specification fall in two broad classes: strict and nonstrict. A strict race specification is used when unintentional races are suspected and need to be reported. A strict specification can be created using message variables. In contrast, a nonstrict race specification is used to specify that a race is not harmful and does not need reporting. Each use of \textit{pvmME}s is given below with examples.

4.3 \textbf{Intentional Race Specification}

What constitutes an intentional race is largely dependent on the programmer. \textit{pvmME} provides support for expressing what a programmer conceives of as a race. There are two types of \textit{Race Expressions} in \textit{pvmME} that can be used to express intentional races: strict and nonstrict.

\textbf{strict}

A strict expression specifies an intentional race with respect to a set of message variables, i.e., when a strict \textit{Race Expression} evaluates to false on a set of messages, there is no unintentional race. Alternately, if a strict \textit{Race Expression} evaluates to true, an unintended race is indicated. A keyword \texttt{strict} is used to qualify a \textit{pvmME} to be a strict \textit{Race Expression}. Often equality of some of the fields of a message to corresponding fields of another message constitutes an unintentional race.

\textbf{EX 1. strict:} \texttt{msgid == 3 && field1 == field1'};
In some other cases, if values of specific fields fall between certain ranges, it is an unintentional race.

**EX 2. strict:** \( \text{msgid} == 3 \land \text{field1} > 2 \land \text{field1} < 5 \land \text{field1}' > 2 \land \text{field1}' < 5 \);

In Ex 3, two messages that have the same value for \( \text{field1} \) will not race, if the values of \( \text{field2} \) are different in these messages, and vice versa.

**EX 3. strict:** \( \text{msgid} == 4 \land \text{field1} == \text{field1}' \land \text{field2} == \text{field2}' \);

Thus, as long as this pvmME is unsatisfied on pairs of messages, there can be many messages with either the same \( \text{msgid} \) and \( \text{field1} \), or the same \( \text{msgid} \) and \( \text{field2} \), but not with all the fields the same.

**nonstrict**

Whenever a race is detected, the default action is to stop any further race detection. On the other hand, if we want any message race with a particular \( \text{msgid} \) to be ignored, then we need to qualify the pvmME used with the **nonstrict** keyword.

**EX 4. nonstrict:** \( \text{msgid} == 4 \);

In this case, whenever a message race with a value of 4 for the \( \text{msgid} \) is detected, the race is ignored, thereby considering it intentional.

### 4.4 Detecting Messages

Many times it is required to detect the presence (or the absence) of messages. Therefore, we provide a means to express the presence of a message. Note that when detection of the presence of a message is always enabled, failure to report the presence of a message is equivalent to reporting the absence of a message up to the current state of computation.

We use a keyword **exists**, to show that a pvmME is used to detect the presence of a message. Such expressions are called Existence Expressions. A pvmME that is qualified with the keyword **exists** will be used to evaluate every message, one at a time. Therefore, such a pvmME must not contain any prime fields.

**EX 5. exists:** \( \text{msgid} == 3 \land (\text{field1} < 3) \);

This will return true as soon as a message arrives whose \( \text{msgid} \) is 3 and whose value in \( \text{field1} \) is less than 3.

### 4.5 Message Evaluation

Another class of pvmMEs, called Evaluation Expressions, are used to analyze messages at certain specific points in an execution. Such pvmMEs are qualified with the keyword **eval**.

**EX 6. eval:** \( \text{msgid} == 5 \land \text{field1} > 2 \land \text{field2} < 3 \);
This expression returns a boolean value of true when \( \text{field1} \) is greater than 2 and \( \text{field2} \) is less than 3 in a message; else it returns false.

\text{EX 7. eval: if (msgid == 4) field1;}

This will return the value of \( \text{field1} \) in every message with \( \text{msgid} == 4 \). These pvmMEs may be evaluated over the set of messages in the available set at a suspend point while debugging. The pvmMEs can also be used while testing for continuous evaluation of the messages.

5 Conclusion

We have described in this paper a new class of expressions, called Message Expressions, for specification of nondeterministic behavior of programs. Message Expressions have the following advantages over event specification techniques that use patterns of run-time events: (a) Writing a Message Expression does not require a knowledge of the exact pattern of run-time events. Instead, Message Expressions can be used to specify nondeterministic behavior based on variables used in messages, (b) Message Expressions can be evaluated without intrusive access of program variables. Besides, Message Expressions can be integrated with other means of run-time event specifications. The use of Message Expressions is not limited to the description of nondeterministic behavior of programs. Message Expressions can be used for detecting messages with specific attributes, and also for evaluating messages.

We have integrated Message Expressions in a distributed dynamic debugger called \( \text{mdb}[7] \). Message Expressions have been useful for unambiguous specification of race conditions in programs. The debugger is in use at Los Alamos National Laboratory. We are also investigating the incorporation of Message Expressions in a trace based debugger called Xpvm.

References


