FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Specifying Performance Properties Using Compound Runtime Events - APART Technical Report -

Felix Wolf, Bernd Mohr

FZJ-ZAM-IB-2000-10

August 2000

(letzte Änderung: 24.08.2000)

Specifying Performance Properties Using Compound Runtime Events

APART Technical Report [1]

http://www.fz-juelich.de/apart

Workpackage 2

Identification and Formalization of Knowledge

Felix Wolf
f.wolf@fz-juelich.de

Bernd Mohr
b.mohr@fz-juelich.de

Central Institute for Applied Mathematics
Research Centre Juelich

August 24, 2000

[1]The ESPRIT IV Working Group on *Automatic Performance Analysis: Resources and Tools* is funded under Contract No. 29488

Acknowledgments

We would like to thank all our partners in the APART Working Group for their contribution to this topic. In particular, we would like to express our gratitude to Michael Gerndt (Technische Universität München) for many helpful discussions and comments contributing to improve this document.

Abstract

Event traces provide a very fine grained view on the performance behavior of a parallel application. Based on this view, performance properties which cannot be represented by profiling data can be described in terms of compound runtime events. In many cases, a compound runtime event indicating the existence of a performance property exhibits a quite complex structure. Most of the relationships by which its constituents are interconnected depend on a specific programming model which makes it difficult to capture all such situations by means of one general representation method.

To overcome this problem, we present a generic technique for defining programming model specific abstractions allowing a simple description of compound runtime events in the context of that programming model. In addition, we show that this approach can be easily integrated into the APART specification language and propose appropriate language extensions.

Contents

1 Motivation **1**

1.1 Introduction 1

1.2 Compound Runtime Events 1

1.3 Outline 2

2 Basic Event Trace Model **2**

2.1 Generating Event Traces 2

2.2 The Basic Model 3

3 Model Enhancement **3**

4 Example: Message Passing **5**

4.1 Model Enhancement 6

4.2 Collective MPI Operations 8

5 Specifying CREs **8**

6 CREs in the Context of Message Passing **10**

6.1 Late Sender 10

6.2 Late Receiver 12

6.3 Receiving Messages in Wrong Order 13

6.4 Remarks 14

7 Event Trace Model Enhancement and ASL **14**

7.1 Extending the ASL Language 15

7.1.1 Expression Syntax 15

7.1.2 Specifying CREs 15

7.1.3 Pattern Matches 16

7.1.4 Pattern Templates 17

7.2 Extending the ASL Specification Model 17

8 MPI Related CRE Specification in ASL **19**

8.1 Extending the MPI Part of the ASL Specification Model 19

8.2 MPI Related CREs in ASL 19

8.2.1 Late Sender 21

8.2.2 Late Receiver 21

8.2.3 Messages in Wrong Order 21

8.3 Taking Advantage of Pattern Templates 23

8.4 Using Patterns in Property Definitions 23

8.4.1 late_sender 23

8.4.2 late_receiver 24

8.4.3 wrong_order_ls 24
8.4.4 wrong_order_lr 24

9 Related Work **24**

10 Conclusion **25**

1 Motivation

1.1 Introduction

The development of parallel applications is still a very complex and expensive process. This is essentially a result of the close relationship between the algorithm to be implemented on the one hand and the properties of the target platform in conjunction with the employed programming model on the other hand. Only when all three components of a parallel solution fit together in the right way, the application is able to achieve the desired performance.

The complexity of current systems involves incremental performance tuning through successive observations and code refinements. A critical step in this procedure is deriving the performance properties of a parallel application from the collected performance data. This is necessary in order to yield a useful statement about inefficient program behavior.

A performance property characterizes a specific performance behavior. Both the kind of data available and the tools we can use to interpret the data have a major impact on the informative value of the performance properties that can be proven based on this data.

In many cases, summary information collected by profiling tools is sufficient to build a useful statement. However, there are performance properties which are not visible in this kind of information. A much more detailed view on the behavior of the parallel application can be gained by event tracing. That means, we record those events during runtime that determine the performance properties of the application. Then, the presence of a performance property can be proven by the occurrence of a compound runtime event (CRE) which has to be detected in the event trace after program termination. A CRE is built from primitive events such as those associated with entering a program region or sending a message. A CRE can also be regarded as an event pattern which can be found by applying pattern matching algorithms. An additional advantage of this technique is the existence of many powerful graphical visualization tools like VAMPIR [1] which may help to identify the desired CREs manually. But automatically locating and classifying performance properties would accelerate this process considerably.

Currently, we are investigating methods for specifying CREs that represent performance properties of parallel applications. These specifications should be used by human tool builders or even generators as a basis for a new kind of automatic performance tools which are able to identify and assess the specified performance properties in a parallel application without user intervention. An alternative approach would be to use the specifications as input for a generic tool capable of handling an arbitrary set of CREs. The latter method was adopted in the field of MPI applications by the EXPERT tool architecture [14].

However, CREs indicating the existence of certain performance properties are quite complex and difficult to formalize. For this reason, we present a generic technique leading to simple specifications of CREs produced by parallel and distributed applications. The essential ideas used here are strongly influenced by the experiences gained during the design of the EARL trace analysis language [13].

1.2 Compound Runtime Events

A CRE representing a performance property is a set of primitive events. A primitive event corresponds to a single record from the event trace. The primitive events a CRE comprises

are called its *constituents*. There are various relationships between the constituents of a CRE. Some of which are temporal, but mostly they are relationships depending on the employed programming model. For instance, sending a message and receiving it are interconnected by a relationship which is derived from the message passing programming model. Of course, these relationships may include in their definitions temporal aspects, as well.

Another observation is that CREs through which performance properties become manifest exhibit some form of locality within the event trace. Usually the constituents of such a CRE share some context which refers to the execution *state* of the system or the program at the time when the CRE occurs. This means that in most cases the constituents refer to the same set of ongoing activities of the program in contrast to activities which already belong to the execution history. An example for such a system state is e.g. the set of messages being in transfer at a given moment. However, the nature of such system states again depends on the programming model used.

For this reason, it is difficult to devise a formal representation of CREs which is general enough to cover all possible programming models on the one hand and which is powerful enough to express quite complex and specific CREs on the other hand.

Instead, it would be desirable to provide for each programming model appropriate abstractions which could be used to keep the CRE descriptions as simple as possible. However, to alleviate the loss of generality introduced by model specific description methods we propose a generic technique for creating such abstractions.

1.3 Outline

In section two we explain how current tracing environments suggest a basic model of the events occurring in an event trace. Afterwards, we present in section three a general method for providing higher-level abstractions on top of the basic model which can be used to easily describe performance properties by means of CREs. In order to give an example, the method is applied to the message passing programming model in section four. Section five introduces a guiding principle aimed at formally specifying CREs which is used in section six to define concrete message passing related CREs.

Section seven and eight discuss how the method presented here fits the APART specification language (ASL) [4] and propose some extensions aimed at making our results available to ASL. The ASL is a novel approach to the formalization of performance properties and the associated data. Finally, we consider related work and conclude the paper.

2 Basic Event Trace Model

2.1 Generating Event Traces

The process of preparing an application for event tracing is called *instrumentation*. This involves a modification of the executable application either by inserting event logging statements into the source code prior to compilation or by modifying the object code directly. The latter is typically performed by linking function or subroutine calls to appropriate wrapper functions which usually record the call and return events as well as function specific events such as sending or receiving a message.

The application developer must decide where in the program, i.e. for which program

regions, runtime events should be recorded and which information these event records should contain. We differentiate between static and dynamic instrumentation. Whereas for static instrumentation this decision is made once before running the application, dynamic instrumentation allows to change theses parameters during runtime.

There are a multitude of tools for static as well as dynamic instrumentation like PAT [5] and Dyninst [8]. In most cases, the information recorded for a runtime event includes at least a time stamp, the location, e.g. the process or node, where the event happened, and the type of the event. Depending on the event type additional information may be supplied such as the function name for function call events. Message event records typically contain details on the message processed like the source and destination location or message tags. However, in order to keep instrumentation simple the information included in such an event record is restricted to the data available at the location where and at the moment when the event occurs.

After program termination the event traces which are generated independently for each location are merged and sorted according to their time stamps. Systems that only rely on local clocks have to adjust the time stamps with respect to chronological displacements and clock drifts.

2.2 The Basic Model

The data formats of trace files produced by typical tracing environments constitute a basic model of an event trace. In most cases, there is a small number of record types with an associated set of well-defined data fields. Hence, the basic model consists of a set of event types, where each is characterized by a set of attributes.

In most cases, there are event types which share a subset of their attributes. For this reason, it is convenient to create a type hierarchy containing concrete leaf event types as well as abstract base event types at different levels which isolate common attributes.

Since the time stamp (*time*) and the location (*loc*) attributes are always shared by all event types, we can define an abstract event type *Event* constituting the root of the type hierarchy. All event types inherit from *Event*.

An event type t is defined by a set of attributes $\{a_1, \ldots, a_{n_t}\}$. A subset $\{a_1, \ldots, a_{b_t<m_t}\}$ of these attributes may be associated with more general base types. In the remainder we will use the point notation $e.attr$ to refer to an attribute $attr$ of an event e.

An event trace is a sequence $E = \{e_1, \ldots, e_{max}\}$ of events, where $\forall e_i, e_j \in E$ the following two conditions hold:

1. $i < j \quad \Rightarrow \quad e_i.time \leq e_j.time$
2. $i < j \quad \wedge \quad e_i.loc = e_j.loc \quad \Rightarrow \quad e_i.time < e_j.time$

Note that an event trace E contains leaf event types only.

3 Model Enhancement

The process of creating programming model specific abstractions from a given basic event trace model is called *model enhancement*. It results in an enhanced model which is built on top of the basic model in two steps:

1. Defining system states
2. Extending the basic event types

Defining system states. A system state definition is a mapping $S : E \to \mathcal{P}(E)$ which maps an event $e_i \in E$ from the event trace onto a subset $S(e_i) \subseteq E$ of the events from the event trace, where

$$\forall e_j \in S(e_i) : \quad j \leq i,$$

i.e. system states only depend on historic data and not on future events. In addition, we define for each system state mapping S an initial state $S_0 = \emptyset$. A system state $S_i = S(e_i)$ of an event e_i should reflect one aspect of the execution state of the application after the event e_i took place.

A system state is defined by a set of transition rules. For each leaf event type t in the model a transition rule σ_t defines how a state S_{i+1} is created from a state S_i and an event e_{i+1} of type t by applying σ_t to S_i and e_{i+1}. Thus, we have $S_{i+1} = \sigma_t(S_i, e_{i+1})$, where t is the type of the event e_{i+1}. If there is no explicitly defined transition rule σ_t for an event type t and a state function S, σ_t is assumed to be the identity function. A transition rule defined for an abstract base event type covers all derived event types. But it may be overwritten by a transition rule defined for a more specialized type. The state S_1 is created from the initial state S_0 and the event at the first position e_1.

System states are abstractions used to provide context information for a CRE. In other words, the intent of system states is to separate activities that are still going on with respect to a certain point in time from activities that are already completed with respect to that point in time. Note that in most cases it is useful to define several system state functions $S^1, \ldots S^{n_s}$ per model.

Extending the basic event types. Another useful abstraction is to link related events together, so that one can navigate from an event to another related event. This permits to navigate along a path of related events and to define relationships between constituents of a CRE using such paths. A natural way of representing such links is to provide event attributes with pointer semantics. Extending the event types of the basic model including the abstract base event types by adding pointer attributes is the second step we consider. Of course, there is no reason to restrict event type extensions to pointer attributes. But pointer attributes are the most common case for applying this technique.

For each event e_i of type t we may define additional attributes $\{e.p_1, \ldots, e.p_{m_t}\}$, where

$$e_i.p_j = f_j(e_i.a_1, \ldots, e_i.a_{n_t}, S^1_{i-1}, \ldots S^{n_s}_{i-1}).$$

That means, such an additional attribute depends on the attributes defined in the basic model and on the system states of the event immediately before the event instance in consideration. The rationale behind this definition is to disallow functional dependencies on events belonging to already completed activities. This is not really a restriction, since in most cases related events occur during the same activity. But it allows to calculate the additional attributes by remembering only a small set of relevant events which is important with respect to a later implementation. Also note that pointer attributes never point to future events.

The concept of model enhancement is not only a convenient method for defining CREs as we will demonstrate in the next section. Since the locality mentioned earlier is explicitly modeled in form of system states, we yield also a foundation for efficient search methods to detect CRE instances in an event trace.

4 Example: Message Passing

To give an impression of how model enhancement can be used to provide a simple description of complex CREs, we chose the message passing programming model, one of the most popular programming models on todays widespread massively parallel systems, as an example. Although most of the things presented here are independent of a specific message passing library, the terminology used is based on MPI [7].

Typical event tracing environments used in the context of MPI such as [1, 6] define trace file formats whose event records provide the following contents:

- All event records contain a process identifier as well as a time stamp.
- There are two record types for entering (Enter) and leaving (Exit) a program region. Typically, one data field denotes the *region* entered or left. We assume that a region instance is only left after all enclosed region instances have been left.
- There are two record types for sending (Send) or receiving (Receive) a message which provide data fields for the *source* (*src*) and *destination* (*dest*) processes, the message *tag* and *communicator* (*com*), as well as the message *length*.

Thus, we have four concrete event types which can be arranged in a type hierarchy as depicted in Fig. 1 using UML notation. Besides the event type Event at the top there are two abstract event types RegionEvent and MsgEvent at an intermediate level of the hierarchy. RegionEvent covers event types associated with moving from region to region and MsgEvent covers event types associated with message exchange.

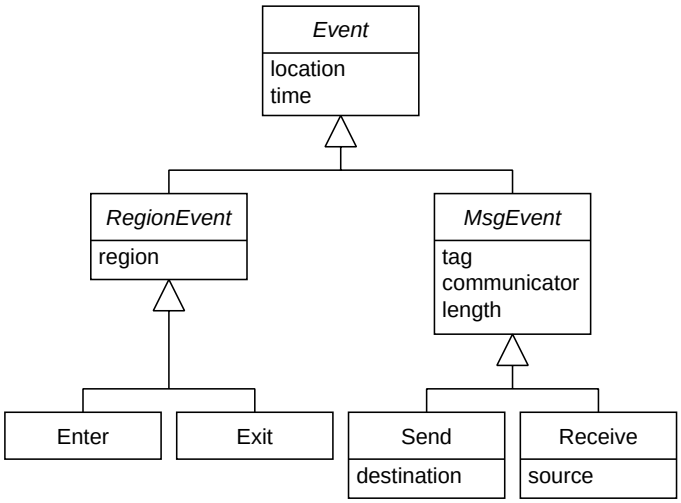

Figure 1: The basic event trace model for message passing

4.1 Model Enhancement

Defining system states. The activities performed by an MPI application at a given moment are best described by means of the program region instances in which the application is executing and the messages which are being in transfer at that moment. The program region instances can be easily represented by the set of Enter events that determine their beginnings and the messages are best specified by the set of their respective Send events. So we define two (groups of) system state functions R^l and $M^{s,d}$ and call R^l the region stack for location l and $M^{s,d}$ the message queue for traffic from source s to destination d. The corresponding state transition rules $\rho^l : R_i^l \mapsto R_{i+1}^l$ and $\mu^{s,d} : M_i^{s,d} \mapsto M_{i+1}^{s,d}$ triggered by an event e_{i+1} are defined in Fig. 2.

$$\rho^l_{Enter}: \quad R^l_{i+1} = \begin{cases} R_i^l \cup \{e_{i+1}\} & \text{if} \quad e_{i+1}.loc = l \\ R_i^l & \text{else} \end{cases}$$

$$\rho^l_{Exit}: \quad R^l_{i+1} = \begin{cases} R_i^l \setminus \{e_j\} & \text{if} \quad e_{i+1}.loc = l \\ R_i^l & \text{else} \end{cases}$$

$$where \quad \neg\exists e_k \in R_i^l : \quad (\quad e_j.time < e_k.time \quad)$$

$$\mu^{s,d}_{Send}: \quad M^{s,d}_{i+1} = \begin{cases} M_i^{s,d} \cup \{e_{i+1}\} & \text{if} \quad (\quad e_{i+1}.loc = s \quad \wedge \quad e_{i+1}.dest = d \quad) \\ M_i^{s,d} & \text{else} \end{cases}$$

$$\mu^{s,d}_{Receive}: \quad M^{s,d}_{i+1} = \begin{cases} M_i^{s,d} \setminus \{e_j\} & \text{if} \quad (\quad e_{i+1}.src = s \quad \wedge \quad e_{i+1}.loc = d \quad) \\ M_i^{s,d} & \text{else} \end{cases}$$

$$\begin{aligned} where \quad & (\quad e_j.tag = e_{i+1}.tag \quad \wedge \quad e_j.com = e_{i+1}.com \quad) \quad \wedge \\ & \neg\exists e_k \in M_i^{s,d} : \quad (\quad e_j.tag = e_k.tag \quad \wedge \\ & \qquad\qquad\qquad\qquad e_j.com = e_k.com \quad \wedge \\ & \qquad\qquad\qquad\qquad e_j.time > e_k.time \quad) \end{aligned}$$

Figure 2: State transition rules for R^l and $M^{s,d}$.

The initial states R_0^l and $M_0^{s,d}$ are defined as empty sets $\emptyset$. ρ^l_{Enter} is responsible for adding Enter events representing active region instances to the region stack and ρ^l_{Exit} is responsible for removing them from the region stack as soon as the corresponding region instances are completed. $\mu^{s,d}_{Send}$ and $\mu^{s,d}_{Receive}$ perform a similar task on Send events in order to keep the set of messages currently being in transfer up to date.

The condition which characterizes the element that will be removed in case of applying the rule ρ^l_{Exit} requires that the element to be removed is the youngest in R_i^l. In case of $\mu^{s,d}_{Receive}$ the condition requires that the element to be removed is the oldest in $M_i^{s,d}$ with matching *tag* and *com* attributes. Note that the transition rules for R^l also remain valid in case of recursive function resp. region calls.

The advantage of the definitions above is that they provide a convenient vocabulary we can use when describing performance properties related to message passing as we will see

later.

Extending the basic event types. Both region instances and messages can be represented by pairs of matching events. A region instance is characterized by its Enter and Exit events and a message is described by its Send and Receive events. For this reason, it would be reasonable to provide a link connecting both sides of each pair, i.e. a link from the Exit event to its corresponding Enter event and a link from the Receive event to its corresponding Send event. The direction of these links follows our definition from the previous section which requires all links to point backwards.

However, the relationship connecting an Exit event with its matching Enter event is actually a specialization of a general relationship between an arbitrary event and the Enter event of the region instance in which the event takes place. Therefore, the link should connect an arbitrary event with the Enter event of its enclosing region instance.

We define the following pointer attributes by using conditions which are similar to those we have already used for the definition of system states. The meaning of these attributes is illustrated in Fig. 3.

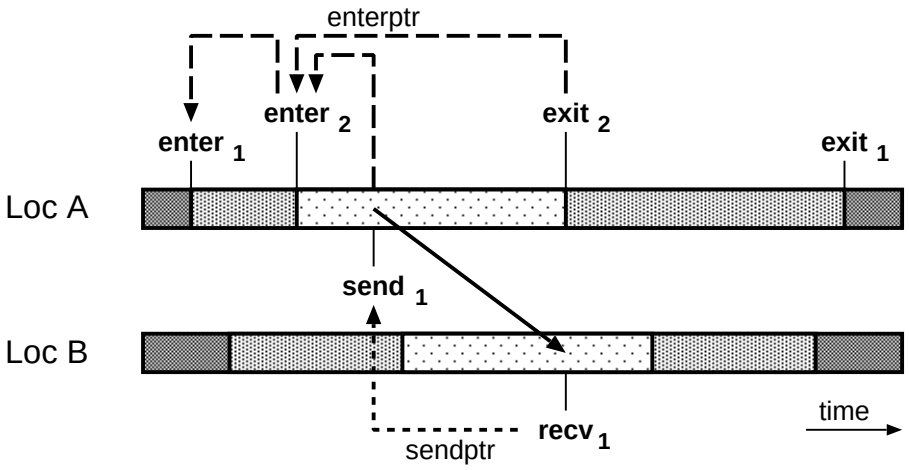

Figure 3: References provided by *enterptr* and *sendptr*

- For an arbitrary event e_i we define an attribute named *enterptr* which points to the Enter event of the region instance in which the event e_i takes place: $e_i.enterptr = e_j \in R^l_{i-1}$, where

  1. $e_i.loc = l$
  2. $\neg\exists e_k \in R^l_{i-1}: \quad (\quad e_j.time < e_k.time \quad)$

  If such an e_j does not exist, the attribute is set to '*null*'. The new attribute is added to the attributes of the root event type Event.

- For an Receive event e_i we define an attribute named *sendptr* which points to the corresponding Send event: $e_i.sendptr = e_j \in M^{s,d}_{i-1}$, where

  1. $(\quad e_i.src = s \quad \wedge \quad e_i.loc = d \quad \wedge \quad e_i.tag = e_j.tag \quad \wedge \quad e_i.com = e_j.com \quad)$

$$2.\quad \neg\exists e_k \in M_{i-1}^{s,d}: \quad \left(\begin{array}{lcll} e_j.tag & = & e_k.tag & \wedge \\ e_j.com & = & e_k.com & \wedge \\ e_j.time & > & e_k.time &) \end{array} \right.$$

The enhanced model we defined so far is illustrated in Fig. 4. Note that in the figure the system state functions are interpreted as operations of the root event type Event since each event can be mapped to corresponding system state instances.

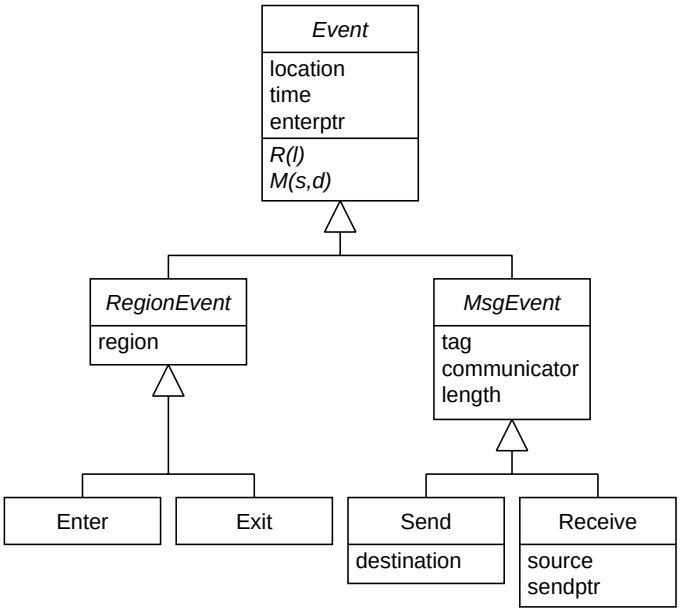

Figure 4: The enhanced model for message passing

However, we must admit that this model does not support collective operations. The events involved in collective operations form another class of related events which are important in the context of MPI related performance properties. The reason for this shortcoming lies in the missing provision for collective operations on the part of current instrumentation systems. But recent developments have begun to fill this gap, so we hope to adapt our model in the near future. In the following subsection we give a short idea of integrating collective operations.

4.2 Collective MPI Operations

Tracing environments such as VAMPIR [1] place a pseudo event record between the Enter and Exit events of a region instance belonging to a collective MPI operation. The pseudo event contains the communicator whose size is given in a separate communicator definition record. By means of these pseudo events the Exit events belonging to the same collective operation instance can be identified and collected by defining an appropriate system state function.

5 Specifying CREs

A specification method for CREs representing performance properties of a parallel or distributed application should meet the following requirements:

1. It should be simple even in case of complex CREs.

2. It should allow for an efficient implementation.

The first requirement demands to specify the relationships between the constituent events of a CRE on a very high level of abstraction. The second requirement concerns the efficiency of possible search methods. This is especially important in the face of the huge amount of data typically involved in event tracing. We think that both requirements can be fulfilled on the basis of enhanced event trace models. We will first introduce a general scheme for specifying CREs and then explain how enhanced event trace models in conjunction with the scheme proposed here are able to meet the two requirements.

A CRE C consists of a set of primitive events, i.e. its constituents. A constituent event is an instance of an event type defined in an enhanced event trace model. The set of constituents can be partitioned into a set of disjoint subsets:

$$C = \bigcup_{i \in I} C_i, \quad I = \{1, \ldots, n\}$$

This partitioning reflects the logical structure of the CRE. The relationships by which these subsets are connected can be expressed by functional dependencies in a form such as the following:

$$C_i = f_i(C'), \quad C' = \bigcup_{j \in J} C_j, \quad J \subset I$$

The functions f_i involved in the specification of a CRE map a set of events onto another set of events from the trace. They may use in their definitions the abstractions provided by an enhanced event trace model, i.e. system state functions and the event attributes including the additional attributes defined during model enhancement. Apart from that, they may access arbitrary events from the event trace and may evaluate conditions over them. A possible addressing mechanism could use the absolute event positions, i.e. the index of an event within the event trace sequence, or the position relative to other events. We do not suggest any notation for specifying these functions. Their specifications may be procedural or functional - whichever method fits the needs best.

However, to be useful for our purposes this scheme must meet several conditions. First, we must ensure that the corresponding dependency graph is acyclic. Furthermore, each C_i except one *root* partition must have at least one predecessor $C_{j \neq i}$ it depends on or it may contain constant events only. So all C_i can be calculated from the root partition by evaluating the functional dependencies. Of course, it is possible that an evaluation step fails, so each f_i can be considered as a predicate, as well. Without loss of generality let C_0 denote the root partition having no predecessor it depends on. As a final condition we require C_0 to have exactly one element which is called the *root* event of the CRE.

Note that in most cases all subsets of C will consist of only a single event. But permitting sets of events might be useful, e.g. to make complete system states or collective MPI operations being part of a CRE.

To complete our scheme we define a predicate *root* which can applied to an arbitrary event in order to decide whether it is a possible root event. Now, we are ready to present a simple pattern matching algorithm:

Algorithm. Searching for all occurrences of a CRE C in an event trace E is straightforward. For all events $e \in E$ do the following:

1. Apply the root predicate to e.
2. If successful, instantiate all constituents which are reachable from the root event by evaluating the functional dependencies.
3. Instantiate all constituents which are reachable from the constituents being already instantiated.
4. Repeat the previous step until all constituents are instantiated or an instantiation step fails.

Remember the two requirements from the beginning of this section. An enhanced event trace model provides just those abstractions that correspond to the vocabulary of the programming model used. So it should be possible to yield a quite simple and understandable specification for most of the CREs representing typical performance properties by using these abstractions when defining the functions involved in the definition of a whole CRE. We will give a demonstration in the next section.

The efficiency of a possible implementation of the scheme presented here relies heavily on the mechanism used for accessing events. Let us consider the following typical scenario: In order to apply the algorithm described above for locating instances of a CRE a search tool walks sequentially through the event trace. If an event fulfills the root predicate, the tool will start to evaluate the tree of functional dependencies used for defining the CRE. It will try to access other events from the event trace. As already mentioned, observations show that most of these events belong to the context of the root event, i.e. events from the corresponding system state or events belonging to the recent past of the root event. All the tool would have to do is to keep track of the context belonging to the events it accesses during its sequential walk and to provide efficient access to this context, e.g. by buffering. Experiences gathered with EARL showed that in most cases this approach leads to a quite workable efficiency.

Of course, the scheme itself does not impose any restrictions on the complexity of the functions f_i, so these have to be defined carefully. But our experiences suggest that in the context of performance analysis the complexity of the required functions is manageable.

6 CREs in the Context of Message Passing

In this section we present three examples[1] of how to represent performance properties in MPI applications as CREs. Note that all three examples cannot be expressed using profiling data only. The specifications are based on the enhanced event trace model we presented before.

6.1 Late Sender

The first example describes the situation which occurs, when an MPI_RECV operation is posted before the corresponding MPI_SEND has been started (Fig. 5). The receiver

[1] The partitions C_i of the CREs presented here will always contain a single event which is indicated by using a symbol beginning with a lower-case letter.

remains idle during the interval between the two calls instead of doing useful computation.

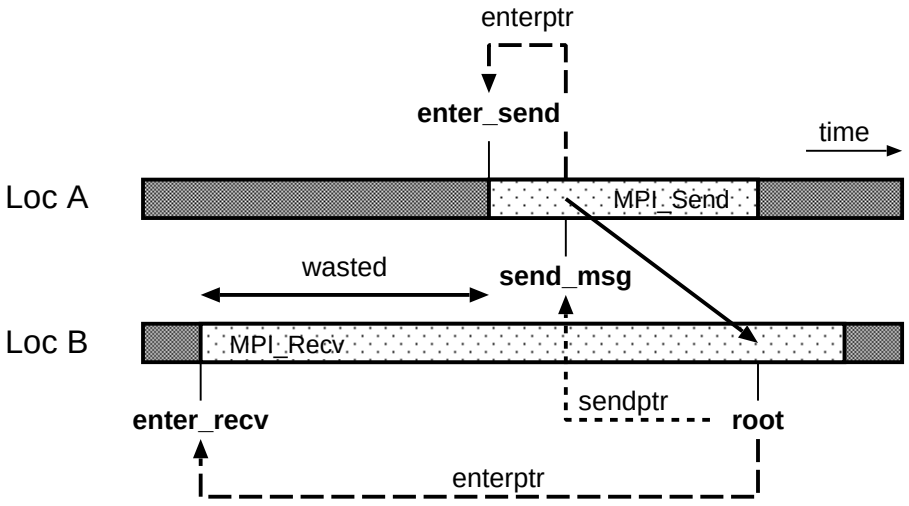

Figure 5: Late sender CRE

This CRE consists of four partitions $\{root, send\_msg, enter\_recv, enter\_send\}$, each containing a single event only: The root event resp. partition is just the event indicating the message arrival, i.e. an event of type Receive. Thus, we have the following root predicate:

$$type(root) = \text{Receive}$$

The other three events are the event of sending the message ($send\_msg$), the event of entering the MPI_SEND region ($enter\_send$), and the event of entering the MPI_RECV region ($enter\_recv$). They are defined as follows:

$$
\begin{aligned}
send\_msg &= root.sendptr \\
enter\_recv &= \begin{cases} root.enterptr & \text{if } root.enterptr.region = \text{MPI\_RECV} \\ fail & \text{else} \end{cases} \\
enter\_send &= \begin{cases} send\_msg.enterptr & \text{if } send\_msg.enterptr.region = \text{MPI\_SEND} \wedge \\ & \quad enter\_send.time > enter\_recv.time \\ fail & \text{else} \end{cases}
\end{aligned}
$$

Applying the algorithm to this CRE specification would result in the following sequence of actions. When a potential candidate for the root event has been found by evaluating the root predicate, i.e. when a event of type Receive has been found, the algorithm traces back the *sendptr* attribute of the root event to locate the corresponding send event ($send\_msg$). Now the event by which the MPI_RECV has been entered ($enter\_recv$) is determined by navigating along the *enterptr* attribute of the root event. To be sure that this event really refers to a region instance of MPI_RECV, the *region* attribute is checked. In the same manner $enter\_send$ is instantiated. But here an additional constraint is taken into consideration which is essential for the whole CRE. The MPI_RECV has to be called before the MPI_SEND. So the two time stamps must be compared. After instantiation of all CRE constituents a tool might consider to compute the amount of wasted time by subtracting the two time stamps.

6.2 Late Receiver

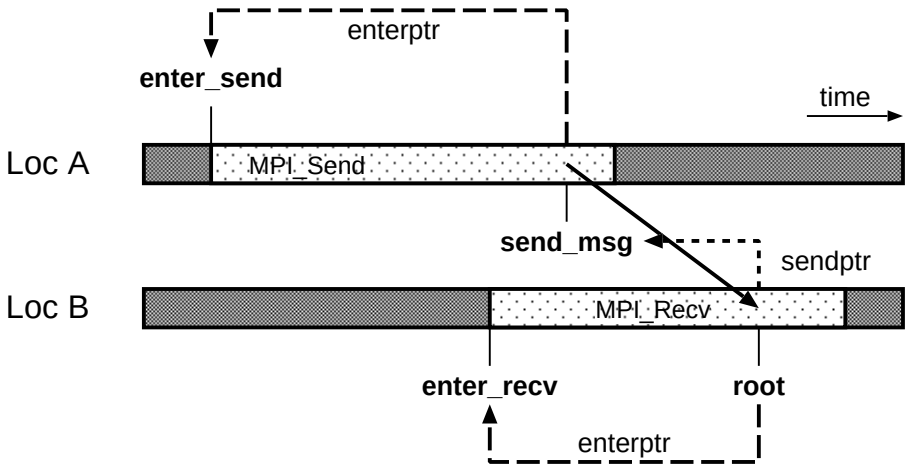

Figure 6: Late receiver CRE

This CRE refers to the inverse case. An MPI_SEND blocks until the corresponding receive operation is called. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default or the size of the message to be sent exceeds the available buffer space and the operation blocks until the data is transfered to the receiver. The behavior is similar to an MPI_SSEND waiting for message delivery. The situation is depicted in Fig. 6. The definition is similar to the previous CRE, in particular the root predicate is the same, so we do not show it here again.

$$
\begin{aligned}
send\_msg &= root.sendptr \\
enter\_recv &= \begin{cases} root.enterptr & \text{if } root.enterptr.region = \text{MPI\_RECV} \\ fail & \text{else} \end{cases} \\
enter\_send &= \begin{cases} send\_msg.enterptr & \text{if } send\_msg.enterptr.region = \text{MPI\_SEND} \wedge \\ & \quad enter\_send.time < enter\_recv.time \wedge \\ & \quad enter\_send \in R^{send\_msg.loc}(enter\_recv) \\ fail & \text{else} \end{cases}
\end{aligned}
$$

An important difference to the previous CRE is the condition appearing in the definition of *enter_send*. Of course, now *enter_send.time* must be less than *enter_recv.time*, since the receiver must be later than the sender. In addition, the MPI_SEND operation must not have finished before the MPI_RECV has been called. So we look at the region stack of the location from where the message was sent and at the time just after the MPI_RECV call was posted ($R^{send\_msg.loc}(enter\_recv)$). If *enter_send* is an element of this set, MPI_SEND and MPI_RECV overlap in time.

However, we must admit that this criterion does not prove waiting of MPI_SEND due to lack of buffer space with maximum reliability. But this criterion is a necessary condition and it is the strongest we can prove based on the data available in an event trace delivered by current instrumentation systems. A detailed description of the performance problem related to this CRE can be found in [7].

6.3 Receiving Messages in Wrong Order

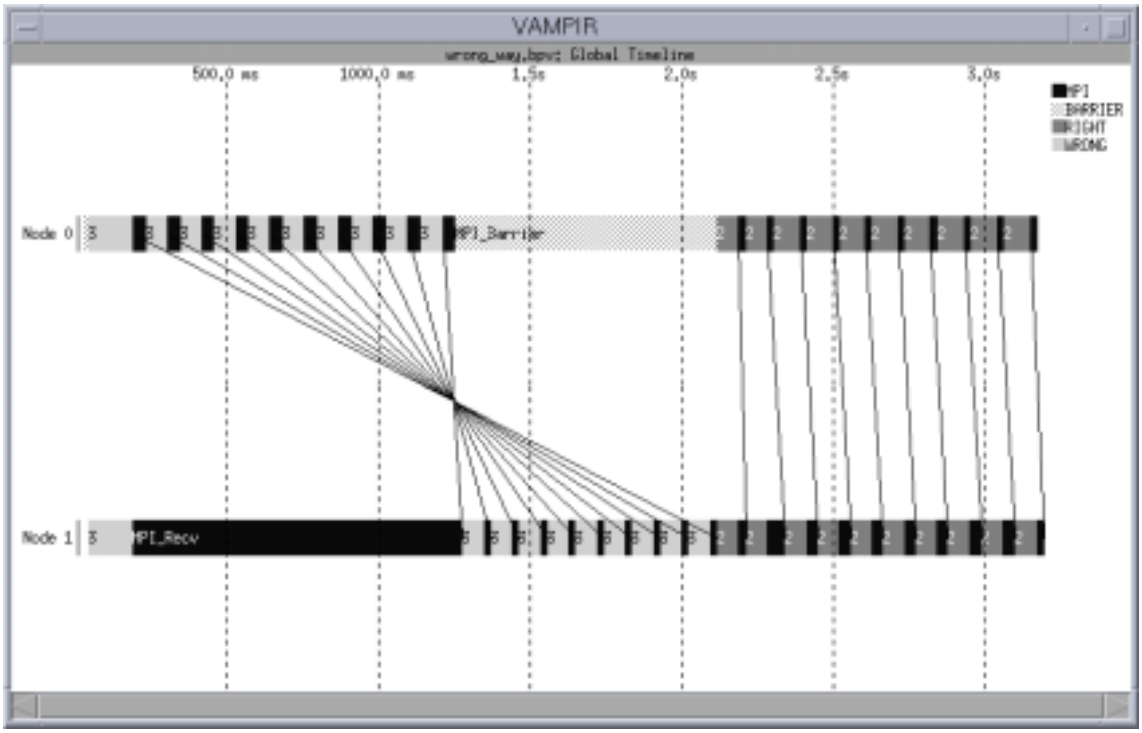

Figure 7: Passing messages out of order

The third example is taken from the *Grindstone Test Suite for Parallel Performance Tools* [9] and highlights the problem of passing messages out-of-order. This problem can arise if one process is expecting messages in a certain order, but another process is sending messages which are not in the expected order. In Fig. 3 an extreme example is shown: In the left part of the picture node 1 is processing incoming messages in the opposite order as sent from node 0. Processing them in the order they were sent would not only speed up the program but would also require much less buffer space for storing unprocessed messages. This is shown in the right part of the picture.

We model this situation as a message which is sent later but received earlier than another message with the same sender and receiver. For this reason, the CRE consists of two partitions, each containing only a single event. The root event is the message receipt, the other event is the message dispatch. Again, the root predicate requires only the event type to be Receive. The CRE is defined as follows:

$$send\_msg \quad = \quad \begin{cases} root.sendptr & \text{if} \quad \exists e \in M^{root.src,root.loc}(root) : \\ & \qquad e.time < root.sendptr.time \\ fail & \text{else} \end{cases}$$

The condition on the right side requires that there are older messages in the 'message queue' for traffic between the source and the destination location of the message in consideration which have not been received at the time that message has been received.

6.4 Remarks

Note that the examples presented in the previous subsections expose an interesting property of the description method used here. The CRE definitions can be divided into two parts. The first part is responsible for locating, i.e. instantiating, the constituents of the CRE. The second part places additional constraints on the constituents which are essential for the performance property they describe. Since checking for these constraints is not required for a correct instantiation nor it depends on any evaluation order, we could consider to explicitly separate the constraint part from the instantiation part. These considerations also include the root predicate. The root predicate can be divided into a condition requiring a certain type and optional constraints if necessary.

In particular the definition of large CREs would benefit since the constraint part could be organized in semantically related groups. E.g. the definition of the second example might be written like this:

Root Type

Receive

Instantiation

$send\_msg$	$= root.sendptr$	
$enter\_recv$	$= root.enterptr$	
$enter\_send$	$= send\_msg.enterptr$	

Constraint

$root.enterptr.region$	$=$ MPI_RECV	$\wedge$
$send\_msg.enterptr.region$	$=$ MPI_SEND	$\wedge$
$enter\_send.time$	$< enter\_recv.time$	$\wedge$
$enter\_send$	$\in R^{send\_msg.loc}(enter\_recv)$	

7 Event Trace Model Enhancement and ASL

The APART specification language ASL [4] which has been developed by the APART Esprit IV Working Group on *Automatic Performance Analysis: Resources and Tools* is a novel approach to the formalization of performance properties and the associated performance related data. ASL provides a formal notation for defining performance properties related to different programming models. It allows to reference performance related data items by means of an object-oriented specification model. ASL distinguishes between static data which is known at compile time, e.g. information on source code entities, and dynamic data which is generated at runtime, e.g. CPU time summaries.

In the ASL terminology a performance property represents one aspect of performance behavior. To test whether such a property is present in an application, an associated condition must be evaluated based on the current performance data. The confidence of a property specifies the reliability of the test condition. If the condition evaluates to true, the severity of a property indicates its relative importance with respect to other properties.

However, the current ASL data model mainly concentrates on profiling data, i.e. summary information, and does not take advantage of the more detailed information contained in event traces. The very fine grained view on the execution behavior provided by event traces can be used to identify hidden idle times, to detect programming errors, or to trace back performance problems to source code entities in a way being not supported by

profiling data. In particular the notion of CREs indicating the existence of performance properties is not part of the current ASL specification. But the very general design of ASL permits the easy integration of our results into the language, thereby requiring only minor extensions which are presented in the next subsections. The extensions are divided into two parts. The first part deals with the ASL language, the second part with the ASL specification model.

7.1 Extending the ASL Language

7.1.1 Expression Syntax

To be able to express state transition rules we need conditional expressions, so we extend the grammar symbol *expr* by adding a further alternative. In addition, we provide a possibility to create a set from a list of single elements. The empty set is created by substituting the symbol *reference-list* by an empty list. Finally, we introduce the `NULL` value literal indicating a void reference. The required grammar extensions are depicted in Fig. 8[2].

expr	**is**	[...]
	or	*cond-expr*
cond-expr	**is**	*if-part elif-part* $*$ [*else-part*]
if-part	**is**	`IF` '(' *bool-expr* ')' *expr*
elif-part	**is**	`ELIF` '(' *bool-expr* ')' *expr*
else-part	**is**	`ELSE` *expr*
set-expr	**is**	[...]
	or	'{' *reference-list* '}'
reference	**is**	[...]
	or	`NULL`

Figure 8: Expression syntax extensions.

7.1.2 Specifying CREs

CREs are specified in ASL by a new language construct *pattern*. Its name is motivated by thinking of CREs as event patterns. Its syntax is defined in Fig. 9. Since ASL is more intended to specify CREs than to implement efficient matching algorithms, the *pattern* construct is designed according to the remarks in section 6.4.

CRE specifications can be parameterized by declaring formal parameters in the *arg-list*. These parameters as well as the local definitions from the optional `LET` clause can be used in the subsequent parts of the CRE specification. The `ROOTTYPE` clause contains the type of the root event. If it is necessary to allow the root event to have multiple types, a common base type can be used here. The CRE partitions are defined in the

[2] In the figure we use [...] as an abbreviation for the unchanged parts of the production rules as defined in [4].

pattern **is** `PATTERN` *p-name* '(' *arg-list* ')' '{'
`[LET`
def ∗
`IN ]`
p-roottype
p-instantiation
p-constraint
p-export
'};'

arg **is** *type ident*
p-roottype **is** `ROOTTYPE` ':' *ident* ';'
p-instantiation **is** `INSTANTIATION` ':' *const-def* ∗
p-constraint **is** `CONSTRAINT` ':' *bool-expr*
p-export **is** `EXPORT` [*m-name*]':' *const-def* ∗

Figure 9: Pattern specification syntax.

`INSTANTIATION` clause. Partitions consisting of only a single element are expressed by simple constant definitions. Partitions consisting of more than one element have to be defined using a set type. It is possible to use conditional expressions here, if a correct instantiation only can be guaranteed by evaluating a condition. If an instantiation step fails, the expression used in the constant definition should evaluate to `NULL`. A condition representing additional CRE properties which are not needed for instantiation can be placed in the `CONSTRAINT` clause. The `EXPORT` clause defines attributes whose values are computed from the CRE constituents. The attributes can be accessed through match objects of the pattern. So the export clause implicitly defines a class to which the match objects will belong. If necessary, the class can be given a name *m-name*.

The root event as well as the complete event trace can be referenced in a CRE definition by the two keywords `ROOT` and `TRACE`. In a future implementation these keywords are bound to the current root event and the event trace being investigated by the search algorithm.

The *pattern* construct should not be seen as an alternative to the *property* construct. It is rather an instrument to increase its expressiveness as we will see later.

7.1.3 Pattern Matches

A pattern specifies two things. Firstly, it specifies a parameterized algorithm to detect CREs representing a certain behavior of the parallel application. Instances of the algorithm are created by using `p-name()` as constructor and by providing actual parameters according to the signature of the pattern. Such an instantiated algorithm is considered to be an instance of class `p-name`. Secondly, the pattern specifies a class representing pattern matches by defining its attributes in the export clause.

The pattern matches contained in an event trace can be obtained by an external function which is implicitly defined together which each pattern definition:

```
setof m-name PATTERN_MATCHES(p-name p, setof events trace);
```

External means that the semantics of this function are not defined explicitly in ASL. But the function is defined implicitly by our algorithm from section 5. The restriction to pattern instances occurring while the program is executing in a region `r` can be implemented by supplying the region as an argument to the pattern constructor and by checking the region stack in the constraint clause. So it is possible to define exactly which constituents should be inside and which should be outside the region.

Note that defining more than one pattern will lead to an overloaded `PATTERN_MATCHES()` function whose return type depends on the first argument. This is a consequence of the flexibility introduced by the export clause. Another way would be to restrict pattern definitions to a fixed set of exported attributes. But this would also restrict the usefulness of patterns in property definitions.

7.1.4 Pattern Templates

Similar to property specifications, pattern specifications can also benefit from templates. Pattern templates can provide a convenient way to define related patterns by using a defined function or comparison operator as a parameter, e.g.:

```
PATTERN TEMPLATE LatePartner <boolean rel_op(float a, float b)>
```

It is also conceivable to define a new pattern based on existing pattern specifications by using a meta-pattern. Meta-patterns would again be defined as pattern templates but using an already defined pattern as a parameter. However, we will not go further into detail here.

7.2 Extending the ASL Specification Model

The integration of pattern based performance properties also requires an extension of the ASL specification model. In this subsection we describe the programming model independent extensions. The extensions comprise both new classes (Fig. 10) and new functions (Fig. 11, 12). The state functions as well as the pointer attributes are defined as global functions. The transition rules of the state functions can be easily transformed into recursive definitions.

```
class RegionEvent extends Event {
  Region region;                    // region entered or left
}

class Enter extends RegionEvent {
}

class Exit extends RegionEvent{
}
```

Figure 10: New classes added to the ASL specification model.

```
setof Enter R(Event e, Process p) =

  IF (e == NULL)
    {}
  ELIF (typeof(e) == Enter AND e.process_id == p)
    R(PRED(e), p) + { e }
  ELIF (typeof(e) == Exit AND e.process_id == p)
    R(PRED(e), p) - UNIQUE({
                             f IN R(PRED(e), p) SUCH THAT
                               NEXISTS g IN R(PRED(e), p)
                                 SUCH THAT g.timestamp > f.timestamp
                           })
  ELSE
    R(PRED(e), p);
```

Figure 11: Region stack of a process.

```
Enter enterptr(Event e) =

  IF (R(PRED(e), p) == NULL)
    NULL
  ELSE
    UNIQUE({
             f IN R(PRED(e), e.process_id) SUCH THAT
               NEXISTS g IN R(PRED(e), e.process_id)
                 SUCH THAT g.timestamp > f.timestamp
           });
```

Figure 12: The `enterptr()` function representing the *enterptr* attribute.

New classes. Entering and leaving a *Region* is a more general concept outside any specific programming paradigm, so we define the event types associated with entering and leaving a region at this more general level. The new classes are defined according to the basic model from section 4.

New functions. Naturally, the system state function(s) R^l has to be defined at this level, as well. But first, we declare an external predecessor function as a prerequisite needed later which maps an event to its predecessor within the event trace.

```
Event PRED(Event e);
```

This function has not to be defined explicitly in ASL, since it is defined implicitly by the event order produced by common instrumentation systems according to our event trace model from section 2.2. Now, R^l can be expressed by a single function with the corresponding *process_id* as a parameter. A function representing the *enterptr* attribute is also defined.

Note that we could shorten the definition, if we used the `enterptr()` function for defining the `R()` system state function. Apart from that, a more convenient way of expressing these functions would be to define them as operations of class `Event` and its subclasses, thereby making use of virtual operations. Unfortunately, ASL currently does not support operations associated with a class.

8 MPI Related CRE Specification in ASL

This section demonstrates the new features of ASL in the context of the message passing paradigm. We try to express the MPI specific part of the enhanced event trace model as well as the examples from section 6 using the new ASL features.

8.1 Extending the MPI Part of the ASL Specification Model

In this subsection we define some extensions to the MPI related part of the ASL specification model which are required as prerequisites for our later CRE specifications.

New classes. We introduce two further concrete event types `Send` and `Receive` representing the dispatch and the receipt of a message (Fig. 13). The message properties are defined in an abstract base event type `MsgEvent`.

```
class MsgEvent extends Event {
  int tag;                          //message tag
  Communicator communicator;        //communicator
  int length;                       //message length in bytes
}

class Send extends MsgEvent {
  Process destination;              //id of destination process
}

class Receive extends MsgEvent {
  Process source;                   //id of source process
}
```

Figure 13: New classes added to the MPI related specification model.

New functions. We define the state function(s) $M^{s,d}$ similar to the region stack function(s) R^l (Fig. 14). Apart from that, we define a function `sendptr()` representing the *sendptr* attribute (Fig. 15). Of course, `sendptr()` can only be applied to instances of class `Receive`.

8.2 MPI Related CREs in ASL

Based on the definitions from the previous subsection, we can now start specifying the MPI related CREs in ASL using the new *pattern* construct:

```
setof Send M(Event e, Process s, Process d) =

  IF (e == NULL)
    {}
  ELIF (typeof(e) == Send AND e.process_id == s AND e.destination == d)
    M(PRED(e), s, d) + { e }
  ELIF (typeof(e) == Receive AND e.source == s AND e.process_id == d)
    M(PRED(e), s, d) - UNIQUE({
                               f IN M(PRED(e), s, d) SUCH THAT
                               (e.tag == f.tag AND
                                e.communicator == f.communicator AND
                                NEXISTS g IN M(PRED(e), p)
                                  SUCH THAT
                                    (f.tag == g.tag AND
                                     f.communicator == g.communicator AND
                                     g.timestamp < f.timestamp )) })
  ELSE
    M(PRED(e), s, d);
```

Figure 14: Message queue for traffic from a source to a destination location.

```
Send sendptr(Receive r) =

  UNIQUE({
           f IN M(PRED(e), r.source, r.process_id) SUCH THAT
             (r.tag == f.tag AND
              r.communicator == f.communicator AND
              NEXISTS g IN M(PRED(e), r.source, r.process_id)
                SUCH THAT
                  (f.tag == g.tag AND
                   f.communicator == g.communicator AND
                   g.timestamp < f.timestamp ))
       });
```

Figure 15: The `sendptr()` function representing the *sendptr* attribute.

8.2.1 Late Sender

```
PATTERN LateSender(Region r) {

  ROOTTYPE: Receive;

  INSTANTIATION:
    Send  send_msg   = sendptr(ROOT);
    Enter enter_recv = enterptr(ROOT);
    Enter enter_send = enterptr(send_msg);
  CONSTRAINT:
    EXISTS e IN R(ROOT, ROOT.process_id) SUCH THAT e.region == r AND
    enterptr(ROOT).region     == MPI_RECV                        AND
    enterptr(send_msg).region == MPI_SEND                        AND
    enter_send.timestamp > enter_recv.timestamp;
  EXPORT:
    float idle_time = enter_send.timestamp - enter_recv.timestamp;
}
```

The first sub-proposition of the conjunction in the constraint clause requires the root event to occur when the process of the root event is executing in region `r`. Note that this only one possible way of defining a constraint concerning the region in which a pattern match occurs.

8.2.2 Late Receiver

```
PATTERN LateReceiver(Region r) {

  ROOTTYPE: Receive;

  INSTANTIATION:
    Send  send_msg   = sendptr(ROOT);
    Enter enter_recv = enterptr(ROOT);
    Enter enter_send = enterptr(send_msg);
  CONSTRAINT:
    EXISTS e IN R(ROOT, ROOT.process_id) SUCH THAT e.region == r AND
    enterptr(ROOT).region     == MPI_RECV                        AND
    enterptr(send_msg).region == MPI_SEND                        AND
    enter_send.timestamp < enter_recv.timestamp                  AND
    EXISTS e IN R(enter_recv, send_msg.process_id)
      SUCH THAT e == enter_send;
  EXPORT:
    float idle_time = enter_recv.timestamp - enter_send.timestamp;
}
```

8.2.3 Messages in Wrong Order

This pattern can introduce costs in several ways. First, it can lead to a situation equal to *Late Sender* as depicted in Fig. 7. But is is also possible, that costs are introduced owing

to a blocking sender. If the messages sent before the current message allocated all the buffer space owned by the sending process, the sender might block until the corresponding receive operation is posted. This is equal to *Late Receiver*. But note that this situation may also end up in a deadlock, if the receiver is waiting for another message not being sent because of a blocked sender. In our ASL definition we want to distinguish between these two cases and therefore define two versions of the pattern. The difference is reflected in the constraint clause as well as in the export clause. Thus, the two patterns can be considered as specializations of the previous ones.

```
PATTERN WrongOrderLs(Region r) {

  ROOTTYPE: Receive;

  INSTANTIATION:
    Send  send_msg   = sendptr(ROOT);
    Enter enter_recv = enterptr(ROOT);
    Enter enter_send = enterptr(send_msg);
  CONSTRAINT:
    EXISTS e IN R(ROOT, ROOT.process_id) SUCH THAT e.region == r AND
    EXISTS s IN M(ROOT, send_msg.process_id, ROOT.process_id)
      SUCH THAT s.timestamp < send_msg.timestamp                 AND

    // constraints refering to late sender

    enterptr(ROOT).region     == MPI_RECV                        AND
    enterptr(send_msg).region == MPI_SEND                        AND
    enter_send.timestamp > enter_recv.timestamp;
  EXPORT:
    float cost = (enter_send.timestamp - enter_recv.timestamp);
}

PATTERN WrongOrderLr(Region r) {

  ROOTTYPE: Receive;

  INSTANTIATION:
    Send  send_msg   = sendptr(ROOT);
    Enter enter_recv = enterptr(ROOT);
    Enter enter_send = enterptr(send_msg);
  CONSTRAINT:
    EXISTS e IN R(ROOT, ROOT.process_id) SUCH THAT e.region == r AND
    EXISTS s IN M(ROOT, send_msg.process_id, ROOT.process_id)
     SUCH THAT s.timestamp < send_msg.timestamp                  AND

    // constraints refering to late receiver

    enterptr(ROOT).region     == MPI_RECV                        AND
    enterptr(send_msg).region == MPI_SEND                        AND
```

```
    enter_send.timestamp < enter_recv.timestamp                    AND
    EXISTS e IN R(enter_recv, send_msg.process_id)
     SUCH THAT e == enter_send;
  EXPORT:
    float cost = (enter_recv.timestamp - enter_send.timestamp);
}
```

8.3 Taking Advantage of Pattern Templates

Note that the *LateSender* and *LateReceiver* pattern expose a very similar structure, so we might consider using pattern templates resp. meta-patterns. Since both patterns share the same instantiation clause and substantial parts of the constraint clause, it would be natural to isolate this part in a separate definition.

However, if we parameterize the time comparison in the constraint clause, the relationship between the two patterns becomes a generalization/specialization relationship which might be represented by some kind of inheritance.

The *WrongOrder* pattern family also contains common elements which might be subject to reuse.

8.4 Using Patterns in Property Definitions

Remember that the original purpose of patterns was to make the very detailed information contained in event traces available to property definitions. In order to meet our goal, we redefine the *late_sender/late_receiver* properties and add two new properties related to sending messages in wrong order.

8.4.1 late_sender

```
PROPERTY late_sender(Region r, Experiment e, Region rank_basis){

LET
  float idle_time = SUM m.idle_time
    WHERE m IN PATTERN_MATCHES(LateSender(r), e.trace);
IN
  CONDITION: idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis,e);
}
```

The algorithm checks for pattern instances which occurred when the application was executing in a region `r`. As opposed to the former definition from [4] which associates this property with a point-to-point primitive belonging to a message passing library, the property becomes now associated with a region defined in the parallel application itself. This is obviously more useful, since it helps more in locating a potential performance bottleneck.

8.4.2 late_receiver

```
PROPERTY late_receiver(Region r, Experiment e, Region rank_basis){

LET
  float idle_time = SUM m.idle_time
    WHERE m IN PATTERN_MATCHES(LateReceiver(r), e.trace);
IN
  CONDITION: idle_time>0;
  CONFIDENCE: medium;
  SEVERITY: idle_time/duration(rank_basis,e);
}
```

8.4.3 wrong_order_ls

```
PROPERTY wrong_order_ls(Region r, Experiment e, Region rank_basis){

LET
  float cost = SUM m.cost
    WHERE m IN PATTERN_MATCHES(WrongOrderLs(r), e.trace);
IN
  CONDITION: cost>0;
  CONFIDENCE: 1;
  SEVERITY: cost/duration(rank_basis,e);
}
```

8.4.4 wrong_order_lr

```
PROPERTY wrong_order_lr(Region r, Experiment e, Region rank_basis){

LET
  float cost = SUM m.cost
    WHERE m IN PATTERN_MATCHES(WrongOrderLr(r), e.trace);
IN
  CONDITION: cost>0;
  CONFIDENCE: medium;
  SEVERITY: cost/duration(rank_basis,e);
}
```

9 Related Work

An alternative approach to describe complex event patterns was realized in [2]. The proposed Event Definition Language (EDL) is focused on specifying incorrect behavior of distributed systems and allows the definition of compound runtime events in a declarative manner based on extended regular expressions where primitive events are clustered

to higher-level events by certain formation operators. Relational expressions over the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. Abstraction mechanisms allow the reuse of already defined event patterns to form custom hierarchies of events. However, problems arise when trying to describe events that are associated with some kind of state.

Another method of representing performance problems is applied in the well-known Paradyn project [10]. In contrast to our approach Paradyn uses hypotheses and corresponding tests based on *metric-focus* grids. A metric-focus grid is a two dimensional matrix. The first dimension represents performance metrics, such as CPU time, blocking time, message rates, I/O rates, or number of active processors. The second dimension represents individual program components, such as procedures, processor nodes, disks, message channels, or barrier instances. The matrix cells contain the value of a metric with respect to one program component.

The method presented here is strongly influenced by the experiences gained during the design of the EARL trace analysis language [13]. EARL is aimed at providing basic building blocks for automatic performance analysis of message passing programs based on event traces. Just as in this paper a performance property is considered as a compound runtime event occurring in an event trace produced by a parallel application. The EARL trace analysis language helps to easily specify an appropriate search algorithm by providing useful abstractions allowing the algorithm to have a very simple structure even in case of complex event patterns. The abstractions are created as described in section 4 and are implemented as an extension to common scripting languages like Perl [11], Python [3] or Tcl [12]. So the remaining parts of the search algorithm can be built using one of the three scripting languages.

The EXPERT [14] tool architecture is designed on top of EARL and provides a Python class library for the detection of typical problems affecting the performance of MPI programs. EXPERT is characterized by a separation of the performance property specifications from the actual analysis process. This separation enables EXPERT to handle an arbitrary set of performance properties. A graphical user interface makes utilizing the class library for detection of typical MPI performance properties straightforward. In addition, a flexible plug-in mechanism allows the experienced user to easily integrate property specifications specific to a distinct parallel application without modifying the tool.

10 Conclusion

Event traces provide a very fine grained view on the performance behavior of a parallel application. Based on this view, performance properties which cannot be represented by profiling data can be specified in terms of compound runtime events (CREs). Using such specifications an automatic tool should be able to report the application designer on reasons for inefficient program behavior.

In many cases, a CRE indicating inefficient performance behavior exhibits a quite complex structure. Most of the relationships by which its constituents are interconnected depend on a specific programming model which makes it difficult to capture all such situations by means of one general representation method.

To overcome this problem, we presented a generic technique named *event trace model enhancement* for defining programming model specific abstractions allowing a simple de-

scription of CREs in the context of that programming model. The abstractions include system states and links between related events. System states separate ongoing activities from activities that are already completed. Based on the observation that in most cases the constituents of a CRE refer to a common set of ongoing activities, the definition of system states provides also a foundation for efficient search methods. Links pointing from one event to other related events allow to easily define a relationship between the constituents of a CRE along a path of relationship primitives. Now, a CRE can be specified on top of these abstractions by functional dependencies between its logical parts.

We demonstrated our approach by defining three example CREs representing typical performance properties of message passing programs. Although all three CREs are quite complex, their corresponding definitions are surprisingly short. This is essentially a result of defining them in terms of the message passing programming model which has become possible through event trace model enhancement.

In addition, we showed, that our approach can be easily integrated into the APART specification language, a novel research project directed towards a formal representation of performance properties.

Event trace model enhancement is especially well-suited for hybrid programming models which are of major importance in the emerging field of SMP cluster computing. The increased architectural complexity of these systems often demands the concurrent usage of more than one programming model in the same application. Assuming that there is already an enhanced event trace model for one programming model, a hybrid enhanced model can be easily constructed, since the system state and pointer attribute definitions of an enhanced model are invariant with respect to new event types which may be added later.

Last, it should be pointed out that all information delivered by profiling tools can also be derived from the information contained in event traces. Additionally, based on our ideas a profiling system might be envisioned which is able to deliver the necessary data calculated at runtime using the specified CREs as a basis but without generating the potentially large event traces.

References

[1] A. Arnold, U. Detert, and W.E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[2] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, February 1986.

[3] D. M. Beazley. *Python Essential Reference.* New Riders, October 1999.

[4] T. Fahringer, M. Gerndt, and G. Riley. Knowledge Specification for Automatic Performance Analysis. Technical report, ESPRIT IV Working Group APART, 1999.

[5] J. Galarowicz and B. Mohr. Analyzing Message Passing Programs on the Cray T3E with PAT and VAMPIR. In H. Lederer and F. Hertwick, editors, *Proceedings of Fourth European CRAY-SGI MPP Workshop*, pages 29–49, Garching/Munich (Germany), October 1998.

[6] W. Gropp and E. Lusk. *User's Guide for MPE: Extensions for MPI Programs.* Argonne National Laboratory, 1998. http://www-unix.mcs.anl.gov/mpi/mpich/.

[7] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI - Portable Parallel Programming with the Message-Passing Interface.* MIT Press, 1994.

[8] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proc. of the Scalable High-performance Conputing Conference (SHPCC)*, Knoxville, Tennessee, May 1994.

[9] J. K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, Oktober 1996.

[10] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[11] S. Srinivasan. *Advanced Perl Programming.* O'Reilly, August 1997.

[12] B. B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall, second edition, 1997.

[13] F. Wolf and B. Mohr. EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In A. Hoekstra and B. Hertzberger, editors, *Proc. of the 7th International Conference on High-Performance Computing and Networking (HPCN'99)*, pages 503–512, Amsterdam (The Netherlands), 1999.

[14] F. Wolf and B. Mohr. Automatic Performance Analysis of MPI Applications Based on Event Traces. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Munich (Germany), August 2000. To be published.