Monterey Phoenix

System and Software Architecture and Workflow Modeling Language (version 3.5)

Mikhail Auguston
Computer Science Department
Naval Postgraduate School
Monterey, California, USA
maugusto@nps.edu
Public servers running web app with MP editor, trace generator, and trace graph visualization are available:

MP version 3 at http://firebird.nps.edu/

MP wiki site: https://wiki.nps.edu/display/MP/Monterey+Phoenix+Home

This is a draft document for MP beta-testers. MP is still a work in progress.

Not yet implemented MP constructs are highlighted in red.

Material contained herein is made available for the purpose of peer review and discussion and does not necessarily reflect the views of US Department of the Navy or US Department of Defense.
Table of Contents

1. Introduction .......................................................................................................................... 5

2. Behavior models in MP ........................................................................................................... 6
   2.1 Event concept ..................................................................................................................... 6
   2.2 Event grammar .................................................................................................................. 6
   2.3 Behavior composition ....................................................................................................... 8
      2.3.1 Aspect-Oriented Paradigm in MP ................................................................................ 11
   2.4 Data items as behaviors .................................................................................................... 11
   2.5 Behavior of the environment ......................................................................................... 12
   2.6 Context conditions with ENSURE .................................................................................. 14
   2.7 Nested composition operations and queries ...................................................................... 15
   2.8 Threads with SUCH THAT condition, $$scope meta-symbol, reshuffling option..16
   2.9 Assertion checking ........................................................................................................... 18
   2.10 Trace annotation and queries about traces .................................................................... 19
   2.11 Business process modeling ......................................................................................... 22
   2.12 Process modeling: takeoff into the next dimension ......................................................... 24
   2.13 User-defined relations .................................................................................................... 28
   2.14 Event attributes ............................................................................................................. 32
      2.14.1 Interval attribute type................................................................................................. 34
      2.14.2 Timing attributes ..................................................................................................... 36
   2.15 Avoiding deadlocks in threads – Dining Philosophers ................................................... 40
   2.16 Modeling Finite State Machines .................................................................................... 42
      2.16.1 Modeling Model Checker and temporal logic formulae ............................................ 44
      2.16.2 Modeling Petri net in MP ........................................................................................ 47
   2.17 Coordinating asynchronous threads and tracking dependency chains ......................... 49
   2.18 Probabilistic models in MP ............................................................................................. 51
      2.18.1 Probability of event trace ........................................................................................ 52
      2.18.2 Probability of an event within a trace ...................................................................... 54
   2.19 MP model reuse ............................................................................................................. 56
      2.19.1 Compiler’s front end model reuse .......................................................................... 57
      2.19.2 Reuse with the MAP composition operation ............................................................ 61

3. Notes on MP semantics – how trace derivation works............................................................. 64

4. Some heuristics for MP code design ..................................................................................... 67
   4.1 Use scope with caution...................................................................................................... 67
   4.2 Use composition operations as early as possible ............................................................... 67
   4.3 Use BUILD blocks ........................................................................................................... 67
   4.4 Optimization of COORDINATE operations .................................................................. 68
   4.5 Use asynchronous coordination with caution .................................................................... 68
   4.6 Coordination and iteration ............................................................................................... 68
   4.7 Coordination and alternatives ........................................................................................ 69
   4.8 Synchronizing iteration cycles and coordination ............................................................... 69
   4.9 Merge and conquer ......................................................................................................... 71
   4.10 Divide and conquer ........................................................................................................ 73

5. MP language constructs ....................................................................................................... 80
   5.1 Lexics ............................................................................................................................... 80
5.2 Extended BNF notation for MP syntax .................................................................80
5.3 Event grammar rules ...............................................................................................80
5.4 Event patterns ..........................................................................................................81
5.5 Navigation directions ...............................................................................................83
5.6 Composition operations ...........................................................................................86
5.7 Event attributes ........................................................................................................95
5.8 Build blocks ............................................................................................................96
5.9 Expressions ................................................................................................................96
5.10 Aggregate operations .............................................................................................100

6. Appendices ..................................................................................................................101
  6.1 Axioms for basic relations .......................................................................................101
  6.2 Relational expressions ............................................................................................101
  6.3 Timing attribute calculation algorithms .................................................................102

7. Lists of MP keywords and meta-symbols ..................................................................103
  7.1 MP keywords and pre-defined attribute names .......................................................103
  7.2 MP meta-symbols ....................................................................................................104

8. Brief related work survey ............................................................................................104

9. Acknowledgments .......................................................................................................105

10. References ..................................................................................................................106
1. INTRODUCTION

Monterey Phoenix (MP) is a framework for software system architecture and business process (workflow) specification based on behavior models [Auguston, 2009a, 2009b, 2014]. Architecture plays a role as a bridge between requirements and implementation of a system, and represents a stepwise refinement in the design process with specific objectives and stakeholders. Business process model often represents a part of the system’s requirements, and interactions between the software and its environment is an integral part of architecture specification.

Software design involves finding an algorithm for solving the problem at hand and mapping it on the appropriate computational platform. An algorithm commonly is specified as a behavior applying a step-by-step procedure to solve the problem at hand. An architecture description belongs to a high level of abstraction, ignoring many details of the algorithm implementation and data structures.

In common software architecture models the main elements are components (representing the functionality) and connectors (representing the information flow between components). Both components and connectors are abstractions of behaviors. MP provides an executable architecture model and behavior modeling is at the core of MP approach. The main MP concepts are activities (or events) and coordination between them, based on the following principles.

- A view of the architecture as a high level description of possible system behaviors, emphasizing the behavior of subsystems (components) and interactions between subsystems. MP introduces the concept of event as an abstraction of activity.
- The separation of the interaction description from the components behavior is an essential MP feature. It provides for a high level of abstraction and supports the reuse of architectural models. Interactions between activities are modeled using event coordination constructs.
- The environment’s behavior is an integral part of the system architecture model. MP provides a uniform method for modeling behaviors of the software, hardware, business processes, and other parts of the system.
- The event grammar models the behavior as a set of events (event trace) with two basic relations, where the PRECEDES relation captures the direct dependency abstraction, and the IN relation represents the hierarchical relationship. Since the event trace is a set, additional constraints can be specified using set-theoretical operations and predicate logic.
- The MP architecture description is amenable to deriving multiple views (or abstractions), and provides a uniform basis for specifying both structural and behavioral aspects of a system.
- MP models are executable. MP tools support automated and exhaustive (for a given scope) scenario generation for early system architecture verification. The Small Scope Hypothesis [Jackson 2006] states that most flaws in models could be demonstrated on relatively small counterexamples. Testing and debugging of architecture models is supported by assertion checking for verifying the behavior properties.
- Humans understand examples better than formal models. Scenarios (or use cases) generated by MP can facilitate communication with system’s stakeholders.
- MP framework can assist in unifying UML activity and sequence diagrams, Statechart notations, and the recent Executable UML Alf language [OMG 2010] for behavior specification. MP may be used in addition to the existing architecture definition tools and methodologies.

MP is intended for the use in software and system architecture design and maintenance as a lightweight Formal Method. It provides an ecosystem for sanity checking tools, reusable architecture patterns, reusable assertions, queries, and tools for extracting architecture views.
In general, there are two major approaches to formal specification languages: data-centered and action-centered. Examples of data-centered approaches include type theories, first order predicate logic, entity-relation models, and relational logic [Jackson 2006]. Action-centered formalisms include finite state machines (finite automata, Statecharts, Petri nets, model checking), different kinds of flowcharts, temporal logics, process algebras [Hoare 1985], [Milner 1989], and algebraic specification systems (axioms about operations on data structures). Of course, some formal specification languages attempt to merge both paradigms. MP can be considered as action-centered formalism with emphasis on system behavior specification, although the data items when needed can be managed as well (see Sec. 2.4 for more details).

2. BEHAVIOR MODELS IN MP

In a certain sense, the source code of a program is a compact description for a set of required behaviors. The source code in any programming language – a finite object by itself – specifies a potentially infinite number of execution paths. The behavior of the system is usually the main concern for the developer, and the presence of unintended behaviors manifests errors in the design. A system is operating in a certain environment, which has its own behavior and interacts with the system. The objective of the MP approach is to provide a framework for specifying behaviors of the system, its parts and its environment, and interactions between them.

The purpose of any model is to help answer questions. MP model is designed to answer questions about system’s behaviors, including such aspects as structure of behavior, dependencies between actions involved in the behavior, constraints on behaviors, and to provide a source for different visualizations or views of behaviors.

2.1 Event concept

The MP behavior model is based on the concept of event as an abstraction of activity. The event has a beginning and an end, and may have duration (a time interval during which the action is accomplished).

The behavior of a system is modeled as a set of events with two binary relations defined for them: precedence (PRECEDES) and inclusion (IN) – the event trace. One action is required to precede another if there is a dependency between them, e.g. the Send event should precede the Receive event. Events may be nested, when a complex activity contains a set of other activities. Defining one of these basic relations for a pair of events represents an important design decision. Usually system behavior does not require a total ordering of events. Both PRECEDES and IN are partial ordering relations. If two events are not ordered, they may overlap in time, in particular, they may occur concurrently. Section 6.1 provides axioms specifying the properties of basic relations.

2.2 Event grammar

The MP model of system’s behavior is written as a schema – a collection of rules describing behavior of components within the system, external actors, and their interactions. The structure of possible event traces is described by event grammar. A grammar rule specifies structure for a particular event type (in terms of IN and PRECEDES relations) and has a form

A: pattern_list;

where A is an event type name and pattern_list is composed from event patterns. Event types that do not appear in the left hand part of rules are considered atomic and may be refined later by adding corresponding rules.
An instance of an event trace satisfying the grammar rule can be visualized as a directed graph with two types of edges (one for each of the basic relations). Events are visualized as boxes, and basic relations as arrows. Fig. 1 outlines the event patterns for use in the grammar rule’s right hand part. Here B, C, D stand for event type names or event patterns.

Sequence denotes ordering of events under the PRECEDES relation. The grammar rule A: B C means that an event a of the type A contains ordered instances of events b and c matching B and C, correspondingly, and relations b IN a, c IN a, and b PRECEDES c hold. A grammar rule may contain a sequence of several events, like A: B C D. In Fig. 1 boxes denote event instances, dashed arrows – IN relations, and solid arrows – PRECEDES relations.

Pattern (+ B +) denotes a sequence of one or more event B instances under PRECEDES relation. It is similar to (* B *) with one additional constraint: there should be at least one B instance in the sequence.

{+ B +} denotes a set of one or more events B.

In all cases it is assumed that iterated event instances are unique. Iteration is used to describe repeated behavior patterns.

An event grammar is a grammar for directed acyclic graphs of vertices (events) with edges representing relations IN and PRECEDES.

An event trace represents an instance of a particular execution of the system or a use case, especially if the behavior of the environment is included. Event traces can be effectively derived from the event grammar rules and then adjusted and filtered according to the composition operations and constraints in the schema. This justifies the term *executable architecture model*. For a given MP schema it is possible to obtain all valid event traces up to a certain limit. Usually such a limit (scope) is set as the upper limit on the number of iterations in grammar rules (recursion can be limited in a similar way, but current Firebird MP tool does not allow explicit or implicit recursion in event grammar rules). For many purposes a modest limit of 3 iterations will be sufficient. The process of generating and inspecting event traces for the schema is similar to the traditional software testing process.
Since it is possible to automatically generate all event traces within the given scope for MP model, it provides for exhaustive testing – a feature usually not available for traditional software testing. Careful inspection of generated traces (scenarios/use cases) may help developers to identify undesired behaviors. Usually it is easier to evaluate an example of behavior (particular event trace) than the generic description of all behaviors (the schema). The Small Scope Hypothesis [Jackson 2006] states that most errors can be demonstrated on relatively small counterexamples. Assertion checking and SAY clauses in MP make the event trace inspection easier.

Event grammar notation is conceptually close to the pseudo-code notation used in software design [Knuth, 1984]. MP architecture model can be refined up to the quite detailed design model. Event names can be considered as pseudo-code statements, and the imperative control structures can be modeled in MP as follows.

Behavior of the conditional statement

if Condition then Statement1 else Statement2

is modeled in MP with the event pattern

Check_Condition
( Condition_is_true    Perform_Statement1 |
  Condition_is_false   Perform_Statement2 )

Behavior of terminating loop, where loop body is executed zero or more times

while(Condition) do Loop_body

is modeled as

Check_Condition
(* Condition_is_true
  Perform_Loop_body
  Check_Condition
*)
Condition_is_false

Behavior of terminating loop, where loop body is executed one or more times

Repeat Loop_body until (Condition)

is modeled as

(* Perform_Loop_body
  Check_Condition
  Condition_is_false
*)
Perform_Loop_body
Check_Condition
Condition_is_true

2.3 Behavior composition

The behavior of a particular system is specified as a set of possible event traces using a schema. The purpose is to define the structure of event traces in terms of IN and PRECEDES relations using event grammar rules and other constraints. A schema contains a collection of events called roots
representing the behaviors of parts of the system (components and connectors in common architecture descriptions), composition operations specifying interactions between these behaviors, and additional constraints on behaviors.

There is precisely one instance of each root event in a trace. A schema can contain auxiliary grammar rules for composite event types used in other rules. In principle, schema may define both finite and infinite traces, but most analysis tools for reasoning about a system’s behavior assume that a trace is finite. Only finite event traces are considered in current MP tools (including Firebird).

The schema represents instances of behavior (event traces) in the same sense as Java source code represents instances of program execution. Just as a particular execution path can be extracted from a Java source code by running it on a JVM, a particular event trace specified by MP schema can be derived from the event grammar rules by applying composition operations and constraints.

Example 1. Simple interaction pattern.

```
SCHEMA simple_message_flow
ROOT Task_A: (* send *);
ROOT Task_B: (* receive *);
COORDINATE $x: send FROM Task_A,
      $y: receive FROM Task_B
      DO ADD $x PRECEDES $y; OD;
```

The composition operation `COORDINATE` coordinates behaviors of two root events sending and receiving messages. This trace operation takes two root event traces and produces a modified event trace (merging behaviors of `Task_A` and `Task_B`) by adding the `PRECEDES` relation for the selected `send` and `receive` pairs. Essentially it is a loop performed over the structure of coordinated events, hence the `DO - OD` notation for the loop body.

This `COORDINATE` composition uses event selection patterns to specify subsets of root traces that should be coordinated. The `send` pattern identifies the set of events selected from `Task_A`. The default for event selection in the coordination source within `COORDINATE` preserves the order of events generated during the derivation process. The default means that the ordering of events in selected sets will remain “as is”.

Both selected event sets should have the same number of elements (`send` events from the first trace and `receive` events from the second), and the pair coordination follows their ordering (synchronous coordination), i.e. first `send` is paired with first `receive`, second with the second, and so on. MP variables $x$ and $y$ provide access to the pair of events matching the selection pattern within each iteration. The `ADD` operation completes the behavior adjustment, specifying additional `PRECEDES` relation for each pair of selected events. Behavior specified by this schema is a set of matching event traces for `Task_A` and `Task_B` with the modifications imposed by the composition. If the numbers of events in the selected event sets (coordination sources) are not equal, the coordination operation fails to produce a resulting trace. The reshuffling operations may be applied to the default set before the coordination to rearrange events within the selected set (see Publish/Subscribe Example 7 in section 2.8, and ring topology Example 14 in section 2.13).

The selection pattern may be either an event type name (atomic or composite), or an alternative pattern composed of event type names.
Fig. 2 gives a sample of event trace satisfying the schema `simple_message_flow`. It resembles a UML sequence diagram’s “swim lanes”.

Different views (or abstractions) for different stakeholders can be extracted from MP schemas. For example, each root may be visualized as a box. If there is a composition operation specifying an interaction (or coordination) between root behaviors, the boxes are connected by an arrow as illustrated in Fig. 3. This diagram is obtained by collapsing root events in the Firebird’s graph window for a trace containing a representative set of events and relations. The root behavior by itself may be visualized with UML Activity Diagram [Booch et al. 2000].

![Diagram of event trace]

Fig. 2. Example of a composed event trace for the `simple_message_flow` schema (scope 2).

Fig. 3. Architecture view for the `simple_message_flow` schema.

Another way of selecting event pairs is called `asynchronous coordination`. In this case the coordination source intended for asynchronous coordination should be marked with a symbol `<!>`. That is an event reshuffling operation performed before the body of coordination operation. For example,

```
COORDINATE <!>  $x: E1  FROM A,
               $y: E2  FROM B
DO    ADD $x PRECEDES $y;   OD;
```

Matching event sets from A and B should contain an equal number of selected events E1 and E2, correspondingly. But now the resulting merged traces will include all permutations of events E1 from A paired with the same (default) sequence of events E2 from B, with the `PRECEDES` relation imposed on each selected pair. This assumes that other constraints, like the axioms from Sec. 6.1, are satisfied. Each permutation yields one potential instance of a resulting trace for the schema deploying this composition. Use of `<!>` may significantly increase the number of composed traces. Read about other options for event reshuffling within the coordination source in the rules `event_reshuffling_option(31)` and `reshuffling_unit(33)`.

The composition operation may be considered as an abstract interaction (interface) description for behaviors. The separation of the interaction description from the components behavior is an essential MP feature. The same component behavior may be reused with different interaction descriptions – a useful composition/reuse aspect.
2.3.1 Aspect-Oriented Paradigm in MP

The COORDINATE operation supports a “cause-effect” refinement for the behavior of two components and it bears a certain similarity to Aspect-Oriented Programming (AOP) paradigm [Kiczales et al. 1997] (or read https://en.wikipedia.org/wiki/Aspect-oriented_programming). For example, the following AOP pattern could be modeled by MP schema where event coordination implements AOP join point and advice coordination.

Suppose that the main stream of execution contains calls to methods M1 and M2 as join points, and the aspect behavior requires a call to Prolog before, and to Epilog after each method call as an advice. The corresponding MP model may look like the following.


```
SCHEMA AOP
ROOT Main: (* ( M1 | M2 | do_something_else ) *);
ROOT PreAdvice: (* Prolog *);
ROOT PostAdvice: (* Epilog *);

COORDINATE $jp: ( M1 | M2 ) FROM Main,
              $a1: Prolog FROM PreAdvice,
              $a2: Epilog FROM PostAdvice
DO ADD $a1 PRECEDES $jp,
       $jp PRECEDES $a2;
OD;
```

The COORDINATE operation may have multiple coordination sources (in a valid trace the number of selected events in each should be the same), and the same ADD operation may process several relations.

2.4 Data items as behaviors

Data items in MP are represented by actions (events) that may be performed on that data. This principle follows the Abstract Data Type (ADT) concept introduced in [Liskov, Zilles 1974].

Example 3. Data flow.

```
SCHEMA Data_flow
   ROOT Writer: (* ( working | writing ) *);
/* writing events should precede reading */
   ROOT File: (+ writing +) (* reading *);

Writer, File SHARE ALL writing;

ROOT Reader: (* ( reading | working ) *);

Reader, File SHARE ALL reading;
```

The behavior of File requires writing events to be completed before reading events, and there should be at least one writing event. The SHARE ALL composition operations ensure that the schema admits only event traces where corresponding event sharing is implemented.
Event sharing is defined as following (here X, Y are events, Z is an event type, and FROM is a transitive closure of IN, <-> means “if and only if”).

\[ X, Y \text{ SHARE ALL } Z <-> \{ v: Z \mid v \text{ FROM } X \} = \{ w: Z \mid w \text{ FROM } Y \} \]

Events to be shared are of the same type (have the same event type name) and hence have the same event attribute signature (see 2.14). If the values of corresponding event attributes are not equal, the SHARE ALL operation fails, and the failure is propagated to the enclosing BUILD block or schema.

There is one exception. Time attributes depend on basic relations and are recalculated automatically each time when basic relations of an event are adjusted in the derivation process. When a pair of events is SHAREd, their relations (basic and user-defined) are merged, and this may trigger time attribute recalculations.

The order of shared events is either the default, or can be specified by the SHARE_clause(41) combined with event reshuffling option (31). Event sharing is yet another way of behavior coordination. It is assumed that shared events may appear within an event at any level of nesting.

At the architectural level data items are inputs or outputs of activities and are modeled as operations that may be performed on them. This is simple and uniform concept.

Fig. 4. An example of composed event trace for the Data_flow schema.

2.5 Behavior of the environment

The following example demonstrates how to integrate the behavior models of an environment and a system that operates in this environment. The ATM_withdrawal schema specifies a set of possible interactions between the Customer, ATM_system, and Data_Base.

Example 4. Withdraw money from ATM.

SCHEMA ATM_withdrawal
ROOT Customer: (* insert_card

   ( identification_succeeds
     request_withdrawal
     ( get_money | not_sufficient_funds ) )

   identification_fails

*)

ROOT ATM_system: (* read_card

   validate_id

)
( id_successful
  check_balance
   ( sufficient_balance
       dispense_money
     |
       unsufficient_balance
   )
   |
   id_failed
  )

COORDINATE $x$: insert_card FROM Customer,
  $y$: read_card FROM ATM_system
  DO ADD $x$ PRECEDES $y$; OD;

COORDINATE $x$: request_withdrawal FROM Customer,
  $y$: check_balance FROM ATM_system
  DO ADD $x$ PRECEDES $y$; OD;

COORDINATE $x$: identification_succeeds FROM Customer,
  $y$: id_successful FROM ATM_system
  DO ADD $y$ PRECEDES $x$; OD;

COORDINATE $x$: get_money FROM Customer,
  $y$: dispense_money FROM ATM_system
  DO ADD $y$ PRECEDES $x$; OD;

COORDINATE $x$: not_sufficient_funds FROM Customer,
  $y$: unsufficient_balance FROM ATM_system
  DO ADD $y$ PRECEDES $x$; OD;

COORDINATE $x$: identification_fails FROM Customer,
  $y$: id_failed FROM ATM_system
  DO ADD $y$ PRECEDES $x$; OD;

ROOT Data_Base: (* validate_id [ check_balance ] *)
/* the structure of Data_Base behavior follows the pattern of events validate_id and check_balance
 in ATM_system, so that these two streams of events could be coordinated with SHARE ALL
 for any scope */
Data_Base, ATM_system SHARE ALL validate_id, check_balance;
An event trace generated from the schema can be considered as a use case example. The event trace on Fig. 5 can be viewed also as an analog of UML sequence diagram’s “swim lanes” for Customer and ATM_system interactions.

The concept of environment in an architecture model includes behavior of other systems, hardware, business processes, and any other behaviors, which are not part of the system under consideration, but may interact with it. In particular, this approach may be of use for analyzing emergent behaviors of System of Systems, when the model of SoS is composed from the models of its components.

2.6 Context conditions with ENSURE

The event trace is a set of events and the Boolean expressions in MP can embrace the traditional predicate calculus notation. A set of behaviors (event traces) may be defined by the event grammar rules, composition operations, and some additional constraints – ENSURE conditions.

Example 5. Stack behavior.

Stack implements the “last in, first out” behavior for storing/retrieving elements. It is assumed that initially Stack is empty.

```
SCHEMA Stack_behavior

ROOT Stack: (* ( push | pop ) *)
  BUILD {
    /* If an element is retrieved, it should have been stored before */
    ENSURE FOREACH $x: pop FROM Stack
      ( #pop BEFORE $x < #push BEFORE $x );
  };
```

Fig. 5. An example of event trace for ATM_withdrawal schema.
This rule specifies the behavior of a stack in terms of stack operations \texttt{push} and \texttt{pop}. The \texttt{BUILD} block associated with a root or composite event contains composition operations performed when event trace segment representing the behavior of that root or composite is generated. The \texttt{ENSURE} Boolean expression provides a condition that each valid trace should satisfy. The domain of the universal quantifier is the set of all \texttt{pop} events inside \texttt{Stack}. The \texttt{FROM Stack} part is optional, by default it is assumed to be \texttt{FROM THIS}.

The function \texttt{#pop BEFORE $x$} yields the number of \texttt{pop} events preceding $x$. The set of valid event traces specified by this schema contains only traces that satisfy the constraint. This example presents a filtering operation as yet another kind of behavior composition, and demonstrates an example of combining imperative (event grammar) and declarative (Boolean expressions) constructs for behavior specification.

Example 15 provides more advanced version of Stack behavior model.

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{Stack_behavior.png}
\caption{Stack behavior, trace #22 for scope 5}
\end{figure}

\textbf{2.7 Nested composition operations and queries}

The following example demonstrates some MP techniques for coordination and simple queries about trace properties.

\textit{Example 6. Car race.} \\

\texttt{SCHEMA Car\_Race}  \\
\texttt{/* root event Cars is a set of concurrent events, each modeling the behavior of a single Car; there should be at least one Car in the car race */}  \\
\texttt{ROOT Cars: \{ + Car + \}}  \\
\texttt{/* following are context conditions local for the root Cars */}  \\
\texttt{BUILD\{}}  \\
\texttt{/* everybody who finishes drives the same number of laps */}  \\
\texttt{ENSURE FOREACH DISJ $c1$: Car, $c2$: Car}  \\
\texttt{(\#finish FROM $c1 == 1 \text{ AND} \#finish FROM $c2 == 1 \rightarrow}  \\
\texttt{\#drive\_lap FROM $c1 == \#drive\_lap FROM $c2) \text{ AND } \texttt{AND}}  \\
\texttt{/* if it breaks, then earlier */}  \\
\texttt{}}
(#finish FROM $c1 == 1 AND break FROM $c2 == 1 ->
  #drive_lap FROM $c1 >= #drive_lap FROM $c2);

/* there always will be at most one winner */
ENSURE #winner <= 1;

/* if at least one car finishes, there should be a winner */
ENSURE #finish > 0 -> #winner > 0;
}

/* Composite event Car encapsulates the behavior of a single car */
Car:  start
  (* drive_lap *)
  /* not every car will finish, sometimes a car that finishes is also a winner */
  ( finish [ winner ] | break);

ROOT Judge:  provide_start_signal
  watch
  /* Judge will monitor the finish events for Cars */
  (* finish *);

/* one-many coordination can be achieved with nested COORDINATE */
COORDINATE $a: provide_start_signal FROM Judge
  DO COORDINATE $b: start FROM Cars
    DO ADD $a PRECEDES $b; OD;
  OD;

/* ordering of finish events in concurrent Car threads can be achieved by coordinating them
  with finish event sequence in the Judge root */
Cars, Judge SHARE ALL finish;

/* now, when the ordering of finish events is ensured, we can specify another constraint:
  the winner is the car, that finishes first */
COORDINATE $w: winner
  DO ENSURE #finish BEFORE $w == 1; OD;

/* this is an example of a query annotating the trace with data specific for the trace */
SAY("In this race " #finish " cars have finished");

2.8 Threads with SUCH THAT condition, $$scope meta-symbol, reshuffling option

Example 7.  Publish/Subscribe architecture model.

Publisher sends a notification to all active Subscribers when new information is available.

SCHEMA Publish_Subscribe

ROOT Publisher:  Register
  (+ <2.. 3 + $$scope * 2> ( Register | Unsubscribe | Send ) +)
  /* Iteration_scope <2.. 3 + $$scope * 2> is needed to provide enough events for
  coordination, since each Client has Register/Unsubscribe event pair,
plus at least three Send events */

BUILD{
/* The Register and Unsubscribe events should be balanced */
  ENSURE #Register == #Unsubscribe;

  ENSURE FOREACH $u: Unsubscribe
      #Register BEFORE $u > #Unsubscribe BEFORE $u;
/* Publisher will Send only if there is at least one active Client.
 This condition is specified with ENSURE. */

  ENSURE FOREACH $s: Send
      #Register BEFORE $s > #Unsubscribe BEFORE $s;
};

ROOT Subscribers: {+ Client +};

Client: Register (* Receive *) Unsubscribe;

/***** Coordination between Publisher and Subscribers *****/
/* The ordering of composition operations is important, since Register and Unsubscribe events are
used to coordinate Send/Receive between Publisher and Client.
First, we do SHARE ALL for Register events, next will try all valid permutations of
Register/Unsubscribe, and then proceed with further coordination of Send/Receive,
using the shared Register and Unsubscribe events as pivots to coordinate Send events for each
Client */

/* Need to try all possible permutations of Register/Unsubscribe pairs,
 hence the use of <!> event reshuffling */
COORDINATE $r1: Register FROM Publisher,
    <!> $r2: Register FROM Subscribers
  DO SHARE $r1 $r2; OD;

COORDINATE $u1: Unsubscribe FROM Publisher,
    <!> $u2: Unsubscribe FROM Subscribers
  DO SHARE $u1 $u2; OD;

/* the following coordination is performed for each valid permutation of Register/Unsubscribe */
COORDINATE $client: Client FROM Subscribers
    DO COORDINATE $reg: Register FROM $client,
       $uns: Unsubscribe FROM $client

/* For Send event selection to coordinate with a particular Client event, the SUCH THAT thread
condition is used. Client will Receive all and only Send between his Register and Unsubscribe */
    DO COORDINATE $r: Receive FROM $client,
       $s: Send FROM Publisher
       SUCH THAT $s AFTER $reg AND $s BEFORE $uns

/* Register and Unsubscribe events have been shared and now can be used to select appropriate
Receive events from the Client and Send events from the Publisher for coordination */
    DO ADD $s PRECEDES $r; OD;
/* It might be interesting to find out whether there are Send events received by several Clients. MARK traces containing such event configuration, and place annotation pointing to the Send event of interest*/
COORDINATE $s: Send
DO
    IF (#Receive FOLLOWS $s > 1) THEN
        MARK;
        ADD SAY("this Send is received by "
            #Receive FOLLOWS $s " Clients")
        PRECEDES $s;
    FI;
OD;

Fig. 7. Publish_Subscribe example, Gryphon screenshot for scope 2, trace #10.

2.9 Assertion checking

MP models need to be tested and debugged like any other programming artifacts. Trace generation for a given scope (usually up to scope 3) yields exhaustive set of valid event traces. Browsing these traces may result in finding examples of behavior that are unwanted. Then we should fix the issue in the MP code, and run the trace generation again, until our expectations are satisfied. Browsing dozens or hundreds of event traces may be time consuming and error prone. The CHECK construct makes it possible to use automated trace monitoring. If the property (Boolean expression in the CHECK) is not satisfied, the trace will be marked and available for further inspection. This can significantly speed up the testing process. Of course, if the property should always hold, it makes sense to convert it into the ENSURE condition.

Example 8. Communicating via unreliable channel.

SCHEMA AtoB
ROOT TaskA: (* A_sends_request_to_B

OD;
OD;
( A_receives_data_from_B  |  
   A_timeout_waiting_from_B  )
*);

ROOT Connector_A_to_B:
( * A_sends_request_to_B
   ( B_receives_request_from_A
       B_sends_data_to_A
       ( A_receives_data_from_B  |  
         A_timeout_waiting_from_B  )
       A_timeout_waiting_from_B
   )
   A_timeout_waiting_from_B
 );

/* A_timeout_waiting_from_B may happen because Connector_A_to_B fails to pass a message
from A to B, or from B to A */

TaskA, Connector_A_to_B SHARE ALL  
A_sends_request_to_B,
A_timeout_waiting_from_B,
A_receives_data_from_B;

ROOT TaskB:  (* B_receives_request_from_A
        B_sends_data_to_A
 );

TaskB, Connector_A_to_B SHARE ALL  
B_receives_request_from_A,
B_sends_data_to_A;

/* assertion checking will mark examples of traces where TaskA does not receive
the requested data and will attach a message to the trace */
CHECK  #A_sends_request_to_B FROM TaskA ==
       #A_receives_data_from_B FROM TaskA
ONFAIL SAY("loss of reception detected");

Fig. 8. Example of AtoB on Gryphon, scope 2, trace #8

2.10 Trace annotation and queries about traces
Assertion checking with the CHECK clause may be the simplest and most common tool for finding
counterexamples of traces, which violate some property. Traces violating the CHECK condition are
marked, and the message provides some hint about the cause. The message is associated with the root, composite, or the whole schema, and further details may require browsing of the relevant trace segment. The following example demonstrates how to make messages more specific and focused.

Let’s consider again the Stack model, but this time without the ENSURE constraint. This Stack certainly will produce unwanted behaviors. The MP code illustrates how to pinpoint events that might be violating certain property, and how to annotate these particular events in the trace with additional messages. Programmers will recognize the use of debugging print statements for traditional program testing/debugging.


SCHEMA Unconstrained_Stack
ROOT Stack: (* ( push | pop [ bang ] ) *) /* bang event models the Stack underflow event */
BUILD {
  /* by default the domain for event selection is FROM THIS */
  /* this is to make sure that each pop attempting to pop empty Stack is followed by bang event */
  ENSURE FOREACH $p: pop
    /* pop associated with the bang event does not affect Stack contents, 
    hence only successful pop events count */
    (#pop BEFORE $p - #bang BEFORE $p) >= #push BEFORE $p
    <-> #bang FOLLOWS $p == 1;

  COORDINATE $b: bang
    /* pick up pop event immediately preceding bang */
    DO COORDINATE $x: pop SUCH THAT $x PRECEDES $b
      DO
        ADD SAY("This event tries to pop empty Stack")
          PRECEDES $x;
        MARK;
      OD;
    OD;
  };

Now traces where Stack behavior is incorrect will be MARKed and corresponding pop events will be pointed to by PRECEDES arrow leading from the message box associated with this event. Note that the sequence of pop push pop will trigger the message for the first pop event only, assuming that the second pop is not applied to empty stack.

Trace annotation may improve the overall readability of use cases, generated by MP tools.

Example 10. Queries about event trace. Pipe/Filter architecture.

Trace annotation with SAY message boxes may facilitate better understanding of behaviors and help to navigate the list of derived traces.

The assumptions for this pipe/filter architecture model with 2 Filters are as follows.

- Items send by Producer not always will be received by Consumer.
- For each Filter the number of sent messages cannot exceed the number of received messages.
• Received messages can stay in the Filter for while, before being send (if at all).

SAY messages can contain information specific for the trace instance (in this example, number of lost messages for Consumer), and could be used as queries to obtain specific data from the trace. For example, we could annotate only traces where messages were lost in both Filters.

 Attaching SAY boxes to particular events improves readability and helps to understand the behavior in more detail (in this example, annotations pointing to particular Filters, where a message got lost).

Please also notice how to describe patterns for better coordination. We could use 

Filter: (* (* ( send | receive ) *) *) or Filter: (* ( send | receive ) *)

to provide sufficient number of send/receive event sequences for coordination, but then the number of traces goes up dramatically and many of the valid traces are duplicates (because the same sequence of send/receive can be obtained from different configurations of these patterns). So, the proper choice of patterns for coordination may be very important.

The Filter’s behavior has a clear pattern: send can be done only after at least one receive.
Filter: (* receive (* send *) *)

This pattern yields an exhaustive set of possible send/receive sequences without duplication for any scope.

SCHEMA Pipe_Filter
ROOT Producer: (* send *);

ROOT Filter1: (* receive (* send *) *) */ Filter can send only after receive */
BUILD {
    ENSURE FOREACH $s: send (#receive BEFORE $s > #send BEFORE $s);
    /* annotate root’s trace if not everything received will be sent */
    IF #send != #receive THEN
        SAY("some messages have been lost in Filter1" );
    FI;
};

COORDINATE $s: send FROM Producer,
             $r: receive FROM Filter1
DO ADD $s PRECEDES $r; OD;

ROOT Filter2: (* receive (* send *) *)
BUILD {
    ENSURE FOREACH $s: send (#receive BEFORE $s > #send BEFORE $s);
    IF #send != #receive THEN
        SAY("some messages have been lost in Filter2" );
    FI;
};

COORDINATE $s: send FROM Filter1,
             $r: receive FROM Filter2
DO ADD $s PRECEDES $r; OD;

ROOT Consumer: (* receive *);

COORDINATE $s: send FROM Filter2, $r: receive FROM Consumer
DO ADD $s PRECEDES $r; OD;

/* Annotate and MARK trace if Consumer does not receive all messages. This is an example of query providing data specific for this trace. */
IF #receive FROM Consumer != #send FROM Producer THEN
  SAY( #send FROM Producer - #receive FROM Consumer " messages did not reach Consumer");
MARK;
FI;

Fig. 9. Pipe_Filter example, scope 2, trace #6

2.11 Business process modeling
The web site [http://www.infoq.com/articles/bpelbpm](http://www.infoq.com/articles/bpelbpm) “Why BPEL is not the holy grail for BPM “ provides an example of a process that can be described in BPMN but not in BPEL. It is claimed to be a challenge for process specification. Here is the description.

“When a new employee arrives at a company, a workflow is instantiated. First a record needs to be created in the Human Resources database. Simultaneously, an office has to be provided. As soon as the Human Resources activity has been completed, the employee can undergo a medical check. During that time, a computer is provided. This is only possible when both an office has been set up and an account created in the information system from the human resource database. When both the computer and the medical check have been performed, the employee is ready to work. Of course, you may want to model this simple process differently, but the point here is that you cannot define a structured parallel workflow [in BPEL] that is equivalent to this one..."

The advantage of MP is in the ability to coordinate events at any location in the behavior hierarchy.


```
SCHEMA Employee_Arrival
ROOT Employee: SendArrivalDate
               MedicalCheck
```
ROOT Employer: EmployeeArrival
/* these are concurrent threads */
{ Fill_HR_DB MedicalCheck,
  ProvideOffice ProvideComputer }
ReadyToWork;

Employee, Employer SHARE ALL MedicalCheck, ReadyToWork;

COORDINATE $a: SendArrivalDate FROM Employee,
$b: EmployeeArrival FROM Employer
DO ADD $a PRECEDES $b; OD;

/* coordination between two events deep inside the process */
COORDINATE $a: Fill_HR_DB FROM Employer,
$b: ProvideComputer FROM Employer
DO ADD $a PRECEDES $b; OD;

Fig. 10. An example of event trace for Employee_Arrival schema.

Example 12. Workflow pattern.

The web site http://workflowpatterns.com/ contains a rich collection of workflow patterns developed at Eindhoven University of Technology and Queensland University of Technology. The patterns are specified using Petri net notation.

Pattern 9 (Structured Discriminator)

When handling a cardiac arrest, the check_breathing and check_pulse tasks run in parallel. Once the first of these has completed, the triage task is commenced. Completion of the other task is ignored and does not result in a second instance of the triage task.
SCHEMA Cardiac_Arrest_Triage

ROOT Phase1: {  check_breathing [finish_first],
    check_pulse [finish_first]  }
BUILD{  ENSURE #finish_first == 1;  };

ROOT Triage:  
    identify_the_patient
    record_assessment_findings
    identify_the_priority;

COORDINATE  
    $a$: finish_first,
    $b$: identify_the_patient
    DO ADD $a$ PRECEDES $b$; OD;

Fig. 11. An example of event trace for Cardiac_Arrest_Triage schema.

2.12 Process modeling: takeoff into the next dimension

The following example is adopted from [Giammarco et al. 2014]. Consider the flight scenario depicted in Figures 6a and 6b, which provide a simplistic view of an uneventful Standard Flight through the National Airspace System. This Systems Modeling Language (SysML) activity diagram spans the two figures and illustrates the sequential interactions of the passenger, the pilot (or aircraft) and the various air traffic controllers through each phase of flight, from Preflight through Landing.

Continental United States flights enter the En Route phase of flight but do not necessarily enter the Oceanic phase. Consider, for example, an alternate path for flights that enter the Oceanic phase of flight. Furthermore for simplicity, there are only two additional alternate paths in this scenario representing times when a controller may ask a pilot to enter a holding sequence: Hold in queue during the Take Off phase and Hold during the Approach phase. After the aircraft taxis to the runway, the controller may or may not issue a Ground Hold due to inclement weather conditions. Then the aircraft enters a departure queue and awaits clearance for takeoff. On approach, the controller may direct the aircraft to Hold due to congestion at the airport, the final embedded alternate path. The aircraft circles at a set location until receiving clearance to land. Of course an actual flight would have many more alternate paths.
The SysML activity diagram in Figures 6a and 6b illustrates the flight scenario and has served as the source material to express the behaviors and interactions of the MP model that follows. MP code is used to describe the behaviors and interactions of the main actors and phases.

Fig. 12. Standard Flight: Preflight, Takeoff, Departure Phases

Fig. 13. Standard Flight: En Route, Descent, Approach, Landing Phases

This MP schema illustrates the following:

- Each of the main flight phases in the SysML diagram is modeled as a root event.
- Each of the main actors in the SysML diagram is modeled as a root event.
- Interactions and overlapping between phases and actors are modeled with SHARE ALL.
and COORDINATE.

Example 13. Standard flight model.

/****************************/
/* Main flight phases */
/****************************/

SCHEMA Flight
ROOT Preflight: BoardAircraft
    FlightCheck
    DepartureClearance
    Pushback
    IssueGroundInstruction
    Taxi
    Clear_for_takeoff;

ROOT Takeoff: (* Hold_in_queue *)
    Clear_for_takeoff
    Liftoff
    Handoff_to_TRACon;

ROOT Departure: ChangeFrequency
    IssueClearances
    Handoff_to_ARTCC;

ROOT EnRoute: IssueInstruction
    ( OceanicExtension | Follow_route )
    ChangeFrequency2
    Handoff_to_TRACon2;

ROOT Descent: Clear_descent
    Maneuver_toward_Airport;

ROOT Approach: (* Hold *)
    Clear_approach
    Enter_approach_line
    Handoff_to_tower;

ROOT Landing: Clear_landing
    Land
    Taxi_instruction
    Taxi_to_gate
    Disembark;

/****************************/
/* Main actors */
/****************************/

ROOT Passenger: BoardAircraft
    InsideCabin
    Disembark;

ROOT Pilot: FlightCheck
Pushback
Taxi
(* Hold_in_queue *)
Clear_for_takeoff
Liftoff
ChangeFrequency
( OceanicExtension | Follow_route )
ChangeFrequency2
Maneuver_toward_Airport
(* Hold *)
Enter_approach_line
Land
Taxi_to_gate;

ROOT Controller: DepartureClearance
IssueGroundInstruction
Clear_for_takeoff
Handoff_to_TRACon
IssueClearances
Handoff_to_ARTCC
IssueInstruction
Handoff_to_TRACon2
Clear_descent
Clear_approach
Handoff_to_tower
Clear_landing
Taxi_instruction;

/*---------------------------------------------------------------*/
/* Interactions between phases */
/*---------------------------------------------------------------*/
COORDINATE  $a: Taxi FROM Preflight,
             $b: Clear_for_takeoff FROM Takeoff
                    DO ADD $a PRECEDES $b; OD;

COORDINATE  $a: Handoff_to_TRACon FROM Takeoff,
             $b: IssueClearances FROM Departure
                    DO ADD $a PRECEDES $b; OD;

COORDINATE  $a:Maneuver_toward_Airport FROM Descent,
             $b:Clear_approach FROM Approach
                    DO ADD $a PRECEDES $b; OD;

/*---------------------------------------------------------------*/
/* Overlapping between actors and process phases */
/*---------------------------------------------------------------*/
Passenger, Preflight SHARE ALL BoardAircraft;
Passenger, Landing SHARE ALL Disembark;
Pilot, Preflight SHARE ALL FlightCheck,
                     Pushback,
                     Taxi;
Pilot, Takeoff  SHARE ALL  Hold_in_queue, Liftoff;
Pilot, Departure  SHARE ALL  ChangeFrequency;
Pilot, EnRoute  SHARE ALL  OceanicExtension, Follow_route, ChangeFrequency2;
Pilot, Descent  SHARE ALL  Maneuver_toward_Airport;
Pilot, Approach  SHARE ALL  Hold, Enter_approach_line;
Pilot, Landing  SHARE ALL  Land, Taxi_to_gate;
Controller, Preflight  SHARE ALL  DepartureClearance, IssueGroundInstruction;
Controller, Takeoff  SHARE ALL  Clear_for_takeoff, Handoff_to_TRACON;
Controller, Departure  SHARE ALL  IssueClearances, Handoff_to_ARTCC;
Controller, EnRoute  SHARE ALL  IssueInstruction, Handoff_to_TRACON2;
Controller, Descent  SHARE ALL  Clear_descent;
Controller, Approach  SHARE ALL  Clear_approach, Handoff_to_tower;
Controller, Landing  SHARE ALL  Clear_landing, Taxi_instruction;

2.13 User-defined relations
The purpose of user-defined relation in MP is similar to the association concept in UML [Booch et al. 2000]. Events may have additional binary relations besides of basic IN and PRECEDES. These relations are visualized in the event trace graph and may be used for navigation in the trace. The following is a model of network ring, where each node interacts with its left and right neighbors only.

Example 14. User-defined relations.

SCHEMA Ring_topology
Node: ( * ( send_left | send_right |
    receive_from_left | receive_from_right ) * ));

ROOT Ring: { + Node +}
/* The following coordination is done over the root Ring */
BUILD{
  /* Assuming that a ring with a single Node cannot have a meaningful behavior (a single Node
does not send/receive messages to/from itself), a constraint is set for the whole model */
  ENSURE #Node > 1;

  COORDINATE $x1: Node, /* This thread produces a default set */

  <SHIFT_LEFT> $x2: Node
  /* SHIFT_LEFT shuffling means that after the default set of Node events is selected (all Node
  events from THIS), it takes the first event from the beginning and puts it to the end of
  sequence. This adjusted thread is used by COORDINATE to produce the ring topology by
  adding user-defined relations left_neighbor_of and right_neighbor_of. */
  DO
    /* The COORDINATE loop proceeds to set up new relations between $x1 and its neighbor */
    ADD $x1 left_neighbor_of $x2,
    $x2 right_neighbor_of $x1;
  OD;

  /* We assert the single ring property. ^R yields a transitive
  closure of the relation R. Ring topology means that every node
  can be reached from any node in both directions. Although it is possible to prove
  that the previous coordination sets the ring properly, the following assertion check
  is provided for clarity to demonstrate how such properties can
  be expressed in MP. If uncommented, it will increase the trace generation time.
  Assertions are mostly used for model testing/debugging. */
  CHECK FOREACH $a: Node, $b: Node
    ( $a ^left_neighbor_of $b
      AND
      $b ^right_neighbor_of $b )
  ONFAIL SAY("ring property violation detected!");

  /* After the ring topology has been created, we coordinate message exchange between
  neighbors, i.e. define the behavior of the ring */
  COORDINATE $m: Node /* Default set, do the following for each Node */
  DO COORDINATE $left: Node
    SUCH THAT $left left_neighbor_of $m,
    DO
      COORDINATE $s1: send_left FROM $m,
      $r1: receive_from_right FROM $left
      DO ADD $s1 PRECEDES $r1; OD;
      COORDINATE $s1: send_right FROM $left,
      $r1: receive_from_left FROM $m
      DO ADD $s1 PRECEDES $r1; OD;
    OD;
  OD;
}; /* end of the BUILD block for the root Ring */
Fig. 14. Event trace for **Ring_topology** schema, scope 2.

Here is yet another example of user-defined relation. Example 5 provides a model of stack behavior. But the relationship between pop and push events is not clear – it is not well visualized. By using additional user-defined relations it becomes possible to clearly show that relation. Similar visualization is amenable for many other models, like queue behavior.

**Example 15.**  *Stack behavior with pop/push relationship visualized.*

**SCHEMA Stack_behavior**

```plaintext
ROOT Stack: (* ( push | pop ) *)
BUILD { /* if an element is retrieved, it should have been stored before */
    ENSURE FOREACH $x: pop FROM Stack
    (#pop BEFORE $x < #push BEFORE $x);

    /* The following composition operation establishes push/pop pairs in the Stack behavior. For each 'pop' the closest in time preceding 'push' is selected, unless it has been already consumed by another 'pop' */
    COORDINATE $get: pop
    DO COORDINATE <REVERSE> $put: push SUCH THAT $put BEFORE $get
        DO IF NOT ( EXISTS $g: pop $put Paired_with $g OR
                      EXISTS $p: push $p Paired_with $get )
            THEN ADD $put Paired_with $get; FI;
    OD;
    OD;
} /* end BUILD for Stack */
```

Example 16. **Queue behavior.**
Queue implements the “first in, first out” behavior for storing/retrieving elements. It is assumed that initially Queue is empty. The following schema specifies behavior of a queue in terms of enqueue and dequeue operations.

```plaintext
SCHEMA Queue_behavior

ROOT Queue: (* ( enqueue | dequeue ) *)
BUILD {
  /* If an element is retrieved, it should have been stored. This constraint is similar to Stack behavior */
  ENSURE FOREACH $x: dequeue
  ( #dequeue BEFORE $x < #enqueue BEFORE $x );

  /* The following composition operation establishes enqueue/dequeue pairs in the Queue behavior. For each 'dequeue' the earliest in time 'enqueue' is selected, unless it has been already consumed by another 'dequeue' */

  COORDINATE $get: dequeue
  DO COORDINATE $put: enqueue
    SUCH THAT $put BEFORE $get
    DO IF NOT ( EXISTS $g: dequeue $put Paired_with $g OR EXISTS $p: enqueue $p Paired_with $get )
    THEN ADD $put Paired_with $get; FI;
  OD;
} ;
```

---

Fig. 15. Example of Stack behavior, trace 17 for scope 5

---
2.14 Event attributes

Event attribute defines a binary relation between an event instance and a value. It is a functional relation, meaning that any event instance may have only one attribute value associated with it at any time moment. But an event instance may have several different attributes associated with it.

An attribute has a name and a type. The following are available attribute types:
- **number** – numerical values, like 3, -12, 2.5, 6.77E2 (both integer and float numbers as in common programming languages, like C or Java)
- **interval** – represents an interval of number values (see Sec. 2.14.1)
- **bool** – logical value true or false

There are pre-defined operations available for attribute type values.
- Arithmetic operations +, -, *, / for **number** and **interval**.
- Arithmetic operations max(n1, n2) and min(n1, n2) for **number**.
- Logic operations NOT, AND, OR for **bool**.
- Comparison operations >, >=, <, <= for **number** and **interval**.
- Comparison operations ==, != for all types.
- Assignment operation := for all types,
- Compound assignment operations +:=, -=, *=, /= for **number** and **interval**,
  - max:=, min:= for **number**,
  - AND:=, OR:= for **bool**.

In the case of division by 0, like 4/0, the result is ‘nan’.

In the assignment operation type of the attribute on the left-hand side and type of the right-hand value should be compatible. Interval type attribute can be assigned a number value. In this case the number N is extended into interval [N..N].

Compound assignment **E.attr op:= val** is an abbreviation for **E.attr := E.attr op val**, where **op** is a binary operation. Type compatibility requirements are the same as for plain assignment. Arithmetic operation requires **number** or **interval** type, logic operations requires **bool**.
Attribute of each type has pre-defined default value:

- **number** has value 0
- **interval** has value [0..0]
- **bool** has value `false`

Attributes are defined within ATTRIBUTES clause, like

```
ATTRIBUTES{ number a1, a2; interval a3; bool a4; }
```

Attribute definition is global, if an attribute has been defined, it may be associated with any event instance in the trace. Event attribute's value may be modified using attribute assignment operations during the derivation. Definition of attribute should precede its use in an expression or assignment operation.

Attribute

**number trace_id**

is pre-defined and cannot be re-assigned. If used in the BUILD block of a root or composite event, it yields 0 (since the trace does not yet exist when the BUILD block is executed during the derivation). When used in an expression outside of BUILD block, it yields the trace's id – a unique number of trace instance under the derivation (these numbers are shown also in the trace scroll bar at the right side of Firebird’s window).

Although the event trace structure is specified in MP using event grammar rules, and event attributes are associated with the event grammar’s symbols (atomic, composite, and root events) the concept of event attribute in MP is not exactly the concept of attribute grammar in traditional syntax analysis [Knuth, 1966]. The main reason for that is to integrate event attribute calculations into specifics of MP derivation process. Event attributes in MP model are used in IF conditions, ENSURE, and SAY clauses.

**Example 17. Example of attribute use.**

**SCHEMA Shopping_Spree**

**ROOT Buyer:** (*<0..2 * $$scope> Buy *); /* Increase the iteration to ensure enough Buy events to coordinate with two Shops */

```
ATTRIBUTES{number price, total_cost;};
```

**ROOT Shop_A:** (* (Sell_Item_1 | Sell_Item_2) *)
/* Set attribute values in the context of composite event where these are used */

```
BUILD{ COORDINATE $s: Sell_Item_1 DO $s.price:= 8; OD;
          COORDINATE $s: Sell_Item_2 DO $s.price:= 20; OD;
};
```

**ROOT Shop_B:** (* (Sell_Item_2 | Sell_Item_3) *)
/* In Shop_B the cost of Item_2 may be different */

```
BUILD{ COORDINATE $s: Sell_Item_2 DO $s.price:= 22; OD;
          COORDINATE $s: Sell_Item_3 DO $s.price:= 30; OD;
};
```

/* Schema’s attribute total_cost is automatically initialized by 0.0.
Absence of event reference means THIS by default */

```
COORDINATE $sell: (Sell_Item_1 | Sell_Item_2 | Sell_Item_3),
```
$buy:  Buy
    DO /* to render the dependency between sell and buy events */
    ADD $buy PRECEDES $sell;
    OD;

/* example of pre-defined attribute use */
SAY(" Trace #" trace_id);

/* use of aggregate operation(76) to calculate total_cost as an attribute of the whole schema */
total_cost := SUM{ $item: (Sell_Item_1 | Sell_Item_2 | Sell_Item_3)
    APPLY $item.price};

SAY("Bought " #Buy " items. Total purchase cost is: " total_cost
    " average cost: " total_cost/#Buy);

23  = Trace #23

  = Buyer
  = Shop A
  = Shop B

  = Buy
  = Sell Item 2
  = Sell Item 3

  = Bought 3 items.
  = Total purchase cost is: 72 average cost: 24

Fig. 17. Shopping_Spree example, trace 23, scope 2, on Gryphon

Yet another example of event attribute use for modeling Bayesian logic is provided in Sec. 2.18.2.

An event occupies a time interval with beginning and end, and may have non-zero duration. For some models it may be of interest to consider attribute values separately at the beginning and at the end of event. The solution is straightforward – to introduce separate attributes, like value_of_A_at_beginning and value_of_A_at_end and to maintain them accordingly.

### 2.14.1 Interval attribute type

In many cases the precise single value for a number attribute cannot be determined. Rather there is some known interval of values for the attribute. Interval temporal logics and interval algebras have been explored by many Computer Science researchers, for instance in [Allen 1983]. The interval attribute type is defined by a pair of numbers and represents the whole set of values between these two limits. Interval value is specified as [value1 .. value2], where value1 and value2 may be any expressions of the type number. Assignment operation is defined for intervals as well. If attribute A of event E is of type interval, it can be assigned value by an assignment operation

\[ E.A := [v1 .. v2]; \]

There are predefined parameter values for interval type.

- \( A.\text{smallest} \)  \( = \) \( \text{min}(v1, v2) \)
- \( A.\text{largest} \)  \( = \) \( \text{max}(v1, v2) \)
\[ A.\text{len} \quad = \quad A.\text{largest} \ - \ A.\text{smallest} \]

The ordering of \( v_1 \) and \( v_2 \) does not matter, so \([v_1..v_2]\) defines the same interval as \([v_2..v_1]\).

If \( v_1 = v_2 \) the interval contains just a single value, and \( \text{len} = 0 \).

Interval attribute can be assigned a single number, like
\[ E.A := N; \]

This is equivalent to the assignment
\[ E.A := [N..N]; \]

Each interval attribute has a pre-defined initial value of \([0..0]\).

The following operations support basic computations with intervals. Suppose that \( I_1 \) and \( I_2 \) are interval values.

**Arithmetic operations +, -, *, /**

\[
\begin{align*}
I_1 + I_2 & = [I_1.\text{smallest} + I_2.\text{smallest} .. I_1.\text{largest} + I_2.\text{largest}] \\
I_1 - I_2 & = [I_1.\text{smallest} - I_2.\text{largest} .. I_1.\text{largest} - I_2.\text{smallest}] \\
I_1 * I_2 & = [I_1.\text{smallest} * I_2.\text{smallest} .. I_1.\text{largest} * I_2.\text{largest}] \\
I_1 / I_2 & = [I_1.\text{smallest} / I_2.\text{largest} .. I_1.\text{largest} / I_2.\text{smallest}] 
\end{align*}
\]

**Compound assignment operations +:=, -=, *=, /=**

**Comparison operations ==, !=, >, >=, <=**

\[
\begin{align*}
(I_1 == I_2) & = \\
& \quad I_1.\text{smallest} = I_2.\text{smallest} \text{ AND } I_1.\text{largest} = I_2.\text{largest} \\
(I_1 != I_2) & = \\
& \quad I_1.\text{smallest} \neq I_2.\text{smallest} \text{ OR } I_1.\text{largest} \neq I_2.\text{largest} \\
(I_1 < I_2) & = \\
& \quad I_1.\text{smallest} > I_2.\text{largest} \\
(I_1 > I_2) & = \\
& \quad I_1.\text{smallest} > I_2.\text{smallest} \text{ OR } I_1.\text{largest} \geq I_2.\text{largest} \\
(I_1 <= I_2) & = \\
& \quad I_1.\text{smallest} \geq I_2.\text{smallest} \text{ OR } I_1.\text{largest} = I_2.\text{largest} \\
(I_1 <= I_2) & = \\
& \quad I_1.\text{smallest} \geq I_2.\text{smallest} \text{ OR } I_1.\text{largest} \leq I_2.\text{largest} \\
\end{align*}
\]

Arguments of arithmetic or comparison operations may be interval or number values, like in 
\([2..6] < 10\)

In such cases the stand alone number value \( N \) is converted into interval \([N..N]\).

**Example 18. Use of interval attributes.**

The task is to load a backpack with items not exceeding the total weight of 30 units. There may be several instances of the same item.

**SCHEMA backpack**

**ATTRIBUTES{ interval weight; };**

/* Composite events may be used to define attribute values for all instances of a particular event */

\[
\begin{align*}
\text{item1:} & \quad \text{BUILD}\{ \text{weight} := [2..5]; \}\}; \\
\text{item2:} & \quad \text{BUILD}\{ \text{weight} := [10..16]; \}\};
\end{align*}
\]

**ROOT Backpack: (+ (item1 | item2) +)**
BUILD{ COORDINATE $item: (item1 | item2)
  DO  SAY("selected " $item " with weight " $item.weight);
     /* interval weight is initialized by [0..0] */
     weight +:= $item.weight;
  OD;
  /* ensure that the backpack’s weight does not exceed maximum */
  ENSURE weight > 0 AND weight <= 30; }

SAY("selected " #(item1 | item2) " items");
SAY("backpack weight is within interval " Backpack.weight);

Fig. 18. Backpack example, scope 3, trace 8.

2.14.2 Timing attributes

Many logics and modeling frameworks have been designed to specify and reason about real-time system behaviors [Formal Methods for Real-Time Computing, 1996]. These include temporal logics, finite state machines and process algebras for behavior specification and verification.

In MP this aspect is supported by timing attributes associated with events. That approach provides means to specify and verify some timing properties of system behavior and to answer some queries about timing in the event traces. As demonstrated in Sec. 2.16, MP may also model finite state machines and temporal logic formulas.

The concept of event is the main MP abstraction for any activity. An event occupies a time interval with beginning and end, and may have non-zero duration. Pre-defined MP event timing attributes handle time. Time in MP is measured in abstract time units. The following timing attributes are pre-defined for each event, including the whole event trace execution.

**interval start, end, duration;**

As any **interval** attribute, timing attribute has initial value [0..0].

Following constraints hold for each event E. Recall that comparison operations for intervals allow
also stand alone numbers as arguments (Sec. 2.14.1).

\[
\begin{align*}
E.\text{start} & \geq 0 \\
E.\text{end} & \geq 0 \\
E.\text{duration} & \geq 0 \\
E.\text{end} & = E.\text{start} + E.\text{duration}
\end{align*}
\]

\[
\begin{align*}
E.\text{start}\text{.smallest} & \leq E.\text{end}\text{.smallest} \\
E.\text{start}\text{.largest} & \leq E.\text{end}\text{.largest}
\end{align*}
\]

For each pair of events \(E_1\) and \(E_2\) the following holds.

If \(E_1\) BEFORE \(E_2\) then
\[
E_1.\text{end}\text{.largest} \leq E_2.\text{start}\text{.smallest}
\]
or equivalently
\[
E_1.\text{end} \leq E_2.\text{start}
\]

If \(E_2\) FROM \(E_1\) then
\[
E_1.\text{start}\text{.smallest} \leq E_2.\text{start}\text{.smallest}
\]
\[
E_2.\text{end}\text{.largest} \leq E_1.\text{end}\text{.largest}
\]

During the trace derivation value assigned to the timing attribute may be changed by the automated recalculation depending on the basic relations IN and PRECEDES in which the event participates. The values of start, end and duration attributes may be automatically adjusted only forward (i.e. may only increase) as a result of the automated recalculation. See the details of timing attribute calculation algorithm in Sec. 6.3.

The value of timing attribute can be re-assigned by SET assignment operation(46). This operation is different from the attribute_assignment(45). During the derivation process assignment operation SET \(E.\text{attribute AT LEAST A}\) is performed as:
\[
E.\text{attribute} := [\text{max}(A.\text{smallest}, E.\text{attribute}\text{.smallest}) .. \text{max}(A.\text{largest}, E.\text{attribute}\text{.largest})]
\]

See also Example 19 and Example 20 below.

Since the value of event timing attribute depends on the basic relations, it makes sense to use these values in conditions, ENSURE, and SAY clauses only after the trace derivation is complete and no more basic relation adjustments are expected, i.e. no more new root segments and ADD operations are expected in the derivation. Of course, if needed user may define and maintain their own timing attributes based on the pre-defined attributes or calculated independently.

**Example 19. Timing attribute use.**

It is assumed that each of the Visitor events contributes to the work delay. For timing attributes in this example automated attribute recalculation is sufficient.

This example also demonstrates how to render a set of events and their attributes, if needed.

**SCHEMA Delays**

**ROOT** Worker:
\[
(+ ( \text{Work} \mid \text{Get_distracted}) +);
\]
Work:
BUILD{
  SET duration AT LEAST 1;
  /* The default value for the Work event duration attribute is set as an interval [1..1] */
};

Get_distracted: (+ Visitor +);

Visitor:
  /* set default duration for Visitor */
BUILD{ SET duration AT LEAST [2..3];};

ATTRIBUTES{ interval total_delay; }

/* In the following operation total_delay is THIS.total_delay, or attribute of the whole schema event, 
and is used to hold specific data extracted from the trace */
COORDINATE $distraction: Get_distracted
  DO total_delay += $distraction.duration; OD;

/* show all events present in the trace with timing attributes, preserving the order of derivation, 
top-down, left-to-right */
COORDINATE $x: $$EVENT
DO SAY( $x " start " $x.start 
  " duration " $x.duration 
  " end " $x.end );
OD;

SAY(" Delay created by " #Visitor " visitors is " total_delay);
The system operates a gate at a railroad crossing. A sensor determines when each train enters and exits the crossing’s region. The main safety requirement: if a train is in the crossing region, then gate must be down.

SCHEMA Railroad_Crossing

ROOT Train:
( + Enter_Restricted_Region
   Goes_through_Restricted_Region
   Enter_Crossing_Region
   Goes_through_Crossing_Region
   Leaves_Crossing_Region
 + )
;

ROOT Crossing:
Up
( + Move_Down
   Down
   Move_Up
   Up
 + )
;

/* Set the default values for duration attributes */
Goes_through_Restricted_Region:
BUILD{ SET duration AT LEAST [10..12]; };

Move_Down:
BUILD{ SET duration AT LEAST [5..8]; };

COORDINATE $enter: Enter_Restricted_Region FROM Train,
   $down: Move_Down FROM Crossing
DO ADD $enter PRECEDES $down; OD;

COORDINATE $enter: Enter_Crossing_Region FROM Train,
   $down: Down FROM Crossing
DO ADD $down PRECEDES $enter; OD;

COORDINATE $leaves: Leaves_Crossing_Region FROM Train,
   $up: Move_Up FROM Crossing
DO ADD $leaves PRECEDES $up; OD;

/* The main safety condition */
CHECK FOREACH $e: Goes_through_Crossing_Region FROM Train
   NOT EXISTS $up: Up FROM Crossing
      MAY_OVERLAP $e $up
ONFAIL SAY("Crossing is open when train is passing through!");

COORDINATE
   $goes_through: Goes_through_Restricted_Region FROM Train,
$\text{down}: \text{Move\_Down}$

```plaintext
FROM Crossing
DO
  SAY("Train enters crossing at " $\text{goes\_through}\$.start);
  IF $\text{goes\_through}.duration < $\text{down}.duration$ THEN
    SAY("Not enough time to close crossing");
  ELSE SAY("Crossing was successfully closed");
  FI;
OD;
```

In this example there always will be enough time for the crossing bar to go down before the train enters crossing region.

Fig. 20. Trace example for Railroad\_Crossing.

2.15 Avoiding deadlocks in threads – Dining Philosophers

Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. It was originally formulated in 1965 by Edsger Dijkstra.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by one philosopher only and so a philosopher can use the fork only if it's not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.

Eating is not limited by the amount of spaghetti left: assume an infinite supply. The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve; i.e., can forever continue to alternate between eating and thinking assuming that any philosopher cannot know when others may want to eat or think.


This MP model provides abstract specification of all correct behaviors, but does not provide implementation details. Since MP supports automated use case extraction, this model may be used as a source of test cases for testing the implementation.
Example 21. Dining Philosophers.

SCHEMA Dining_Philosophers

ROOT Philosophers: {
  +<$$scope> Philosopher +;}
Philosopher: (+ <1> think eat +)
/* Philosopher eats at least once, but is limited to reduce the number of traces */;

eat: { left_fork, right_fork };

ROOT Forks: { +<$$scope> Fork +;}

Fork: (* <2> (left_fork | right_fork) *)
/* limit the number of fork use to reduce the number of traces, this in fact limits each Philosopher to
one "eat" event */
/* Each fork is used at least once by each neighbor */
BUILD{ ENSURE #left_fork > 0 AND #right_fork > 0;};

/* Assign forks to the Philosophers. Philosophers $p1 and $p2 share fork $f.
Shift_left resuffling takes the first event from the beginning and puts it to the end of sequence.
This adjusted thread is used to produce the ring topology, and to assign two neighbors $p1 and
$p2 for sharing fork $f */

COORDINATE $p1: Philosopher,
      <SHIFT_LEFT> $p2: Philosopher,
      $f: Fork
DO

/* Default sequence is adequate here:
the order of fork use in Philosopher and in Fork are the same. */
$p1, $f SHARE ALL right_fork;
$p2, $f SHARE ALL left_fork;
OD;

/* to avoid deadlocks 'eat' events competing for resources are put in order */
COORDINATE $e1: eat
DO COORDINATE $e2: eat
  DO
    IF EXISTS $f1: (left_fork | right_fork) FROM $e1,
    $f2: (left_fork | right_fork) FROM $e2
    ($f1 PRECEDES $f2)
    THEN
      ADD $e1 PRECEDES $e2;
    FI;
  OD;
OD;

/* Ensure that both forks can be used simultaneously */
COORDINATE $e: eat
DO COORDINATE $l: left_fork FROM $e,
    $r: right_fork FROM $e
2.16 Modeling Finite State Machines

Finite state transition diagrams are a popular technique for modeling system behaviors. The following example provides a template that can be applied to many similar finite state diagram models. The main ideas are:

- The modeling of the state diagram includes state visits and actions triggering state transitions. Only paths starting with Start and ending with End are valid.
- Behavior of each state $S_i$ is represented by a separate root event.
- Each transition in the diagram $S_i \rightarrow S_j$ is modeled by a composite event $S_i_{\text{to}}S_j$.
- Event $S_i_{\text{to}}S_j$ contains the symbol and/or action triggering the transition.
- Events $S_i_{\text{to}}S_j$ are shared by corresponding state behavior roots.

The following MP schema models behavior of a finite state diagram with loops and non-deterministic transitions. In addition, it contains MP code for visualizing event traces as paths in the diagram (views of the event traces).

---

Example 22. Modeling finite state machine.

```mp
SCHEMA Finite_State_Diagram
ROOT S1_behavior:  Start S1
  (* (S1_to_S3 | S1_to_S4) S2_to_S1 S1 *)
  (S1_to_S3 | S1_to_S4);
S1_to_S3: a;
S1_to_S4: (a | b);

ROOT S2_behavior:  (* S4_to_S2 S2 S2_to_S1 *)
  S4_to_S2 S2 End;
S2_to_S1: b;

/* Coordination should be done as soon as possible, in order to prune the search,
but only after both root S1_behavior and S2_behavior traces are available */
S1_behavior, S2_behavior SHARE ALL S2_to_S1;

ROOT S3_behavior:  (* S1_to_S3 S3
  (* S3_to_S3 S3 *))
```
S3_to_S4 *

S3_to_S4: a;
S3_to_S4: b;

S3_behavior, S1_behavior SHARE ALL S1_to_S3;

ROOT S4_behavior: (* (S1_to_S4 | S3_to_S4) S4 S4_to_S2 *);
S4_to_S2: a;

S4_behavior, S1_behavior SHARE ALL S1_to_S4;
S4_behavior, S2_behavior SHARE ALL S4_to_S2;
S4_behavior, S3_behavior SHARE ALL S3_to_S4;

/* The Path root serves as a container for the path sequence. */
ROOT Path:;

/* First, assemble all state visits and transition events into the Path */
COORDINATE $a: (S1 | S2 | S3 | S4 | a | b | Start | End)
DO ADD $a IN Path; OD;

/* The reshuffling on two instances of the same sequence is needed to insert PRECEDES between the
adjacent events in the path – states and transition symbols */
COORDINATE <SORT CUT_END> $a: $$$EVENT SUCH THAT $a IN Path,
<SORT CUT_FRONT> $b: $$$EVENT SUCH THAT $b IN Path
DO ADD $a PRECEDES $b; OD;

Fig. 21. Event trace #4 for scope 2.

After collapsing S1_behavior root events, hiding them and the Path root, and reshuffling the
remaining events in the trace, the following clear view of the path in the diagram can be obtained.
2.16.1 Modeling Model Checker and temporal logic formulae

Model checking tools usually involve the use of Kripke structures (finite state diagrams) as models for system behaviors and temporal logic formulae to specify properties of behavior. MP can provide yet another way of modeling a finite state diagram and an assertion about the behavior simulating CTL formula.

Set of paths in a finite state diagram can be specified using a regular expression. The algorithm for converting a finite state diagram into a regular expression is well known. The idea is to convert the original diagram into a generalized nondeterministic finite automaton, where the transitions are marked with regular expressions describing all possible paths from one state to another [Sipser 1996, pp. 64 – 70]. The algorithm works by eliminating states one by one replacing the transitions with regular expressions until only Start and End states remain. The transition between these states is marked with a regular expression specifying all possible paths on the original diagram from Start to End.

The following microwave example has been adapted from [Clarke et al. 1999, pp. 38 – 39]. The finite state diagram modeling the microwave behavior has been converted into a set of regular expressions including the commands performed on the microwave, like close_door, start_cooking, and states S1, S2, ... Each state has a set of atomic propositions assumed to be true within the state. Atomic propositions are represented as inner events. Absence of atomic proposition implies its negation.

The finite state diagram modeling microwave behavior (Kripke structure) can be described as a set of the following transitions. Execution path always starts and ends in the state S1.

\[
\begin{align*}
S1 & \rightarrow \text{start\_oven} \rightarrow S2 \\
S1 & \rightarrow \text{close\_door} \rightarrow S3 \\
S2 & \rightarrow \text{close\_door} \rightarrow S5 \\
S3 & \rightarrow \text{open\_door} \rightarrow S1 \\
S3 & \rightarrow \text{start\_oven} \rightarrow S6 \\
S4 & \rightarrow \text{open\_door} \rightarrow S1 \\
S4 & \rightarrow \text{done} \rightarrow S3 \\
S4 & \rightarrow \text{cook} \rightarrow S4 \\
S5 & \rightarrow \text{open\_door} \rightarrow S2 \\
S5 & \rightarrow \text{reset} \rightarrow S3 \\
S6 & \rightarrow \text{warm\_up} \rightarrow S7 \\
S7 & \rightarrow \text{start\_cooking} \rightarrow S4
\end{align*}
\]
The states have following atomic propositions associated with them. The ‘~’ symbol denotes negation of the atomic proposition.

\[ \begin{align*} 
S1: & \sim\text{Start}, \sim\text{Close}, \sim\text{Heat}, \sim\text{Error} \\
S2: & \text{Start}, \sim\text{Close}, \sim\text{Heat}, \text{Error} \\
S3: & \sim\text{Start}, \text{Close}, \sim\text{Heat}, \sim\text{Error} \\
S4: & \sim\text{Start}, \text{Close}, \text{Heat}, \sim\text{Error} \\
S5: & \text{Start}, \text{Close}, \sim\text{Heat}, \text{Error} \\
S6: & \text{Start}, \text{Close}, \sim\text{Heat}, \sim\text{Error} \\
S7: & \text{Start}, \text{Close}, \text{Heat}, \sim\text{Error} 
\end{align*} \]

This behavior should be verified with respect to CTL (Computation Tree Logic) formula
\[ \text{AG} (\text{Start} \rightarrow \text{AF Heat}) \]
meaning that for each path in the diagram and for each state on this path, if the atomic proposition Start is true in this state, then for each path from that state somewhere in future should be reachable a state where proposition Heat is true.

Example 23. Microwave oven. Modeling model checking.

```
SCHEMA microwave_oven
ROOT Microwave: S1 (* R7 S1 *);
/* Regular expressions, or paths, including state visits and actions triggering state transitions */
/* Eliminating S7, paths from S6 to S4 via S7 */
R1: warm_up S7 start_cooking;
/* eliminating S6, paths from S3 to S4 via S6 */
R2: start_oven S6 R1;
/* eliminating S5, paths from S2 to S3 via S5 */
R3: close_door S5 reset;
/* eliminating S5, paths from S2 to S2 via S5 */
R4: close_door S5 open_door;
/* eliminating S2, paths from S1 to S3 via S2 */
R5: start_oven S2 (* R4 S2 *) R3;
/* eliminating S4, paths from S3 to S1 via S4 */
R6: R2 S4 (* cook S4 *) open_door;
/* eliminating S3, paths from S1 to S1 via S3 */
R7: (close_door | R5) S3
   (* R2 (* S4 cook *) S4 done S3 *)
   (open_door | R6);
/* States with atomic propositions represented as inner events, absence of atomic proposition implies its negation */
S1: /* ~Start, ~Close, ~Heat, ~Error */
S2: {Start, Error}; /* Start, ~Close, ~Heat, Error */
S3: Close; /* ~Start, Close, ~Heat, ~Error */
S4: {Close, Heat}; /* ~Start, Close, Heat, ~Error */
```
S5: {Start, Close, Error}; /* Start, Close, ~Heat, Error */
S6: {Start, Close}; /* Start, Close, ~Heat, ~Error */
S7: {Start, Close, Heat}; /* Start, Close, Heat, ~Error */

/* MP assertion to check CTL formula AG(Start -> AF Heat)
=====================================*/
CHECK (FOREACH $s: Start EXISTS $h: Heat $h AFTER $s)
  ONFAIL SAY("no Heat after Start detected");

/************** create Path view ******************/
ROOT Path;
/* The Path root serves as a container for the path sequence */
COORDINATE  $a: (S1 | S2 | S3 | S4 | S5 | S6 | S7 |
  warm_up | start_cooking | start_oven |
  close_door | reset | open_door | cook | done)
  DO ADD $a IN Path;
OD;

/* States and transitions are derived top-down and left-to-right, so the default selection preserves the
order, we just need to add PRECEDES to render that order explicitly.
The reshuffling on two instances of the same sequence is needed to add PRECEDES between the
adjacent events */
COORDINATE  <CUT_END> $a: $$EVENT SUCH THAT $a IN Path,
  <CUT_FRONT>  $b: $$EVENT SUCH THAT $b IN Path
  DO ADD $a PRECEDES $b; OD;

Counterexamples for the assertion can be obtained even for scope 1 (28 traces generated, 2 MARKed by the assertion violation).
2.16.2 Modeling Petri net in MP

The template for modeling a Petri net in MP is similar to the other finite state machines. Behaviors of places and transitions are modeled by root events, where each transition is represented by a pair of events sends_to and receives_from containing events send_token and get_token, correspondingly. The BUILD block for each place behavior ensures that number of send tokens does not exceed the number of received tokens – the main condition of the Petri net behavior.

The following example has been adapted from Wikipedia (https://en.wikipedia.org/wiki/Petri_net).

Example 24. Modeling Petri net.

SCHEMA Petri_Net
ROOT P1_behavior: get_token (+ (* P1_receives_from_T2 *)
                      P1_sends_to_T1
                     +)
BUILD{ ENSURE FOREACH $s$: P1_sends_to_T1
  #get_token BEFORE $s$ > #send_token BEFORE $s$;};

P1_sends_to_T1: send_token;
P1_receives_from_T2: get_token;

ROOT T1_behavior: (* T1_receives_from_P1 T1_fires
  (T1_sends_to_P2, T1_sends_to_P3) *)
T1_sends_to_P2: send_token;
T1_sends_to_P3: send_token;

COORDINATE $s$: P1_sends_to_T1,
  $r$: T1_receives_from_P1
  DO ADD $s$ PRECEDES $r$; OD;

ROOT P2_behavior: (* (* P2_receives_from_T1 *)
  P2_sends_to_T2 *)
BUILD{ ENSURE FOREACH $s$: P2_sends_to_T2
  #get_token BEFORE $s$ > #send_token BEFORE $s$;};
P2_receives_from_T1: get_token;
P2_sends_to_T2: send_token;

COORDINATE $s$: T1_sends_to_P2,
  $r$: P2_receives_from_T1
  DO ADD $s$ PRECEDES $r$; OD;

ROOT P3_behavior: (* (* P3_receives_from_T1 *)
  P3_sends_to_T2 *)
BUILD{ ENSURE FOREACH $s$: P3_sends_to_T2
  #get_token BEFORE $s$ > #send_token BEFORE $s$;};
P3_receives_from_T1: get_token;
P3_sends_to_T2: send_token;

COORDINATE $s$: T1_sends_to_P3,
  $r$: P3_receives_from_T1
  DO ADD $s$ PRECEDES $r$; OD;

ROOT T2_behavior: (* {T2_receives_from_P2, T2_receives_from_P3}
  T2_fires
  [{T2_sends_to_P1, T2_sends_to_P4}] *)
/* T2_sends_to events are made optional to terminate the loop from T2 to P1 for a finite scope */
T2_sends_to_P1: send_token;
T2_sends_to_P4: send_token;

COORDINATE $s$: P2_sends_to_T2,
  $r$: T2_receives_from_P2
  DO ADD $s$ PRECEDES $r$; OD;

COORDINATE $s$: P3_sends_to_T2,
  $r$: T2_receives_from_P3
DO ADD $s$ PRECEDES $r$; OD;

COORDINATE $s$: T2_sends_to_P1,
     $r$: P1_receives_from_T2
DO ADD $s$ PRECEDES $r$; OD;

ROOT P4_behavior: get_token (* P4_receives_from_T2 *);
     P4_receives_from_T2: get_token;

COORDINATE $s$: T2_sends_to_P4,
     $r$: P4_receives_from_T2
DO ADD $s$ PRECEDES $r$; OD;

/* report number of tokens available in the places */
COORDINATE $b$: ( P1_behavior | P2_behavior |
     P3_behavior | P4_behavior )
DO SAY("At the end of simulation place "$b" has "
     #get_token FROM $b - #send_token FROM $b " tokens");
     OD;

Fig. 25. Event trace for Petri_Net schema, scope 1, after some box reshuffling.

2.17 Coordinating asynchronous threads and tracking dependency chains

Sometimes we need to distinguish chains of events involved into processing of a particular item. User-defined relations can be useful to connect chains of related events. The following example demonstrates how to emphasize events involved into processing of a particular order. User-defined relation related_to connects events involved into a single order.

In addition, user-defined relation Is_supplier_for establishes a connection between Supplier event that delivers the order and Consumer event that receives the order.

Example 25. Coordinating asynchronous threads and tracking event dependency.
User-defined relation related_to connects events within each single Request_delivery -> Consume chain.

SCHEMA Suppliers

ROOT Consumers: {+ Consumer +};
Consumer: (* Request_delivery
    ( Receive_supply | Not_available ) *) ;

/* Supply_Office communicates asynchronously with Producers and Suppliers */
ROOT Supply_Office: (*$scope * $scope> /* Supply_Office serves several Consumers, hence the iteration should be increased */
    Receive_order_and_buy *);

COORDINATE <!> $request: Request_delivery FROM Consumers,
    $receive_request: Receive_order_and_buy FROM Supply_Office
    DO ADD $request PRECEDES $receive_request;
    ADD $request related_to $receive_request;
    OD;

ROOT Producers: {+ Producer +};
Producer: (* ( Sell | Not_available ) *) ;

COORDINATE $receive_order: Receive_order_and_buy FROM Supply_Office,
    /* Receive_order_and_buy and Sell/Not_available events may be asynchronous */
    <!> $sell: (Sell | Not_available) FROM Producers
    DO ADD $receive_order PRECEDES $sell;
    ADD $receive_order related_to $sell;
    OD;

COORDINATE $sell: Sell FROM Producers,
    /* Sell and Receive_supply may be asynchronous */
    <!> $receive: Receive_supply FROM Consumers
    DO ADD $sell PRECEDES $receive;
    ADD $sell related_to $receive;
    OD;

ENSURE FOREACH $request: Request_delivery,
    $receive: Receive_supply
    ( $request ^related_to $receive -> $request BEFORE $receive);

ENSURE /* preventing mismatched delivery */
    NOT EXISTS DISJ $c1: Consumer, $c2: Consumer
    EXISTS $request: Request_delivery FROM $c1,
    $receive: Receive_supply FROM $c2,
    $order: Receive_order_and_buy FROM Supply_Office
    ( $request ^related_to $order AND
COORDINATE <!>$refusal1: Not_available FROM Producers,
  $refusal2: Not_available FROM Consumers
DO SHARE $refusal1 $refusal2; OD;

ENSURE /* preventing mismatched refusals */
  NOT EXISTS DISJ $c1: Consumer, $c2: Consumer
  EXISTS $request: Request_delivery FROM $c1,
           $receive: Not_available FROM $c2,
           $order: Receive_order_and_buy FROM Supply_Office
           ($request ^related_to $order AND
            $order ^related_to $receive );

/**** add yet another relation for Supplier/Consumer ****/
COORDINATE $c: Consumer
  DO COORDINATE $p: Producer
    DO IF EXISTS $request: Request_delivery FROM $c,
        $sell: Sell FROM $p
        ( $request ^related_to $sell )
    THEN ADD $p Is_supplier_for $c; FI;
  OD;
OD;

Run for scope 2 yields 213 traces, in approx. time 19 sec.

Fig. 26. Event trace for Suppliers schema.

2.18 Probabilistic models in MP
Using probabilities for systems behavior modeling and simulation is pretty common, and there are
many techniques and tools designed for such purpose. MP can be used for probabilistic modeling in
several different ways.

The following three parts are necessary in any probabilistic model.

1. Well-defined selection space of objects/activities to which we want to assign probabilities. Usually
   it is a finite set.
2. Probability calculation rules. Probability is just a numeric value within the range 0..1 assigned to
an object in the selection space. It may be independent or calculated from data available in the selection space. Bayesian logic (https://en.wikipedia.org/wiki/Bayesian_probability) is an example, and we can apply probability calculation rules using MP constructs IF, COORDINATE, etc. to define values of probability attributes. Calculation of probability for an object may use conditions (dependencies) involving other objects.

3. Probabilities calculated in step 2 are just numbers. We need to define the meaning for them. The statement “probability of event E is P” should have a clear explanation.

Currently there are two types of probabilistic models in the MP domain. Each serves a specific purpose and may be used to answer specific questions.

2.18.1 Probability of event trace

Type 1. Probability of the whole event trace as an element of the set of all valid traces.

The selection space is the set of all valid event traces generated for a given MP scope. This set is always finite.

The rationale for probability calculation rules is in considering the work of MP trace generator as a Monte Carlo process. We can use probability during trace derivation when we encounter either alternative event pattern (have to choose an alternative), or iteration pattern (have to choose the number of iterations). Selection set is determined by the given generation scope and by the probabilities assigned to alternatives in MP event grammar rules.

Each trace under derivation has a probability implied by the alternative selections producing the trace. It is assumed that the derivation process makes independent probabilistic decisions at each alternative point. Not all traces for which we start the derivation will be valid, some will be rejected, and we'll need to prorate the probabilities of valid traces to get the probability sum equal to 1 and to retain the frequency ratio.

Trace probabilities are shown in the trace scroll bar on the right side of the MP GUI window. MP trace generator produces exhaustive set of all valid behaviors for a given scope and renders prorated probability for each scenario in that set. There is no need to run Monte Carlo simulation for a large number of cases. Instead we can just obtain the complete list of valid traces and calculate probability for each. Even if the probability of valid trace is calculated as 0 (because some probabilities in the alternatives were defined as 0), the trace will appear in the valid trace list. But when scope is changed, the number of valid traces and their probabilities also may change.

Typical questions that may be answered using such models are: “what is probability of this particular valid trace?”, “what is probability to get a valid trace satisfying certain property?”.

Example 26. Example of Type 1 probability.

Here is a simple stack behavior model. Valid behaviors don’t permit a scenario when Pop operation is applied to the empty stack. For the example’s purpose we assume probability of Push is higher than probability of Pop. It is assumed also that probabilities to perform 0, 1 or more iterations by the iterative event pattern (* … *) are equal.

```
SCHEMA Stack_behavior
ROOT Stack: (* (<<0.75>> Push | <<0.25>> Pop ) *)
   BUILD { /* If an element is retrieved, it should have been stored before */
```
ENSURE FOREACH \$x\: \text{Pop FROM Stack} \\
( \text{\#Pop BEFORE \$x < \#Push BEFORE \$x}) ; };

Following event traces (both valid and rejected) are produced by MP trace generator for scope 2.

<table>
<thead>
<tr>
<th>Trace #</th>
<th>Trace</th>
<th>Number of iterations</th>
<th>Probability for iteration</th>
<th>Probability of the trace</th>
<th>Valid or rejected</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>empty</td>
<td>0</td>
<td>1/3</td>
<td>1/3</td>
<td>valid</td>
</tr>
<tr>
<td>2</td>
<td>Push</td>
<td>1</td>
<td>1/3</td>
<td>1/3 * 3/4 = 1/4</td>
<td>valid</td>
</tr>
<tr>
<td>3</td>
<td>Pop</td>
<td>1</td>
<td>1/3</td>
<td>1/3 * ¼ = 1/12</td>
<td>rejected</td>
</tr>
<tr>
<td>4</td>
<td>Push Push</td>
<td>2</td>
<td>1/3</td>
<td>1/3 * ¾ * ¾ = 3/16</td>
<td>valid</td>
</tr>
<tr>
<td>5</td>
<td>Push Pop</td>
<td>2</td>
<td>1/3</td>
<td>1/3 * ¾ * ¼ = 1/16</td>
<td>valid</td>
</tr>
<tr>
<td>6</td>
<td>Pop Push</td>
<td>2</td>
<td>1/3</td>
<td>1/3 * ¼ * ¾ = 1/16</td>
<td>rejected</td>
</tr>
<tr>
<td>7</td>
<td>Pop Pop</td>
<td>2</td>
<td>1/3</td>
<td>1/3 * ¼ * ½ = 1/48</td>
<td>rejected</td>
</tr>
</tbody>
</table>

There are 4 valid traces with summary probability \(1/3 + ¼ + 3/16 + 1/16 = 5/6\). Since only valid traces will be produced, to retain the frequency, probabilities of valid traces are normalized.

<table>
<thead>
<tr>
<th>Trace #</th>
<th>Trace</th>
<th>Initial probability of the trace</th>
<th>Normalized probability (Type 1 calculations results)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>empty</td>
<td>1/3</td>
<td>((1/3)/(5/6) = 2/5)</td>
</tr>
<tr>
<td>2</td>
<td>Push</td>
<td>1/3 * ¾ = 1/4</td>
<td>((1/4)/(5/6) = 3/10)</td>
</tr>
<tr>
<td>4</td>
<td>Push Push</td>
<td>1/3 * ¾ * ¾ = 3/16</td>
<td>((3/16)/(5/6) = 9/40)</td>
</tr>
<tr>
<td>5</td>
<td>Push Pop</td>
<td>1/3 * ¾ * ¼ = 1/16</td>
<td>((1/16)/(5/6) = 3/40)</td>
</tr>
</tbody>
</table>

Similar probability calculation rules are applied also to the shared events, when appearing in alternative branches.

Alternative with a single branch and probability, like \((<<p>> A)\), is equivalent to the optional event pattern \([<<p>> A]\).

Example 27. Shared events with probabilities.

SCHEMA example

ROOT R1: (<<0.6>> a | b);
ROOT R2: (<<0.8>> a | c);

R1, R2 SHARE ALL a;

<table>
<thead>
<tr>
<th>Valid segment</th>
<th>Probability of this segment</th>
<th>Prorated trace probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1: a</td>
<td>0.6</td>
<td></td>
</tr>
<tr>
<td>R1: b</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td>R2: a</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>R2: c</td>
<td>0.2</td>
<td></td>
</tr>
<tr>
<td>Schema: R1 R2 a</td>
<td>0.6 * 0.8 = 0.48</td>
<td>0.48 / (0.48 + 0.08) = 6/7 ≈ 0.857143</td>
</tr>
<tr>
<td>Schema: R1 R2 b c</td>
<td>0.4 * 0.2 = 0.08</td>
<td>0.08 / (0.48 + 0.08) = 1/7 ≈ 0.142857</td>
</tr>
</tbody>
</table>
Traces (Schema: R1 R2 a c) and (Schema: R1 R2 b a) are rejected during the derivation because SHARE ALL coordination constraint cannot be satisfied. For remaining valid traces probabilities are normalized to get the sum of probabilities equal 1.

2.18.2 Probability of an event within a trace

Type 2. Probability of an event appearing within a particular event trace.

The concept of dependency should be fundamental in all kinds of reasoning about behaviors, including the probabilistic analysis. If some activities are performed in a certain order, this may affect the way probabilities of these are calculated. The initial rationale is based on the Bayesian logic.

In MP the basic relations between events are the simplest and most easily identifiable forms of dependencies. If we state that A PRECEDES B that means that B depends on A, otherwise there is no reason to impose the ordering between A and B. And since a task depends on its subtasks, B IN A also clearly states a dependency relation. We can build probabilistic dependencies along the dependencies provided in the MP model.

The selection set is a set of all pairs (T, E) for a given MP scope. Here T is a valid event trace and E is an event instance in the T. This set is finite. The object is an event instance in the given event trace. This means that all probability calculations are confined within a single event trace.

This approach is supported by event attributes and attribute evaluation constructs in MP v.4. Probability of an event E is an attribute E.probability of type number and the rules for calculating this attribute’s value are specified using corresponding MP constructs.

When considering an object (T, E) it is assumed that the instance of E is a part of T. The meaning of calculating E.probability = P may be expressed as “the probability for E to appear in T was P”.

Example 21. Example of Type 2 probabilities calculation.
Suppose there are 3 red and 2 green balls in a box. We pick at random three balls, one at a time. The result of the process can be denoted as RRR, RRG, or RGG, representing sets of selected balls.

The following derivation tree shows the sequences of events and probabilities of events for all 7 valid scenarios.

Fig. 28. Derivation tree for Example 21.

Probabilities for resulting event traces are:
- trace containing RRR, $p = \frac{1}{10}$, total 1 trace;
- trace containing RRG, $p = \frac{6}{10}$, total 3 traces;
- trace containing RGG, $p = \frac{3}{10}$, total 3 traces.

The following MP schema demonstrates how to obtain all valid traces and to calculate probabilities for the results.

**Example 28. Event probability calculations.**

**SCHEMA RedGreen**

**ATTRIBUTES**

- number selection_probability;

**ROOT Selection:**

```
(+<3> ( Select_Red | Select_Green ) +)
   ( RRR | RRG | RGG )
```

/* Constraints to shape the valid traces */

```
ENSURE (#RRR == 1 <=> #Select_Red == 3) AND
   (#RRG == 1 <=> #Select_Red == 2) AND
   (#RGG == 1 <=> #Select_Red == 1);
```
/* Attribute calculations are done here */
COORDINATE $res: ( RRR | RRG | RGG )
DO
    /* prepare for further calculations */
    probability := 1;

    COORDINATE $s: ( Select_Red | Select_Green )
    DO
        /* Probability of the result depends on the ball selection order */
        IF $s IS Select_Red THEN
            /* probability to select this ball is: */
            selection_probability *:= (3 - #Select_Red BEFORE $s)/
                                (5 - #( Select_Red | Select_Green ) BEFORE $s);
        ELSE
            selection_probability *:= (2 - #Select_Green BEFORE $s)/
                                (5 - #( Select_Red | Select_Green ) BEFORE $s);
        FI;
    OD;
/* Report the results */
SAY( "Probability of result " $res " was "
    selection_probability );
OD;

Notice this is probability for RRG to appear in a particular trace, the total probability for RRG to appear in all valid traces is 0.6 (see the derivation tree above).

Fig. 29. RedGreen example, trace 2, scope 1, on Gryphon

Type 1 trace probability still is calculated as 1/7 ~ 0.142857, since Type 1 assumes that probability of selecting an alternative is constant for the whole derivation process.

2.19   MP model reuse
Software artifact reuse requires two activities. First, we need to find out the object for reuse, usually stored in some repository. Second, it is very unlikely that the reuse can be accomplished just by copy/paste, in most cases some adjustment may be needed to integrate the reusable object into the
system under development. For instance, in traditional programming languages typical reuse framework is implemented as subroutine library with parameter passing mechanism for adjusting subroutine behavior, or preprocessor with macro commands with parameters, or template library in C++ with corresponding type parameters for adjustments.

In the case of MP typical reuse may involve either another MP model, like architecture template, or part of it, like some particular root events with interactions between them. These items can be stored in separate folders, and since the size of MP code is relatively small, just copied/pasted into the MP model under development. But they still may require adjustment. Typically that is an adjustment of relations between the reused parts and the rest of MP model. Interactions in MP (coordination operations) are separated from the behavior descriptions (event grammar rules), and such adjustment can be done in a declarative fashion by coordinating the reusable MP code and the MP code under development. The following examples demonstrate how it can be done. The map_composition(37) is a convenient abbreviation for coordinating several relations at once.

2.19.1 Compiler’s front end model reuse
The following compiler's front-end architecture model is inspired by [Perry, Wolf 1992], [Shaw, Garlan 1996] and the unforgettable picture of compiler architecture from the “Dragon Book” [Aho, Sethi, Ullman 1986](page 13). The following examples demonstrate component reuse with different interaction patterns and emphasize the advantages of separation between the specification of component behavior and the specification of interactions between components. This is a model of bottom-up parser with lexical analyzer based on regular expression matching. It models the behavior of Lex/Yacc generated compiler's front end. The first model represents an architecture where lexer stores tokens in the intermediary data structure before parser starts to access it, and the second is a model where parser works with the lexer interactively. The Lexer part models the behavior of a typical Lex machine. The behavior of Stack is integrated into Parser’s behavior.

Example 29. Compiler’s front end in batch processing mode.

The Token_recognition event defines Lexer's behavior according to the semantics when a regular expression in each LEX rule is applied independently, and hence no ordering on RegExpr_Match events is imposed. Each RegExpr_Match performs until all finite automata involved in the token recognition enter the Error state. Then the winner is selected and look-ahead characters beyond the recognized lexeme are returned into the input stream by unget_some_characters. The ordering of put_token and get_token events in the Token_list ensures that production of the token set will precede consumption.

SCHEMA compiler1

ROOT Source_code: (* get_characters
    unget_some_characters *);

/**************************** lexical analysis ****************************/
/************************************************************************************/

Token_recognition: get_characters
  {+ RegExpr_match +}
  unget_some_characters
BUILD{ /* only one RegExpr_match succeeds */
    ENSURE #largest_successful_match == 1; }

RegExpr_match: match_string
  [ largest_successful_match
put_token ];

ROOT Lexer: (+ Token_recognition +);

Source_code, Lexer SHARE ALL get_characters, unget_some_characters;

/!*------------------ intermediate token list -----------------*/
/!*================================================================*/
ROOT Token_list:
  (+ put_token +) (+ get_token +)
BUILD{ ENSURE #put_token >= #get_token; }

Token_list, Lexer SHARE ALL put_token;

/!* ---------------- bottom-up parsing ------------------------*/
/!*================================================================*/
ROOT Parser: {Shift_Reduce, Stack}
BUILD{ COORDINATE $s1: Shift_Reduce,
  $s2: Stack
  DO $s1, $s2 SHARE ALL Pop, Push; OD; }

Shift_Reduce:
  Push /* push the start non-terminal symbol */
  get_token
  (+ ( Shift | Reduce Put_node )
  +)
  ( Reduce
    Put_node
    Parsing_complete | Report_syntax_error );

Shift: Push
get_token;

Reduce: (+<1..2> Pop +) /* to tame the combinatorial explosion for nested iterations */
  Push ;

Stack: (*<2 .. 2 * $scope + 3 > (Push | Pop) *)
  /* extended iteration to provide sufficient numbers of Pop and Push events */
BUILD{ ENSURE FOREACH $x: Pop
  #Pop BEFORE $x < #Push BEFORE $x; }

/!*---------- lexer and parser in the batch mode -------------*/
/!*================================================================*/
Token_list, Parser SHARE ALL get_token;
Fig. 30. Example of compiler's front end in batch mode, scope 2, trace #17

Example 30. Compiler’s front end in incremental mode.

The last COORDINATE composition operation provides for an interaction between Lexer and Parser.

SCHEMA compiler2
ROOT Source_code: (* get_characters
             unget_some_characters *);}
/*--------------------------  lexical analysis --------------------------*/
Token_recognition: get_characters
  { + RegExp_match +}
unget_some_characters
BUILD{ /* only one RegExp_match succeeds */
  ENSURE #largest_successful_match == 1; };
RegExp_match: match_string
  [ largest_successful_match
    put_token ];
ROOT Lexer: (+ Token_recognition +);
Source_code, Lexer SHARE ALL get_characters, unget_some_characters;

/* ----------------------- bottom-up parsing -----------------------*/
/*==================================================================*/
ROOT Parser: {Shift_Reduce, Stack}
BUILD{
  COORDINATE $s1: Shift_Reduce,
    $s2: Stack
    DO $s1, $s2 SHARE ALL Pop, Push; OD; };
Shift_Reduce:
  Push /* push the start non-terminal symbol */
  get_token
  (+ (  Shift       |
        Reduce Put_node )
  +)
  (  Reduce
      Put_node
      Parsing_complete |
      Report_syntax_error ) ;
Shift: Push
  get_token ;
Reduce: (+<1..2> Pop +) /* to tame the combinatorial explosion for nested iterations */
  Push ;
Stack: (*<2 .. 2 * $scope + 3 > (Push | Pop) *)  /* extended iteration to provide sufficient numbers of Pop and Push events */
BUILD{ ENSURE FOREACH $x: Pop
     #Pop BEFORE $x < #Push BEFORE $x; };
/*---------------------- lexer and parser in the interactive mode -------*/
/*==================================================================*/
COORDINATE $produce_token: put_token FROM Lexer,
    $consume_token: get_token FROM Parser
    DO ADD $produce_token PRECEDES $consume_token; OD;
Fig. 31. Example of compiler’s front end in interactive mode, scope 2, trace #15.

Another example of MP code reuse can be found in Sec. 2.14.2, Example 19 with the COORDINATE operation creating an event list following the trace derivation order. This reusable MP code snippet can be adjusted to select only events satisfying certain conditions and to show only specific event attributes.

2.19.2 Reuse with the MAP composition operation

Reuse of a behavior defined by a schema may require mapping some activities in one schema onto activities in another, thus defining a composite behavior. The map_composition operation performs event mapping between two instances of behaviors.

Example 31. Reuse of an authentication subsystem.
The root events Data_Base, Requester, and Authentication_Service represent typical authentication system’s behavior. The Requester requests access to services. The
Authentication_Service requests that specific credentials be provided. If the supplied credentials are valid, the system authorizes the Requester to access the services, otherwise it notifies the Requester that the received credentials are invalid and the Requester may re-attempt access up to two more times. At any time the Requester may decide to abandon the access request. Such reusable MP model can be stored in the reusable model folder and brought into a new MP code by copy/paste. Since it is expected that most MP model’s code is short enough, current MP version does not provide any type of Include pre-processor command.

This architecture template is reused in the system under design represented in this model by the User root. The MAP operation merges the behavior of Requester into the behavior of User by making a copy of all relations, basic and user-defined, in which Requester participates to the User event. In addition COORDINATE/ADD operation further synchronizes some events in User and in Requester.

Run for scope 3 or higher to see attempts_exhausted event.

SCHEMA  Reuse_of_a_system

/*============== reused MP code starts here ==============*/
ROOT Data_Base: (* check_ID *);

ROOT Authentication_Service:
    request_ID
    (* creds_invalid  request_ID *)
    
(  creds_invalid
    [ attempts_exhausted ]
    cancel_access_request  |
    
    creds_valid
    access_granted        )

BUILD{ /*---  constraints for this root ---*/
    ENSURE #creds_invalid <= 3; /* no more than 3 attempts to get access */
    ENSURE #attempts_exhausted == 1 <--> #creds_invalid == 3;
    IF #attempts_exhausted > 0 THEN SAY(" attempts exhausted!");FI;
}

request_ID: check_ID;

Authentication_Service, Data_Base SHARE ALL check_ID;

ROOT Requester:
    request_access
    provide_ID
    (* creds_invalid  request_access  provide_ID *)
    
(  creds_valid
    ( access_granted | abandon_access_request)                    |

    creds_invalid
    ( attempts_exhausted | abandon_access_request ) )

62 of 108
Requester, Authentication_Service SHARE ALL
  creds_valid, creds_invalid,
  attempts_exhausted, access_granted;

COORDINATE $a: request_access FROM Requester,
   $b: request_ID FROM Authentication_Service,
   $c: provide_ID FROM Requester
DO ADD $a PRECEDES $b;
   ADD $b PRECEDES $c;
OD;
/*================= end of reused MP code =======================*/

/*==============
========================================*/
/*
new system, which will reuse authentication system's behavior
*/
ROOT User:
  ( login_succeeds
    ( work |
      /* User may change their mind and cancel login */
      cancel ) |

  login_fails )
;
/*====================================================================*/
/* the following adjustments of the new model and the reused one are needed to integrate
their behaviors.
Reuse of authentication system's behavior:
  mapping actions of the Requestor from Authentication_Service on actions of the User. */
/* Requester's behavior is included into the behavior of User */
MAP Requester ON User;
/* synchronizing particular events from the authentication system's behavior with events in
the new system */
COORDINATE $ready: access_granted FROM Requester,
   $go: login_succeeds FROM User
DO ADD $ready PRECEDES $go; OD;
3. NOTES ON MP SEMANTICS – HOW TRACE DERIVATION WORKS

The event trace emerges as a result of derivation process, which is guided by grammar rules and composition operations. The event trace derivation determines the MP semantics. Trace derivation for schema, roots, and composite events is performed top-down and from left to right following the event grammar rules. Composition operations act like “crisscrossing” derivation rules that may add new relations to the trace under derivation, or filter out some trace segments from the result.

As John Hughes (Hughes, 1989) has put it in his influential paper: “The way in which one can divide up the original problem depends directly on the ways in which one can glue solutions together”. Event grammar rules in MP provide a way of modularization, or dividing the system’s behavior model into a hierarchy of component behaviors. Composition operations are separated from the grammar rules, and define interactions (or dependencies) between behaviors of components. This separation and supporting it mechanism of “gluing” behaviors together are the core features of MP.

For the schema

```plaintext
SCHEMA S
ROOT R1: ...;
ROOT R2: ...;
...
ROOT Rn: ...;
```

event trace derivation starts with an implicit grammar rule

```plaintext
S: { R1, R2, ... , Rn }
```

and deploys the composition operations interlacing the root rules and defined in the BUILD blocks attached to the root and composite event grammar rules.

For a given scope N derivation of all valid event traces starts with derivation of all trace segments within scope N for each composite event, followed by derivation of all valid trace segments for root events, and, finally, assembles all valid trace segments for the whole schema from root segments. Coordination operations are used on each phase to filter out trace segments as needed. The derivation process traverses the search space of possible trace segments in order to select valid trace segments and assemble them into next segment sets.

The search space may grow exponentially depending on the number and complexity of root and composite events in the model. This “combinatorial explosion” may be a serious problem for
managing large and complex MP models, if not addressed properly it may bring the trace derivation process to a standstill. To provide some control over the derivation process, MP deploys “layered derivation” for search space trimming. The purpose is to speed up the derivation by avoiding invalid trace segments early. There are several techniques to tame the “combinatorial explosion”, and in many cases to get the results in a reasonable time, see Section 4 for more details.

Any root definition (event grammar rule) should appear in the MP source code before the root name is used in composition operation, so that the trace derivation from the grammar rules could be accomplished before application of composition operations on those traces.

The Firebird event trace generator implements the following derivation steps for the given scope N.

1. First, all valid event traces (segments) for the scope N for each composite event defined in the schema are derived and stored. The order of composite event derivation is determined automatically by the dependencies between them, so that if composite event A is used in the right-hand part of composite event B, derivation for A will be accomplished before B. Composition operations included in the BUILD blocks attached to the composite event rules are performed at this phase, potentially reducing the number of derived event traces. No recursion (direct or indirect) in composite event grammar rules is allowed in this MP version.

2. Next, traces for the scope N for each root event are derived and stored in the order of root event appearance in the schema. Composition operations included in the BUILD blocks attached to the root event rules are performed at this phase, potentially reducing the number of derived event traces.

3. Each valid event trace for the schema is assembled from the segments of root event traces. Root segments for the trace assembly are selected following the order of root event appearances in the schema. Composition operations interleaving root events in the MP code are also performed in the order of their appearance. If a composition operation (SHARE ALL, COORDINATE, or ENSURE) involves a root event, that root event should appear in the schema before the operation, so that this root’s segment could be fetched before the composition operation. The Console window in Firebird provides statistics about the intermediate and final trace assembly.

4. Composition operation may impose constraints on the derived traces, like the equal numbers for coordinated events, or ENSURE constraints. Use of a composition operation as early as possible may speed up trace generation by pruning the search for valid traces. The ordering of root events and ordering of composition operations in the schema can provide a powerful trace generation speed improvement (see Sec. 4.2 for more details).

5. Traces for roots and composite events are generated independently before the whole schema’s trace is assembled. Hence, the FROM clauses in the BUILD blocks for composite and root events are limited to THIS and to variables containing inner events only.

6. All generated trace segments are checked for compliance with the Axioms (Sec. 6.1). If a trace segment fails to satisfy Axioms, it is rejected.

An example of reusable MP code can be found in Sec. 2.14.2, Example 19 with the COORDINATE operation creating an event list following the trace derivation order. You may use this MP code to learn about the event derivation order in your model.

Some suggestions to keep in mind as you progress in modeling effort.

- Proceed with the MP model building incrementally. Start with a simple (bare bones) model of the main actors/behaviors. Add actors (root events), events, and coordination for them one at a time. Introduce composite events to encapsulate the logically complete actions. Run your code with
scope 1 to weed out obvious errors.

- Don’t forget introductory comments with the name of author, date, and short description of the purpose of this MP model. Save/backup your MP file frequently.

- Make sure that event names are readable and informative. In many cases event names will be pseudo-code statements describing activities within the system and its environment.

- Use indentation to emphasize the order of events and main control structures (alternatives and iterations). Try to have one event name per line of MP code. Usually we read MP code top-down, and it always helps when while reading the code we can figure out what will be the order of events evolving from that code.

- After each update run the MP model to detect/fix coding errors and omissions in the MP code, and to decide about the appropriate scope, which is an essential aspect for non-trivial MP models. This may include adding specific iteration scope descriptions for selected iterators, like (*<1..4> A *) to steer the number of iterations. This way you can get around the scope limits imposed by Firebird, if needed.

- Validation begins when MP model starts to run smoothly. Now it is time to select the graph visualization mode (Sequence, Swim Lane, etc.) and to start browsing generated scenarios/use cases with respect to how well it captures your intent.

- The number of generated traces may start to grow into hundreds and thousands. It becomes cumbersome to find traces satisfying particular properties, for instance, you’d like to see whether there are traces containing the event Bell_Rings. To help, MP has IF and MARK operations, and adding to your model something like IF #Bell_Rings > 0 THEN MARK; FI; will mark traces satisfying the condition, and make it easier to find them in the trace browsing scrollbar.

- After the initial draft of the model is stabilized, add incrementally more actors/behaviors, repeating the test/debug activities after each increment. Insert comments in your MP code explaining your major design decisions and rationale for MP constructs used in the code. The best comments answer the question “Why?”, e.g. why this alternative or iteration has been placed in this particular position in the model, or why this coordination construct is here.

- Models are built by humans and usually need testing and debugging. For this purpose MP has CHECK construct for assertion checking (or for counterexample rendering) and SAY clause for annotating generated traces with messages. It is similar to the print statements used for debugging source code in traditional programming languages. After all, MP is executable modeling language. SAY clauses may provide answers to queries about number of events of interest already assembled into the trace segment, or indicate reaching certain point in the derivation process. This can be done both in BUILD blocks and in the schema’s code. The MARK construct combined with IF composition operation makes it possible to highlight traces of interest for testing/debugging purposes.

- MP is not a programming language, like C, C++, or Java. It is a behavior specification language. MP grammar rules are supposed to define all behaviors you want to have. So, if you want to describe behaviors with and without Bell_Rings event, make sure that your event grammar contains all alternatives - if the grammar does not contain a behavior you want to have, there is no way how you will obtain it. Now, it may contain also many unwanted behaviors, but we can weed them out using ENSURE, COORDINATE, and SHARE ALL. Consider these constructs as a powerful event trace filters. These are the principles of writing MP code.

The philosophy for verifying executable MP architecture and process models may be summarized as following.
Small Scope Hypothesis. Most flaws in models can be demonstrated on small counterexamples. MP tool generates exhaustive set of all possible behaviors within a small scope, and assertion checking with CHECK can produce all counterexamples of traces violating behavior’s property within the scope (these traces will be highlighted by MARK command). Assertions in the MP model may be checked both for the whole schema and for standalone root or composite event rules (CHECK clause in corresponding BUILD blocks). In the latter case, a counterexample of assertion violation may be just a trace segment for a particular root or composite event.

4. SOME HEURISTICS FOR MP CODE DESIGN

The trace generation time may be one of the main concerns when using MP tool. With scope increase it may grow rapidly. Generation time depends on several factors.

- The number and complexity of composite and root events.
- The number and size of traces generated for each root/composite event (this information is available in Firebird’s Console window).
- The number of composition operations and their place within MP code.
- The structure of composition operations (in particular, the nesting within COORDINATE).
- The ordering of root events in the schema’s code.

The derivation process is based on the search tree (selection of trace segments for assembly top-down and left-to-right). Pruning the search tree as early as possible is the main principle for speed-up. Here are some heuristics that may help to reduce the generation time.

4.1 Use scope with caution

The number of traces (and trace generation time) may increase dramatically with the scope. Start MP model’s testing/debugging with scope 1 to detect the initial obvious mistakes, and proceed to the higher scopes with caution. In many cases small scope (up to 3) is sufficient to detect/fix most of bugs and to verify all event traces within a scope with the CHECK constructs (Small Scope Hypothesis).

4.2 Use composition operations as early as possible

COORDINATE and SHARE ALL require coordinated threads of events to have the same size. If this condition does not hold, the derivation process backtracks one step back and picks up another root/composite event segment. When performed early in the derivation, it can prune a significant part of the search tree. Composition operation works as a powerful search tree’s pruning tool. ENSURE also acts as a filter rejecting assembled trace segment, when it violates context condition (assertion).

For the schema’s trace assembly root event segments are picked up in the order of root event rule appearance in the schema. Place a composition operation in the MP schema code immediately after all participating root event rules.

4.3 Use BUILD blocks

Root and composite event segments are derived before the schema’s trace assembly begins. This is the principle of “layered derivation” implemented in MP. In next MP implementations this feature could be supported by running segment derivation on parallel processors. Composition operations placed within BUILD blocks may reduce the number of segments derived for the composite/root event, and as a result reduce the total search. This is yet another application of the general principle: “prune the search as early as possible”.

67 of 108
4.4 Optimization of COORDINATE operations

Event thread selection for coordination in COORDINATE or SHARE ALL is time-consuming. If the same event thread participates in several coordination operations, it makes sense to merge them.

For example:

COORDINATE  $x$: a FROM A,
            $y$: b FROM B
        DO ADD $x$ PRECEDES $y$; OD;

COORDINATE  $x$: a FROM A,
            $z$: c FROM C
        DO ADD $x$ PRECEDES $z$; OD;

Since the thread is a FROM A the same in both operations, they can be merged as:

COORDINATE  $x$: a FROM A,
            $y$: b FROM B,
            $z$: c FROM C
        DO ADD $x$ PRECEDES $y$,
              $x$ PRECEDES $z$; OD;

The merge eliminates the repeated processing of the coordination thread for A. Notice that ADD may process several relations.

4.5 Use asynchronous coordination with caution

Use asynchronous coordination (event reshuffling with <!> or <!CHAIN>) only when it is necessary, like for coordinating asynchronous event threads. An example of Publish/Subscribe architecture model in Sec. 2.8 uses <!> event reshuffling in order to try all possible permutations of Register/Unsubscribe pairs. Such coordination may cause dramatic increase in the number of generated traces (and the generation time), since MP will generate all possible permutations of events to coordinate with other sources. If the selected event set has N events, then there are N! possible permutations. Since we expect traces to be generated for a reasonably small scope, the size of the selected event set will be also small and, correspondingly, the number of permutations also is expected to be modest.

4.6 Coordination and iteration

Coordination of several event threads requires that the numbers of selected events in each thread are equal. If selected threads have different number of events, the coordination fails, the trace under derivation is rejected, the derivation backups and proceeds with the next step. In some cases, when the designer overlooks potential event number conflicts, the whole derivation process ends up with zero traces. If this happens, the first debugging activity should be focused on finding the conflicts within coordination operations.

The inconsistence between the numbers of events in the coordinated threads may be a typical MP coding mistake. For example, in the following snippet a valid trace will be produced only once, when the number of B is precisely 1.

ROOT R1: A;
ROOT R2: (* B *);
COORDINATE $x$: A, $y$: B
DO ADD $x$ PRECEDES $y$; OD;

### 4.7 Coordination and alternatives

Another typical mistake with coordination may appear when the event selected for a coordination thread appears as an alternative. Then only traces containing that event will be accepted for coordination, but traces that select other alternatives will be rejected. Here is an example.

```plaintext
ROOT R1: ( A | B | C);
ROOT R2: D;
COORDINATE $x$: B, $y$: D
   DO ADD $x$ PRECEDES $y$; OD;
```

Only traces for root R1 containing B will be selected, traces for R1 containing A or C will be rejected because root R2 will have only traces with D. The solution for this issue (if we want to coordinate only B and D) is to make the coordination conditional on the presence of B.

```plaintext
ROOT R1: ( A | B | C);
ROOT R2: D;
IF #B > 0 THEN
   COORDINATE $x$: B, $y$: D
      DO ADD $x$ PRECEDES $y$; OD;
FI;
```

### 4.8 Synchronizing iteration cycles and coordination

Coordinated events may appear inside iteration as alternatives. There are at least two options for coordination.

*Option 1.* Coordinate event pairs even without any concern whether they appear in the same cycle of iteration or not. The following example illustrates this. Notice that the default event sequence (the order of event appearance during the derivation) is used for event selection in coordination threads.

```plaintext
SCHEMA S1
ROOT R1: (+ (A | B) C +);
ROOT R2: (+ (D | E) F +);
COORDINATE $a$: A FROM R1, $d$: D FROM R2
   DO ADD $a$ PRECEDES $d$;
   OD;
COORDINATE $b$: B FROM R1, $e$: E FROM R2
   DO ADD $b$ PRECEDES $e$;
   OD;
```

It may result in a trace like this:
Option 2. Coordinate event pairs, but only if they appear in the same iteration cycle. If selected alternatives within the same cycle cannot be coordinated, this trace is rejected.

SCHEMA S2
ROOT R1: (+ (A | B) C +);
ROOT R2: (+ (D | E) F +);

COORDINATE $a: (A | B)$ FROM R1,
$d: (D | E)$ FROM R2

DO
  IF $a$ IS A AND $d$ IS D OR $a$ IS B AND $d$ IS E THEN
    /* This pair is selected from the same cycle,
    since the default event sequence is used for coordination threads */
    ADD $a$ PRECEDES $d$;
  ELSE REJECT; /* otherwise reject the trace under derivation */
  FI;
OD;
Now the coordination is synchronized with the iteration cycles.
4.9 Merge and conquer

Complex MP models may have large number of root events with several coordination operations performed on clusters of roots. This may be system’s architecture model with a large number of actors (components) and interactions (connectors) between them. Usually coordination operations will be placed in schema’s body following the corresponding root event descriptions and performed during the derivation as described in Section 3. Derivation for such “heavy” MP models in Firebird may take too much time even for scope 1. Fortunately, the “layered derivation” strategy implemented in MP may help to speed up the trace derivation, and in many cases makes it possible to work with a pretty large and complex MP models. The idea is simple – organize the derivation as a hierarchy of smaller derivations.

Here is an example of how to benefit from the “layered derivation”. Schema S1 has several interacting root events, where roots B and C have several coordination operations between them. These coordination operations will be performed repeatedly each time when derivation process backtracks and performs new top-down pass through schema’s code for the next trace derivation.

Example 32. Merging root events.

SCHEMA S1
ROOT A: (* ( work | send_to_B) *);

ROOT B: (*<1.. $$scope + 1>
( work | receive_from_A | send_to_C ) *);
COORDINATE $a: send_to_B,
$b: receive_from_A
DO ADD $a PRECEDES $b; OD;

ROOT C: (*<1.. $$scope + 1>
( work | receive_from_B | send_to_D ) *);
COORDINATE $a: send_to_C,
$b: receive_from_B
DO ADD $a PRECEDES $b; OD;
B, C SHARE ALL work;

ROOT D: (* (work | receive_from_C ) *);
COORDINATE $a: send_to_D,
$b: receive_from_C
DO ADD $a PRECEDES $b; OD;

B and C have two coordination operations between them: COORDINATE and SHARE ALL. To delegate this “heavy” coordination to a separate derivation task we introduce a separate root event B_and_C to encapsulate the trace segment derivation (including the coordination) for B and C.

SCHEMA S2
ROOT A: (* ( work | send_to_B) *);
ROOT B_and_C: {B, C}
/* this root defines a new component encapsulating B and C and interactions in which B and C participate */

BUILD{

COORDINATE $one: B, $two: C
DO
    COORDINATE $a: send_to_C,
            $b: receive_from_B
    DO ADD $a PRECEDES $b; OD;

    $one, $two SHARE ALL work;
    OD;

};

COORDINATE $a: send_to_B,
            $b: receive_from_A
    DO ADD $a PRECEDES $b; OD;

B: (*<1.. $$scope + 1>
     ( work | receive_from_A | send_to_C ) *);

C: (*<1.. $$scope + 1>
     ( work | receive_from_B | send_to_D ) *);

ROOT D: (* (work | receive_from_C ) *);

COORDINATE $a: send_to_D,
            $b: receive_from_C
    DO ADD $a PRECEDES $b; OD;

Derivation for the schema S2 runs almost 20% faster than for S1 for scopes 2 and 3, because the coordination between B and C is performed now only when deriving segments for B_and_C. There is a price for it – additional memory is needed to store segments for B_and_C, a common trade in software design. The new event hierarchy is visible in the generated traces.

Fig. 33. Event trace for schema S2 demonstrating the result of “layered derivation”.
First candidates for a merge are root events with several coordination operations between them, and of course, more than two roots could be merged into a single “envelope”. Merge and conquer!

4.10 Divide and conquer
MP model complexity (the size of the search space) and trace generation time depend on several factors:
- Number of alternative and iteration event patterns in the MP code,
- Number and position in the code of root event definitions,
- Number and position in the code of coordination operations,
- Scope.

The way we can speed up trace generation depends on how early we can trim the derivation process. By applying “layered derivation” or by splitting the derivation process into separate “layers” for composite and root events, we can do a lot of trimming in the BUILD blocks for composites and roots, thus significantly reducing the search and backtracking in the next derivation layer. Positioning coordination operations as early in the MP code as possible also does the early trimming. All this has been explained in Sections 4.1 - 4.8.

But it is possible to add more derivation “layers” on the top of all that and thus significantly improve the performance of trace derivation. If the MP model is too complex and requires too long derivation time, we can select some property P and split the model into two: one where all traces have property P and another where all traces don’t have P. Typically P will state presence or absence of some events in the trace. Notice that traces in one model are all different from traces in another – because either the property P holds or not. The union of two event trace sets yields the final result for the original MP model. All meaningful scenarios are either in one or another set. In most cases the total derivation time spent on the “split” models will be several times less neither the derivation time of the original model.

If the “split” model still requires too much derivation time, we can continue the same trick: select another property P1 and again split the complex MP model into two – one that satisfies P1, and another that does not. Each new MP model is significantly simpler than previous one, because by selecting P or not P we can remove several alternatives and coordination operations, which significantly reduces the time. The union of sets of traces produced by these reduced MP models still is the complete set of scenarios (traces) for the original MP model. We just have introduced a new layer of derivation.

Run time for each simplified MP model may be dramatically smaller, so that the total time to run all of them should be smaller than the total time for the original “heavy” MP model. The set of traces for the same scope is the same as the set of traces for the original MP model, it will be just split into several separate sets.

The method certainly is generic, for any complex MP model we can find a good property P to split it into two sub-cases, and we can continue such splitting until we get simple and fast enough MP models, retaining the final result - the complete and exhaustive set of generated traces.

This method of fighting “combinatorial explosion” is unique for MP. In MP it is often easy to perform “partial evaluation” of the MP event grammar rules excluding/retaining alternatives, which support the property P or its negation. By removing these alternatives we downsize the complexity and run time.
Most of existing formal methods, like traditional model checking or Alloy cannot support this way of optimization. If you add a new constraint “should satisfy P” or “should not satisfy P”, it will only increase the number of propositional formulae required to satisfy the model and will increase the size of search space and the run time.

In MP we have the flexibility and ease to split behaviors, which satisfy or don’t satisfy P into separate and simpler models. The following toy example illustrates the model “splitting” method, but the approach for splitting MP model into sub-models will be the same for many other cases.

Example 33. Model of application process.

Applicant submits an application and needs to get it approved by the bureaucratic chain of two officials. If the first official approves, he forwards the application to the second official. Application is approved only after it receives both approvals. Each official can approve/reject the application or request it reworked and resubmitted. Rework and resubmit events may be repeated several times. Each rejection is final and cannot be followed by rework and resubmit events.

The statistics of trace numbers and derivation times for different scopes is presented in a table at the end of this section.

The original MP model is as follows.

```plaintext
SCHEMA Application_approval_process
ROOT Applicant:
    prepare_application
    submit_application
    (* rework
      submit_application *)
    ( application_is_approved | application_is_rejected );

ROOT Official_1:
    (+ receives_application_from_Applicant
      (approves_and_forwards_to_Official_2 | request_rework | reject ) )
    +)
BUILD{ ENSURE #reject <= 1;
        ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; }

COORDINATE $s: submit_application FROM Applicant,
              $r: receives_application_from_Applicant FROM Official_1
DO ADD $s PRECEDES $r; OD;

ROOT Official_2:
    (* receives_application_from_Official_1
      (approves_and_forwards_to_Applicant | request_rework | reject )
    *)
BUILD{ ENSURE #reject <= 1;
```
ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; 

COORDINATE $s: approves_and_forwards_to_Official_2 FROM Official_1,
   $r: receives_application_from_Official_1 FROM Official_2
   DO ADD $s PRECEDES $r; OD;

/*--- interactions with Applicant ---------*/
COORDINATE $r: reject, /* this may come from several actors */
   $rr: application_is_rejected FROM Applicant
   DO ADD $r PRECEDES $rr; OD;

COORDINATE $r: request_rework, /* this may come from several actors */
   $rr: rework FROM Applicant
   DO ADD $r PRECEDES $rr; OD;

COORDINATE $a: approves_and_forwards_to_Applicant FROM Official_2,
   $aa: application_is_approved FROM Applicant
   DO ADD $a PRECEDES $aa; OD;

The fist round of splitting this MP model into two separate models is based on the property “rework was required”. By performing “partial evaluation” on the original MP code, the following two MP models are obtained. Please notice how parts of the model that are not needed anymore have been removed from the original code.

This MP model accepts only event traces that don’t have events rework and request_rework. The MP code has been obtained from the original Application_approval_process model by deleting parts containing rework and request_rework (these parts are highlighted in red).

SCHEMA Application_approval_no_rework
ROOT Applicant:
   prepare_application
   submit_application
   (* rework
      submit_application *)
   ( application_is_approved | application_is_rejected )

ROOT Official_1:
   (+ receives_application_from_Applicant
     (approves_and_forwards_to_Official_2 | request_rework | reject )
   )
BUILD{ ENSURE #reject <= 1;
   ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; }

COORDINATE $s: submit_application FROM Applicant,
   $r: receives_application_from_Applicant FROM Official_1
   DO ADD $s PRECEDES $r; OD;
ROOT Official_2:

(*)
receives_application_from_Official_1
(approves_and_forwards_to_Applicant | request_rework | reject )

*)
BUILD{ ENSURE #reject <= 1;
ENSURE FOREACH $r: reject #$$EVENT AFTER $r == 0; };

COORDINATE $s: approves_and_forwards_to_Official_2 FROM Official_1,
$r: receives_application_from_Official_1 FROM Official_2
DO ADD $s PRECEDES $r; OD;

COORDINATE $r: reject, /* this may come from several actors */
$rr: application_is_rejected FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE $r: request_rework, /* this may come from several actors */
$rr: rework FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE $a: approves_and_forwards_to_Applicant FROM Official_2,
$aa: application_is_approved FROM Applicant
DO ADD $a PRECEDES $aa; OD;

The next MP model accepts only event traces that contain rework and request_rework. The MP code has been obtained from the original Application_approval_process. Modified parts are highlighted in red.

SCHEMA Application_with_rework
ROOT Applicant:
prepare_application
submit_application
(+ rework
   submit_application +) /* modified to ensure the presence rework events */
/* eliminates empty alternative for the iteration, reducing the number of trace segments */
( application_is_approved | application_is_rejected );

ROOT Official_1:
(+ receives_application_from_Applicant
(approves_and_forwards_to_Official_2 | request_rework | reject )
+ )
BUILD{ ENSURE #reject <= 1;
ENSURE FOREACH $r: reject #$$EVENT AFTER $r == 0; };

COORDINATE $s: submit_application FROM Applicant,
$r: receives_application_from_Applicant FROM Official_1
DO ADD $s PRECEDES $r; OD;

ROOT Official_2:
(*  receives_application_from_Official_1
   (approves_and_forwards_to_Applicant |
    request_rework  |
    reject )
*)
BUILD{ ENSURE #reject <= 1;
    ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; }

COORDINATE $s: approves_and_forwards_to_Official_1 FROM Official_1,
    $r: receives_application_from_Official_1 FROM Official_2
DO ADD $s PRECEDES $r; OD;

/*--- interactions with Applicant --------*/
COORDINATE    $r: reject, /* this may come from several actors */
               $rr: application_is_rejected FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE    $r: request_rework, /* this may come from several actors */
               $rr: rework FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE    $a: approves_and_forwards_to_Applicant FROM Official_2,
               $aa: application_is_approved FROM Applicant
DO ADD $a PRECEDES $aa; OD;

Further split is done on the Application_with_rework model. The characteristic property used for the split is whether the trace contains event application_is_approved or application_is_rejected. Deleted and modified parts are highlighted in green.

SCHEMA Application_with_rework_approved
ROOT Applicant:
  prepare_application
  submit_application
  (+ rework
    submit_application +) /* modified to ensure the presence rework events */
  /* eliminates empty alternative for the iteration, reducing the number of trace segments */
  ( application_is_approved | application_is_rejected );

ROOT Official_1:
(+ <2..$$scope> /* will receive application at least twice */
  receives_application_from_Applicant
  (approves_and_forwards_to_Official_2 |
   request_rework  |
   reject )
+);
BUILD{ ENSURE #reject <= 1;
ENSURE FOREACH $r: reject \$\$$EVENT AFTER $r == 0; };

COORDINATE $s: submit_application FROM Applicant, 
  $r: receives_application_from_Applicant FROM Official_1 
DO ADD $s PRECEDES $r; OD;

ROOT Official_2: 
  (* receives_application_from_Official_1 
    (approves_and_forwards_to_Applicant | 
      request_rework | 
      reject 
    ) 
  *) ;
BUILD{ ENSURE #reject <= 1; 
  ENSURE FOREACH $r: reject \$$EVENT AFTER $r == 0; };

COORDINATE $s: approves_and_forwards_to_Official_2 FROM Official_1, 
  $r: receives_application_from_Official_1 FROM Official_2 
DO ADD $s PRECEDES $r; OD;

/*@--- interactions with Applicant --------*/
COORDINATE $r: reject, /* this may come from several actors */
  $rr: application_is_rejected FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE $r: request_rework, /* this may come from several actors */
  $rr: rework FROM Applicant
DO ADD $r PRECEDES $rr; OD;

COORDINATE $a: approves_and_forwards_to_Applicant FROM Official_2, 
  $aa: application_is_approved FROM Applicant 
DO ADD $a PRECEDES $aa; OD;

Deletion of several alternatives and COORDINATE significantly reduces the complexity and derivation time for this model.

The model for the case when rework was required but application was rejected. Deleted and modified parts are highlighted in green.

SCHEMA Application_with_rework_rejected
ROOT Applicant:
  prepare_application
  submit_application
 (+ rework
   submit_application +) /* modified to ensure the presence of rework events */
/* eliminates empty alternative for the iteration, reducing the number of trace segments */
  ( application_is_approved |
    application_is_rejected );

ROOT Official_1:
 (+ receives_application_from_Applicant
(approves_and_forwards_to_Official_2 | 
  request_rework | 
  reject )
+
BUILD{ ENSURE #reject <= 1; 
  ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; }

COORDINATE $s: submit_application FROM Applicant, 
  $r: receives_application_from_Applicant FROM Official_1 
DO ADD $s PRECEDES $r; OD;

ROOT Official_2:
(* receives_application_from_Official_1 
  (approves_and_forwards_to_Applicant | 
    request_rework | 
    reject ) 
*)
BUILD{ ENSURE #reject <= 1; 
  ENSURE FOREACH $r: reject #$EVENT AFTER $r == 0; }

COORDINATE $s: approves_and_forwards_to_Official_2 FROM Official_1, 
  $r: receives_application_from_Official_1 FROM Official_2 
DO ADD $s PRECEDES $r; OD;

/*--- interactions with Applicant --------*/
COORDINATE $r: reject, /* this may come from several actors */
  $rr: application_is_rejected FROM Applicant 
DO ADD $r PRECEDES $rr; OD;

COORDINATE $r: request_rework, /* this may come from several actors */
  $rr: rework FROM Applicant 
DO ADD $r PRECEDES $rr; OD;

COORDINATE $a: approves_and_forwards_to_Applicant FROM Official_2, 
  $aa: application_is_approved FROM Applicant 
DO ADD $a PRECEDES $aa; OD;

Statistics for these models is assembled in the following table. The measurements have been performed with MP Gryphon running on iMac workstation with 4 GHz Intel Core i7 processor and memory 8 GB. Even for this toy model it appears that the same set of event traces can be obtained in a significantly reduced time after the original model has been split into three sub-models. The main advantage of the split is in the significant reduction of the search space for each model.

<table>
<thead>
<tr>
<th></th>
<th>Scope 1</th>
<th>Scope 2</th>
<th>Scope 3</th>
<th>Scope 4</th>
<th>Scope 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original MP model</td>
<td>3 traces in 0.00095 sec.</td>
<td>9 traces in 0.011822 sec.</td>
<td>18 traces in 0.142988 sec.</td>
<td>30 traces in 1.29319 sec.</td>
<td>45 traces in 9.52105 sec.</td>
</tr>
<tr>
<td>Model no_rework</td>
<td>3 traces in 0.000638 sec.</td>
<td>3 traces in 0.000921 sec.</td>
<td>3 traces in 0.00129 sec.</td>
<td>3 traces in 0.001956 sec.</td>
<td>3 traces in 0.003417 sec.</td>
</tr>
</tbody>
</table>
MP v.3.5
Draft modified: 9 September 2018

<table>
<thead>
<tr>
<th>Model with_rework and approved</th>
<th>No traces</th>
<th>6 traces in 0.010938 sec.</th>
<th>15 traces in 0.141649 sec.</th>
<th>27 traces in 1.16603 sec.</th>
<th>42 traces in 9.13005 sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total for no_rework and with_rework, time ratio</td>
<td>3 traces in 0.000638 sec. 67%</td>
<td>9 traces in 0.011859 sec. 100.3%</td>
<td>18 traces in 0.142939 sec. 99.96%</td>
<td>30 traces in 1.167986 sec. 90%</td>
<td>45 traces in 9.133467 sec. 95.9%</td>
</tr>
<tr>
<td>Model with_rework and rejected</td>
<td>No traces</td>
<td>2 traces in 0.00266 sec.</td>
<td>5 traces in 0.033998 sec.</td>
<td>9 traces in 0.29292 sec.</td>
<td>14 traces in 2.15931 sec.</td>
</tr>
<tr>
<td>Model with_rework and rejected</td>
<td>No traces</td>
<td>4 traces in 0.002946 sec.</td>
<td>10 traces in 0.02632 sec.</td>
<td>18 traces in 0.13386 sec.</td>
<td>28 traces in 0.592665 sec.</td>
</tr>
<tr>
<td>Total for no_rework, with_rework and approved, with_rework and rejected, time ratio</td>
<td>3 traces in 0.000638 sec. 67%</td>
<td>9 traces in 0.006527 sec.</td>
<td>18 traces in 0.061608 sec.</td>
<td>30 traces in 0.428262 sec.</td>
<td>45 traces in 2.755392 sec.</td>
</tr>
</tbody>
</table>

5. MP LANGUAGE CONSTRUCTS

5.1 Lexics

'ttt' - terminal string
Id - identifier, sequence of upper or lower case letters, digits, and underscore, starting with a letter. Identifiers are case-sensitive.

Current implementation limits identifier’s length to 80 characters.

float_constant - float constant has decimal point, like 24.5. It may have unary minus preceding it, like -24.5. Use of exponent also is allowed, like 24E2 or 24.5E-2

integer_constant - integer constant should be decimal integer, like 25, and may have unary minus preceding it, like -25

"characters" - string constant with zero or more characters, which may be any printable ASCII characters, except escaped characters, like ‘\n’. Current implementation limits string constant length to 80 characters.

comments - /* text */

Spaces, eol, and tabs in MP code are not syntactically significant, except when separating keywords and/or identifiers.

5.2 Extended BNF notation for MP syntax

P, P1, and P2 stand for grammar’s terminals, non-terminals, or groups, where E stands for a terminal symbol.

(* P *) - P repeated zero or more times
(+ P +) - P repeated one or more times
(* P * E) - P repeated zero or more times, separated by E
(+ P + E) - P repeated one or more times, separated by E
( P1 | P2 ) - alternative (may have one or more alternatives)
[P] - optional P

5.3 Event grammar rules

(1) schema:
'SCHEMA' schema_name(2)
(* (rule(3) | composition_operation(25) | attribute_declarations(47) ) ';

*})
;
Derivation of the schema's trace proceeds top-down and from the left to right. Composition operations are performed following the order of derivation, immediately after the root derivation and composition operations appearing in the schema before them were completed, and hence may speed up the trace derivation by rejecting the previously derived root traces before proceeding with the next roots.

(2) schema_name: Id -- schema's name is an identifier 
;

(3) rule:
( 'ROOT' root_name(4) | event_name(7) ) ':'
	pattern_list(5)

[ Build_block(51)]
;
Absence of the ROOT keyword yields a composite event declaration.

Operations in BUILD blocks defined for roots and composites are performed each time when the trace derivation for that root or composite is accomplished. These are local “crisscrossing” derivation rules, which can add more relations within the root or composite trace. Notice the absence of semicolon before BUILD block.

(4) root_name: Id -- root name is an identifier 
;

5.4 Event patterns

(5) pattern_list:
(* pattern_unit(6) *)
;

(6) pattern_unit:
( event_name(7) | alternative(8) | iterator(10) | iterator_plus(16) | set(17) | set_iterator(18) | set_iterator_plus(19) | optional(20) )
;
No recursion (direct or indirect) in composite event grammar rules is allowed.
(7) event_name: Id
;
Event name is an identifier, atomic or composite event type name.

(8) alternative:
    '({' (+ [ probability(9) ] pattern_list(5) +'|' ) '})'
;
The 'probability' option is the probability to select the alternative in the random trace generation mode, each selection is independent from the previous history. Empty option for pattern list is needed to balance probabilities assignment, e.g. in ( A | B | ) empty alternative has probability 0.3333333, but in [(A|B)] it will have probability 0.5. [A] is equivalent to (A| )

Alternative pattern with a probability and a single branch (<<p>> A) is equivalent to the optional(20) pattern [<<p>> A], or to the (<<p>> A | )
Alternative pattern ( A ) has default probability 1.0 and is equivalent to the pattern A.

(9) probability:
    '<<' float_number(55) '>>'
;
Probability for selecting an alternative in the random trace generation process. Current MP implementation does not have Monte Carlo simulation option for random trace generation, but, nevertheless, the probability of a trace can be calculated. See Example 26.

float_number should be in the range 0.0 – 1.0
Sum of probabilities within the alternative pattern unit should be 1.0
Probabilities not provided explicitly, are prorated evenly to supplement the total sum to 1.0

(10) iterator:
    '(*' [ iteration_scope(11)] pattern_list(5) '*)'
;
Probability of selecting particular number of iterations in the derivation process can be specified using alternative event pattern with probabilities attached to the branches of alternative, like
    (<<0.7>> (*<1> E *) | <<0.3>> (*<2> E *) )

(11) iteration_scope:
    '<' [ scope_expression(12) '..' ] scope_expression(12) '>
;
The value of scope_expression should be non-negative integer.
The default for (* ... *) iteration is < 0 .. $$scope >. Iteration scope <min .. max> implies that during trace derivation the number of iterations will proceed from min to max; < n > is the same as <n .. n>. The scope_expression may be helpful to ensure that the number of iterations is sufficient to produce enough events to coordinate with multiple sources (see Publish_Subscribe example in Sec. 2.8, the iteration in the Publisher root).

(12) scope_expression:
    scope_expr_additive_elt(13)
    (* ('+' | '-') scope_expr_additive_elt(13) *)
;
(13)  `scope_expr_additive_elt:`
    `scope_expr_term(14)`
    `(\* ('*' | '/') scope_expr_term(14) \*)`;

(14)  `scope_expr_term:`
    `( integer_number(54) | (' ' scope_expression(12) ')' | scope_metavariable(15) ) )`;

(15)  `scope_metavariable: '$$scope'`;

The value of $$scope is set up before the trace derivation begins, and hence is available for use in iteration_scope(11), reshuffling_unit(33), and in expression(64) in MP code.

(16)  `iterator_plus:`
    `('+ [ iteration_scope(11)] pattern_list(5) '+')`;

    The default for iteration scope is < 1 .. $$scope >, the lower limit can not be less than 1;

(17)  `set:`
    `'{ ' (* pattern_list(5) * ',', ' ) '}'`;

    set requires only IN relation between the set and its elements;

(18)  `set_iterator:`
    `'{* [ iteration_scope(11)] pattern_list(5) '*}')`;

    The default for iteration scope is < 0 .. $$scope >; < n > is the same as <n .. n>

(19)  `set_iterator_plus:`
    `'+ [ iteration_scope(11)] pattern_list(5) '+')`;

    The default for iteration scope is < 1 .. $$scope >, the lower limit can not be less than 1

(20)  `optional:`
    `'[ [ probability(9)] pattern_list(5) ]]'`;

    The default probability is 0.5

5.5 Navigation directions

(21)  `navigation_direction:`
    `relational_expression(22)`;
Navigation within event traces is one of the main MP features, and uses binary relations between event instances - basic (IN and PRECEDES), or user-defined. This “Time Machine” is deployed in bool_expr3(60), and may be used in quantified_bool_expr(61), and special_function(70) explicitly, or via thread(24) with SUCH THAT clause in coordination operations containing asynchronous_source(29) or coordination_source(30).

\[(22)\] relational_expression:
\[
(\text{`IN'} | \text{`PRECEDES'} | \text{`ENCLOSING'} | \text{`FROM'} | \text{`CONTAINS'} | \text{`FOLLOW'S'} | \text{`BEFORE'} | \text{`AFTER'} |
\]
relation_name(52) |
user_defined_relation(53) |
\text{`( relational_expression ')'} |
\text{`~ relational_expression } |
\text{`^ relational_expression }
\]

For a binary relation \(R\) (basic or user-defined), \(~R\) stands for the reverse \(R\), and \(^R\) for the transitive closure (the relation that contains \(R\) and is transitive). Operations \(~\) and \(^\) are right-associative. Note that \(^(~R)\) is equivalent to \(~(^R)\). See Sec.6.2 for more details.

For the convenience of navigation within the trace, besides of the basic relations IN and PRECEDES, there are additional relation names for reversed and transitive closures of basic relations:

ENCLOSING is a reverse of IN, or \(~IN\),
FROM is a transitive closure of IN, or \(^IN\)
CONTAINS is a reverse of FROM, or \(~FROM\), or \(~(^IN)\),
FOLLOW'S is a reverse of PRECEDES, or \(~PRECEDES\),
BEFORE is a transitive closure of PRECEDES, or \(^PRECEDES\),
AFTER is a reverse of BEFORE, or \(~BEFORE\), or \(~(^PRECEDES)\).

Sec.6.1 provides ten axioms specifying properties of basic relations. This implies that valid event trace always is a directed acyclic graph.

The following picture illustrates possible navigation relations between events \(a\) and \(b\). The REL stands for a user-defined binary relation name in \(a \text{ REL } b\).
variable: $Id$

$MP$ variable is an identifier preceded by "$", like $a$, and can be assigned an event instance as a value. It can be used in several $MP$ constructs: thread(24), shared_composition(26), coordination_source(30), add_relation(34), map_unit(38), bool_expr(60), event_instance(68), special_function(70).

thread:

```
variable(23) ':' selection_pattern(35)
['FROM' event_instance(68)]
['SUCH' 'THAT' bool_expr(57)]
```

The thread is kind of a set comprehension, and specifies a set of events selected by applying selection pattern and selection conditions (SUCH THAT). If an event instance has been shared with other root or composite event, only a single instance of shared event will be included in the thread.

Threads may be used as sources of events in coordination_source(30), in asynchronous_source(29), in the # special_function(70). The scope of variable bound to the selection_pattern depends on the operation/expression, which deploys the thread.

Search for events matching the selection_pattern is performed within the set determined by the FROM clause. Absence of the FROM clause is equivalent to FROM THIS, where THIS refers to the root or composite event in the BUILD block of which the thread appears, or to the whole schema, if the thread appears in the schema’s body.
If the thread appears in the BUILD block of a root or composite event, root_name in the FROM clause is not allowed. Root names as event_instance in FROM may be used in schema’s body.

**Example.** Assume that the selection is done from the schema’s trace segment (FROM THIS), where A, B are root events. SUCH THAT clause makes it possible to assemble a thread from different events and from several sources. It can be used to express set union and set intersection for event sets. The following thread is a union of two sets - events of type a from A, and events of type b from B:

\[ Sx: (a | b) \text{ SUCH THAT } (Sx \text{ FROM } A \text{ AND } Sx \text{ IS } a) \text{ OR } (Sx \text{ FROM } B \text{ AND } Sx \text{ IS } b) \]

If two events within the search domain are shared as a result of previous coordination with SHARE or MAP, only one instance (which was added to the trace first) will be included in the thread.

### 5.6 Composition operations

(25) **composition_operation:**

```
( shared_composition(26)
  | asynchronous_coordinate_composition(27)
  | coordinate_composition(28)
  | MAP_composition(37)
  | conditional_composition(39)
  | simple_action(40)
)
```

Composition operations act like “crisscrossing” derivation rules that may add new events and basic relations to the trace under derivation.

(26) **shared_composition:**

```
(+ (root_name(4) | variable(23) ) + ',',')
  'SHARE' 'ALL'
(+ event_name(7) + ',',')
```

Clause root_name is not allowed if a composition operation appears in the BUILD block of a root or composite event, it may be used in schema’s body only.

The sharing composition operation is performed on the default sequences of events. It does not have explicit reshuffling option, but if needed, reshuffling for SHARE ALL can be done by combining coordinate composition operation (28) or (27) and SHARE_clause(41), as demonstrated in the example for SHARE_clause(41).

Events to be shared are of the same type (have the same event type name) and hence have the same event attribute signature. If the values of corresponding event attributes are not equal, the SHARE ALL operation fails, and the failure is propagated to the enclosing BUILD block or schema.

There is one exception. Time attributes depend on basic relations and are recalculated automatically each time when basic relations of an event are adjusted in the derivation process. When a pair of events is SHARED, their relations (basic and user-defined) are merged, and this may trigger time attribute recalculation.

(27) **asynchronous_coordinate_composition:**

‘COORDINATE’
Asynchronous coordinate composition should contain at least one asynchronous source(29) and may produce several different event traces, one for each permutation in each asynchronous source. It contributes to the trace derivation process in a similar way as the alternative(8) pattern does.

Asynchronous coordinate composition cannot be nested into another asynchronous_coordinate_composition(27), coordinate_composition(28), or conditional_composition(39).

It may contain several asynchronous sources(29) and/or coordination sources(30), and may contain other composition operations (25) in the DO/OD body (including coordinate compositions(28), but except another asynchronous_coordinate_composition).

COORDINATE is essentially a loop over one or more event sets of equal size (coordination sources). It starts with the tuple of first event from each coordination source followed by performing the DO/OD on that tuple, then a tuple of second events in each coordination source is selected and DO/OD is performed again, and so on. The composition operations enclosed within DO/OD are performed for each tuple of events selected from the coordination sources.

By default the order of event selection follows the order of events in the trace generated during the derivation process. In most cases this will be sufficient. The coordination_source(30) may provide an option for event reshuffling to change the order of coordination for the selected sets.

Variables bound to the selection patterns within coordination sources of the composition are visible in the DO/OD body of this composition (including nested threads), but not in other threads coordinated by this coordination composition.

If all coordination sources yield empty sets, no operations in the DO/OD body will be performed but the coordination operation will still succeed.
<!> means asynchronous coordination, and may produce several event sets, each of which will be tried by enclosing COORDINATE. The traces resulting from the coordination operation will include all permutations of events from the coordination source. This assumes that other constraints, like partial ordering axioms (Sec. 6.1) are satisfied. Each valid permutation yields an instance of a resulting trace for the composite event, root, or schema deploying this composition. If there are $N$ events in the default sequence, <!> will produce $1 \times 2 \times 3 \times \ldots \times (N-1) \times N = N!$ sequences. Asynchronous coordination may cause dramatic increase in the number of generated traces and the generation time. See examples in Sec. 2.8 and 2.14.

<!CHAIN> is similar to <!>, it produces all permutations of events from the coordination source. In addition it will add the PRECEDES relation between each pair of adjacent events in the resulting sequence. If the insertion of PRECEDES violates partial ordering axioms (Sec. 6.1), the generated permutation is rejected, coordination backtracks and proceeds with the next permutation.

$$\text{(30) coordination_source:}$$
\[
[ \text{event_reshuffling_option(31)} ] \text{ thread(24)}
\]
$$;$$

$$\text{(31) event_reshuffling_option:}$$
\[
( \text{ '!>>' } | \text{ '<' } \text{ sequence_of_reshufflings(32) '>' } )
\]
$$;$$

The default for event selection in the coordination source within COORDINATE preserves the order of events generated during the derivation process. The default means that the thread will be used “as is”. The reshuffling operations may be applied to the default set before the coordination to rearrange events within the selected set.

Strict synchronous coordination !>> sorts selected event set with respect to the transitive closure of PRECEDES (BEFORE relation), and the coordination will follow this ordering. If event set cannot be totally ordered, the coordination operation fails, and the trace derivation backtracks.

Event reshuffling may be useful when we need to control the coordination order, but in most cases the default option will be sufficient. The same consideration applies also to SHARE ALL composition operation. It does not have explicit reshuffling option, but if needed, reshuffling for SHARE ALL can be obtained by combining COORDINATE and SHARE clause, as demonstrated in the example for SHARE_clause(41).

$$\text{(32) sequence_of_reshufflings:}$$
\[
( + \text{ reshuffling_unit(33) +} )
\]
$$;$$

$$\text{(33) reshuffling_unit:}$$
\[
( \ ( \ \text{ 'SORT' } | \text{ 'REVERSE' } | \\
\text{ 'SHIFT_RIGHT' } | \text{ 'SHIFT_LEFT' } | \\
\text{ 'CUT_FRONT' } | \text{ 'CUT_END' } | \\
\text{ 'FIRST' } | \text{ 'LAST' } )
[ \text{ '(' } \text{ scope_expression(12) ')} ] \\
\text{ '(' } [ \text{ '<' } \text{ scope_expression(12) '>' } ] \\
\text{ sequence_of_reshufflings(32) ')*' } )
\]
SORT (plain synchronous coordination) performs topological sort on the selected event set with respect to the transitive closure of PRECEDES (BEFORE relation), and the coordination will follow this ordering. This means that if A BEFORE B holds for events A and B, event A will appear in the sorted sequence before B. If the result does not yield a total ordering, the coordination will still proceed.

REVERSE produces the reversed sequence. For example, it may be useful when modeling Last-In-First-Out (or LIFO) behaviors.

SHIFT_RIGHT(N) takes the last N events from the default sequence and moves them to the beginning (circular shift N to the right). This may be useful for coordinating event with its neighbors. SHIFT_RIGHT is equivalent to SHIFT_RIGHT(1).

SHIFT_LEFT(N) takes the first N events from the default sequence and moves them to the end (circular shift N to the left). See the Ring example in Sec. 2.13. SHIFT_LEFT is equivalent to SHIFT_LEFT(1).

The reshuffling operations below have similar constraint: if the selected event sequence contains less than N elements, the enclosing coordination operation fails, and the trace derivation backtracks. The same happens if any of these operations are applied to the empty set.

CUT_FRONT(N) deletes N events from the front of selected sequence. CUT_FRONT is equivalent to CUT_FRONT(1).

CUT_END(N) deletes N events from the end of selected sequence. CUT_END is equivalent to CUT_END(1).

FIRST(N) returns the first N elements from the sequence and cuts the rest. FIRST is equivalent to FIRST(1).

LAST(N) returns the last N elements from the sequence and cuts the rest. LAST is equivalent to LAST(1).

(* <N> sequence *) will repeat the sequence of reshuffling operations N times. Please notice that symbols ‘<’ and ‘>’ are used to enclose both sequence of reshuffling units and the iteration number (which may be an expression).

Sequence of reshuffling operations is performed from left to right. Reshuffling operations provide certain flexibility to shape the set of events selected for coordination.

Operations CUT, FIRST, and LAST may change size of the selected set. Checking for the equal size of coordination sources within the COORDINATE is performed after the reshuffling for each source is accomplished.

Please notice that “!>>” and “<!>” are not allowed within the reshuffling_unit.

Examples of event reshuffling options composed from simple reshuffling operations.

- Two shifts left can be described as <SHIFT_LEFT  SHIFT_LEFT> or as <(* 2>  SHIFT_LEFT *)>
- <SHIFT_RIGHT> is equivalent to <REVERSE  SHIFT_LEFT  REVERSE>
• Event set in descending topologic order: <SORT REVERSE>

(34) add_relation:
   'ADD'
   (+
   ( event_instance(68) | SAY_clause(44) )
   ( 'IN' | 'PRECEDES' | user_defined_relation(53) )
   ( event_instance(68) | SAY_clause(44) )
   +'
   )

; If add_relation operation appears in the BUILD block of a root or composite event, root_name in event_instance is not allowed. It may be used in schema’s body only.

Notice that relations ENCLOSING, FROM, CONTAINS, FOLLOWS, BEFORE, and AFTER are reserved and cannot be added.

SAY_clause is a special case of composition operation that creates a new instance M of message event (visualized in the Firebird event trace as a “dog ear” box) and initializes its text attribute with the message text. SAY operation by default always performs ADD M IN THIS operation, and the “dog ear” box will be associated with the schema or with the root, composite event in which BUILD block it appears. By placing SAY as argument in ADD it becomes possible to bind the message event with some other event with IN, PRECEDES, or user-defined relation, thus providing more focused visualization of the message (see Example 9).

User-defined binary relations can be used in the navigation_direction(21) via relational_expression(22).

(35) selection_pattern:
   ( Single_event_selection(36) | '
   ( ' (+ Single_event_selection(36) + '|') ')'
   )

; Selection_pattern is used in coordinate_composition(28) via thread(24), quantified_bool_expr(61), and in special_function(70)

(36) Single_event_selection:
   ( event_name(7) | '$$EVENT'
   | '$$ROOT'
   | '$$COMPOSITE'
   | '$$ATOM'
   )

; The meta-symbol $$EVENT matches any event type, including the whole trace event (which always is present in the generated trace). This event pattern could be used with the ENSURE operation for user-defined relation properties, similar to the examples in Sec. 6.1, or to provide the total number of events within another event E, like $$EVENT FROM E. $$EVENT will yield the total number of events within THIS.

$$ROOT matches root events only, $$COMPOSITE matches composite events (but not root events), and $$ATOM matches atomic events only.
(37) \[
\text{map\_composition:
  \(\text{(+ 'MAP' map\_unit(38) 'ON' map\_unit(38) +',')\)}
\];
\]

The purpose of the map construct is to implement behavior reuse.

MAP A ON B updates the context and structure of event B by adding B to all relations in which A participates - IN, PRECEDES, and user-defined relations (53). MAP does not affect event attributes, except time attributes, which depend on basic relations and are recalculated automatically each time when basic relations of an event are adjusted in the derivation process.

If REL is basic or user-defined relation, then MAP A ON B is equivalent to performing the following operations for all REL.

COORDINATE $x$: $\$\text{EVENT}$
\[
\text{DO IF }$x$ \text{ REL } A \text{ THEN ADD }$x$ \text{ REL } B \text{; FI;}
\]
\[
\text{IF } A \text{ REL }$x$ \text{ THEN ADD } B \text{ REL }$x$ \text{; FI;}
\]
\[
\text{OD;}
\]

Suppose that original behaviors of A and B are specified as
A: pattern1;
B: pattern2;
Because all IN relations in which A participates are copied to B, the behavior of B after MAP A ON B can be described as a set of original behaviors, i.e.
B: \{ pattern1, pattern2 \};

Thus, MAP is a shortcut for a whole group of coordination operations, which otherwise should be written separately.

After MAP A ON B, B also inherits all behaviors in which A participates (the whole context of A). The difference with SHARE_clause(41) being that the instance of A and its relations still remain in the trace unchanged.

(38) \[
\text{map\_unit: } \ ( \ \text{root\_name(4)} \ | \ \text{variable(23)} \ )
\];
\]

If map composition operation appears in the BUILD block of a root or composite event, root\_name is not allowed. It may be used in the schema’s body only.

(39) \[
\text{conditional\_composition:}
\\ 'IF' \ \text{bool\_expr(57)}
\\ 'THEN' \ (* \ \text{composition\_operation(25)} ';;' *)
\\ ['ELSE' (* \ \text{composition\_operation(25)} ';;' *)]
\\ 'FI'
\];
\]

This control structure provides additional conveniences for trace generation, including context condition check for filtering traces, trace annotation with messages, assertion checking.

See also ensure_request(42) and assertion_check(43), which are abbreviations based on the use of conditional\_composition.
(40) simple_action:
    (  `REJECT'  |  
    `MARK'    |  
    add_relation(34)  |  
    SHARE_clause(41)  |  
    ensure_request(42)  |  
    assertion_check(43)  |  
    SAY_clause(44)  |  
    attribute_assignment(45)  |  
    timing_attribute_adjustment(46)  )

;  

REJECT terminates the derivation for the current event trace or root/composite segment, and the generator proceeds to the next instance. This operation may be useful for early pruning in the derivation process. Typical use is demonstrated by the ensure_request(42). Sometimes you may just want to temporarily delete some traces from the browsing list.

MARK can be performed in a BUILD block of root or composite, or in schema’s body, and will ensure that the complete trace under derivation will be highlighted to make it distinct in the graph window navigation list. To MARK a trace in the process of model testing/debugging may be useful in order to indicate that it contains something of interest (more details could be provided in the messages produced by SAY_clause(44) and attached to the event boxes in the trace graph). For instance, MARK is used to highlight assertion counterexamples in assertion_check(43).

MARK performed in any BUILD block is propagated during the derivation, and traces including a MARKed root/composite segment also become MARKed, up to the top trace.

The Firebird’s graph window navigation list contains all MARKed and un-MARKed traces for which the derivation within the scope has been completed.

(41) SHARE_clause:
    `SHARE' variable(23) variable(23)

;  

This operation merges instances of two events. If event types are distinct, the operation fails, and the failure is propagated to the enclosing BUILD block or schema.

If both are atomic events of the same type, operation may succeed and event instances are merged into a single event instance inheriting all relations from each. For composite events both arguments should have the same structure of events inside them.

Yet another necessary condition for successful event sharing is the same event attribute values at the derivation moment when the SHARE_clause is performed, otherwise the operation fails.

There is one exception. Time attributes depend on basic relations and are recalculated automatically each time when basic relations of an event are adjusted in the derivation process. When a pair of events is SHAREd, their relations (basic and user-defined) are merged, and this may trigger time attribute recalculation.

The shared_composition(26) operation  
A, B SHARE ALL x;
can be expressed in terms of coordinate_composition(28) and SHARE_clause, as
COORDINATE $a: x$ FROM A,  $b: x$ FROM B
    DO SHARE $a$ $b$; OD;

The coordination operation above is applied to the default event sets. In most cases it is sufficient. With the event_reshuffling_option(31) in the COORDINATE it is possible to SHARE all permutations of event sequences, like in the following example.

COORDINATE $a: x$ FROM A,  <!> $b: x$ FROM B
    DO SHARE $a$ $b$; OD;

ensure_request:
    ‘ENSURE’ bool_expr(57)
    ;
ENSURE A;
    is an abbreviation for
    IF A THEN /* continue derivation */ ELSE REJECT; FI;

When used in the Build_block(51) of a composite event or root, or in the schema’s body(1) this action filters the trace derivation by applying context conditions to the already derived segment of event trace.

assertion_check:
    ‘CHECK’ bool_expr(57) ‘ONFAIL’ SAY_clause(44)
    ;
CHECK A ONFAIL SAY(“Message”); is an abbreviation for
    IF A THEN /* continue derivation */ ELSE SAY("Message"); MARK; FI;

Assertion checking may be included in the Build_block(51) or in the schema’s body(1), usually for model’s testing/debugging. If the trace segment under derivation violates the assertion, an instance of the trace including this segment will be MARKed (highlighted) and available for browsing with a message explaining the cause. If the property described by the assertion should be always satisfied, it is better to include it as ENSURE construct. CHECK is the most common pattern for assertion checking, providing a message associated with a root, composite event, or the whole trace. More detailed assertion checking for advanced model testing/debugging with message boxes bound to particular events in the trace can be achieved by using a combination of COORDINATE, ADD and SAY constructs (see Example 9).

SAY_clause:
    ‘SAY’ ‘(’
    (+
        ( expression(64) | interval_expression(71) | string_constant(56) | variable(23) )
    )
    ‘)’
    ;
SAY_clause generates messages (events) that are visible in the event trace graph as distinct boxes. It may be used to provide trace annotations. By default the message event is under IN relation with the root/composite or schema event, depending on where (in the BUILD block, or in schema’s body) it has been produced. Additional IN and PRECEDES relations may be added to the message event by placing SAY_clause as an argument in ADD operator, see add_relation(34) rule for details.

SAY_clause may render the output of queries evaluated on event trace instances. Expression(64) value is included in the message, where stand alone variable(23) yields an event name. Thus it becomes possible to enhance the event trace with data specific for this trace.

Examples of queries/comments
1) SAY ("Total number of send/receive within Task_A is " # (send | receive) FROM Task_A );
2) CHECK FOREACH $a: pop (#pop BEFORE $a < #push BEFORE $a)
   ONFAIL SAY("too many pop events before pop event ");

(45) attribute_assignment:
    [ event_instance(68) '.' ] attribute_name
    [ ( '+' | '-' | '*' | '/' |
        'max' | 'min' | 'AND' | 'OR' ) ]
    ':'
    ( expression(64) |
      bool_expr(57) |
      interval_expression(71) )
;

Type of the attribute on the left-hand side and type of the right-hand value should be compatible. Interval attribute can be assigned a number value. In this case the number N is extended into interval [N..N].

Compound assignment E.attr op:= val is an abbreviation for E.attr := E.attr op val, where op is a binary operation. Type compatibility requirements are the same as for plain assignment. Arithmetic operations ‘+’, ‘-’, ‘*’, ‘/’ require number or interval type, logic operations require bool, compound operations max and min are allowed for number type only.

Absent event_instance by default means THIS.

(46) timing_attribute_adjustment:
    'SET' event_instance(68) '.'
    ( 'start' | 'end' | 'duration' )
    'AT' 'LEAST' (expression(64) | interval_expression(71))
;
This operation is different from the attribute_assignment(45). Value assigned to the timing attribute may be changed by the automated recalculation if it is not consistent with timing attribute rules (see. Sec.0 for details). The values of start and end attributes may be automatically adjusted only forward (i.e. may only increase), and the value of duration attribute may only increase as a result of the automated recalculation. Number value N from expression(64) will be converted into interval [N..N].
5.7 Event attributes

Event attribute defines a binary relation between an event instance and a value. It is a functional relation, meaning that any event instance may have only one attribute value associated with it at any time moment. Attributes have pre-defined default values. Event attribute’s value may be modified using attribute assignment operations during the derivation.

(47) **attribute declarations:**

```
'ATTRIBUTES' '{'

(+ attribute_definition(48) ';' +)

'}'
```

Attribute declarations are global for the MP schema and may appear in several places within MP source code, but follow the rule “attribute has to be declared before used.” This means that the definition of attribute should appear in MP source code before the attribute name appears in an expression or in an attribute assignment operation.

(48) **attribute definition:**

```
attribute_type(50) (+ attribute_name(49) +',')
```

(49) **attribute name:** Id

(50) **attribute type:**

```
( 'number' | 'interval' | 'bool' )
```

Attribute of each type has specific default value.

- **number** has default value **0**
- **interval** has default value **[0..0]**
- **bool** has default value **false**

There are some pre-defined attributes, which cannot be overloaded.

Timing attributes are pre-defined for all events and maintained within MP trace generator. The default values are **[0..0]**, the same as for any **interval** attribute, but their values are automatically recalculated when basic relations are added during the trace derivation.

- **interval start**;
- **interval end**;
- **interval duration**;

Rules for timing attribute re-calculation are explained in Sec. 0 and in Sec. 6.3

The whole event trace can be considered as an event of executing the schema, and in addition to timing attributes has pre-defined attributes:

- **number trace id**;
- **number trace probability**;

Values of these attributes are calculated automatically and appear in the trace scroll bar on the right side of Firebird window.
5.8 Build blocks

Composition operations within BUILD block are performed immediately after the derivation from a grammar rule for an instance (segment) of root or composite event is completed. These may add new relations and events, reject the whole derived segment (depending on a context condition), add an annotation, or provide a result of a query (via SAY).

(51) Build_block:
    'BUILD' '{
      (* composition_operation(25) ';'; *)
    }
    ;

(52) relation_name: Id
    ;
    Relation names IN, PRECEDES, AFTER, BEFORE, CONTAINS, ENCLOSING, FOLLOWS, FROM are predefined and require that the objects bound by these relation are event instances.

(53) user_defined_relation: Id
    ;
    MP keywords (including pre-defined relation names, like IN, PRECEDES, BEFORE) cannot be used as user-defined relation names.

(54) integer_number: integer_constant
    ;
    integer_constant is a non-negative decimal integer, like 25, or may have unary minus preceding it, like -25

(55) float_number: float_constant
    ;
    float_constant has decimal point, like 24.5. It may have unary minus preceding it, like -24.5. Use of exponent also allowed, like 24E2 or 24.5E-2

(56) string_constant: "characters"
    ;
    Zero or more characters may be any printable ASCII characters, but escaped characters (like \n).

5.9 Expressions

(57) bool_expr:
    bool_expr1(58) (* ('->' | '<->') bool_expr1(58) *)
    ;
    '->' stands for the implication operator, and '<->' for the iff operator ('if and only if'). These are left associative operations: A -> B -> C is equivalent to (A -> B) -> C.

(58) bool_expr1: bool_expr2(59) (* 'OR' bool_expr2(59) *)
    ;

(59) bool_expr2: bool_expr3(60) (* 'AND' bool_expr3(60) *)
bool_expr3:
    'NOT' bool_expr3(60) | Argument for NOT operation should be of type bool.
    'true' |
    'false' |
    event_instance(68) navigation_direction(21)
    event_instance(68) |

to check event’s type
    ( variable(23) 'IS' event_name(7) ) |

expression(64)
the following part may be used for comparing number and interval values
    [ comparison_operation(63) expression(64) ] |

    ( 'MAY_OVERLAP' variable(23) variable(23) ) |

quantified_bool_expr(61) |

Boolean_aggregate_operation(75)

; navigation_direction(21) may be a relation name, or relational_expression(22) with reverse and/or transitive closure operations.

If one argument of comparison_operation is number N and another is interval, the number N is converted into interval [N..N].

attribute_reference should be an attribute of type bool.

MAY_OVERLAP $x \ $y is an abbreviation for NOT($x BEFORE $y OR $y BEFORE $x), meaning that events $x and $y may overlap in time, i.e. be concurrent, or under FROM relation.

quantified_bool_expr:
    quantifier(62)
    (+ variable(23) `:' selection_pattern(35)
        [ `FROM` event_instance(68) ]
    + `,')
    bool_expr(57)

; The scope of variable under the quantifier extends to the quantified_bool_expr body.
If the quantified_bool_expr appears in the BUILD block of root or composite event, root_name in the FROM clause is not allowed. It may be used in the schema’s body only. Absence of the FROM clause is equivalent to FROM THIS.

quantifier: (‘FOREACH’ | ‘EXISTS’) [ ‘DISJ’ ]

‘DISJ’ may be used when there are several variables under the same quantifier. It means that the Boolean expression bound by the quantifier is true only for a combination of disjoint events bound by the variables. Events are disjoint if they are not merged by SHARE ALL or SHARE operations.

FOREACH DISJ $a: A, $b: A, $c: A \mathcal{F} \text{ is equivalent to}
FOREACH $a: A, $b: A, $c: A ((\forall a, \forall b, \forall c \text{ are disjoint event instances}) \rightarrow \mathcal{F})

(’\rightarrow’ stands here for the implication operator)

and

EXISTS DISJ $a: A, $b: A, $c: A \mathcal{F} \text{ is equivalent to}
EXISTS $a: A, $b: A, $c: A ((\exists a, \exists b, \exists c \text{ are disjoint event instances}) AND \mathcal{F})

comparison_operation:
( ‘<’ | ‘<=’ | ‘==’ | ‘!=’ | ‘>=’ | ‘>’ )

expression:
additive_element (65)
(* (‘+’ | ‘-’) additive_element(65) *)

additive_element:
term(66) (* (‘*‘ | ‘/’) term(66) *)

term:
(integer_number(54) | float_number(55) | scope_metavaraible(15) | ‘(‘ bool_expr(57) ‘)’ | ‘-‘ term(66) | for number type values only
(‘max’ | ‘min’ )
(‘expression(64) ‘,’ expression(64) ‘)’ |
special_function(70) |
numerical_aggregate_operation(76) |
attribute_reference(67)
  the optional part is valid for interval values only
  [ '.' ('smallest' | 'largest' | 'len')) |
interval_constructor(68)
)
;

(67) attribute_reference:
  [ event_instance(68) '.' ] attribute_name(49)
;
  Absence of event_instance means THIS.

(68) interval_constructor:
  ['[ expression(64) '..' expression(64) ']' ]
;

(69) event_instance:
  ( root_name(4) | variable(23) | 'THIS' )
;
  THIS refers to the schema, or to the root or composite event in the BUILD block of which it appears.

(70) special_function:
  ( '#' selection_pattern(35)

    [ navigation_direction(21) event_instance(69) ] |

    '#' '{' thread(24) '}' -- number of events in the source thread
  )
;

  This function counts number of events in the selected set. The scope for selecting events for counting
  is FROM THIS, unless it is explicitly overloaded by navigation_direction. THIS refers to the schema,
  root or composite event in the BUILD block of which the expression appears. Root name for the
  navigation_direction clause cannot be used inside the root or composite event BUILD blocks, since
  root segments may not yet be available when composite segments for this event are derived. Absence
  of the navigation_direction clause is equivalent to FROM THIS. The second form with thread may be
  used when SUCH THAT condition is needed to filter out events for counting.

  The range for special function appearing in a BUILD block is limited to that composite or root event
  trace segment, since it will be calculated at the time when the composite or root event trace segment
  is derived, and the rest of trace is not yet available. The same consideration holds when the special
  function appears in the schema’s context – its range is constrained by the progress of derivation
  accomplished so far. If later composition operations add some events and relations to the trace under
derivation, these are not yet available when this particular special operation is performed.

The number of events counted by the special function does include only one instance of a shared event at this point in derivation process.

In a simple case of existential quantifier, when only the fact of event’s presence should be established, it is recommended to use special function, like (\#A FROM $x > 0), which is equivalent to the assertion “there exists at least one event A inside the event $x.”

If a value N of the type number appears in interval_expression(71) as term(66), it is converted into interval [N..N].

5.10 Aggregate operations

(75) Boolean_aggregate_operation:
( 'OR' | 'AND' )
'{' thread(24) 'APPLY' bool_expr(57) '}'
;

(76) numerical_aggregate_operation:
( 'SUM' | --numerical sum
'TIMES' | -- numerical product
'MAX' | -- numerical max
'MIN' | -- numerical min )
The aggregate operation returns a number or a bool value depending on the operation and APPLY clause. The main rationale for the use of aggregate operations is the presence of threads with SUCH THAT clause for event selection. Scope of the variable defined in the thread extends to the APPLY clause. See also special_function(70).

In fact, FOREACH $a: P FROM S \mathcal{F}$ is an abbreviation for $\text{AND}\{\text{\$a: P FROM S APPLY \mathcal{F}}\}$ Correspondingly, EXISTS $\text{\$a: P FROM S \mathcal{F}}$ is an abbreviation for $\text{OR}\{\text{\$a: P FROM S APPLY \mathcal{F}}\}$

The special_function(70), for instance, $\#\{\text{\$a FROM S SUCH THAT E}\}$ is an abbreviation for $\text{SUM}\{\text{\$x: \$a FROM S SUCH THAT E APPLY 1}\}$ as well.

Use of abbreviations (like quantified_bool_expr(61) and special_function(70)) improves readability, reduces typing, and supports optimizations for the event trace derivation implementation.

6. APPENDICES

6.1 Axioms for basic relations

Each of basic relations IN and PRECEDES defines a partial ordering of events.

Assume FROM is a transitive closure of IN, and BEFORE is a transitive closure of PRECEDES, as defined in relational_expression(22). Trace derivation procedure ensures the following ten axioms will be satisfied for any generated event trace. It is assumed that variables $a$, $b$, $c$ here match the $$EVENT pattern (36), which stands for any event type.

Mutual Exclusion

Axiom 1) FOREACH $\text{\$a, \$b (\$a \text{ BEFORE } \$b \rightarrow \neg (\$a \text{ FROM } \$b )})$
Axiom 2) FOREACH $\text{\$a, \$b (\$a \text{ BEFORE } \$b \rightarrow \neg (\$b \text{ FROM } \$a )})$
Axiom 3) FOREACH $\text{\$a, \$b (\$a \text{ FROM } \$b \rightarrow \neg (\$a \text{ BEFORE } \$b )})$
Axiom 4) FOREACH $\text{\$a, \$b (\$a \text{ FROM } \$b \rightarrow \neg (\$b \text{ BEFORE } \$a )})$

Non-commutativity

Axiom 5) FOREACH $\text{\$a, \$b (\$a \text{ BEFORE } \$b \rightarrow \neg (\$b \text{ BEFORE } \$a )})$
Axiom 6) FOREACH $\text{\$a, \$b (\$a \text{ FROM } \$b \rightarrow \neg (\$b \text{ FROM } \$a )})$

Irreflexivity for BEFORE and FROM follows from non-commutativity.

Transitivity

Axiom 7) FOREACH $\text{\$a, \$b, \$c (\$a \text{ BEFORE } \$b \text{ AND } \$b \text{ BEFORE } \$c) \rightarrow (\$a \text{ BEFORE } \$c)}$
Axiom 8) FOREACH $\text{\$a, \$b, \$c (\$a \text{ FROM } \$b \text{ AND } \$b \text{ FROM } \$c) \rightarrow (\$a \text{ FROM } \$c)}$

Distributivity

Axiom 9) FOREACH $\text{\$a, \$b, \$c (\$a \text{ FROM } \$b \text{ AND } \$b \text{ BEFORE } \$c) \rightarrow (\$a \text{ BEFORE } \$c)}$
Axiom 10) FOREACH $\text{\$a, \$b, \$c (\$a \text{ BEFORE } \$b \text{ AND } \$c \text{ FROM } \$b) \rightarrow (\$a \text{ BEFORE } \$c)}$

6.2 Relational expressions

For a binary relation R (basic or user-defined), $\sim R$ stands for the reversed R and $^\text{R}$ for the transitive
closure (the smallest relation that contains R and is transitive [Jackson 2006]).

For the convenience of navigation within the trace, besides of the basic relations IN and PRECEDES there are additional relation names for reversed and transitive closure of basic relations.

ENCLOSING is a reverse of IN, or ~IN,
FROM is a transitive closure of IN, or ^IN
CONTAINS is a reverse of FROM, or ~FROM, or ~(^IN),
FOLLOWS is a reverse of PRECEDES, or ~PRECEDES,
BEFORE is a transitive closure of PRECEDES, or ^PRECEDES,
AFTER is a reverse of BEFORE, or ~BEFORE, or ~(^PRECEDES).

Relational expressions are used in navigation_direction(21) and satisfy the following equivalencies (for both basic and user-defined relations). Here R stands for a relation name or a relational expression.

\(^{\neg}R\) is equivalent to \(~^{\neg}R\)
~\(~R\) is equivalent to \(R\)
~\(~^{\neg}R\) is equivalent to \(^{\neg}R\)

6.3 Timing attribute calculation algorithms

Event timing attribute values depend on the basic relations in which the event takes part. Each time a new basic relation IN or PRECEDES is added, timing attributes may need to be recalculated to ensure consistency with the rules explained in Sec. 2.14.2. Automated timing attribute re-calculation is performed during the derivation as needed.

Timing attributes are pre-defined as

```
  ATTRIBUTES{ interval start, end, duration; }
```

All timing attributes are assigned the default value [0..0].

The notation used:

- **E.start**, **E.end**, **E.duration** – values of timing attributes for the event E before re-calculation.
- **E.start’**, **E.end’**, **E.duration’** – values of timing attributes for the event E after re-calculation.
- **E.attribute’** := **expression** – step in the calculation algorithm assigning new value to the timing attribute.

Let A and B be interval values. Following functions are used for the time attribute calculations.

1. \(\text{imax}(A,B)\) stands for \([\text{max}(A\text{.smallest, B\text{.smallest}) .. max}(A\text{.largest, B\text{.largest})}]\)
   During the derivation process assignment operation \(\text{SET} \ E\text{.attribute AT LEAST} A\) is performed as:
   \(E\text{.attribute’} := \text{imax}(A, E\text{.attribute});\)

2. **shift(A, B)** performs "shifting" timing attributes of event B forward in time to ensure that \(A <= B\) holds.
   \(B\text{.start’} :=\)
   \[
   \begin{align*}
   \text{if(time\_start\_attributes[B].smallest < time\_end\_attributes[A].largest)} \{ \\
   \text{time\_start\_attributes[B] = time\_end\_attributes[A];} \\
   \text{B.end’ := B.start’ + B.duration;}
   \}
   \end{align*}
   \]

The following timing attribute recalculations presented in a pseudo-code and are performed as needed during the derivation.
Repeat until there are no more changes to timing attributes {
  For each pair of disjoint events E1 and E2 {

    if E1 BEFORE E2 then shift(E1, E2);

    if E1 CONTAINS E2 then {
      if E2.start.smallest < E1.start.smallest{
        // E2 may start before E1, E2 is shifted inside E1
        E2.start' := imax(E1.start, E2.start);
        E2.end' := E2.start' + E2.duration;
      }

      if(E2.end'.largest > E1.end.largest){
        // Ej may still end after Ei, even if Ej starts inside Ei
        // shift Ei.end and adjust Ei.duration, since Ej.start is already inside Ei

        // forward Ei.end to the E2.end, if needed
        E1.end' := imax(E1.end, E2.end');

        // recalculate E1 duration from E1.start and E1.end, extend E1.end.largest, if needed
        // let E1.duration' be [x..y],
        // and shift_right >= 0 be the extension of E1.end.largest, if needed
        // then the main invariant should hold:
        // (E1.end.smallest – E1.start.smallest) <= x <= y <=
        // (E1.end.largest + shift_right – E1.start.largest)
        dif1 := E1.end.smallest – E1.start.smallest;
        dif2 := E1.start.largest – E1.end.largest;
        shift_right := max((dif1 + dif2), 0);
        // if dif1 + dif2 is negative, no shift right is needed

        E1.duration' := [dif1 .. shift_right - dif2];
      }
    }

    if E1 EQUALS E2 then{
      // set all timing equal to max
      E1.start' := E2.start' := imax(E1.start, E2.start);
      E1.duration' := E2.duration' := imax( E1.duration, E2.duration);
      E1.end' := E2.end' := E1.start' + E1.duration' ;
    }
  }
} // end For
} // end Repeat

This description implies that the values of start and end attributes may be automatically adjusted only forward (i.e. may only increase), and the value of duration attribute cannot be decreased as a result of the automated re-calculation.

7. LISTS OF MP KEYWORDS AND META-SYMBOLS

7.1 MP keywords and pre-defined attribute names
These should not be used as event_name(7) or root_name(4).
7.2MP meta-symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$EVENT$$</td>
<td>Generic event type, matches any event type.</td>
</tr>
<tr>
<td></td>
<td>Used in Single_event_selection(36)</td>
</tr>
<tr>
<td>$$ROOT$$</td>
<td>Generic event type, matches any root event,</td>
</tr>
<tr>
<td></td>
<td>Used in Single_event_selection(36)</td>
</tr>
<tr>
<td>$$COMPOSITE$$</td>
<td>Generic event type, matches any composite event,</td>
</tr>
<tr>
<td></td>
<td>but not root events.</td>
</tr>
<tr>
<td></td>
<td>Used in Single_event_selection(36)</td>
</tr>
<tr>
<td>$$ATOM$$</td>
<td>Generic event type, matches any atomic event.</td>
</tr>
<tr>
<td></td>
<td>Used in Single_event_selection(36)</td>
</tr>
<tr>
<td>$$scope$$</td>
<td>Current scope. Used in iteration_scope(11), and in expression(64)</td>
</tr>
</tbody>
</table>

8. BRIEF RELATED WORK SURVEY
The following ideas of behavior modeling and formalization have provided inspiration and insights for this work.

Literate programming introduced by D.Knuth set the directions for hierarchical refinement of structure mapped into behavior, with the concept of pseudo-code and tools to support the refinement process [Knuth 1984].

[Campbell, Habermann 1974] and [Bruegge, Hibbard 1983] have demonstrated the application of path expressions as appropriate abstraction for program monitoring and debugging. In [Perry, Wolf 1992] path expressions have been used (semi-formally) as a part of software architecture description.

CSP (Communicating Sequential Processes) and other process algebras [Hoare 1985], [Milner 1989], [Roscoe 1997] provided a framework for process behavior modeling and formal reasoning about those
models, including the ideas of individual events, composite events, and event sharing. This behavior modeling approach has been applied to software architecture descriptions for connector protocol specification [Allen 1997], [Allen, Garlan 1997], [Pelliccione et al. 2009].

Rapide [Luckham et al. 1995a, 1995b] uses events and partially ordered sets of events (posets) to characterize component interaction.

“The backbone of the system model should be a hierarchy of activities, ... that capture the functional capabilities of the system - suitably decomposed to a level with which the designer is happy.” [Harel 1992]. Statecharts [Harel 1987] is an example of labeled transition system approach to the behavior modeling. It became one of the most common behavior modeling frameworks, integrated in the broader modeling and specification systems UML [Booch et al. 2000], and AADL [Feiler et al. 2006].

Coordination models and languages advocate separation of the interactional and the computational aspects of software components. Configuration and architectural description languages share these principles with coordination languages [Papadopolous, Arbab 1998], [Carriero, Gelernter 1992]. Separation of the component behavior from the coordination between behaviors in MP follows this principle and extends on it.

[Wang, Parnas 1994] proposed to use trace assertions to formalize the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior. The approach is based on algebraic specifications and term rewriting.

The ideas of Alloy Analyzer [Jackson 2006], in particular, the principles of immediate feedback for model design, Small Scope Hypothesis, relational logic, and model instance visualization, to mention a few, have provided a great inspiration for MP development.

The concept of software behavior models based on event grammars and event traces was also introduced in [Auguston 1991, 1995], [Auguston, Jeffery, Underwood 2002], [Auguston, Michael, Shing 2006] as an approach to software debugging and testing automation. The early drafts of Monterey Phoenix have appeared in [Auguston 2009 a, 2009 b] and [Auguston, Whitcomb 2012].

9. ACKNOWLEDGMENTS

Many people have contributed their time, effort, and insights to the development of MP language and to the implementation of the first MP tool prototypes. I am deeply grateful to all of them.

Kristin Giammarco and Cliff Whitcomb were instrumental in bringing Systems Engineering expertise, vision, and support for the project.

Ph.D. students Joey Rivera and Monica Farah-Stapleton developed MP modeling case studies. Joey Rivera and Alex Gociu designed the first version of Eagle6 with GUI for MP code editing, trace generation, and visualization.

The launch of Firebird public web app in Spring 2015 with its powerful GUI was a milestone in bringing MP to the system and software engineering community. The creative and novel solutions implemented in Firebird GUI by Philip McCullick, Michael Nigh, Mike Northcutt, Leah Frye, Noah Lloyd-Edelman, and Richard von Buelow have brought new approach to the ways how we use MP, and have set standards for MP implementation work.

The work of the Eagle6 team led by Joey Rivera, and including Marco Mason, Austin Meagher, Miroslav Lazarevic, Zoran Zamahajev, Boban Nikolic, and Ljubisa Jovev brought MP tools into the commercial realm.
Collaborative work with National University of Singapore team: Songzheng Song, Jiexin Zhang, Yang Liu, Jun Sun, Jin Song Dong, and Tieming Chen provided the first experience with MP model static analysis using PAT model checking tools [Jiexin Zhang et al. 2012], [Songzheng Song et al. 2014].

Several NPS faculty and students while using MP have provided valuable suggestions for MP improvements and new examples of MP models. Special thanks go to Bruce Allen, Troy Christensen, Katy Giles, Kenneth Hollinger, Richard Moebius, John Quartuccio, Michael Revill, David Shifflett.

Development of Monterey Phoenix was supported by funding from several sponsors:

- Naval Postgraduate School (NPS) Naval Research Program (NRP),
- Consortium for Robotics and Unmanned Systems Education and Research (CRUSER),
- Marine Corps Systems Command,
- Systems Engineering Research Center (SERC),
- Naval Air Systems Command (NAVAIR),
- Naval Air Weapons Station China Lake,  
- Naval Air Warfare Center Weapons Division,
- Office of Naval Research.

10. REFERENCES


HAREL, D., 1992, Biting the silver bullet: toward a brighter future for system development, IEEE Computer, Vol. 25(1), pp.8-20


