A Correlation Framework for the CORBA Component Model

Georg Jung, John Hatcliff, and Venkatesh Prasad Ranganath
{jung,hatcliff,rvprasad}@cis.ksu.edu

Department of Computing and Information Sciences, Kansas State University,
234 Nichols Hall, Manhattan, KS 66502, USA

Abstract. Large distributed systems, including real-time embedded systems, are increasingly being built using sophisticated middleware frameworks. Communication in such systems is often realized using in terms of asynchronous events whose propagation is implemented by an underlying publish/subscribe service that hooks components into a generic event communication channel. *Event correlation* – a mechanism for monitoring and filtering events – has been introduced in some of these systems as an effective technique for reducing network traffic and computation time. Unfortunately, even though event correlation is used heavily in frameworks such as ACE/TAO's real-time event-channel and in mission critical contexts such as Boeing's Bold Stroke avionics middleware, the industry standard CORBA Component Model (CCM) does not include a specification of event correlation. While previous proposals for event correlation usually offer sophisticated facilities to detect combinations in the stream of incoming events, they have not been constructed to fit within the CCM type system, and they offer relatively little support for transforming and rearranging filtered events into meaningful output events. In this paper, we present the design rationale, syntax, and semantics for a new and highly flexible model for event correlation that is designed for integration into the CCM type system. Our model has been integrated and tested in the Cadena development and analysis framework, which has been designed to support development of mission-control applications in the Boeing Bold Stroke framework.

1 Introduction

As software systems become more distributed, developers are increasingly turning to component-based development frameworks such as Java Enterprise Beans (EJB) and the CORBA Component Model (CCM) to manage the complexities associated with building and deploying distributed systems. A major advantage of such component based systems working on sophisticated middleware in general is the clear separation of concerns, which distinctly isolates the stages of the development process as well as it divides business logic from infrastructure, allowing to synthesize substantial parts of the implementation directly from the specification. Further, CCM as an established industry standard based on

CORBA, introduces system independence and a high level of interoperablity into the development process. As a result, component-based development is being explored in real-time safety/mission-critical domains as a mechanism for incorporating non-functional aspects such as real-time, quality-of-service, and distribution, permitting the developer to focus on application-specific parts.

Communication between components in such systems is often phrased in terms of asynchronous events whose propagation is implemented by an underlying publish/subscribe service that hooks components into a generic event communication channel. In these event services, *event correlation* – a mechanism for monitoring and filtering events – often plays a crucial role in reducing network traffic and computation time.

To illustrate, consider the common case where one component C receives events a and b from components A and B, respectively, and generates a new output event c which is synthesized in some way from a and b – specifically, C requires *both* a and b before it can generate its own output event. If component A issues events at a higher frequency than component B (as is often the case in real-time periodic systems), many a events will be discarded by C as it awaits for an accompanying b. Obviously a communication channel which is able to filter out such additional events saves logic and computation time in the receiving component and reduces network traffic in the system. Specifically, we would like the event communication layer itself to monitor the event flow and forward an a and b together to C only when both events have occurred. Depending on the complexity of communication this improvement is often substantial.

Unfortunately, even though event correlation is used heavily in frameworks such as ACE/TAO's real-time event-channel and in mission critical contexts such as Boeing's Bold Stroke avionics middleware, the CCM specification does not include a specification of event correlation. While previous proposals for event correlation usually offer sophisticated facilities to detect combinations in the stream of incoming events, they have not been constructed to fit within the CCM type system, and they offer relatively little support for transforming and rearranging filtered events into meaningful output events.

In this paper, we present a new and highly flexible model for event correlation that is simple in syntax and rich enough in features to specify complicated correlations. Increased flexibility is achieved by splitting an event correlator into two phases: first, a *filter phase* monitors the event flow for the desired event combination, then a second closely-interacting *transformation phase* disassembles input events and reassembles payloads from the input events into output events in a programmable manner. The splitting of correlation into these two phases (specifically, the introduction of the programmable transformation facility) allows our correlation framework to be tightly integrated with the event type system of CCM.

The contributions of the paper are as follows.

- We present a novel event correlation framework that decomposes event correlators into event filter and event transformer stages.

- We define a formal semantics for this correlation framework in terms of a language over event sequences.
- We show how the notion of event transformer allows for the first time a correlation framework to be incorporated into CCM and integrated with the CCM event type system.
- We describe how our correlation framework is implemented in the Cadena environment for development of high-assurance distributed systems, and how correlation specifications given at the modeling level are translated to implementations in the underlying middleware layers.
- We illustrate how our framework can be used to correlation problems that are representative of those in avionics applications but are more difficult to solve using previous frameworks.

These results remove barriers that previously prevented applications that relied heavily on correlation (such as those built in Boeing's Bold Stroke program) from being transitioned to a CCM framework where standardization and a richer deployment framework provide a variety of benefits.

The rest of this paper is organized as follows. Section 2 presents the syntax and semantics of event filter expressions Section 3 describes the event transformers. Section 4 illustrates how our correlation framework can be used to implement dynamic changes to correlations. Section 6 discusses related work, and Section 7 concludes.

2 Syntax and Semantics of Filter Expressions

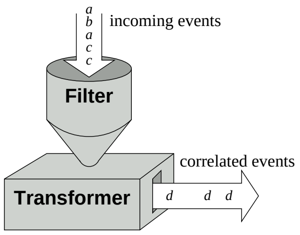
2.1 The Filter Syntax

Previous approaches build on atomic expressions which access the payload of an event and associate truth values according to whether the event with its attributes satisfies the atomic expression. While our model in general is independent from the actual form of the underlying atomic expressions, we chose to only consider the arrival of an event, since this strategy follows the concept of the CCM architecture in the sense that we connect typed source ports to typed sink ports from known entities, allowing the communication channel to have knowledge about the connections but leaving any assessment of the payload values other than rearrangement to the components of the system. The time of the issuing of the event (e. g. represented in the form of a timestamp attached to the event) as well as its source and its type are considered intrinsic properties of the event and hence visible to the correlator. Note that knowledge of the generation time of an event is implicitly assumed by all above mentioned previous works on event correlation [6, 7, 12, 10], otherwise the use of a sequential operator (see below) is infeasible. Accordingly, we can assign a single identifier to every source port connected to a correlator.

In the reminder of this section, we will assume that there are source components A, B, C, ... connected to the Event Channel (i. e. the communication channel provided by the middleware), and that these components issue the events

$$
\begin{array}{rcl}
\textsf{filter} & ::= & \textsf{sequence} \ (\ \textsf{||} \ \textsf{filter} \ )^{*} \\
\textsf{sequence} & ::= & \textsf{collection} \ (\ \textsf{;} \ \textsf{sequence} \ )^{*} \\
\textsf{collection} & ::= & \textsf{accumulation} \ (\ \textsf{|} \ \textsf{collection} \ )^{*} \\
\textsf{accumulation} & ::= & \textsf{atom} \ (\ \textsf{+} \ \textsf{accumulation} \ )^{*} \\
\textsf{atom} & ::= & (\mathit{label}\,\textsf{:})^{?} \ \mathit{event} \\
& | & (\mathit{label}\,\textsf{:})^{?} \ (\ \textsf{sequence} \ )
\end{array}
$$

(a) Filter Expression Grammar



(b) Filter/Transformer

***fig.* 1.** Filter Grammar and Role

a, b, c, $\ldots$ with the types τ_a, τ_b, τ_c, $\ldots$ respectively. As mentioned above, the payload of an event is not accessible to the filter, thus we can identify a finite set $\Sigma = \{a, b, c, \ldots\}$ of possible events occurring in an infinite sequence as input of the correlator by considering two events equal iff they come from the same source port. Similar to the other mentioned approaches, we use an expression which we call the *filter expression* to denote the subsequences which are of interest for the receiver of the correlated event.

The syntax of the filter expressions is closely related to that of previous approaches such as e. g. the ECL expressions in [10], the Event Composition Operators in [7] or the Policy language in [6]. It is further based on our assessment of the Boeing's Bold Stroke and SAE AADL (Avionics Architecture Description Language[1]) frameworks. In this approach we present three basic combinators and a parallel combinator. Informally the three combinators are (1) the *accumulation* of events, i. e. both of two events a and b have to occur, regardless of the order (written a+b), (2) the *collection* of events, i. e. at least one of two events a and b has to occur (written a|b), and (3) the *sequence* of events, i. e. of two events a and b both have to arrive in the given order (written a;b). An abbreviated grammar of the filter expressions is given in *fig.* 1(a). In this grammar, "$(\ldots)^{*}$" stands for zero or more and "$(\ldots)^{?}$" for zero or one instance of the item given inside the parenthesis. Note that all combinators are defined with arity two. While this does not impact the expressiveness of the combinators since they are associative [5], it greatly simplifies the formal definition of dynamic semantics given in section 4. Further, note that our approach enabled us to reduce the number of different combinators as compared to previous models without loosing flexibility. In fact, we believe that in our approach the expressions are more intuitive instead.

2.2 Semantics of the Three Basic Combinators

We now define the semantics of filter expressions in a way similar to regular expressions. This approach provides a firm formal background while leaving the

[1] See `http://www.sae.org/technicalcommittees/aasd.htm`

computational model for implementing a filter open to the choice of the programmer. Essential is the concept of a *match*, which we first define for a basic atomic expression composed of a single event:

Definition 1. *A sequence* $\boldsymbol{s} = e_1 \dots e_n$ *of events* $e_1, \dots, e_n \in \Sigma$ **matches** *a singleton filter expression* a*, iff an event* a *is in* $\boldsymbol{s}$*, i. e. there is an* i *with* $e_i = a$*.*

Next, we define the basic combinators `+`, `|` and `;`. The parallel operator `||` is discussed later.

Definition 2. *A sequence* $\boldsymbol{s}$ *of events* **matches** *a filter expression* $x_1 \oplus x_2$ *iff*

- $\oplus$ *is* `+` *and* $\boldsymbol{s}$ *matches* x_1 and $\boldsymbol{s}$ *matches* x_2*.*
- $\oplus$ *is* `|` *and* $\boldsymbol{s}$ *matches* x_1 or $\boldsymbol{s}$ *matches* x_2*.*
- $\oplus$ *is* `;` *and the sequence* $\boldsymbol{s}$ *can be split into two subsequences* $\boldsymbol{s_1}$ *and* $\boldsymbol{s_2}$ *such that* $\boldsymbol{s} = \boldsymbol{s_1} \cdot \boldsymbol{s_2}$ *and* $\boldsymbol{s_1}$ *matches* x_1 *and* $\boldsymbol{s}_2$ *matches* x_2*.*

To illustrate, consider the expression a+b and a;b. While the former is matched by the sequence *bbca*, the latter is not. Other matched expressions are e. g. b|d or b;a. Note that "+" and "|" are commutative, while ";" is not [5]. We will call the set $\mathrm{M}(x) = \{\boldsymbol{s} \in \Sigma^* \mid \boldsymbol{s} \text{ matches } x\}$ the set of matches of the expression x. Clearly, there are infinite sequences in the set of matches of an expression x. Nevertheless for the filter we are only interested in a notification whenever a match first is complete.

Definition 3. *A* **shortest match** *of an expression* x *is a sequence* $\boldsymbol{s}$ *of events such that* $\boldsymbol{s}$ *matches* x *but no proper prefix of* $\boldsymbol{s}$ *matches* x*.*

In analogy to regular languages we call the set of shortest matches of an expression x the language $\mathrm{L}(x)$ of x.

For example, *aab*, *aaab* and *aaaaaab* are shortest matches for the expression b with $a, b \in \Sigma$, *aaba* is a match, but not a shortest match. Shortest matches for the expression a+b|a+c are e. g. *ba*, *ac*, *bbcbca*, *aac*.

Sequences we can define with these three basic combinators are a subset of regular languages ([5]), hence it is obvious that we can construct an acceptor.

2.3 The Trigger

Definition 2 implies that for the combinators `+` and `|` both subexpressions have to be checked on the same sequence $\boldsymbol{s}$, which means that their acceptors are both executed in parallel. Consider the sequence *abbabaa*... and the expression a;b;a. Then there are two successive shortest matches in the sequence

$$\underbrace{abba}\,baa\dots \qquad \text{and} \qquad abb\,\underbrace{aba}\,a\dots$$

which are overlapping. In the event communication service though, we are interested in separate, non-overlapping occurrences of the correlation. We therefore define the notion of a *trigger* on the sequence $\boldsymbol{s}$.

Definition 4. *A shortest match of an expression x on a sequence $\boldsymbol{s}$ of events is a* **trigger***, iff it does not overlap with a previous trigger on $\boldsymbol{s}$.*

Note that at the beginning of a sequence there can be no previous trigger, hence the definition is well founded. The second match in the above example therefore it is not a trigger. Intuitively, a state based acceptor resets at the point $abba \downarrow baa\ldots$. We say a filter *triggers* whenever it completes a trigger.

2.4 The Parallel Combinator

There can be situations, in which all overlapping triggers are of interest to the receiver of a correlated event. Moreover though, as described in section 4, we want to be able to influence the behavior of a correlator without affecting ongoing computations of the filter. We hence introduce a parallel combinator `||`, which is exactly similar to the collection combinator `|` in its definition of a match and shortest match, but it allows overlapping matches:

Definition 5. *Let $\boldsymbol{s}$ be a sequence, T_1 be the set of triggers of expression x_1 on $\boldsymbol{s}$, and T_2 be the set of triggers of expression x_2 on $\boldsymbol{s}$. Then the set of triggers T for the expression $x_1 \,||\, x_2$ is the union $T_1 \cup T_2$ of the sets T_1 and T_2.*

Intuitively, the expressions x_1 and x_2 run independently from each other. Clearly it is not necessary to allow the use of the parallel combinator anywhere but in top level of the filter expression (this motivates the structure of the grammar in *fig.* 1(a)).

2.5 The Result of the Filter Evaluation

Consider the expression $a \,|\, b$ which triggers whenever a or b appear in the input sequence. In case of triggering we want to know which one of the events, a or b actually arrived, since the result of the correlation delivered by the transformer (see section 3) might depend on it. Therefore we introduce the notion of an *active* subexpression.

Definition 6. *Let $\boldsymbol{s}$ be a sequence of events, and the subsequence $\boldsymbol{s}'$ of $\boldsymbol{s}$ be a trigger of expression x on $\boldsymbol{s}$. A subexpression x' of x is called* **active** *iff $\boldsymbol{s}'$ matches x'.*

E. g. b+c is an active subexpression of the expression a+$c \,|\, b$+c on the trigger ccb, while the subexpression a+c is not. For the trigger ac though, a+c is active, while b+c is not. Finally, for the trigger abc every subexpression of a+$c \,|\, b$+$c$is active.

To make the result of a subexpression accessible to the transformer, it has to be marked with a label.

Definition 7. *A label l' attached to a subexpression x' of an expression x is* **active** *iff x' is active.*

To illustrate we label two of the subexpressions of the previous example as l_1:(a+c)|b+l_2:c. On the trigger ca the label l_1 is active, on the trigger bc not. Note that l_2 is active on every trigger of the expression.

Upon completion of every trigger the filter propagates the set of active labels as result to the transformer. Note that the filter does not terminate after the first trigger, but it continues to trigger throughout the sequence of incoming events.

2.6 The Concrete Syntax of the Correlation

We define the correlators offered by the middleware communication channel in a separate input file called *correlation library.*[2] For simplicity we designed the form of the correlation definition closely similar to usual method or procedure definitions in imperative languages:

output-type `correlation` *name* `(`*typed-identifier*$_1$, *typed-identifier*$_2$, ...`)`
filter-expression `{` *transformer* `}`

Here, the *output-type* is an event type defined in the CORBA IDL file[2] designating the type of the event sent out by this correlator. It is followed by the keyword `correlation` and an identifier *name* denominating the correlation. The events handled by this correlator are then provided as a list of typed identifiers which, in analogy to the parameters of a method, are later bound to event ports in the system assembly. The *filter-expression* defines the event subsequences upon which the correlation is supposed to trigger as an expression over the identifiers from the typed identifier list using combinators and labels in the syntax and semantics given previously in this section. Example:

```
Notification correlation AfterTimeout (TimeOut a, DataAvailable b)
                                  a ; b {...}
```

Here, the identifier `a` will be bound to a port issuing `TimeOut`-events, while `b` is bound to a `DataAvailable` port. The correlation triggers on every sequence which contains a `TimeOut` and later a `DataAvailable`-event. The parenthesis "`{...}`" stands for the transformer, which is discussed in the next section.

3 The Transformer

3.1 Outline of the Transformer

The transformer provides the second step of our two phase model. As indicated in section 2.5, the input for the transformer is a set of active labels delivered by the filter. The function of the transformer is to assemble a new event based on various of the available incoming events and to push that event to the receiver(s) whenever a trigger is complete. In section 4 we will assign further objectives to the transformer, namely the possibility to perform dynamic changes to the correlator's behavior, to add more flexibility.

[2] see section 5

```
     transformer ::= init ( case )*                       label-exp ::= disjunct ( | disjunct )*
            case ::= case label-exp : ( statement )*       disjunct ::= conjunct ( & conjunct )*
       statement ::= push event;                           conjunct ::= !conjunct
           event ::= identifier                                        | ( label-exp )
                     | new event-initial                               | label
   event-initial ::= type { attr-assignments }                         | true
attr-assignments ::= ( attribute := identifier.attribute )*
```

***fig.* 2.** Basic Transformer Grammar

The transformer has two parts. The first part is an initialization, for which the discussion will also be deferred to section 4. The second part consists of case clauses branching on boolean expressions over the labels, assigning the values true to active and false to inactive labels respectively.

3.2 The Basic Transformer Syntax

An abbreviated grammar for the transformer is shown in *fig.* 2. The substantial parts of the transformer are the `case`-clauses. Each case features a boolean expression over the labels. Upon a trigger, the body of *each* case, for which the associated label expression evaluates to *true* is executed.

For now there are three possible actions to take inside the body of a case clause: First, one of the events can be just propagated through, by giving the identifier of the event as argument to the `push` statement. A side condition is that the type of the event is a subtype of the declared output type of the correlation.Second, the correlator can assemble a new event of a given type, which again must be a subtype of the declared output type. The third, and at first sight trivial option is to do nothing at all. The section 3.4 though will show how this adds considerably to the possibilities of our approach.

3.3 The Transformer Output

For the assembly of an event we provide the `new` statement. It receives an event type defined in a separate file using the CORBA Interface Definition Language (IDL)[3] and a comma separated list of attribute assignments enclosed in curly brackets. For example the following lines define two event-types in IDL-syntax, where event `DataNotify` inherits from event `Notify`:

```
eventtype Notify {                 eventtype DataNotify : Notify {
   attribute short SourceID;          attribute float Value;
};                                 };
```

A transformer, given e. g. an input `DataNotify` event a, can assemble an output `Notify` event with the statement

```
push new Notify { SourceID = a.SourceID }
```

[3] See section 5.

```
case l1: push new NavData
  { air = a.air, nav = b.nav }
case l2: push new NavData
  { air = a.air, nav = c.nav }
```

(a) Send data from both events

```
case l1: push new NavData
  { air = a.air, nav = b.nav }
case l2 & !l1: push new NavData
  { air = a.air, nav = c.nav }
```

(b) Take only b if present

```
case true: push new DataAvailable {}
```

(c) Send general notification

***fig.* 3.** Handling a Double Match Trigger

We make use of the inheritance subtyping given by the IDL event declarations, i. e. whenever e. g. `Notify` is defined to be the event type of an incoming event, the correlator accepts events of type `DataNotify` or any other subtype, similarly it can send out events of any type which is subtype of the declared output type.

3.4 Examples

Double Match. In [11], Boeing engineers discuss correlations in the context of the Boeing Bold Stroke framework. In their example, a receiving component is interested in notifications from three different event sources, referred to as A, B and C. A correlation occurs whenever either both, component A and B, or both, component A and C have issued an event.[4] The filter expression describing this pattern is a+b|a+c, or equivalently a+(b|c). Consider the following stream of incoming events: $\ldots cba$. The sequence matches the expression, and hence is, assuming no previous overlapping trigger, a trigger for the filter. Note though, that the sequence is a match for both subexpressions a+b as well as a+c. Naturally, there must be a clear definition on how to handle this case. To discuss these, we label the subexpressions as follows: l_1:(a+b)|l_2:(a+c). *Fig.* 3 presents three different possibilities for the transformer to react to a trigger.

In *fig.* 3(a) the transformer sends an event containing data from the incoming event a if l_1 is active, or an event containing data from event b if l_2 is active. If both labels are active though, as is the case with the above mentioned trigger, the transformer of *fig.* 3(a) will generate *two* events. This complies with the policy informally described in [11].

Fig. 3(b) presents an alternative strategy, where the combination of event a with event b is favored over the combination of a and c. Again, this behavior is easy to specify by accessing the active labels: if l_1 is active the output is

[4] In one of the Bold Stroke examples, the *modal scenario*, this situation is given by two different steering queues, which correlate with a single air-frame as input to the combined display

assembled from b. Only if l_1 is not active, the transformer uses c. A behavior like this, although it suggests itself in many common situations, is extremely complicated to describe in any of the previous approaches.

In many cases, a component may not be interested in particular payload values arriving in correlated events, i.e., the component simply needs to know that a correlation trigger has occurred. In this case, the transformer can be constructed to simply output an event with empty payload as shown in *fig.* 3(c). Here, upon any trigger the same notification event is generated, regardless of the incoming events.

Interleaving Event. A primitive offered by other frameworks is the *non-interleaving* or *do-unless* correlation, expressed e. g. with `{e1;e2}!e3` in [7], or with $\mathbf{do}\{\phi_1\}\mathbf{unless}\{\phi_2\}$ in [10]. Common to these primitives is that some expression is pursued on the stream of incoming events until it is interrupted and reset by an interleaving event. For example, after the occurrence of `e1` in the above expression from the GEM framework, the correlator looks for `e2` or interrupts if `e3` comes in between. Similarly in the Stanford approach the expression ϕ_1 is evaluated in parallel with ϕ_2, and if ϕ_2 succeeds earlier, then the whole expression results in a *fail*.

Our model provides the same functionality without an additional primitive. Consider an expression x_0 which should not be interrupted by the completion of a second expression x_1. The filter expression $l_1 : x_0 \,|\, l_2 : x_1$ executes both expressions in parallel, resetting whenever either one triggers. As shown in *fig.* 4(a) this

```
case l1: push new event
```

(a) Prohibit interleaving

```
case l1: push b
case l2: push a
```

(b) Most recent event

***fig.* 4.** Interleaving Semantics and Most Recent Event

is fully sufficient to prohibit interleaving of the two subexpressions, simply by ignoring the label l_2. Unlike previous approaches though, we can even safely handle a case where, similar to the double match example, both expressions complete with the arrival of the same event, e. g. by explicitly requiring the label l_2 not to be active when sending out the result. This is done by replacing the label expression `l1` by the expression `l1 & !l2` in *fig.* 4(a). Further, whenever this expression is integrated as a subexpression into a larger context, it is easy to refer to either x_0 or x_1 by using the labels, e. g. to catch and handle the interleaving reset event, instead of ignoring it.

Most Recent. Consider a component interested in the accumulation of two events a and b. One possible filter which recognizes this accumulation is a+b. It is possible though to retrieve further information, e. g. about the order in which the events arrive. To achieve this, we expand and label the expression into an equivalent expression l_1:(a;b)|l_2:(b;a). Still, this filter triggers whenever both, a and b are present in the incoming stream of events. The transformer body shown in *fig.* 4(b) transfers the most recent of both events through to the subscriber.

4 Dynamic Changes

4.1 Changing Requirements

The communication structure in component based distributed systems is usually subject to dynamic modifications as it has to adapt to changes in the component's behavior as well as varying environmental properties. Especially the different operandi of the components referred to as *modes* cause frequent alterations to the communication. A correlator which is designed to work throughout the system's runtime has to offer possibilities to adjust both, filtering as well as transformation and propagation of events, to changing requirements.

In analogy to the modes of the components, some earlier approaches provide modes also for the correlator represented by mode guards, each of which in turn encapsulates a complete correlation definition. Nevertheless, according to our assessment of the component scenarios provided to us by the Boeing company, changes to the correlator rarely require an exchange of the whole definition by an entirely different one, instead the most common change is simply temporary absence of single events. Our approach aims to provide easy means to express this common scheme while still supporting more complicated dynamic adaption.

4.2 Cancelling Subexpressions

In our model, dynamic changes are realized by dropping or restoring subexpressions of the filter. We hence introduce two different states for each subexpression, *alive* and *aborted*. Similarly to the filter semantics given in section 2, we identify a label with the subexpression marked by the label, i. e. a label is aborted iff the marked expression is aborted and alive iff the marked expression is alive.

Definition 8. *Let x_a be an* **aborted** *expression. Then for any filter expression x_0 and any combinator $\oplus \in \{$;, |, +, ||$\}$ we have that* $\mathrm{L}(x_a \oplus x_0) = \mathrm{L}(x_0 \oplus x_a) = \mathrm{L}(x_0)$.

In short, an aborted subexpression will simply be ignored by the filter. Note though, that for different combinators this definition has different implications, e. g. in an accumulation an aborted subexpression will be treated as "always present" while in a collection it will be treated as "never occurring".

Similarly to an aborted expression, any literal containing the attached label is also ignored in the label expressions guarding the cases in the transformer.

```
      init ::= ( commuter-stmt; )*        commuter-stmt ::= abort (label (, label)* )
statement ::= push event;                                  | revive (label (, label)* )
            | commuter-stmt;                               | toggle (label (, label)* )
```

***fig.* 5.** Transformer Grammar Extensions

Definition 9. *Let l_a be an* **aborted** *label. Then for any label expression x_0 and any boolean operator $\oplus \in \{\texttt{\&}, \texttt{|}\}$ we have that $\lambda_a \oplus x_0 \equiv x_0 \oplus \lambda_a \equiv x_0$, where λ_a is either l_a or $!l_a$. An empty expression evaluates to* `false`.

Analogous to the filter expressions, this definition interprets an aborted label's literal as `true` in a conjunction and as `false` in a disjunction.

4.3 Additions to the Filter Syntax

Fig. 5 shows the extensions to the transformer grammar which enable the dynamic features of the correlator.[5] With the statements `abort` (l) and `revive` (l) label l and the subexpression marked by l can be dropped and restored, the `toggle` (l) statement switches between abort and alive state. By default every label is initially alive. Whenever a label is supposed to be in abort state initially, it has to be switched off in the initialization part of the transformer.

4.4 Simulation of a Mode-Based Approach

Consider a mode expression from [10] with n mode guard expressions $g_1, \dots g_n$ and enclosed correlator expressions $x_1, \dots x_n$:

$$\textbf{in } (g_1) \textbf{ do } \{x_1\} \qquad \dots \qquad \textbf{in } (g_n) \textbf{ do } \{x_n\}$$

We can easily simulate this functionality with the following filter expression:

$$m_1\texttt{:}\ x_1\ \texttt{|}\ \dots\ \texttt{|}\ m_n\texttt{:}\ x_n\ \texttt{||}\ l_1\texttt{:}\ g_1\ \texttt{|}\ \dots\ \texttt{|}\ l_n\texttt{:}\ g_n$$

and the following transformer

```
{ abort (m1, ... mn);
  case l1: abort (m1, ... mn); revive (m1);
  ...
  case ln: abort (m1, ... mn); revive (mn);
  internal cases of the different expressions }.
```

Therefore the mode based approach offers no expressive power beyond our approach.

[5] Note the basic grammar in *fig.* 2.

4.5 Example: Dropping an Unreliable Source

A common task for a correlator in Distributed Realtime Environment (DRE) applications is to accumulate all incoming events from e. g. a redundant sensor array and to send a combined event to the receiving component. We assume four sources referred to as A, B, C and D, with the events a, b, c and d. We further assume that a controlling component is able to determine the logical validity [9] of the data issued by the sources, and reacts to invalid data by issuing a shutdown event, telling the correlator to ignore the corresponding source. The filter expression hence is

```
ma:a + mb:b + mc:c + md:d || la:ca || lb:cb || lc:cc || ld:cd
```

Accordingly, the transformer is

```
{ case la: abort (ma);
  case lb: abort (mb);
  case lc: abort (mc);
  case ld: abort (md);
  case ma & mb & mc & md: push new DataAvailable {} }
```

If we want to enable the controlling component to be able to revive the dropped sources again, we can similarly use `toggle` instead of `abort`. Note though, that to cover every possible case with n redundant sensors a mode based approach as proposed by [10] would need up to 2^n different modes to achieve the same functionality. We are not aware of any support for dynamic modification of the correlator's behavior in previous works other than the above cited mode based proposal from Stanford.

5 Use in Cadena

We implemented the correlation framework described here in Cadena, a development environment we built for constructing distributed systems using the CORBA Component Model [4]. Cadena provides a wide range of support for modeling, analysis, and automatic code generation including facilities for specifying CCM component interfaces using CCM IDL, editors for allocating component instances and constructing connections between these instances, specifying component attributes, various forms of architectural-level analysis such as model slicing and model-checking against temporal specifications, and the ability to generate component code from IDL specifications using existing CCM implementations such as *OpenCCM*[8] (generating Java code) and *CIAO*[1] (generating C++ code). Cadena is implemented in IBM's *Eclipse*[3] open-source integrated development environment.

In Cadena, development begins by modeling components using CCM IDL which defines the external interfaces of components. Cadena provides an additional *component property specification* (`.cps`) file that is used to various lightweight semantic properties of a component including dependences between

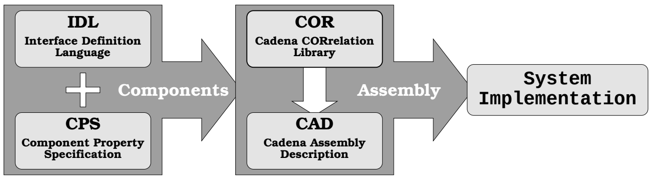


***fig.* 6.** The Elements of a CADENA Specification

actions on a component's ports and an abstract transition semantics for the component (see *fig.* 6).

Once components are modeled, a "system layout" is constructed in which instances of components are allocated and component ports of these instances are hooked together. Component instance connection information is held in *component assembly description* file. As *fig.* 6 illustrates, correlations specified in `.cor` files can also be captured in the model. Various forms of static analysis and behavioral checking of the model work on internal representations formed from the component IDL, `.cps`, `.cad`, and `.cor` files.

Below the modeling level, component implementations are generated from CCM IDL compilers. Not only do these compilers generate the standard CORBA stubs and skeletons, they also generate a substantial amount of the code required to implement component infrastructure such as functionality for connecting ports, communicating events, and otherwise support interaction with a component's context.

In part of our larger project on development of CADENA infrastructure, other researchers at Kansas State have developed a flexible *Event Communication Framework (ECF)* [2] to augment the basic event propagation mechanisms of CCM and the lower CORBA layers. ECF is targeted towards optimizing communication between peers in a distributed system based on middleware in the presence of *event correlation* and *data replication*. Since ECF is independent of implementation (e.g., the underlying ORB or implementation language) and architecture (e.g., the topology of a particular application), it can be used seamlessly in any CCM-based application. CADENA processes correlator definitions in the `.cor` files along with connection information in the `.cad` file and synthesizes correlator implementation code to be linked into the CORBA communication layers and component container and server implementations. In this phase, as an optimization the framework may break a correlator into parts that realize subexpressions of the filter expression depending on the locality of components producing events occurring in the subexpressions. We refer to such subexpressions as *node local subexpressions*. This reduces network traffic by contributing internode network traffic only when a node local subexpression is satisfied. Likewise, as the mode of a component can affect the consumption/production of events by that component, by propogating this information upstream/downstream along the event flow path further optimization can be achieved. At present, ECF can use this information to dynamically configure the correlator, as described in

Section 4, depending on the specified behavior and the usage context of the correlation. This can contribute to reduction in network traffic in a direct or cascading fashion.

Boeing's Open Experimental Platform (OEP) provides a set of scenarios which are representative of how DRE systems are built and used. With the above mentioned features, we have successfully realized all event correlations that occur in the scenarios in the above OEP. As the Boeing OEP is based on Boeing Bold Stroke avionics middleware (built on top of ACE/TAO real-time middleware), the handling of event correlations in a manner that is compatible with CCM specifications has removed one of the major obstacles in migrating the OEP into CCM from the non-standard component model which forms the basis of the concurrent Bold Stroke implementation and which lacks a substantial amount of infrastructure (e.g., deployment facilties and IDL code generation) that is provided by CCM.

6 Related Work

Our work was inspired from the early stages by previous work on event correlation from the GEM project [7] and a group from Stanford [10] also working on Boeing Bold Stroke. The Stanford model provides a rigorous formal background by defining the semantics of their definition language using automata similar to finite automata, called *correlation machines*. While this approach gives a convenient base for an implementation, we abstained from binding our semantics definition to a particular computational model, mainly because the implementation itself (part of our KSU colleagues work on ECF) continues to evolve as we work with middleware experts from the CIAO/ACE/TAO teams from Vanderbilt and Washington Universities to adapt the standard CCM event infrastructure into a form that is better suited for real-time applications. In particular, this effort is investigating, e.g., computation time and network traffic optimizations using partial pre-evaluation of correlations close to the event source.

The Reflex framework [6] provides a fully implemented correlation engine, together with a specification language called *Policy Definition Language*. Reflex generates C++ correlators intended to monitor network events in an internet like structure. Accessible semantic definitions are informal though. Note, that CCM uses the notion of events primarily for communication, while Reflex and other work concentrates on events as a means to monitor system behavior by some kind of controller.

7 Conclusion

We have presented a flexible framework for event correlation based on the two-phased approach of event filters and event transformers. Our directions within the work have been heavily influenced by the special needs of complex high-reliability real-time systems, and our desire to be able to develop such systems using the CCM framework. While previous approaches define the complete correlation in a single expression, we believe that employing our two phase model eases the assembly of meaningful output significantly.

Not only does the separate transformer stage make it easier for the framework to fit into the CCM type system, it also lends itself to a variety of interesting capabilities required for the real-time domain. For example, in collaboration with other middleware researchers, we are investigating approaches for attaching various real-time and quality of service properties to events. Given these extensions, transformers are also used to transform priorities, expiration times, and to implement drop strategies for events. For further development of our model, policies about event overwriting are being investigated, e. g. for events carrying sensory data the most recent event has priority over previous, while for an error message the earliest occurrence is more important. In addition, we are investigating how such policies may be captured in an extended event type system.

References

1. Component-integrated ACE ORB, a C++ implementation of CCM. Available at http://www.cse.wustl.edu/~nanbor/projects/CIAO/.
2. Event communication framework. Available at http://neo.projects.cis.ksu.edu.
3. Eclipse, an open extensible ide and tool platform written in java. Available at http://www.eclipse.org.
4. J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03)*, May 2003.
5. G. Jung, J. Hatcliff, and V. P. Ranganath. A correlation framework for the CORBA component model. Technical Report 03-9, Kansas State University, Department of Computing and Information Sciences, 2003.
6. S. Louvau, D. Chen, S. Jackson, P. Devanbu, and M. Gertz. Reflex – the customizable event correlation system. http://reflex.cs.ucdavis.edu/.
7. M. Mansouri-Samani and M. Sloman. Gem: A generalised event monitoring language for distributed systems, 1997.
8. OpenCCM, a Java implementation of CCM. Available at http://openccm.objectweb.org.
9. R. J. Richards, G. W. Daugherty, D. A. Haverkamp, and C. B. Netto. Middleware-based automatic source selection. Rockwell Collins internal, September 2002.
10. C. Sánches, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dil, and Z. Manna. Event correlation: Language and semantics. In *Proceedings of EMSOFT'03*, volume 2855, pages 323 – 339. ACM, Springer, September 2003.
11. D. Sharp. Challenge problems for the model-based integration of embedded software weapon system open experimental platform. Part of the Boeing OEP software., July 2001.
12. H. Sipma. Event correlation: A formal approach. Technical Report Draft, Stanford University, July 2002.