Material in Response to Comments on: "MuSE Graphs for Flexible Distribution of Event Stream Processing in Networks"

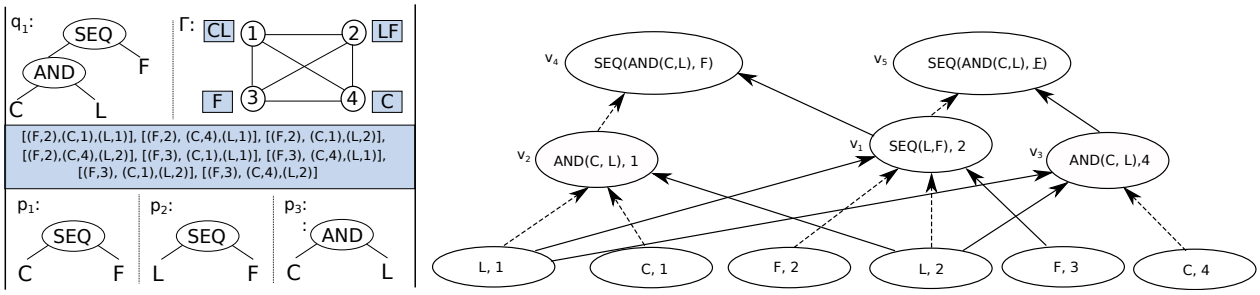
November 12, 2020

1 Additional Overview of Formal Notation

Table 1: Overview of the most important notations of the MuSE graph model.

Notation	Explanation
$\mathfrak{E}(\Gamma, q)$	Set of event type bindings of a query q in a network Γ
$\Pi(q), \pi(q, \mathcal{E})$	Set of all possible projections of q, and projection of q with respect to set of event types $\mathcal{E}$
$G = (V, E, c)$	Multi-Sink Evaluation (MuSE) graph: vertices V, edges E, edge weights $c : E \rightarrow \mathbb{R}$
$c(G)$	Sum of all edges weights of MuSE graph G
$\mathfrak{A}(v)$	Event type bindings covered by vertex v
$c_p(v)$	Placement costs of a vertex $v \in V$ of a MuSE graph $G = (V, E, c)$
$\hat{r}(p)$	Output rate of a projection p
$\mathfrak{c} = (\mathfrak{V}, \beta)$	Combination graph with projections $\mathfrak{V}$ and predecessors $\beta : \mathfrak{V} \nrightarrow \mathfrak{V}^k$
$\mathfrak{C}(q)$	Set of all combinations of a query q

2 Adapted Running Example



(a) (b)

Figure 1: (a) Query q_1 to be evaluated in network Γ (top), with its event type bindings (middle), and query projections p_1 - p_3 (bottom). (b) MuSE graph for the example. Dashed edges have a weight of zero as the respective operators are hosted by the same node.

3 Further Details to Include for the Case Study

3.1 Specification of the Used Queries

```
Query 1:
  PATTERN SEQ(Fail f, Evict e, Kill k, Update1 u)
  WHERE f.uID=e.uID ∧ e.uID=k.uID ∧ k.uID=u.uID
  WITHIN 30min

Query 2:
  PATTERN AND(Finish fi, Fail fa, Kill k, Update1 u)
  WHERE fi.jID=fa.jID ∧ fa.jID=k.jID ∧ k.jID = u.jID
  WITHIN 30min

Query 3:
  PATTERN SEQ(Schedule s, Fail f, Evict e, AND(Update1 u1, Update2 u2))
  WHERE s.jID=f.jID ∧ f.jID=e.jID ∧ e.jID=u1.jID ∧ u1.jID = u1.jID
  WITHIN 30min
```

3.2 Ambrosia Background

Ambrosia [16, in the paper] is a framework for resilient distributed computing implemented in C#. It aims at providing fault-tolerance for distributed applications that typically run in a Cloud environment. Ambrosia offers 'virtual resiliency' by encapsulating the application code running at each cluster node in a so-called immortal. The latter can be seen as a wrapper that handles check-pointing of the application's state and materializes all function calls sent and received by the application in a log. As such, applications running in Ambrosia benefit from comprehensive replayability under exactly-once semantics for all function calls.

3.3 Implementation Details

We partitioned the dataset which comprises the traces that comprise monitoring information of around 12.3k machines randomly into 20 sets. Based thereon, we generated event streams for a network consisting of 20 nodes. Each node generates primitive events based on the respective event streams and evaluates a set of assigned query projections. Based on a given MuSE graph, a node retrieves the set of projections to evaluate locally as well as a protocol that dictates the event flow in the network.

We used an automata-based approach for the evaluation of projections. The input of an automaton for a given projection p can contain arbitrary sub-projections of p. Due to the distribution, results of these sub-projections may arrive in arbitrary order. Hence, we constructed the automatons such that at each state, the result of each sub-projection that is still required to be processed, can actually be evaluated. Constraints on the order of the results of sub-projections are checked at guards assigned to each transition of the automaton.

Using the standard Ambrosia interfaces, each node is implemented by an immortal. To provide seamless recovery of a node failure, the state of each node contains, among others, its current input queue and the set of partial matches.

In the case study, each node was capable of generating each event type. Hence, each multi-sink placement contained all nodes and the number of input events for projections with multi-sink placements was evenly balanced between the nodes. This resulted in a significant difference in the event time latency compared to single-sink placements.

4 Algorithms to Construct MuSE Graphs

4.1 Exhaustive Search to Construct an Optimal MuSE Graph

Algorithm 2: Optimal MuSE Graph Generation

input : A query $q = (O, \lambda, P)$; an event sourced network $\Gamma = (N, f, r)$
output : An optimal MuSE graph G_{opt}

$C \leftarrow \emptyset$ // Set of all combinations at all placements
$\mathcal{G}_{func} \leftarrow \emptyset$ // Set of MuSE graphs as functions
$\mathcal{G} \leftarrow \emptyset$ // Set of MuSE graphs
for $\mathfrak{c} = (\mathfrak{V}, \lambda) \in \mathfrak{C}(\mathfrak{q})$ **do** // For each combination
  $\mathfrak{V}_\mathfrak{c} \leftarrow \mathfrak{V} \setminus \mathfrak{V}_\mathfrak{p}$
  for $placement \in N^{\mathfrak{V}_\mathfrak{c}}$ **do)** // For each possible placement of the projections $\mathfrak{V}_\mathfrak{c}$
  $C \leftarrow C \cup \{(\mathfrak{c}, placement)\}$
$\mathcal{G}_{func} = C^{\mathfrak{E}(q,\Gamma))}$ // Set of all assignment of event type bindings of query q in network Γ to C
for $m \in \mathcal{G}_{func}$ **do** // For each MuSE graph
  $V \leftarrow \emptyset$
  $E \leftarrow \emptyset$
  for $e \in \mathfrak{E}(\Gamma, q)$ **do** // For each event type binding
    $(c = (V, \beta), p) \leftarrow m(e)$
    $V_e \leftarrow \{(proj, n) \mid proj \in \mathfrak{V} \wedge n \in placement(proj)\} \cup e$
    $E_e \leftarrow \{((proj_1, n_1), (proj_2, n_2)) \mid proj_1, proj_2 \in \mathfrak{V} \wedge (proj_1, proj_2) \in \beta\}$
    $V \leftarrow V \cup V_e$
    $E \leftarrow E \cup E_e$
  $c \leftarrow computeEdgeWeights(V, E, P, r)$
  $\mathcal{G} \leftarrow \mathcal{G} \cup \{(V, E, c)\}$
$c_{min} \leftarrow \infty$
$G_{opt} \leftarrow (\emptyset, \emptyset, \emptyset)$
for $G \in \mathcal{G}$ **do**
  if $c(G) \leq c_{min}$ **then**
    $c_{min} \leftarrow c(G)$
    $G_{opt} \leftarrow G$
return G_{opt}

The runtime complexity of Alg. 1 is given by $\mathcal{O}\left(\left(2^{2^{|O_p|}} \cdot |N|^{2^{|O_p|}}\right)^{N^{|O_p|}}\right)$.

4.2 aMuSE/aMuSE*

Algorithm 3: Beneficial Projection and Combination Enumeration

input : A query $q = (O, \lambda, P)$; an event sourced network $\Gamma = (N, f, r)$
output : A set of beneficial projections Π_{ben} and a function C mapping beneficial projections to combinations

$\Pi_{ben}, C \leftarrow \emptyset$ // Beneficial projections and mapping of them to combinations
for $\mathcal{E} \in \mathcal{P}(O_p)$ **do** // For each subset of primitive event types
  $c_{proj} \leftarrow 0$
  for $e \in O_p^{\pi(q,\mathcal{E})}$ **do** $c_{proj} \leftarrow c_{proj} + r(e) \cdot |\{M \mid M \subseteq N \wedge m \in M \Rightarrow e \in f(m)\}|$
  if $\hat{r}(\pi(q, \mathcal{E})) < c_{proj}$ **then** // If the output rate is smaller than the sum of input rates
    $\Pi_{ben} \leftarrow \Pi_{ben} \cup \{\pi(q, \mathcal{E})\}$ // Add to set of beneficial projections
for $q \in \Pi_{ben}$ **do** // For each beneficial projection
  $C \leftarrow (q, \texttt{generateCombinations}\,(q, \Pi_{ben}))$ // Generate all combinations
return C, Π_{ben}
function `generateCombinations`(q, Π_{ben})
  $C_{proj} \leftarrow \emptyset$
  $\Pi_{sub} \leftarrow \texttt{generateSubProjections}\,(q, \Pi_{ben})$
  for $C_{poss} \in \mathcal{P}(\Pi_{sub})$ **do** // For each subset of projections
    if `isCorrectCombination` (C_{poss}, q) **then** $C_{proj} \leftarrow C_{proj} \cup \{C_{poss}\}$
  return C_{proj}
function `generateSubProjections`(q, Π_{ben})
  $\Pi_{sub} \leftarrow \emptyset$
  for $q_{sub} \in \Pi_{ben}$ **do**
    if $O_p^{q_{sub}} \subset O_p^q$ **then**
      if $q_{sub} = \pi(q, O_p^{possSubProj})$ **then** $\Pi_{sub} \leftarrow \Pi_{sub} \cup \{q_{sub}\}$
  return Π_{sub}
function `isCorrectCombination`(C_{poss}, q)
  $O_{prim} \leftarrow \emptyset$
  for $q_{sub} \in C_{poss}$ **do**
    if $(O_p^{q_{sub}} \not\subseteq O_{prim})$ **then** $O_{prim} \leftarrow O_{prim} \cup O_p^{q_{sub}}$
    else return $False$
  if $O_{prim} = O_p^q$ **then return** $True$
  else return $False$

The runtime complexity of Alg. 3, deriving beneficial projections and enumerating all combinations, is given by $\mathcal{O}\left(2^{|O_p|} \cdot 2^{2^{|O_p|}}\right)$. The total runtime complexity of aMuSE is given by $\mathcal{O}\left(2^{|O_p|} \cdot 2^{2^{|O_p|}} \cdot |O_p|^4\right)$.