Chapter 11
Quality of Service-Aware Complex Event Service Composition in Real-time Linked Dataspaces

Feng Gao and Edward Curry

Keywords Complex event processing · Quality-of-service · Modelling · Service composition · Dataspaces · Internet of Things

11.1 Introduction

The proliferation of sensor devices and services along with the advances in event processing brings many new opportunities as well as challenges for intelligent systems. It is now possible to provide, analyse, and react upon real-time, complex events in smart environments. When existing event services do not provide such complex events directly, an event service composition may be required. However, it is difficult to determine which event service candidates (or service compositions) best suit users' and applications' quality-of-service requirements. A sub-optimal service composition may lead to inaccurate event detection and lack of system robustness. In this chapter, we address these issues by first providing a Quality-of-Service (QoS) aggregation schema for complex event service compositions, and then developing a genetic algorithm to create near-optimal event service compositions efficiently. The approach is evaluated with both real sensor data collected via Internet of Things services and synthesised datasets.

The chapter is organised as follows: Sect. 11.2 introduces the technical aspects of complex event processing within dataspaces, including the design of the service, pay-as-you-go service levels, and its life cycle. Section 11.3 presents the Quality-of-Service (QoS) model we use and the QoS aggregation rules we define. Section 11.4 presents the heuristic that enables QoS-aware event service compositions based on Genetic Algorithms. Section 11.5 evaluates the proposed approach. Section 11.6 discusses related works in QoS-aware service planning, and Sect. 11.7 concludes and details future work.

E. Curry, *Real-time Linked Dataspaces*,
https://doi.org/10.1007/978-3-030-29665-0_11

11.2 Complex Event Processing in Real-time Linked Dataspaces

Real-time data sources are increasingly forming a significant portion of the data generated in the world. This is in part due to increased adoption of the Internet of Things (IoT) and the use of sensors for improved data collection and monitoring of smart environments, which enhance different aspects of our daily activities in smart buildings, smart energy, smart cities, and others [1]. To support the interconnection of intelligent systems in the data ecosystem that surrounds a smart environment, there is a need to enable the sharing of data among intelligent systems.

11.2.1 Real-time Linked Dataspaces

A data platform can provide a clear framework to support the sharing of data among a group of intelligent systems within a smart environment [1] (see Chap. 2). In this book, we advocate the use of the dataspace paradigm within the design of data platforms to enable data ecosystems for intelligent systems.

A dataspace is an emerging approach to data management which recognises that in large-scale integration scenarios, involving thousands of data sources, it is difficult and expensive to obtain an upfront unifying schema across all sources [2]. Within dataspaces, datasets *co-exist* but are not necessarily fully integrated or homogeneous in their schematics and semantics. Instead, data is integrated on an *as-needed* basis with the labour-intensive aspects of data integration postponed until they are required. Dataspaces reduce the initial effort required to set up data integration by relying on automatic matching and mapping generation techniques. This results in a loosely integrated set of data sources. When tighter semantic integration is required, it can be achieved in an incremental *pay-as-you-go* fashion by detailed mappings among the required data sources.

We have created the Real-time Linked Dataspace (RLD) (see Chap. 4) as a data platform for intelligent systems within smart environments. The RLD combines the pay-as-you-go paradigm of dataspaces with linked data, knowledge graphs, and real-time stream and event processing capabilities to support a large-scale distributed heterogeneous collection of streams, events, and data sources [4]. In this chapter, we focus on the complex event processing support service of the RLD.

11.2.2 Complex Event Processing

Complex Event Processing (CEP) detects composed/complex events from real-time data streams according to predefined *Event Patterns*. It is a key enabling technology for smart cities [271], due to the inherent dynamicity of data and applications.

However, within a dataspace [19], there is a multitude of heterogeneous event sources to be discovered and integrated [272]. This poses a classic source selection problem where it is crucial to determine which event services should be used and how to compose them to match non-functional requirements defined by users or applications [273]. The problem of source selection for dataspaces is also tackled in Chap. 15.

Consider an intelligent travel-planning system using traffic sensors deployed in a city, for example, traffic sensors in the city of Aarhus, Denmark, as shown in Fig. 11.1. The events produced by the sensors are all made available in a dataspace for smart city data. A user (say "Alice") might need to plan her trip based on the current traffic condition and would like to keep monitoring the traffic during her travel. Her request may form an event request with an event pattern shown as an *Event Syntax Tree (EST)* in Fig. 11.2. Another user (say "Bob") might need to deploy a long-term event request monitoring the traffic condition in his neighbourhood, and thus form a similar event request. In both cases, multiple sensors are involved, their observations are aggregated to produce complex events with coarse-grained information, and the users may have non-functional requirements for those events, for example, having an accuracy above some threshold or a latency below a specific value. Moreover, one complex event might be useful for different event requests, that is, complex events are reusable. Thus, addressing the users' functional and non-functional requirements efficiently and effectively in this context needs to

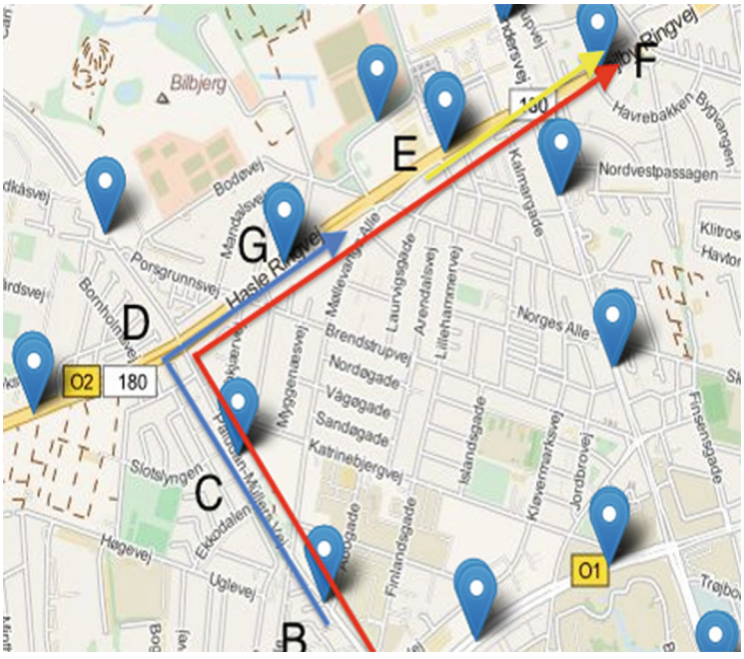

Fig. 11.1 Traffic sensors in Aarhus City

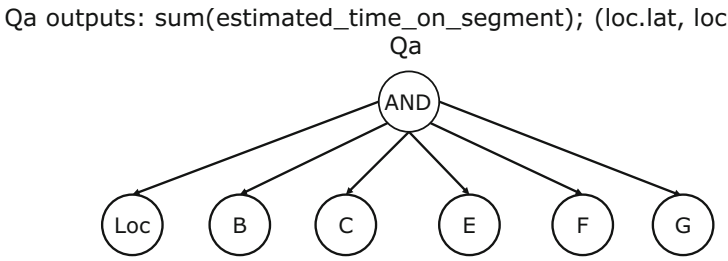

Fig. 11.2 Traffic planning event request for Alice [274]

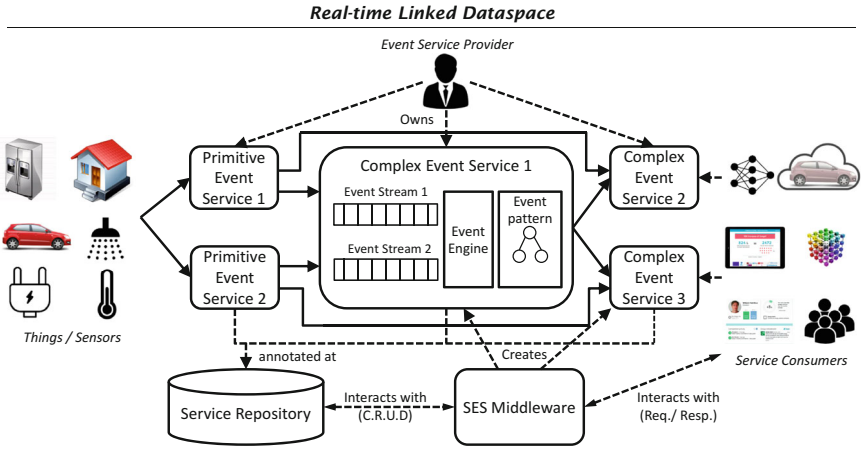

Fig. 11.3 Overview of an event service network [276]

consider the combinations of different event sources in the dataspace as well as reusing events on different abstraction levels.

11.2.3 CEP Service Design

The design of the CEP Service within the RLD [4] is based on our existing work in this area, including [123, 274–276], which is brought together in this chapter. In [275], CEP applications are provided as reusable services called Complex Event Services (CESs), and the reusability of those event services is determined by examining event patterns. Event services can thus collaborate in a distributed, cross-platform environment, creating an *Event Service Network* as shown in Fig. 11.3.

Within the RLD the complex event service composition problem is supported by the use of a specialised service to aid in event service composition.

11.2.4 Pay-As-You-Go Service Levels

Dataspace support services follow a tiered approach to data management that reduces the initial cost and barriers to joining the dataspace. When tighter integration into the dataspace is required, it can be achieved incrementally by following the service tiers defined. The incremental nature of the support services is a core enabler of the pay-as-you-go paradigm in dataspaces. The functionality of the complex event processing service follows the 5 Star pay-as-you-go model (detailed in Chap. 4) of the RLD. The complex event processing service has the following tiered-levels of support:

1 Star **No service:** No complex event processing is supported.
2 Stars **Single-service:** Event patterns are identified within a single stream.
3 Stars **Multi-service:** Event patterns can be composed of multiple event services.
4 Stars **QoS-aware:** Quality-of-Service (QoS) aware service composition of event services.
5 Stars **Context-aware:** Context-aware event processing with the use of knowledge from the dataspace.

In this chapter, we detail the implementation of the CES for the RLD with the aim to enable a QoS-aware event service composition and optimisation.

11.2.5 Event Service Life Cycle

In order to understand the problem of realising event service composition and optimisation, we analyse the different activities related to event services from their creation to termination. We identify the following five key activities in the life cycle of event services, as depicted in Fig. 11.4:

0. *Service Description:* The static description of the service metadata is created and stored in the service repository. Describing services and storing the descriptions is a preliminary step for any service requests to be realised by the described services.
1. *Request Definition:* An event service consumer identifies the requirements on the interested complex events (as well as the services that deliver the events) and specifies those requirements in an event service request.
2. *Planning:* An agent receives a consumer's request and matches it against service descriptions in the service repository. If direct matches are found, the matching service descriptions are retrieved, and the matching process ends. Otherwise,

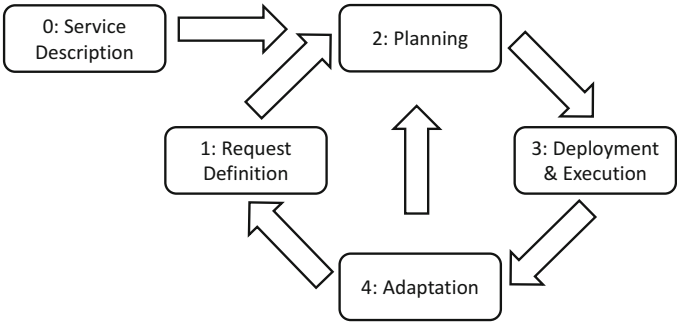

Fig. 11.4 Life cycle of an event service [276]

existing event services are composed to fulfil the requirements, and composition plans are derived.

3. *Deployment and Execution:* An agent establishes connections between the event service consumer and providers by subscribing to event services (for the consumer) based on a composition plan, then it starts the event detection (if necessary) and messaging process.
4. *Adaptation:* An agent monitors the status of the service execution to find irregular states. When irregular states are detected, the planning activity is invoked to create new composition plans and/or service subscriptions. If the irregular states occur too often, it may suggest that the service request needs to be re-designed.

We consider efficient and effective management of the event service life cycle having the following three basic requirements:

- *User-Centric Event Request Definition:* The event requests should reflect each individual user's requirements or constraints on both Functional Properties (FP) and Non-Functional Properties (NFP) of complex events. Users should be able to specify different events they are interested in by specifying FP, for example, event type and pattern. Additional to FP, it is very likely that different users may have different sets of preferences for the NFPs: some may ask for accurate results while others may ask for more timely notifications. The implemented event services should be capable of tackling these requirements and constraints.
- *Automatic Event Service Planning:* The service planning activity should be able to automatically discover and compose CESs according to users' functional and non-functional requirements. Planning based on the functional aspects requires comparing the semantic equivalence of event patterns, while planning based on the non-functional aspects requires calculating and comparing the composition plans with regard to the QoS parameters. To fully benefit from automatic implementation and enable an on-demand event service implementation, the automatic planning should be efficient to be carried out at run-time.

- *Automatic Event Service Implementation:* The deployment of the composition plans should also be automatic to facilitate automatic execution. The adaptation activity should have the ability to automatically detect service failures or constraint violations according to users' requirements at run-time and make appropriate adjustments, including re-compose and re-deploy composition plans, to adapt to changes. The adaptation process should be efficient to minimise information loss and maximise the performance of the event services over time.

Within the RLD, the complex event service composition problem is supported by the use of a specialised service to aid in event service composition. Two issues should be considered in the design of the CES: QoS aggregation and composition efficiency. The QoS aggregation for a complex event service relies on how its member event services are correlated and composed. The aggregation rules are inherently different from conventional web services. Efficiency becomes an issue when the complex event consists of many primitive events, and each primitive event detection task can be achieved by multiple event services. We address both issues by (1) creating QoS aggregation rules and utility functions to estimate and assess QoS for event service compositions, and (2) enabling efficient event service compositions and optimisation regarding QoS constraints and preferences based on Genetic Algorithms.

11.3 QoS Model and Aggregation Schema

To have a comprehensive approach for QoS-aware event service composition and optimisation within a dataspace, we need an objective function. In this section, we first discuss the relevant QoS dimensions for event services. Then, we briefly explain how we aggregate them in an event service composition and how we derive the event QoS utility as our objective function.

11.3.1 QoS Properties of Event Services

For an event service (or event service composition), its overall QoS can be discussed on several dimensions. In this work, we consider QoS attributes from [65] that are relevant for QoS propagation and aggregation, including:

- *Latency (L):* Describes the delay in time for an event transmitted by the service*Price (P):* Describes the monetary costs for an event service.
- *Energy Consumption (Eng):* Describes the energy costs for an event service.
- *Network Consumption (Net):* Describes the usage of a network of an event service, measured by messages consumed per unit time.
- *Availability (Ava):* Describes the possibility (in percentages) of an event service being accessible.

- *Completeness (C):* Describes the completeness (in percentages) of events delivered by an event service.
- *Accuracy (Acc):* Describes the possibility (in percentages) of getting correct event messages.
- *Security (S):* Describes the security levels (higher numerical value indicates higher security levels).

By the above definition, a quality vector $Q = <L, P, Eng, Net, Ava, C, Acc, S>$ can be specified to indicate the QoS performance of an event service in eight dimensions.

11.3.2 QoS Aggregation and Utility Function

The QoS performance of an event service composition is influenced by three factors: *Service Infrastructure*, *Composition Pattern*, and *Event Engine*. The *Service Infrastructure* refers to computational hardware, service Input/Output (I/O) implementation, and the physical network connection; it determines the inherent I/O performance of a service. The *Composition Pattern* refers to the way that the member event services are correlated, expressed in event patterns. The internal implementation of the *Event Engine* also has an impact on the QoS. Table 11.1 summarises how the different QoS parameters of an event service composition are calculated based on these three factors. In this chapter, we do not elaborate on the impact of service infrastructure or event engine but focus on the QoS aggregation over the composition pattern, that is, how different QoS dimensions propagate over

Table 11.1 Overall Quality of Service calculation [274]

Dimensions	QoS symbols			Overall QoS calculation
	Service infrastructure	Composition pattern	Event engine	
Latency	L_i	L_c	L_e	$L = L_i + L_c + L_e$
Price	P_i	P_c	–	$P = P_i + P_c$
Energy	Eng_i	Eng_c	Eng_e	$Eng = Eng_i + Eng_c + Eng_e$
Network consumption	–	Net_c	–	$Net = Net_c$
Availability	Ava_i	Ava_c	–	$Ava = Ava_i \times Ava_c$
Completeness	C_i	C_c	C_e	$C = C_i \times C_c \times C_e$
Accuracy	Acc_i	Acc_c	Acc_e	$Acc = Acc_i \times Acc_c \times Acc_e$
Security	S_i	S_c	–	$S = \min(S_i, S_c)$

different event service correlations, which is summarised in Table 11.2. We apply

Table 11.2 Quality of Service aggregation rules based on composition patterns [274]

QoS dimensions for event service $\mathscr{E}$	Aggregation rules	Applicable event operators
$P_c(\mathscr{E})$	$\sum_{e \in \mathcal{E}_{ice}} P_c(e)$	And, Or, Sequence, Repetition
$Eng_c(\mathscr{E})$	$\sum_{e \in \mathcal{E}_{ice}} Eng_c(e)$	And, Or, Sequence, Repetition
$Net_c(\mathscr{E})$	$\sum_{e \in \mathcal{E}_{ice}} C_c(e) \cdot f(e)$	And, Or, Sequence, Repetition
$Ava_c(\mathscr{E})$	$\prod_{e \in \mathcal{E}_{ice}} Ava_c(e)$	And, Or, Sequence, Repetition
$Acc_c(\mathscr{E})$	$\prod_{e \in \mathcal{E}_{ice}} Acc_c(e)$	And, Or, Sequence, Repetition
$S_c(\mathscr{E})$	$\min \{S_c(e) \mid e \in \mathcal{E}_{ice}\}$	And, Or, Sequence, Repetition
$L_c(\mathscr{E})$	$L_c(e)$, e is the last event in $\mathcal{E}_{dse}$	Sequence, Repetition
	$\text{avg} \{L_c(e) \mid e \in \mathcal{E}_{dse}\}$	And, Or
$C_c(\mathscr{E})$	$\frac{\min \{C_c(e) \cdot f(e) \mid e \in \mathcal{E}_{dse}\}}{\text{card}(\mathcal{E}) \cdot f(\mathcal{E})}$	And, Sequence, Repetition
	$\frac{\max \{C_c(e) \cdot f(e) \mid e \in \mathcal{E}_{dse}\}}{\text{card}(\mathcal{E}) \cdot f(\mathcal{E})}$	Or

the rules in Table 11.2 from leaves to the root of a composition pattern to derive the overall QoS step-by-step. We refer readers to [275] for a more thorough explanation of Tables 11.1 and 11.2.

11.3.3 Event QoS Utility Function

Given a quality vector of an event service composition Q = <L, P, Eng, Net, Ava, C, Acc, S > representing the service QoS capability, we denote q as one of the eight quality dimensions in the vector, O(q) as the theoretical optimum value (e.g. for latency the optimum value is 0 s) in the quality dimension of q, C(q) as the user-defined value specifying the hard constraints (i.e. worst acceptable value, e.g. 1 s for latency) on the dimension, and $0 \leq W(q) \leq 1$ as the weighting function of the quality metric, representing users' preferences (e.g. W(L) = 1 means latency is very important for the user and W(L) = 0 means latency is irrelevant for the user). Furthermore, we distinguish between QoS properties with positive or negative tendency: $Q = Q_+ \cup Q_-$, where $Q_+ = \{Ava, C, Acc, S\}$ is the set of properties with the positive tendency (larger values the better), and $Q_- = \{L, P, Eng, Net\}$ is the set of properties with the negative tendency (smaller values the better). The QoS utility U is derived by: $= \sum_{q_i \in Q_+} \frac{W(q_i) \cdot (q_i - C(q_i))}{O(q_i) - C(q_i)} - \sum_{q_j \in Q_-} \frac{W(q_j) \cdot (q_j - O(q_j))}{C(q_j) - O(q_{j.})}$

According to the above equation, the best event service composition should have the maximum utility U. A normalised utility with values between [0,1] can be derived using the function $\overline{U} = (U + |Q_-|)/(|Q_+| + |Q_-|)$.

11.4 Genetic Algorithm for QoS-Aware Event Service Composition Optimisation

Event service composition is inherently an NP-hard problem; hence, we propose a Genetic Algorithm (GA) to find a near-optimal solution in a reasonable time. Typically, a GA-based search iterates the process of population initialisation, select, crossover, and mutation to maximise the "fitness" (QoS utility introduced in Sect. 11.3.3) of the solutions in each generation.

11.4.1 Population Initialisation

During population initialisation, we generate individual composition plans as Concrete Composition Plans (CCPs). CCPs are event patterns with specific event service correlations and service bindings for the implementation of event service compositions, that is, each CCP is an individual solution for the event service composition problem. CCPs are generated from Abstract Composition Plans (ACPs), which are composition plans without service bindings. ACPs come from the event service composition request; we mark the reusable nodes in the requested event pattern (as shown in Fig. 11.5) by identifying isomorphic sub-graphs (as in [275]). Then, by enumerating all combinations of the implementation of the sub-patterns with reusable nodes as roots, we can list all ACPs. Finally, we pick a random subset of ACPs and generate a set of CCPs by binding event services.

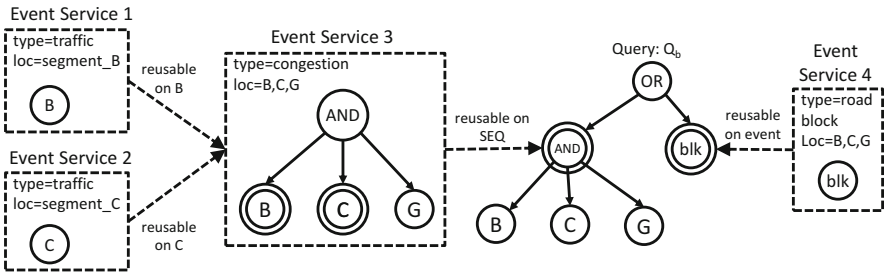

Fig. 11.5 Marking the re-usable nodes [274]

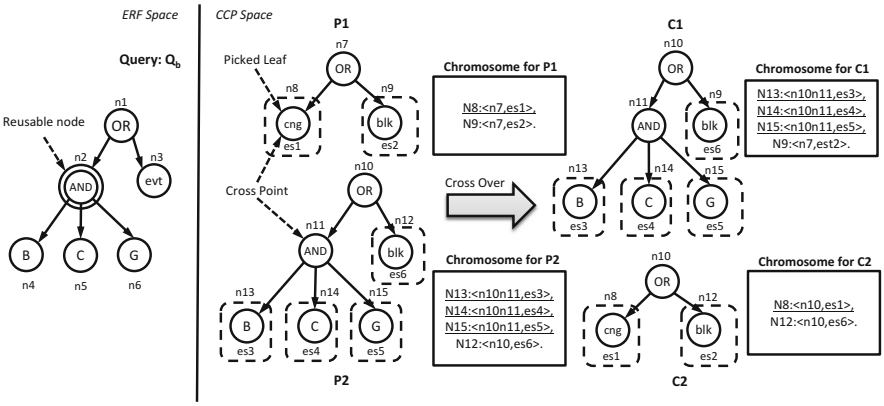

Fig. 11.6 Example of genetic encodings and crossover [274]

11.4.2 Genetic Encodings for Concrete Composition Plans

Individuals (CCPs) in the population need to be genetically encoded to represent their various characteristics (composition patterns). Conventionally, web service compositions are encoded with a sequence of service identifiers, the index of each service identifier correlates it to the specific service task in the composition. We follow this principle and encode each leaf in a CCP with an event service identifier in tree traversal order. However, since the event syntax tree is partially ordered, the position or index of the event service identifier is insufficient to represent its corresponding task.

Moreover, ancestor operators of the leaf nodes can help with identifying the role of the leaf nodes in the CCPs. Therefore, global identifiers are assigned to all the nodes in the CCPs, and a leaf node in a CCP is encoded with a string of node identifiers as a prefix representing the path of its ancestors and a service identifier indicating service binding for the leaf, as shown in Fig. 11.6. For example, a gene for the leaf node "n13" in P_2 is encoded as a string with the prefix "n10n11" and a service ID for the traffic service candidate for road segment B, that is, "es3"; hence the full encoding of n13 is <n10n11,es3>. The complete set of encodings for every gene constitutes the chromosome of P_2.

11.4.3 Crossover and Mutation Operations

After the population initialisation and encoding, the algorithm iterates the cycle of select, crossover, and mutation to find optimal solutions. The selection is trivial; individuals with better finesses (i.e. QoS utility) are more likely to be chosen to reproduce. In the following, we explain the details on the crossover, mutation, and elitism operations designed for GA-based event service composition.

11.4.3.1 Crossover

To ensure that the crossover operation produces valid child generations, parents must only exchange genes representing the same part of their functionalities, that is, the same (sub-) event detection task, specified by semantically equivalent event patterns. An example of crossover is illustrated in Fig. 11.6. Given two genetically encoded parent CCPs P_1 and P_2, the event pattern specified in the query Q and the Event Reusability Hierarchy *(ERH)*,[1] the crossover algorithm takes the following steps to produce the children:

1. Pick a leaf node l1 randomly from P_1; create the node type prefix ntp_1 from the genetic encoding of P_1, that is, $code_1$, as follows: replace each node ID in the prefix of $code_1$ with the operator type.
2. For each leaf l_1 in P_2, create the node type prefix ntp_2 from $code_2$ (i.e. encodings for l_2) and compare it with ntp_1. If $ntp_1 = ntp_2$ and the event semantics of l_1 and l_2 are equivalent, that is, they are merged into the same node in the *ERH*, then mark l_1, l_2 as the crossover points n_1, n_2. If $ntp_1 = ntp_2$ but the pattern of l_1 is reusable to l_2 or l_2 is reusable to l_1, then search back on $code_1$, $code_2$ until the cross points n_1, n_2 are found on $code_1$, $code_2$ such that $T(n_1) \doteq T(n_2)$, that is, the sub-patterns of P_1, P_2 with n_1, n_2 as the root node of the *Event Syntax Tree* (EST) of the sub-patterns are semantically equivalent.
3. If ntp_1 is an extension of ntp_2, for example, $ntp_1 = (And;Or;Seq)$, $ntp_2 = (And;Or)$ and the pattern of l_1 is reusable to l_2 in the *ERH*, then search back on $code_1$ and try to find n_1 such that the sub-pattern with EST $T(n_1)$ is equivalent to l_2. If such n_1 is found, mark l_2 as n_2.
4. If ntp_2 is an extension of ntp_1, do the same as step 3 and try to find the cross point n_2 in $code_2$.
5. Whenever the cross points n_1, n_2 are marked in the previous steps, stop the iteration. If n_1 or n_2 is the root node, return P_1, P_2 as they were. Otherwise, swap the sub-trees in P_1, P_2 whose roots are n_1, n_2 (and therefore the relevant genes), resulting in two new CCPs.

11.4.3.2 Mutation and Elitism

We apply a *Mutation* operation (with a certain possibility called mutation rate) after each crossover. The mutation operation randomly changes the composition plan for a leaf node in a CCP. The result of the mutation could be a different service binding for the leaf or replacing the leaf node with a new composition using the leaf node as an event request.

[1] An ERH is a DAG with nodes representing event patterns and edges representing the reusable relations, we introduce the ERH in our previous work in [123].

We apply an *Elitism* operation after each selection of a generation and add an exact copy of the best individual from the previous generation. Elitism allows us to ensure the best individual will survive over multiple generations.

11.5 Evaluation

In this section, we present the evaluation results of the proposed approaches. We put our experiments in the context of an intelligent travel-planning system using both real and synthetic sensor datasets for the city of Aarhus. In this scenario, a user will select a travel route and make an event request which tries to continuously monitor the traffic condition using the sensors deployed along the route that are available in the dataspace. The evaluation has two parts: in the first part, we analyse in detail the performance of the GA. In the second part, we demonstrate the usefulness of the QoS aggregation rules. All experiments are carried out on a machine with a 2.53 GHz duo core CPU and 4 GB 1067 MHz memory. Experiment results are an average of 30 iterations.

11.5.1 Part 1: Performance of the Genetic Algorithm

In this part of the evaluation, we compare the QoS utility derived by Brute-Force (BF) enumeration and the developed GA. Then, we test the scalability of the GA. Finally, we analyse the impact of different GA parameters and provide guidelines to identify optimal GA parameter settings.

11.5.1.1 Datasets

Open Data Aarhus (ODAA) is a public platform that publishes sensor data and metadata about the city of Aarhus. Currently, there are 449 pairs of traffic sensors in ODAA. Each pair is deployed on one street segment for one direction and reports the traffic conditions on the street segment. These traffic sensors are used in the experiments to answer requests on travel planning. We also include some other sensors in our dataset that might be used in traffic monitoring and travel planning, for example, air pollution sensors and weather sensors. These sensors are not relevant to requests like Alice's (i.e. denoted Qa in Fig. 11.2) or Bob's (denoted Qb), that is, they are noise to queries like Qa and Qb (but could be used in other travel-related queries). In total, we use 900 real sensors from ODAA, in which about half of them are noise. We denote this dataset sensor repository R0.

Each sensor in R0 is annotated with a simulated random quality vector <L, Acc, C, S>, where L $\in$ [0 ms, 300 ms], Acc, C $\in$ [50%, 100%], S $\in$ [1, 5], and frequency f $\in$ [0.2 Hz, 1 Hz]. We do not model price or energy consumption in the

Table 11.3 Simulated sensor repositories [274]

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
N	1	2	3	4	5	6	7	8	9
Total size	1800	2700	3600	4500	5400	6300	7200	8100	9000

Table 11.4 Queries used in experiments [274]

Query	Description	Nodes
Q_a	Alice's query on estimated travel time on the route	1 AND, 6 streams
Q_b	Bob's query on traffic condition	1 AND, 1 OR, 4 streams
Q'_a	A variant of Q_a with more nodes	1 AND, 3 random operators, 8 streams
Q'_b	A variant of Q_b with more nodes	1 AND, 1 OR, 10 streams

experiments because their aggregation rules are similar to network consumption. For similar reasons, we also do not model availability. To test the algorithms on a larger scale, we further increase the size of the sensor repository by adding N functionally equivalent sensors to each sensor in R0 with a random quality vector, resulting in the nine different repositories as shown in Table 11.3. In the experiments, we use a loose constraint to enlarge the search space, and we set all QoS weights to 1.0. The queries used in the experiments are summarised in Table 11.4.

11.5.1.2 QoS Utility Results and Scalability

In this set of experiments, we first demonstrate the usefulness of the GA by comparing it to a BF algorithm and a random pick approach. Figure 11.7 shows the experimental results for composing Q_a over R_3 to R_9 (R_1 and R_2 are not tested here because their solution spaces are too small for GA), where Q_a has six service nodes and one operator. A more complicated variant of Q_a with eight service nodes and four operators is also tested, denoted Q'_a.

The best utility obtained by the GA is the highest utility of the individual in the last generation before the GA stops. In the current implementation, the GA is stopped when the current population size is less than five or the difference between the best and the average utility in the generation is less than 0.01, that is, the evolution has converged. Given the best utility from BF $\overline{U}_{bf}$, best utility from GA $\overline{U}_{ga}$, and the random utility of the dataset $\overline{U}_{rand}$}, we calculate the degree of optimisation as $d = (\overline{U}_{ga} - \overline{U}_{rand})/(\overline{U}_{bf} - \overline{U}_{rand})$. From the results in Fig. 11.7, we can see that the average is $d = 89.35\%$ for Q_a and Q'_a. In some cases, the BF algorithm fails to complete, for example, Q_a over R_8 and R_9, because of memory limits (heap size set to 1024 MB). We can see that for smaller repositories, d is more significant. This is because, under the same GA settings (initial population size: 200, crossover rate: 95%, mutation rate: 3%), the GA has a higher chance of finding the global optimum during the evolution when the solution space is small, and the

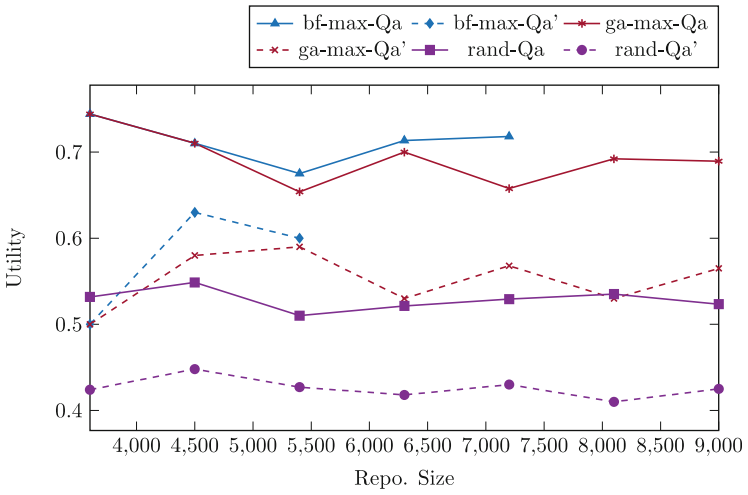

Fig. 11.7 QoS utilities of BF, GA, and random pick [274]

elitism method described in Sect. 11.4.3 makes sure that, if found, the global optimum "survives" till the end of evolution, for example, in the GA results for Q_a over R_3 and R_4 in Fig. 11.7.

It is evident that a BF approach for QoS optimisation is not scalable because of the NP-hard nature of the problem. We analyse the scalability of the GA using different repository sizes, query sizes (total number of event operator nodes and event service nodes in the query), as well as different number of CESs in the Event Reusability Hierarchy (ERH).

From the results in Fig. 11.8a, we can see that the composition time of Q_a grows linearly for GA when the size of the repository increases. To test the GA performance with different query sizes using different operators, we use the EST of Q_b as a base and replace its leaf nodes with randomly created sub-trees (invalid ESTs excluded). Then we test the GA convergence time of these queries over R_5. Results from Fig. 11.8b indicate that the GA execution time increases linearly regarding the query size.

In order to test the scalability over a different number of CESs in the ERH (called ERH size), we deploy 10–100 random Complex Event Services (CESs) to R_5, resulting in ten new repositories. We test the GA on a query created in the previous step (denoted Q'_b) with the size of 12 nodes (two operators, ten sensor services) and record the execution time in Fig. 11.8c. To ensure each CES could be used in the composition plan, all CESs added are sub-patterns of Q'_b. From the results, we can see that although the increment of the average execution time is generally linear, in some rare test instances there are "spikes", such as the maximum execution time for ERHs of size 40 and 80. After analysing the results of those cases, we found that most (over 90%) of the time is spent on population initialisation, and the complexity of the ERH causes this, that is, the number of edges considered during ACP creation.

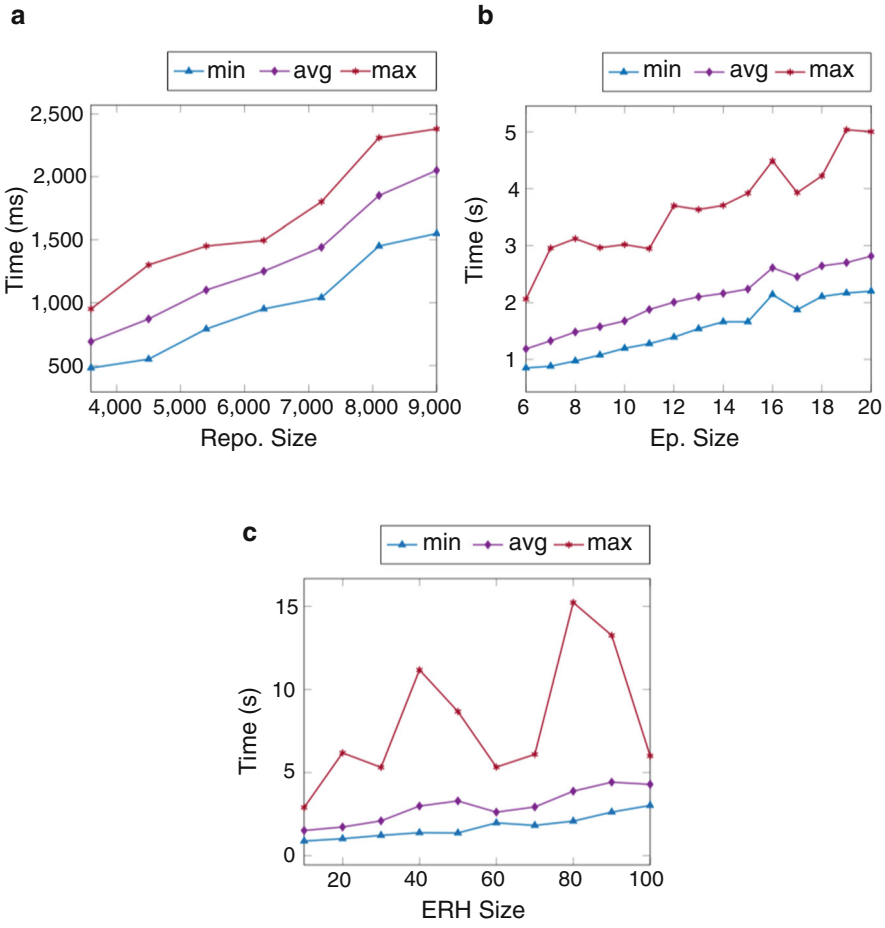

Fig. 11.8 (**a**) GA scalability over repository size, (**b**) GA scalability over EP size, (**c**) GA scalability over ERH size [274]

11.5.1.3 Fine-Tuning the Parameters

In the experiments above, a fixed set of settings is used as the GA parameters, including crossover rate, mutation rate, and population size. To find good settings of the GA in our given problem domain, we fine-tune the mutation rate, population size, and crossover rate based on the default setting used above. We change one parameter value at a time while keeping other parameters unchanged.

In order to determine the effect of the parameter tuning, we define a Cost-Effectiveness score (i.e. CE-score) as follows: given the random pick utility of a dataset $\overline{U}_{\text{rand}}$, we have the final utility derived by GA $\overline{U}_{\text{ga}}$ and the number of milliseconds taken for the GA to converge t_{ga}, CE-score $= (\overline{U}_{\text{ga}} - \overline{U}_{\text{rand}}) \times 10^5 / t_{\text{ga}}$. We test two queries Q_a, Q'_b over two new repositories R'_5, R'_9, which are R_5 and R_9

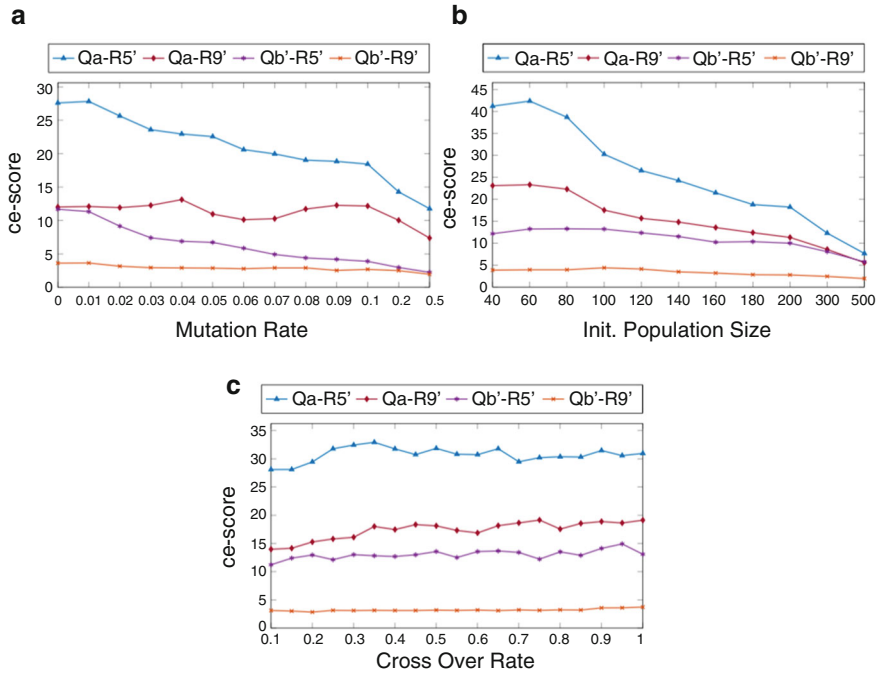

Fig. 11.9 (**a**) CE-score over mutation rate, (**b**) CE-score over population size, (**c**) CE-score over crossover rate [274]

with 50 and 100 additional CESs, respectively. Hence, we have four test combinations on both simple and complex queries and repositories. The results of fine-tuning the mutation rate, population size and crossover rate are shown in Fig. 11.9a–c.

From the results in Fig. 11.9a, we can see that the optimal mutation rate is quite small for all tests, that is, from 0% to 0.4%. Results in Fig. 11.9b indicate that for smaller solutions spaces such as Q_a over R'_5 and R'_9, the optimal initial population size is smaller, that is, with 60 individuals in the initial population. For more complicated queries and larger repositories, using a larger population size, for example, 100, is more cost-efficient. Results from Fig. 11.9c indicate that for Q_a over R'_5, the optimal crossover rate is 35% because the global optimum is easier to achieve, and more crossover operations bring additional overhead. However, for more complicated queries and repositories, a higher crossover rate, for example, from 90% to 100%, is desired. It is worth noticing that in the results from Fig. 11.9b and c, the changes in the score for Q'_b over R'_9 is not significant. This is because the GA spends much more time trying to initiate the population, making the cost-effectiveness score small and the differences moderate.

In the previous experiments, we use the selection policy such that every individual is chosen to reproduce once (except for the elite whose copy is also in the next-generation). This will ensure the population will get smaller as the evolution progresses and the GA will converge quickly. This is desirable in our case because the

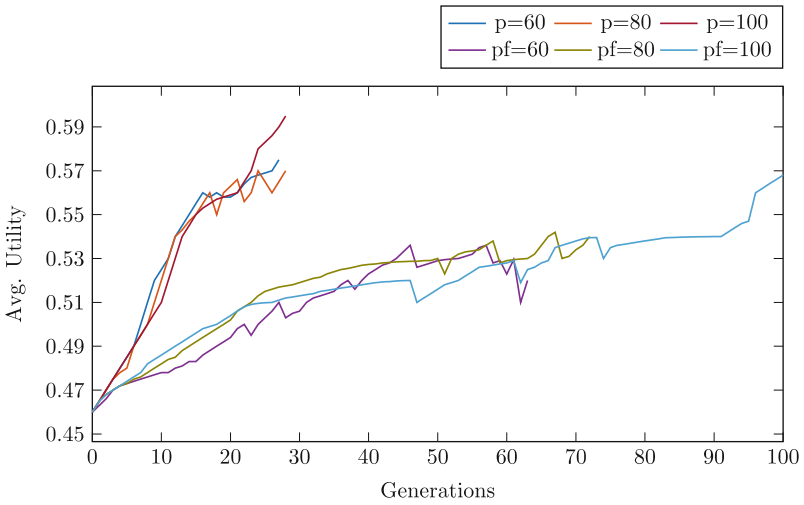

Fig. 11.10 Average utility using flexible ("p = x") and fixed ("pf = x") population size [274]

algorithm is executed at run-time and is time sensitive. However, it is also possible to allow an individual to reproduce multiple times and keep a fixed population size during evolution.

To compare the differences of having a fixed or flexible population size, we show the average utility (of Q'_b over R'_9) over the generations in Fig. 11.10. The results show that the number of generations for flexible population sizes is similar, while larger sizes achieve higher utilities. Also, the duration of generations in fixed population sizes is quite different: for a fixed population size of 60, the GA converges in about 60 generations; and for the size of 100, it lasts more than 100 generations. Larger sizes also produce better results in a fixed population, but it is much slower, and the utilities are lower than those obtained from flexible populations. In summary, we can confirm that using a flexible population size is better than a fixed population size for the GA described in this section.

11.5.2 Part 2: Validation of QoS Aggregation Rules

In this part of the evaluation, we show how the QoS aggregation works in a simulated environment.

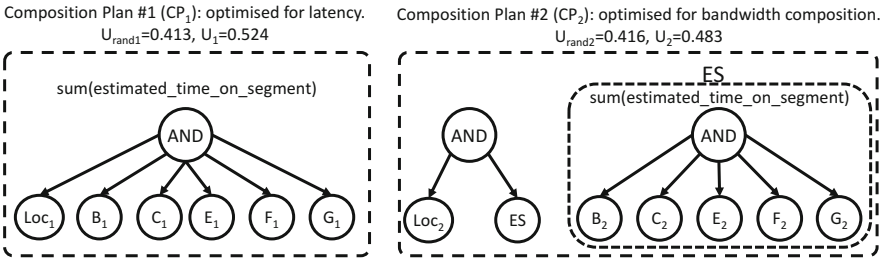

Fig. 11.11 Composition plans for Q_a under different weight vectors [274]

11.5.2.1 Datasets and Experiment Settings

To demonstrate the effect of QoS aggregation and optimisation, we generate two composition plans CP_1 and CP_2 with the GA for Q_a over R'_9 using the same constraints specified in Sect. 11.5.1. CP_1 is optimised for latency, with the weight of latency set to 1.0 and other QoS weights set to 0.1; while CP_2 is optimised for network consumption, with the weight of network consumption set to 1.0 and others set to 0.1. The reason we generate one plan to reduce the latency and the other to reduce network consumption is that the resulting plans are quite different in structure, as shown in Fig. 11.11.

When the two composition plans are generated, we transform the composition plans into stream reasoning queries (e.g., C-SPARQL query). We evaluate the queries over the traffic data streams produced by ODAA sensors. According to the composition plan and the event service descriptions involved in the plans, we simulate the QoS of the event services on a local test machine, that is, we create artificial delays, wrong and lost messages according to the QoS specifications in event service descriptions, and set the sensor update frequency as the frequency annotated (so as to affect the messages consumed by the query engine). Security is annotated but not simulated, because the aggregation rule for security is trivial, that is, estimated to be the lowest security level. Notice that the simulated quality is the *Service Infrastructure* quality. We observe the results and the query performance over these simulated streams and compare it with the QoS estimation using the rules in Tables 11.1 and 11.2, to see the differences between the practical quality of the composed event streams and the theoretical quality as per our estimation.

11.5.2.2 Simulation Results

The results of the comparison between the theoretical and simulated quality of the event service composition are shown in Table 11.5. The first column is the quality dimensions of the two composition plans; the second column is the computed quality values based on the aggregation rules defined in Table 11.2. These rules consider the

Table 11.5 Validation for QoS aggregation and estimation [274]

		Composition pattern	Event engine	End-to-end simulated	End-to-end deviations
Plan 1 (CP_1)	Latency	40 ms	604 ms	673 ms	+4.50%
	Accuracy	50.04%	100%	51.43%	+2.78%
	Completeness	87.99%	97.62%	72.71%	−14.89%
	Network consumption	4.05 msg/s	4.05 msg/s	3.84 msg/s	−5.19%
Plan 2 (CP_2)	Latency	280 ms	1852 ms	2328 ms	+9.19%
	Accuracy	53.10%	100%	51.09%	−3.79%
	Completeness	87.82%	73.18%	46.31%	−17.96%
	Network consumption	0.37 msg/s	0.40 msg/s	0.32 msg/s	−13.51%

Composition Pattern of the query as well as the *Service Infrastructure* quality of the composed services. We denote this quality QoS. However, this is not the end-to-end QoS, because the quality of the event stream engine needs to be considered. To get the stream engine performance we deploy the queries with optimal *Service Infrastructure* quality, that is, no artificial delay, mistake, or message loss, and we record the quality of query executions in the third column. We denote this engine quality QoS_{ee}. The simulated end-to-end quality is recorded in the fourth column, denoted QoS_s. We calculate the theoretical end-to-end quality based on QoS and QoS_{ee} using Table 11.1. Notice that the *Service Infrastructure* qualities of the queries themselves are not considered since we do not measure the results provided to external service consumers. Instead, the quality measurement stops at the point when query results are generated. We denote this theoretical end-to-end quality QoS_t and calculate the deviation $d = (QoS_s/QoS_t) - 1$, which is recorded in the last column. From the results we can see that the GA is highly effective in optimising latency for CP_1 and network consumption for CP_2: the latency of the former is 1/7 of the latter and event messages consumed by the latter are less than 1/8 of the former.

We can also see that the deviations of latency and accuracy are moderate for both plans. However, the completeness estimation deviates about 15–18% from the actual completeness. For the network consumption in CP_1, the estimation is quite accurate, that is, about 5% more than the actual consumption. However, the network consumption for CP_2 deviates from the estimated value by about 13.51%. The difference is caused by the unexpected drop in C-SPARQL query completeness when a CES with imperfect completeness is reused in CP_2, which suggests that an accurate completeness estimation of the service could help improve the estimation of the network consumption for event service compositions using the service.

11.6 Related Work

In general, Dataspaces follow a "best-effort" model for data management which can be seen as providing varying Quality of Service levels; this is evident in the area of search and querying with different mechanisms for indexing and federated queries [2, 121, 122]. In this section, we discuss the state-of-the-art in QoS-aware service composition as well as on-demand event/stream processing.

11.6.1 QoS-Aware Service Composition

QoS models and aggregation rules for conventional web services have been discussed in [273, 277]. In this work, we extract some QoS properties from existing work and define a similar utility function. However, QoS aggregation in complex event services is different: the calculation of aggregated QoS depends on the correlations among member event services, while the impact of event engine performance also needs to be considered. Therefore, a set of new aggregation rules are developed in this work. GA-based service composition and optimisation have been explored previously in [273, 278, 279]. However, they only cater for *Input*, *Output*, *Precondition*, and *Effect* (IOPE) based service compositions. For composing complex event services, a pattern-based reuse mechanism is required [275].

11.6.2 On-Demand Event/Stream Processing

Our work is not the first attempt that combines Service-Oriented Architectures (SOA) with Complex Event Processing to achieve on-demand event processing. Event-driven SOA has been discussed in [280, 281]. However, they only use CEP to trigger sub-sequential services. SARI [282] uses IOPE-based service matchmaking for event services, but it has limited matchmaking capability for logical AND and OR correlations.

A unified event query semantics is essential for a cross-platform and on-demand event processing [283]. EVA extends the semantic framework proposed by [284] and provides a transformation mechanism from EVA to target CEP query languages, but the transformation adaptation happens before run-time, and an on-demand event processing at run-time is not realised. RSQ-QL [283] is a recent unified RDF Stream Processing (RSF) query language that in theory supports event processing and existing RSP engines. However, it has yet to be implemented with a concrete engine.

Semantic Web Service inspired semantic streams [285] uses a prolog-based reasoning system to discover relevant data streams. It provides support for both functional and non-functional requirements, but the matchmaking still depends on the stream types. H2O [286] proposes a hybrid processing mechanism for long-term

(coarse-grained) and real-time (fine-grained) queries, the former type of query can provide partial results for the latter. However, it limits the expressiveness of real-time queries. Dyknow [287] leverages C-SPARQL as its RSP engine and facilitates on-demand semantic data stream discovery using stream metadata annotations. However, Dyknow does not support complex event streams.

11.7 Summary and Future Work

In this chapter, we detail the design of a dataspace support service for the composition of a complex event service. The service uses a GA-based approach to find optimised event service compositions within a dataspace. A QoS aggregation schema is proposed to calculate the overall QoS for an event service composition based on correlations of member event services. A QoS utility function is defined based on the QoS model and serves as the objective function in the GA. Our algorithm is evaluated with both real and synthetic sensor data streams within an intelligent travel-planning system. Results show that we can achieve about 89% optimal results in seconds. We also provide experimental results on fine-tuning GA parameters to further improve the algorithm. Finally, we use experiments to validate our QoS aggregation schema, and the results indicate that our QoS aggregation and estimation do not deviate far from the actual QoS.

We are considering the following future directions. Firstly, we will explore rule-based event service composition as a more general approach for integrating various event services. The GA-based approach proposed in this work is an ad hoc solution in the sense that it relies on the event pattern semantics. Changing the event semantics might introduce significant revisions of the composition algorithm. Using a rule-based composition algorithm, on the other hand, can easily cope with various event semantics. Secondly, we will explore the distributed stream processing mechanisms for RDF streams and find efficient means to support dynamic reasoning in a distributed manner.