Chapter 3. Data Freshness

This chapter describes our proposal for data freshness evaluation and enforcement. We propose freshness evaluation algorithms that take into account the properties of the data integration system that have more impact in data freshness. This approach allows the specialization of evaluation algorithms according to different application scenarios. We also present an approach for data freshness enforcement when freshness requirements cannot be achieved.

1. Introduction

The needs of having precise measures of data freshness become increasingly critical in several fields. Examples are numerous:

- *Information Retrieval*: Given a user query, there may be a great number of web sources providing data to answer the query but having different data quality, in particular, having varied freshness. A big amount of retrieved data is not relevant for users because of its lacks of freshness (e.g. web pages announcing train ticket reductions for expired promotions). The analysis of data freshness is useful for making a pre-filtering of data (or entire data sources), according to freshness requirements. In addition, retrieved data may be sorted according to their freshness, allowing the user to first see the freshest data.
- *Decision making*: When decision making is based on data extracted from autonomous data sources, external to the organization, a fine knowledge of data quality is necessary in order to associate relative importance to data. For example, old euro currencies should not impact decisions in the same way than more recent currencies. In this context, data freshness should be informed to end-users, as an additional attribute qualifying data. Further strategies, as filtering old data may also be carried out.
- *E-commerce*: Many web portals, such as Kelkoo® or Ebay® bring a uniform access to products of several vendors, allowing the comparison of product features and prices. Offerings are so numerous and disparate that users are overloaded with great amounts of data. Frequently, many of the proposed products are out of stock or have changed the offering conditions (e.g. the price); for example, when searching last minute flights, most of the offers are timeout and user loses considerable time. Incorporating data freshness conditions to the query interfaces offers a good possibility for reducing interaction times and providing relevant data.
- *Scientific experiments*: Research experiments, especially in the field of life sciences, produce great amounts of data, which are published in databanks and journals. Searches of related experiments (e.g. about a gene sub-sequence) are frequently carried out in order to cross results and abstract similar behaviors. Comparison is not trivial and requires executing expensive routines, which is worsen by the great amount of data and its wide overlapping. Furthermore, the existence of relatively old data introduces noise to the task. In this context, the analysis of data freshness may help reducing the search space in order to retrieve the most recent experiments.
- *Web-services integration*: Consider a company that uses web service S and wants to find a compatible service provider. The selection among the offered services may be done according to several criteria, for example, response time or service availability. When the service also provides data, data freshness may play an important role, because users may prefer obtaining the most recent data. For example, in an application querying yellow pages providers, data freshness may be critical.
- *Customer relationship management*: Managing obsolete data may become very expensive. A well-known example is the return of mail because customers have changed their addresses while the system continues managing the old ones. Generally, many data qualifying customers are obtained from external sources (e.g. address catalogs, yellow pages, census data). Knowing data freshness is crucial for taking accurate decisions. Furthermore, data freshness may be an important factor when choosing among data providers.

All these scenarios motivate the need of data freshness evaluation methods capable of adapting to different user expectations and different perceptions of data freshness. As argued in previous chapter, data freshness represents a family of quality factors. We recall the two freshness factors that have been proposed in the literature (see Sub-section 2.1 of Chapter 2 for further details):

- ❑ *Currency* describes how *stale* is data with respect to the sources. It captures the gap between the extraction of data from the sources and its delivery to the users. It is often measured as the time elapsed since data was extracted from the source.
- ❑ *Timeliness* describes how *old* is data (since its creation/update at the sources). It captures the gap between data creation/update and data delivery no matter when data was extracted from sources. It is often estimated as the time elapsed from the last update to a source.

We consider both freshness factors. We use the term *data freshness* when the discussion concerns both factors and we refer to *data currency* and *data timelines* only when specific discussion is necessary. We consider set granularity for measures, i.e. a freshness value is associated to each source relation (or equivalent structures when sources are not relational).

In this chapter we deal with data freshness evaluation in data integration systems (DISs). We address the problem of evaluating the freshness of the data returned to users in response to their queries and deciding if users' freshness expectations can be achieved. Initially, we treat the topic of data freshness evaluation, and then, we discuss how freshness measures can be used for improving the DIS and enforcing data freshness.

In order to evaluate the freshness of the data returned to users, we should consider the freshness of source data and also take into account the processes that extract, integrate and convey data to users. In previous chapter we analyzed the various dimensions that influence data freshness, namely, nature of data, architectural techniques and synchronization policies. We now focus on modeling such features and using them in the freshness evaluation process. To this end, we propose a framework which attempts to formalize the different elements involved in data freshness evaluation. Among these elements there are data sources, user queries, processes that extract, integrate and convey data, metadata describing DISs features, quality measures and quality evaluation algorithms.

In our framework, DISs are modeled as workflow processes in which the workflow activities perform the different tasks that extract, integrate and convey data to end-users. For example, in data warehouse refreshment processes, typical workflow activities are the routines that perform the extraction, cleaning, integration, aggregation and customization of data [Bouzeghoub+1999]. Workflow models enable the representation of complex data manipulation operations. Quality evaluation algorithms are based on the workflow's graph representation and consequently, the freshness evaluation problem turns into value aggregation and propagation through this graph.

The idea behind the framework is to define a flexible context which allows specializing evaluation algorithms in order to take into account the characteristics of specific application scenarios. For example, in a DIS that materializes data, the data freshness evaluation method should take into account the delays introduced by data refreshment, while in a virtual DIS such delays are not applicable. We propose a freshness evaluation approach that is general enough to be used in different types of DISs but is flexible enough to adapt to the characteristics of concrete application scenarios.

In addition to allowing the evaluation of data freshness, our framework proposes many facilities for data freshness enforcement. A DIS should provide the data freshness expected by the users. In order to know if user freshness expectations can be achieved by the DIS, we can evaluate the freshness values of conveyed data and compare them with those expected by users. If freshness expectations are not achieved, we may improve DIS design in order to enforce freshness or negotiate with source data providers or users in order to relax constraints. We propose a freshness enforcement approach that supports the analysis of the DIS at different abstraction levels in order to identify its critical points and to target the study of improvement actions for these critical points.

The following sections describe our approach for data freshness evaluation and enforcement: Section 2 describes the framework and presents an overview of the evaluation approach. Section 3 uses the framework for data freshness evaluation, specifically, we model the DISs processes and properties that have impact in data freshness and we implement evaluation algorithms that take into account the processes and properties. Section 4 deals with data freshness enforcement, presenting improvement actions for enforcing data freshness when freshness expectations cannot be achieved by a DIS. Section 5 illustrates the development of a specific improvement action. We conclude, in Section 6, by drawing the lessons learned from our experiments.

2. Data quality evaluation framework

In this section we present a framework for data freshness evaluation in the context of DISs. The framework models data sources, data targets and the DIS processes (which build target data from source data). DIS processes include the tasks for extracting, transforming and integrating data and conveying it to users. We can model a unique DIS or several DISs (e.g. various data marts for different departments of an enterprise).

The goal of the framework is twofold, firstly, helping in the identification of the DIS properties that should be taken into account for freshness evaluation, and secondly, allowing the easy development of evaluation algorithms that consider such properties.

This section describes the quality evaluation framework, specifying the representation of its components and presenting an overview of its usage for data quality evaluation. The rest of the chapter provides detailed description on data freshness evaluation and enforcement, based on this framework.

2.1. Definition of the framework

The quality evaluation framework attempts to formalize the different elements involved in data freshness evaluation. Among these elements there are data sources, data targets, DIS processes, DIS features, quality measures and quality evaluation algorithms. We start defining the framework and along the section we define the framework components.

The proposed framework is defined as follows:

Definition 3.1 (quality evaluation framework). The *quality evaluation framework* is a 5-uple:

<Sources, Targets, QualityGraphs, Properties, Algorithms>

where *Sources* is a set of available data sources, *Targets* is a set of data targets, *QualityGraphs* is a set of graphs representing several DISs processes, *Properties* is a set of properties describing DISs features and quality measures and *Algorithms* is a set of quality evaluation algorithms. □

Example 3.1. Consider a DIS that retrieves meteorological information from three sources: S_1 (real time meteorological data of satellites), S_2 (meteorological dissemination database) and S_3 (climatic sensors). The DIS provides information to three query interfaces: T_1 (historical information about climate alerts), T_2 (aggregated data about climate measurements) and T_3 (detailed data about predictions). The DIS includes processes for extracting, filtering, integrating and aggregating data. Among the DIS features that are relevant for studying data freshness there are, for example, the processing cost of DIS processes and the refreshment frequencies of materialized data. As quality measure, users are interested in data timeliness. Consequently, there is a quality evaluation algorithm for measuring data timeliness in such DIS. The quality evaluation framework allows modeling all these components. □

The framework includes a set of data targets for which the user requires data quality evaluation and a set of data sources providing data for feeding those data targets. Data sources can be relations in data repositories, web pages, user input interfaces or other types of applications producing data. Analogously, data targets can be relations in data repositories, views, user display interfaces or other types of applications consuming data. Sources and targets are defined as follows:

Definition 3.2 (data source). A *data source* is represented by a pair <Name, Description> where *Name* is a String that uniquely identifies the source and *Description* is a free-form text providing additional information useful for end-users to identify the source (e.g. URL, provider, high-level content description). □

Definition 3.3 (data target). A *data target* is represented by a pair <Name, Description> where *Name* is a String that uniquely identifies the data target and *Description* is a free-form text providing additional information useful for end-users to identify the target (e.g. application/process name, interfaces, servers running the application). □

Each DIS extracts data from some data sources and provides some data targets with data. Those sources and targets are included in the sets of sources and targets of the framework. Next sub-section describes a model for DISs processes.

2.1.1. Graph model of the data integration system workflow

A DIS is modeled as a workflow process in which the workflow activities perform the different tasks that extract, integrate and convey data to end-users. Each workflow activity takes data from sources or other activities and produces result data that can be used as input for other activities. Then, data traverses a path from sources to users where it is transformed and processed according to the system logics. We choose workflow models in order to enable the representation of complex data manipulation operations, as in [Bouzeghoub+1999] [Grigori+2005] [Ballou+1998]. In order to perform data quality evaluation, we define the concept of *quality graph*, which is a graph that has the same workflow structure as the DIS and is adorned with additional DIS information that is useful for quality evaluation. Many of existing proposals for workflow specification are graph based [van der Aalst+2002] [Mendling+2006] [Ziemann+2005] [Grigori+2005] and many works have represented DIS as graphs [Theodoratos+1997] [Naumann+1999]. For this reason, we choose to base our quality evaluation approach on graphs*. Using graphs as representation formalism, the quality evaluation problem turns into a graph traversal problem.

A quality graph is a directed acyclic graph. The nodes are of three types: (i) *activity nodes* representing the major tasks of a DIS, (ii) *source nodes* representing data sources accessed by the DIS, and (iii) *target nodes* representing data targets fed by the DIS. Nodes have a name that identifies them; for source and target nodes, their name coincide with those of sources and targets of the framework. Activities can be atomic or composed. They consume input data elements and produce output data elements which may persist in repositories. There are two types of edges: (i) *control edges* expressing the control flow dependencies between activities (e.g. execution precedence), and (ii) *data edges* representing data flow from sources to activities, from activities to targets and between activities (i.e. the output data of an activity is taken as input by a successor activity). In most DISs, control flow is induced by data flow, i.e. there is a control flow edge between two activities if and only if there is a data flow edge between them. Both, nodes and edges can have labels, which are discussed in next sub-section.

In summary, a quality graph is defined as follows:

Definition 3.4 (quality graph). A *quality graph* is a quadruple $G=(V, E, \rho_V, \rho_E)$ where:

- V is the set of nodes. V^s, V^t and V^a are the sets of source, target and activity nodes respectively; with $V = V^s \cup V^t \cup V^a$. Each source or target node corresponds to a source or target of the framework.
- $E \subset V \times V \times T$ is the set of edges. $T = \{c, d\}$ distinguishes between control edges (c) and data edges (d). The edge $(u, v)^t$ originates at node u, terminates at node v and has type t; with $u,v \in V$, $t \in T$.
- $\rho_V : V \rightarrow L_V$ is a function assigning labels to the nodes. L_V denotes the set of node labels.
- $\rho_E : E \rightarrow L_E$ is a function assigning labels to the edges. Analogously, L_E denotes the set of edge labels. □

We consider, without loss of generality, that target nodes have a unique incoming data edge. If we need to model multiple data edges incoming a target node, we can add a virtual activity node that concentrates the incoming edges and has a unique outgoing edge to the target node. This pattern is commonly used in workflow design [van der Aalst+2003]. Analogously, we consider that source nodes have a unique outgoing data edge.

Figure 3.1 sketches the quality graph representation. Activity nodes are represented as circles, while source nodes (with no input edges) and target nodes (with no output edges) are represented as rectangles. Data edges are continuous arrows and control edges are dotted arrows. Labels are written next to nodes and edges. In some figures, when we only want to study the data flow, control edges may be omitted. Analogously, some properties may be omitted.

* We use graphs for representing workflows although the approach can easily be adapted to other formal models such as Petri nets or state-chart diagrams, provided that it is a uniform model.

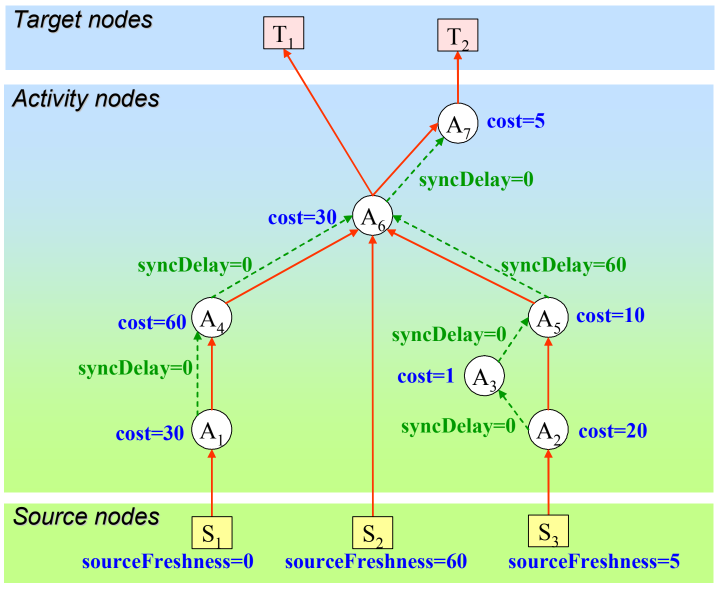

Figure 3.1 – Quality graph

2.1.2. Quality features as graph adornments

Data quality depends on the structure of the quality graph (i.e. how data traverses the graph) but also on data and process properties (e.g. process synchronization policies, data volatility). The idea behind this approach is to adorn quality graphs with *property labels* that allow estimating the quality of the data that can be produced by the DIS, for example, the time an activity needs for executing or a descriptor stating if an activity materializes data or not.

Properties can be of two types: (i) *features*, indicating some characteristic of the DIS (costs, delays, policies, strategies, constraints, etc.), or (ii) *measures*, indicating a quality value corresponding to a quality factor.

Features can represent precise process metadata (as execution cost of activities), estimations (e.g. based on designer experience on similar applications, cost models or upper bounds) or beliefs (e.g. source reputation). Quality measures can be actual values acquired from sources (e.g. source data freshness, measured from data last update) or expected values indicating user high level expectations (as the desired data freshness). Property values (for both, features and measures) can be directly given by system administrators, users or source providers (e.g. source availability windows, DIS refreshment policies, user expected freshness), can be systematically obtained using measurement processes (e.g. response time), can be aggregated from user passed behavior or statistics (e.g. user preferred sources, activity execution time), can be derived from other property values (e.g. activity global cost) or can be calculated in some ad-hoc way. Quality evaluation algorithms calculate further property values (quality measures) as will be discussed in next sub-section.

We define a property as follows:

> **Definition 3.5 (property).** A *property* is a 3-uple `<name, metric, domain>` where *name* is a String that identifies the property, *metric* is a description of the measurement semantics and units, and *domain* describes the domain of the property values. □

Nodes and edges of quality graphs are adorned with property labels of the form: `property = value`, where *property* is a property name and *value* is an element of the property domain. In Figure 3.1, property labels (cost, synchronization delay and source freshness) are written next to nodes and edges.

In the following sub-section we illustrate how we utilize property labels for evaluating data quality.

2.2. The approach for data quality evaluation in data integration systems

Quality evaluation is performed by evaluation algorithms that take as input a quality graph and calculate the quality values (corresponding to data freshness) for the graph. In order to illustrate our quality evaluation approach, consider the following example:

Example 3.2. Figure 3.2 sketches a quality graph representing a simple DIS, which extracts traffic statistics of a unique data source (S_1). A wrapper (activity A_1) extracts data, which is cleaned and prepared by activity A_2 and finally aggregated (activity A_3) and delivered to a user application (T_1).

We aim to estimate data timeliness. Consider that S_1 is a dissemination server that publishes traffic statistics once an hour, so published statistics correspond to traffic events of the passed hour. A_1 extracts data immediately after its publication. Source data timeliness (at extraction time) is an hour (60 minutes) because the oldest source data may be produced at most an hour before. Also consider that the execution costs of activities A_1, A_2 and A_3 are 5, 60 and 15 minutes respectively, as shown in Figure 3.2. These costs should be taken into account in the estimation of the freshness (timeliness) of the data returned to users (through user application T_1). Intuitively, timeliness of resulting data is estimated as 140 minutes, which results from adding the execution cost of activities to the source data freshness (60 + 5 + 60 + 15). □

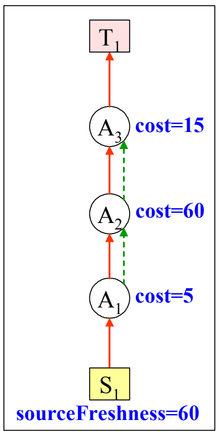

Figure 3.2 – Quality evaluation example

Evaluation algorithms may traverse the graph, node by node, operating with property values. For example, a simple evaluation algorithm may start at source nodes, read the value of the *source freshness* property and move along the data flow adding the value of the *cost* property of each activity, as intuitively explained in previous example. Such algorithm can easily be implemented using default graph traversal methods. This mechanism for calculating data quality applying operations along the graphs is what we call *propagation of quality measures within the graph* (*quality propagation* for short).

As a quality graph describes the DIS integration process and its properties, it contains the input information needed by evaluation algorithms. Evaluation algorithms take as input a quality graph, calculate the quality values corresponding to a quality factor and return a quality graph with an additional property (corresponding to the evaluated quality factor). The quality graph must be labeled with certain property values in order to execute a certain algorithm. For example, the activity nodes of a quality graph must be labeled with their execution cost in order to execute the evaluation algorithm informally described in Example 3.1.

Formally, evaluation algorithms are defined as follows:

Definition 3.6 (quality evaluation algorithm). A *quality evaluation algorithm* is a 7-uple:

<Name, Description, QualityFactor, Input, Output, Preconditions, Postconditions>

where:

- *Name* is a String that identifies the algorithm.
- *Description* is a free-form text describing algorithm evaluation strategy and optional details.
- *QualityFactor* is the quality factor that the algorithm calculates.
- The *Input* is a quality graph.
- The *Output* is a quality graph that results of adding new property values (corresponding to the algorithm *quality factor*) to the *input* quality graph.
- *Preconditions* is a set of pairs <group, property> indicating that a group of nodes/edges (e.g. activity nodes) must be labeled with values of such property (e.g. execution cost). The algorithm can be executed only if preconditions are satisfied.
- *Postconditions* is a set of groups of nodes/edges that will be labeled with new property values (corresponding to the algorithm *quality factor*) after algorithm execution. □

Concerning code, evaluation algorithms have the following signature:

```
FUNCTION AlgorithmName (G: QualityGraph) RETURNS QualityGraph
```

The implementation of evaluation algorithms may vary according to the quality factor and the concrete application scenario. The framework does not constrain the way the algorithms can be implemented. For example, the simple evaluation algorithm of Example 3.1 propagates freshness values adding property values, but another evaluation algorithm may use sophisticated calculation strategies and even user-defined functions. The proposed graph representation facilitates the implementation because it enables to use graph primitives (e.g. getPredecessors, getSuccessors, getProperties) and traversal methods (e.g. findShortestPath, depthFirstSearch).

There are two kinds of quality propagations: (i) propagation of actual values, and (ii) propagation of expected values. In the former (as illustrated in previous example), quality values of source data are propagated along the graph, in the sense of the data flow (from source to target nodes) and combined with property values of nodes and edges. In the latter, quality values expected by users are propagated along the graph but in the opposite sense (from target to source nodes) and combined with property values.

Example 3.3. Consider that users expect freshness values of at most 2 hours (120 minutes) for data produced by the DIS of Figure 3.2. Subtracting activity execution costs from the freshness expected value, we obtain a value of 40 minutes (120 – 15 – 60 – 5). This value means that the source should provide data that is fresher than 40 minutes in order to satisfy user freshness expectations. □

Propagation of actual values serves to inform users of the quality of result data; a comparison with user expectations at target nodes (expected values versus propagated actual values) determines if user quality expectations can be achieved or not. Conversely, propagation of expected values serves to constraint source providers on the quality of source data (or to choose among alternative sources providing the same type of data); a comparison with source actual quality at source nodes (actual values versus propagated expected values) determines if sources provide data with enough quality.

An important remark is that quality propagation can be performed during the execution of DIS processes using precise property values (e.g. exact execution cost obtained during execution), or conversely, it can be performed without executing DIS processes and using estimations of property values (e.g. statistics of costs of previous executions). In the former, quality evaluation is used to inform users of results quality. In the latter, quality estimations can be used to decide whether query results will be adequate for user needs or not (if no adequate, alternative actions may be taken, for example, accessing to sources with higher quality even if source accesses are more expensive). We focus on this latter propagation context, even if our mechanism can be easily applied for calculating data quality during DIS execution.

Finally, note that although the framework was conceived for the evaluation of data freshness, their components were defined in a general way allowing its use for the evaluation of other quality factors. For example, the *response time* quality factor may be measured in a similar way, and the direct use of the framework for its evaluation seems to be possible. In order to evaluate other quality factors, the framework may need to be extended, adding for example, new propagation operations; Chapter 4 illustrates this fact for the data accuracy quality factor.

In the next section we use the framework for data freshness evaluation.

3. Data freshness evaluation

In this section we describe our data freshness evaluation approach. We firstly give an intuitive idea of the freshness calculation strategy and we describe some general properties that support the calculation of data freshness. Then, we present a basic data freshness propagation algorithm based on those properties. Finally, we describe an instantiation approach for adapting the basic algorithm to particular application scenarios.

We firstly explain the propagation of freshness actual values (Sub-sections 3.1 to 3.5); the propagation of freshness expected values is similar and is discussed in Sub-section 3.6. We conclude this section motivating some direct applications of both types of data freshness propagations.

3.1. Basic evaluation algorithm

In this sub-section we propose a basic algorithm for evaluating data freshness. In order to propagate freshness actual values, the algorithm needs input information describing DIS properties.

The freshness of the data delivered to users depends on the freshness of source data but also on the amount of time needed for executing all the activities as well as on the delays that may exist among their executions. We briefly describe such properties, as well as users' freshness expectations:

– *Processing cost*: It is the amount of time that an activity needs for reading input data, executing and building result data (*cost* is used for short in some figures).

– *Inter-process delay*: It is the amount of time passed between the executions of two activities, i.e. between the end of the former and the start of the latter (*delay* for short).

– *Source data actual freshness*: It is the freshness of data in a source, i.e. at the moment of data extraction (*sourceAfreshness* for short). As data currency measures the gap with source data (the time passed since data extraction) data currency of source data is always zero, however, data timeliness of source data can take positive values.

– *Target data expected freshness*: It is the users' desired freshness for result data (*targetEfreshness* for short).

The evaluation algorithm calculates the following property:

– *Actual freshness*: It is an estimation of the actual freshness of data outgoing a node (*Afreshness* for short).

The relevance of these properties depends on the application scenario. For example, the materialization of data and the use of different policies to refresh such data may imply important *inter-process delays* while in virtual systems these delays may be negligible. The calculation (or estimation) of such property values is discussed in Sub-section 3.4, taking into account the particularities of concrete scenarios.

We propose an evaluation algorithm that estimates the freshness of result data based on previous properties. The algorithm propagates freshness actual values traversing the quality graph (following the data flow) and calculating the freshness of the data outgoing each node. The principle is the following:

– For a source node A, the freshness of data outgoing A is calculated as the source data actual freshness.

– For an activity node A with one predecessor P, the freshness of data outgoing A is calculated adding the freshness of data produced by P, the inter-process delay between P and A and the processing cost of A.

– In the general case, if activity A has several predecessors, the freshness of data coming from each predecessor (plus the corresponding inter-process delay) should be combined (synthesizing a unique value) and added to the processing cost of activity A. The typical combination function computes the maximum of the input values, but other user-specific functions may be considered (Sub-section 3.4.2 discusses other combination functions).

The calculation can be sketched as shown in Figure 3.3: First, for each predecessor (P_1… P_n) of activity A, we add the freshness of incoming data and the inter-process delay. Then, we combine such values (combineActualFreshness function) and add the processing cost to the result. The resulting value is associated to the data edges going from A to its successors (C_1… C_m).

```
1.  $v_1$ = Freshness($P_1$,A) + InterProcessDelay($P_1$,A) …
    $v_n$ = Freshness($P_n$,A) + InterProcessDelay($P_n$,A)
2.  F = combineActualFreshness({$v_1$,…$v_n$}) + ProcessingCost(A)
3.  Freshness(A, $C_1$) = F…
    Freshness(A, $C_m$) = F
```

Figure 3.3 – Freshness evaluation strategy

We associate freshness values to the data edges outgoing nodes because we want to emphasize that *freshness* is a property of data not of processes. However, the strategy can be very easily adapted for associating freshness values to nodes. For practical reasons, sometimes we refer to the *freshness of a node*, meaning the *freshness of data produced by the node*, i.e. freshness values associated to outgoing data edges.

Note that in previous formulas (Figure 3.3) we need inter-process delay values to be associated to data edges. However, inter-process delays may be influenced by properties of control flow (e.g. the type of synchronization among activities), which should be taken into account in their calculation. The calculation (or estimation) of property values is discussed in Sub-section 3.4. By the moment, we can consider that property values are labels of the quality graph as in Figure 3.4a. Processing costs are associated to activity nodes and inter-process delays are associated to data edges among activities, however, in order to facilitate the expression of some formulas (e.g. those of Figure 3.3) we often consider that processing costs are associated to all nodes (with zero value for source and target nodes) and that inter-process delays are associated to all data edges (with zero values for edges outgoing source nodes or incoming target nodes).

> **Example 3.4.** Consider the quality graph of Figure 3.4a as input for the propagation of freshness actual values. The freshness of (S_1,A_1) is calculated as the source data actual freshness of S_1, i.e. 10 units of time. As A_1 has a unique predecessor S_1, the freshness of (A_1,A_3) is calculated adding actual freshness of (S_1,A_1) plus the inter-process delay of (S_1,A_1) plus the processing cost of A_1, obtaining a value of 13 (10+0+3) units of time. As A_3 has two predecessors (A_1 and A_2), two input values are combined: 18 (13+5) and 12 (10+2), keeping the maximum, which is added to the processing cost of A3. Then, freshness of (A_3, T_1) is 22 (18+4) units of time. Figure 3.4b shows the output quality graph. □

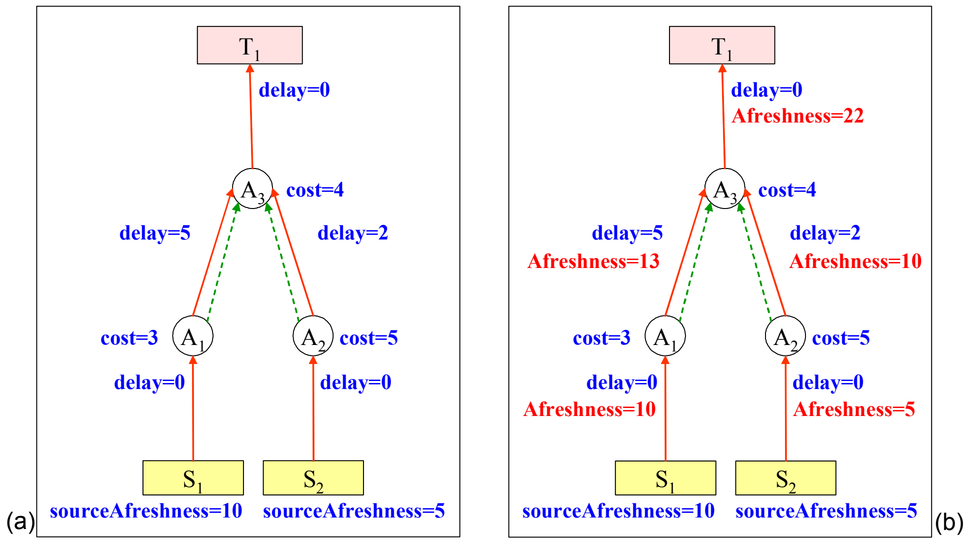

Figure 3.4 – Propagation of freshness actual values: (a) input quality graph, and (b) output quality graph

The previous strategy is implemented in a basic algorithm for freshness propagation: *ActualFreshnessPropagation* (see Algorithm 3.1). It takes as input a quality graph and returns as output the quality graph with additional labels corresponding to data freshness. The *QualityGraph* class has methods for manipulating the classical graph operations (as *getPredecessors*) and for manipulating the property values associated to the nodes and edges (as *addProperty* and *getPropertyValue*). The algorithm first spans source nodes, obtaining source data actual freshness and storing the freshness of data outgoing source nodes. Then, the algorithm traverses activity nodes, obtaining data freshness and inter-process delays of incoming edges and storing such values in a list (valList). The values of the list are combined, added to the processing cost of the activity, and finally stored for all outgoing edges. A pseudocode of the algorithm is sketched in Algorithm 3.1.

We defined the *getSourceActualFreshness*, *getInterProcessDelay* and *getProcessingCost* functions, as abstract functions, which should be instantiated for specific scenarios. The functions calculate the values of the source data actual freshness, inter-process delay and processing cost properties. Their signatures are:

```
FUNCTION getSourceActualFreshness (G: QualityGraph, A: Node) RETURNS INTEGER
FUNCTION getProcessingCost (G: QualityGraph, A: Node) RETURNS INTEGER
FUNCTION getInterProcessDelay (G: QualityGraph, e: Edge) RETURNS INTEGER
FUNCTION ActualFreshnessPropagation (G: QualityGraph) RETURNS QualityGraph
    INTEGER value;
    FOR EACH source node S of G DO
        value= getSourceActualFreshness(G,S);
        Edge e = data edge outgoing S in G
        G.addProperty(e,"ActualFreshness",value);
    ENDFOR;
    FOR EACH activity node A in topological order of G DO
        HASHTABLE valList;
        FOR EACH data edge e incoming A in G DO
            value= G.getPropertyValue(e,"ActualFreshness") + getInterProcessDelay(G,e);
            valList.add(e,value);
        ENDFOR;
        value= combineActualFreshness(G,valList) + getProcessingCost(G,A);
        FOR EACH data edge e outgoing A in G DO
            G.addProperty(e,"ActualFreshness",value);
        ENDFOR;
    ENDFOR;
    RETURN G;
END
```

Algorithm 3.1 – Basic algorithm for propagating freshness actual values

The *combineActualFreshness* function is also an abstract function that combines the freshness values of predecessor nodes (stored in the *valList* array, including the delays) and synthesize a unique freshness value. For example, if users are interested in an upper bound of freshness the *combineActualFreshness* function may return the maximum of predecessors freshness. Its signature is:

```
FUNCTION combineActualFreshness (G: QualityGraph, valList: HashTable) RETURNS INTEGER
```

The previous abstract functions can be overloaded for different scenarios taking into account the characteristics of the scenarios. Next sub-sections discuss these ideas.

3.2. Overview of the instantiation approach

In order to characterize different types of scenarios we consider the freshness factor that users are interested in and the three dimensions that influence data freshness (introduced in Chapter 2 - Sub-section 2.3), namely, nature of data, architectural techniques and synchronization policies. Our goal is to develop data freshness evaluation algorithms specialized for the different scenarios.

The basic evaluation algorithm can be specialized for considering the particularities of the scenarios. Firstly, different DIS properties should be considered in the evaluation. These properties are the ones that allow the estimation of source data actual freshness, processing costs and inter-process delays for a given scenario. For example, when materializing data, the time passed between two consecutive refreshments (refreshment period) may be an important component of the *inter-process delay* while in virtual systems this property has no sense. In addition, the evaluation algorithm should be instantiated to take into account the considered properties. The instantiation consists in overloading the abstract functions according to the scenario properties.

Then, the instantiation method consists of three steps:

1. Modeling the scenario according to the freshness factors and the dimensions that influence data freshness.
2. Identifying the appropriate properties for the scenario.
3. Instantiating the evaluation algorithm.

In order to illustrate the instantiation approach, we introduce the following motivating example:

Example 3.5. Consider three different DIS scenarios that deal with information about cinemas and films, illustrated in Figure 3.5:

- **DIS$_1$**: A DIS that retrieves information about films and the cinemas where these films are in billboard, in response to user queries as "Where can I see a given film?" or "Which films are in billboard now?". It extracts film information (titles, genre, actors, directors, timetables, etc.) from the AlloCiné site and cinema information (cinema, capacity, category, etc.) from the UGC and CinéCité sites. The process model consists of three activities for extracting data from the mentioned sites (A$_1$, A$_2$ and A$_3$), an activity for merging (union) the data extracted from both cinema sites (A$_4$) and an activity for joining film and cinema information (A$_5$). Users expect data freshness of at least a week.
- **DIS$_2$**: A DIS (part of a reservation system) that accesses to information about cinemas and the availability of places for their performances (of the UGC and CinéCité sites) and present the information to the user allowing him to choose a cinema. The process model consists of two activities for extracting data from both sites (B$_1$ and B$_2$) and an activity (B$_3$) that merges the extracted data, formats it and conveys it to the user interface. When activity B$_3$ receives data from an extractor, it waits for data from the other extractor for at most one minute, conveying the available data to the user. User freshness expectations are of at least 5 minutes.
- **DIS$_3$**: A DIS that manages statistics information about films, the number of persons that watched each film and their opinions. Typical user questions are "Which films have the best ranking this week?" or "Which film should I watch?". The DIS extracts film information and audience statistics from the AlloCiné site and critic information (films, opinions, recommendations, etc.) from the CineCritics and FilmCritiquer sites. Such sources are queried at different times (film data is extracted weekly and critic data daily) because of negotiations with source providers, so the extracted information is locally materialized in order to answer user questions. The process model consists of extraction activities (C$_1$, C$_2$ and C$_3$), an activity for reconciling data from two extractors (C$_4$), an activity for joining critic and film data (C$_6$), and two activities for performing aggregations and calculating statistics (C$_5$ and C$_7$). Users expect data freshness of at least a week.

Although all of them integrate data provided by several cinema sites, their characteristics (e.g. nature of data, system implementation) are very different. Furthermore, freshness factors and metrics as well as user freshness requirements are also different. Consequently, the way data freshness is estimated should be adapted to each application scenario. □

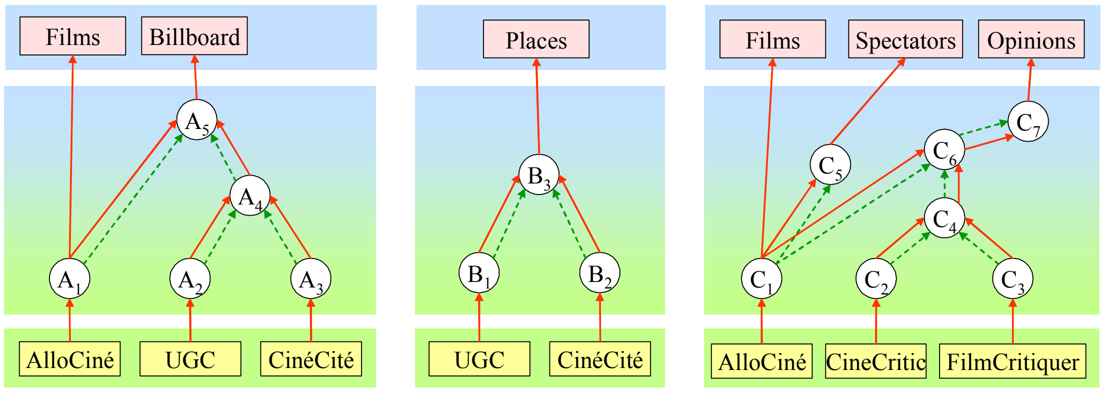

Figure 3.5 – Quality graphs representing the example DISs (omitting property labels)

Next sub-sections describe and apply each instantiation step. We explain the whole method, characterizing the scenarios, determining the properties of each scenario and implementing the overloaded functions. We illustrate the instantiation of the framework to the concrete scenarios introduced in previous example.

3.3. Modeling of scenarios

In this step, DISs processes are classified according to the freshness factor and the four dimensions that influence data freshness. We remind those dimensions (see Sub-section 2.3 of Chapter 2 for details):

- *Nature of data.* This dimension classifies source data according to its change frequency in three categories: *stable*, *long-term-changing* and *frequently-changing* data. When working with frequently changing data, it is interesting to measure how long data can remain unchanged and minimize the delivery of expired data (i.e. evaluate currency). However, when working with data that does not change very often, it is more interesting to measure how old is the data (i.e. evaluate timeliness).
- *Architectural techniques.* This dimension classifies DISs architectural techniques in three categories: *virtual*, *caching* and *materialization* techniques. DISs architectural techniques are very relevant in freshness evaluation because they may introduce significant delays. Specifically, when caching or materializing data, the refreshment frequency causes important delays that should be considered.
- *Synchronization policies.* This dimension classifies DISs synchronization policies according to the interaction between the sources, the DIS and the users (combinations of pull and push policies, in synchronous and asynchronous modes) in 6 categories[*]: *pull-pull*, *pull/pull*, *pull/push*, *push/push*, *push/pull* and *push-push* policies. Asynchronous modes, with data materialization, may introduce important delays.

This classification is useful in the measurement of processing costs, inter-process delays and source data actual freshness because it summarizes the dimensions that may impact in these properties. A first remark is that the magnitude of source data actual freshness, processing costs and inter-process delays should not be considered in the absolute but compared to freshness expectations. For example, if users may tolerate data freshness of "some days", the processing costs of activities ("some minutes") are negligible; however, if users require data "extremely fresh", the processing costs of activities could be relevant. In order to decide which properties are relevant and which are negligible for a given DIS, the classification according to the taxonomy brings a first idea of magnitudes, which will be taken into account in the calculation of property values. For example, the architectural techniques dimension gives a first idea of the magnitude of inter-process delays.

A scenario is characterized by the freshness factor users are interested in and its classification according to the dimensions of the taxonomy. So, a scenario is described giving four components: (i) freshness factor, (ii) nature of data, (iii) architectural techniques, and (iv) synchronization policies.

Example 3.6. Let's classify the DISs of Example 3.5 according to the freshness factor and the three dimensions of the taxonomy and model the respective scenarios.

Users of DIS_2 want to obtain the same data that is stored at the sources, no matter when the information was updated (when was sold the last ticket), so they are interested in *currency*. However, users of DIS_1 and DIS_3 are mainly interested in seeing information about "recent" films, so they are interested in *timeliness*.

The nature of data is different in the various sources. Cinema descriptive information (UGC and CinéCité sites, accessed by DIS_1) is quite stable, film information (AlloCiné site, accessed by DIS_1 and DIS_3) has a relatively long-term change frequency and place availability (UGC and CinéCité sites, accessed by DIS_2) and critic information (CineCritic and FilmCritiquer sites, accessed by DIS_3) frequently changes.

Concerning DIS implementation, DIS_1 is an interactive process, with virtual techniques and pull synchronous policies; activities are simple JSP queries. DIS_2 is an interactive process, with virtual techniques, that synchronizes the extraction of data from two data sources (pull synchronous with timeout policies); activities are simple copies of data. DIS_3 materializes the data produced by some activities (C_1, C_5, C_6 and C_7), so user queries are answered from materialized data. Data is refreshed periodically: film data is refreshed weekly and critic data daily. Activities are complex cleaning, reconciliation and aggregation processes.

The scenarios for the three previous DISs can be characterized as follows:

[*] Each configuration is named with the user-DIS policy followed by the DIS-source policy. Asynchronism is represented by a slash (/), synchronism by a dash (-)

- ❑ Scenario Sc_1 for DIS_1:
  - – Freshness factor: timeliness
  - – Nature of data: stable and long-term-changing data
  - – Architectural techniques: virtual techniques
  - – Synchronization policies: synchronous pull policies (pull-pull)
- ❑ Scenario Sc_2 for DIS_2:
  - – Freshness factor: currency
  - – Nature of data: frequently-changing data
  - – Architectural techniques: virtual techniques
  - – Synchronization policies: synchronous pull policies (with timeout) (pull-pull)
- ❑ Scenario Sc_3 for DIS_3:
  - – Freshness factor: timeliness
  - – Nature of data: long-term-changing and frequently-changing
  - – Architectural techniques: materialization techniques
  - – Synchronization policies: asynchronous (periodic) pull policies (pull/pull) □

Next step analyzes the relevant properties of each scenario.

3.4. Identification of appropriate properties

Data freshness is evaluated based on the source data actual freshness, processing cost and inter-process delay properties, but the way of calculating these properties depends on the particular scenario considered. The calculation may be based on DIS properties (e.g. wrapper extraction frequencies, synchronization policies), which may be set as labels of the quality graph or may be measured by used-defined functions. Analogously, the way of combining several freshness values (when an activity has several incoming edges) depends on the scenario and may be based on DIS properties (e.g. data volatility, source reputation). In this sub-section we deal with the identification of the DIS properties involved in such calculations and we discuss their estimation.

3.4.1. Estimation of property values

The calculation of source data actual freshness, processing cost and inter-process delay properties depends on the application scenario. Specifically, we discuss two aspects: (i) the DIS properties that influence their calculation and (ii) the estimation type. The combination of these two aspects should lead to a calculation method.

The first aspect is how to calculate the source data actual freshness, processing costs and inter-process delay properties. Depending on the scenario, different DIS properties may influence their calculation. For example, several delays may compose processing costs of extraction activities in certain DISs [Hull+1996]: the *communication delay* between a source and the activity, the *source query processing delay* and the *activity query processing delay*. Similarly, many delays may influence the inter-process delay, for example when combining data from two sources, activities may hold data from a source while waiting for data from another source. When materializing data, the time passed from last materialization may be an important delay, bounded by the refreshment frequency [Theodoratos+1999]. Properties associated to control flow may be also taken into account, for example, if two activities are synchronized in a way that the latter executes one hour after the former, such synchronization delay should be taken into account. Scheduling methods like Critical Path Method (CPM) and Program Evaluation and Review Technique (PERT) [Hiller+1991] can be used for setting delays among activities. When control flow is not driven by data flow, the inter-process delay between two activities might be calculated using properties associated to other activities, as illustrated in the following example.

Example 3.7. Consider the portion of a quality graph shown in Figure 3.6. Activity A_5 is synchronized to execute one unit of time after activity A_3 but it takes as input the data materialized by activity A_2. Activity A_3 is a control routine that does not produce any data. The inter-process delay associated to data edge (A_2,A_5) is calculated as the sum of synchronization delays of control edges (A_2,A_3) and (A_3,A_5) and the processing cost of A_3, i.e. 2 (0+1+1) units of time. □

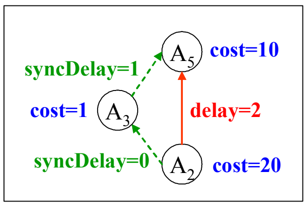

Figure 3.6 – Calculating inter-process delay from control flow properties

Various works propose different estimations for the source data actual freshness (timeliness[*]), for example: using data timestamps [Braumandl 2003] or the *update frequency* of a source [Naumann+1999]. In sources with access constraints (as restrictions on the query frequency) or dissemination frequencies, such properties should be taken into account. In specific scenarios, the values of source data actual freshness, processing cost or inter-process delay may be directly provided by expert users, system administrators or source providers (set as labels of the quality graph) without need of considering additional DIS properties.

The second aspect concerns the type of estimation, which could be a precise measure of data freshness at this moment or an estimation of data freshness in the average or worst case. The former implies storing metadata in each activity execution (e.g. the extraction time) and executing the evaluation algorithm each time the DIS conveys data (e.g. for each user query). The latter implies keeping statistics and bounds for property values which may be used to calculate upper bounds or average case values of data freshness (without need of calculating freshness when conveying data).

In summary, we should compare the magnitude of source data actual freshness, processing costs and inter-process delays with respect to freshness expectations (aided by the scenario modeled in previous step). For the relevant ones, we should identify the DIS properties that influence their calculation and the desired estimation type in order to obtain their calculation strategies. For example, if the processing cost of extraction activities is relevant, communication delay with sources is the preponderant cost and we want a worst case estimation, a good estimation strategy for the processing cost may consist in keeping statistics of communication costs and taking the maximum. We should study how to acquire each DIS property value. Some properties values may be directly provided by expert users, source providers or DIS administrators, or may be measured or estimated using specialized routines, for example, reading statistics.

Example 3.8. Let's analyze the relevant properties for the three DISs of Example 3.5. Users expect freshness values of "a week" for DIS_1 and DIS_3. With such freshness requirements, the "day" is a good unit for measuring freshness and properties values. Properties with values ranging in "some minutes" or less can be neglected. However, as users expect freshness values of at most "five minutes" for DIS_2, the measurement unit for it should be the "minute" and property values ranging in "some seconds" are relevant.

We want to estimate property values in the worst case. For DIS_1 the processing cost of simple JSP queries (seconds) and the inter-process delay for merging source data (seconds) can be neglected. For DIS_3 the relevant processing costs are due to the reconciliation processes (activities C_4 and C_6), which may require human interaction (to solve conflicts or errors) and may last some days. DIS_3 materializes data, so there is an asynchrony between the data extraction and the data delivery. Sources are queried at different times (because of negotiations with source providers), provoking important inter-process delays between the executions of some activities; then, the refreshment frequencies should be considered. For DIS_2, synchronizing extractors introduces a relevant delay, as the data extracted from a source may be held while waiting for data from the other source. The synchronization timeout is an upper bound for such delay that can be used to estimate it in the worst case. The communication delay with the sources can be important too, as well as the processing cost (seconds) of the merge activity.

For DIS_1 and DIS_3, where users are interested in data timeliness, source data actual freshness (days, weeks or months) is relevant. It can be estimated, in the worst case, as the source update frequency. □

The relevant DIS properties are summarized in Sub-section 3.4.3, as well as the properties necessaries for implementing the combineActualFreshness function. The latter is studied in next sub-section.

[*] Remember that source data actual freshness is always zero when measuring data currency.

3.4.2. Combination of input values

As argued in Sub-section 3.1, when an activity has several predecessors, the freshness of data coming from them is combined, synthesizing an input freshness value for the activity. That is, given the input values iv_1, iv_2,… iv_n, where iv_i is the freshness value (plus the inter-process delay) of data coming from the ith predecessor, the combineActualFreshness function returns an estimation of the freshness of the whole input:

```
cv= CombineActualFreshness (iv1, iv2,… ivn)
```

The combination strategy depends on the activity semantics. Firstly, if the activity chooses among inputs (e.g. a mediator that returns only the data extracted from the most reputable source), the synthesized value is the input value of such source. Conversely, if the activity merges data from several inputs, the synthesized value should be calculated as a function of the input ones.

A simple combination function considers the worst case, i.e. it returns the maximum of input values. Analogously, the function can return an average (or weighted average) of input values. Typical weights are data volume, data volatility or source reputation). Furthermore, voting strategies can be taken into account (e.g. given more weight to data that is present in more sources). Specialized functions can be defined considering the characteristics of specific scenarios.

Analogously to the calculation of source data actual freshness, processing costs and inter-process delays, the combination function may be based on some DIS properties. We should identify the DIS properties that influence its calculation and determine their calculation strategies.

Example 3.9. Let's analyze the *combineActualFreshness* functions for the three DISs of Example 3.5. For scenarios Sc_2 and Sc_3 the appropriate implementation is taking the maximum of predecessors freshness. However, in scenario Sc_1 users expect to know how fresh is film information, independently to when the cinema data was last updated. We take into account data volatility (movie information is more volatile than actors' information); the strategy consists in ignoring input values of sources providing more stable data. The nature of data dimension is used to define this strategy: expert users assign volatility values. When input activities have the same data volatility, the function returns the maximum input value. □

Next sub-section summarizes the relevant properties and their calculation strategies.

3.4.3. Summary of relevant properties and their calculation

As a result of this step, we produce a list of DIS properties that are necessary for overloading the *getSourceActualFreshness*, *getInterProcessDelay*, *getProcessingCost* and *combineActualFreshness* functions and we sketch the calculation strategies for their implementation. Both properties and strategies will be used in next step for implementing these functions.

Table 3.1 summarizes the DIS properties that are relevant in the calculation of the overloaded functions for the three scenarios of previous examples, indicating, to which graph components they correspond (for example, indicating that only the processing cost of certain activity is relevant) and how to acquire the property values. Table 3.2 summarizes the calculation strategies for them.

		DIS property	**Scope**	**Acquisition**
Sc_1	**Source actual freshness**	Update frequency	All sources	Source providers
Sc_2	**Processing cost**	Communication delay	Wrappers	Statistics of connections to sources
		Processing cost	Activity B_3	Statistics of executions
	Inter-process delay	Synchronization timeout	Control edges incoming activity B_3	System administrator
	Combine actual freshness	Data volatility	Data edges	Expert users (at source level); replicated to edges
Sc_3	**Source actual freshness**	Update frequency	All sources	Source providers
	Processing cost	Interaction cost	Activities C_4 and C_6	Statistics of executions
	Inter-process delay	Refreshment frequency	All activities	System administrator

Table 3.1 – DIS properties used for overloading functions

	Scenario Sc_1	Scenario Sc_2	Scenario Sc_3
Processing cost	Neglect	For wrappers (B_1 and B_2): - communication delay with sources + processing cost For merge activity (B_3): - processing cost (worst cases from statistics)	Processing cost (worst case from statistics)
Inter-process delay	Neglect	For edges incoming B_3: - synchronization timeout For other edges: - neglect	For edges outgoing activities: - 1 / refreshment frequency of input node For other edges: - Neglect
Source data actual freshness	1 / update frequency	Neglect	1 / update frequency
Combine actual freshness	Maximum of input values	When different data volatility: - input value of the most volatile input When equal data volatility: - maximum of input values	Maximum of input values

Table 3.2 – Calculation strategies for overloading functions

In next sub-section we describe how to take into account these properties and strategies in the implementation of freshness evaluation algorithms.

3.5. Instantiation of the evaluation algorithm

The freshness evaluation algorithm can be instantiated to adapt it to a specific scenario, overloading the *getSourceActualFreshness*, *getInterProcessDelay*, *getProcessingCost* and *combineActualFreshness* functions, in order to consider in each function, the DIS properties that are most relevant for the scenario (as discussed in previous sub-section).

Generally, the implementation of the overloaded functions is very simple and consists in obtaining the values of some DIS properties or invoking routines to read statistics. However, for some particular scenario, more complex functions may be implemented. As an example, Table 3.3 shows a pseudocode of the *getSourceActualFreshness* function for scenario Sc_1. The function reads the *update frequency* property, acquired from source providers (as specified in Table 3.1) and registered as a label of source nodes of the quality graph. The calculation follows the strategy described in Table 3.2. As another example, the *getProcessingCost* function for scenario Sc_3 should invoke an external function for reading statistics of processing costs of activities (stored for example in a log) and obtaining the maximum. The DIS should store such statistics during execution (no necessarily at each execution).

```
FUNCTION getSourceActualFreshness (G: QualityGraph, A: Node) RETURNS INTEGER
    INTEGER frequency = G.getPropertyValue(A,"UpdateFrequency");
    RETURN 1 / frequency;
END
```

Table 3.3 – Overloading of a function for scenario Sc_1

The framework does not constraint the type of code that can be used in the implementation of overloaded functions; all user-defined functions can be invoked. In addition, the functions implemented for a scenario may be reused for other scenarios (e.g. the function of Table 3.3 may be also used in scenario Sc_3). Furthermore, alternative implementations might be provided in order to support different freshness estimations, for example, if some users need a metric but other users want to analyze another one. This gives additional flexibility to our approach.

A real application scenario is studied in Chapter 5 (Sub-section 3.2). We identify the relevant DIS properties and overload the corresponding functions to instantiate the freshness evaluation algorithm. .

Next sub-section discusses the propagation of freshness expected values which allows to constraint source providers on the freshness of source data.

3.6. Propagation of freshness expectations

Analogously to the propagation of freshness actual values, we can propagate freshness expected values from target to source nodes. The propagated freshness expected values may help the DIS designer to know the freshness that he should ask the source providers for. A direct application of this propagation strategy is the comparison of alternative data sources in order to select the one that provides the freshest data.

Sub-section 3.1 presented the propagation of freshness actual values from source to target nodes. The propagation of freshness expected values is quite similar but presents some additional problems. In this sub-section we discuss the propagation of freshness expected values and we present the corresponding propagation algorithm.

The propagation algorithm calculates the following property:

– *Expected freshness*: It is an estimation of the expected freshness for data outgoing a node (*Efreshness* for short).

The propagation principle is to traverse the quality graph (in sense inverse to the edges) calculating the expected freshness for data outgoing each node, i.e. the maximum freshness value that may be tolerated for the data produced by the node, in order to achieve freshness expectations. Intuitively, while for calculating actual values we add processing costs and inter-process delays to source data actual freshness, for calculating expected values, we should subtract them from target data expected freshness. A first intuitive propagation algorithm starts with target data expected freshness, and for each node, subtract the processing cost of the node and the inter-process delay with the predecessor node. Next example illustrates the idea:

> **Example 3.10.** Consider the quality graph of Figure 3.7a as input for the propagation of freshness expected values. Actual freshness was propagated with the basic algorithm described in Sub-section 3.1, taking the maximum as combination function (as illustrated in Example 3.4). The expected freshness for (A_3, T_1) is calculated as the target data expected freshness of T_1, i.e. 30 units of time. The expected freshness for (A_1,A_3) is calculated as the expected freshness of (A_3, T_1) minus the cost of A_3 minus the inter-process delay of (A_1,A_3), obtaining a value of 21 (30-4-5) units of time. The other values are calculated analogously. Figure 3.7b shows the output quality graph. □

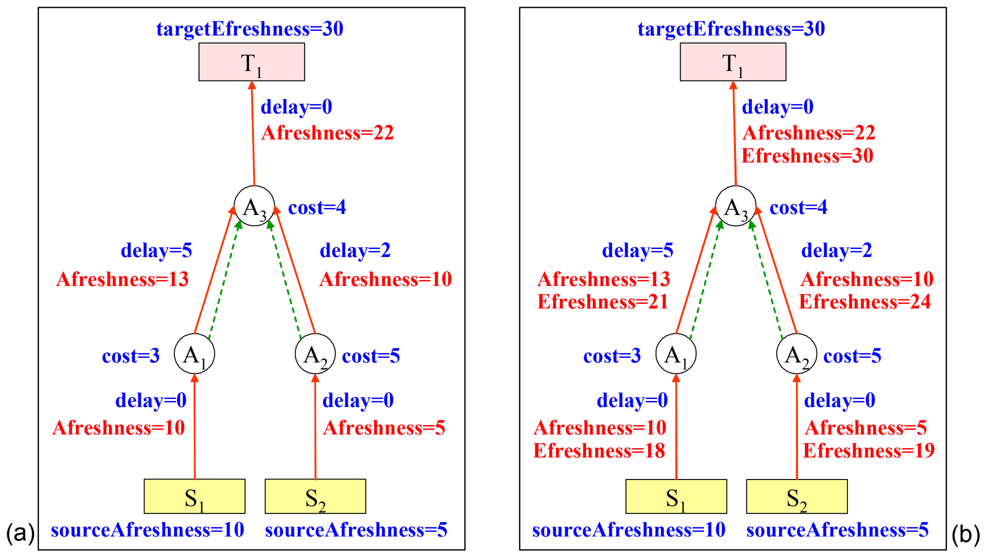

Figure 3.7 – Propagation of freshness expected values: (a) input quality graph, and (b) output quality graph

In previous example, freshness expected values are propagated in the same way to all predecessors. However, in certain scenarios, different values should be propagated to each predecessor. Such differentiation depends on the semantics of the combination function (combineAcutalFreshness), which synthesizes an input freshness value for each node. The following example illustrates this idea:

Example 3.11. Consider again the quality graph of Figure 3.7a but suppose that freshness actual values were propagated using a different combination function which returns the freshness of input data coming from S_1 (for example, because of data volatility). Note that it does not matter which is the freshness of data coming from S_2, if freshness of data from S_1 is good enough, freshness expectations are achieved. For example, if the source data actual freshness of S_2 is 5000 units of time instead of 5 units of time, the actual freshness of (A_3, T_1) continues being 22 units of time, which does not surpass freshness expectations. So, the propagation of freshness expected values in this quality graph should respect this intuition, i.e. the expected freshness for (A_2,A_3) should be infinite. □

When a node has several predecessors, its expected freshness is decomposed among the predecessors obtaining an expected value for each predecessor. We define the *decomposeExpectedFreshness* function for performing such decomposition. Given an input value *iv* corresponding to the expected freshness for data outgoing a node (minus the processing cost of the node), the decomposeExpectedFreshness function returns a set of values $\{v_1, v_2, \ldots v_n\}$ where v_i is the expected freshness value for data incoming the node from the ith predecessor:

$$\{v_1, v_2, \ldots v_n\} = \text{decomposeExpectedFreshness}\ (iv)$$

Previous example motivated that the decomposition of expected values should be coherent to the combination of actual values. Specifically, the decomposeExpectedFreshness function should have the inverse effect of the combineActualFreshness function. The idea is that, if the combination function is applied to the propagated expected values, the synthesized value should satisfy freshness expectations. To see this, let v be the freshness expected value for a node and $v_1, v_2, \ldots v_n$ the freshness expected values propagated to the node predecessors, i.e. decomposeExpectedFreshness (v) = $\{v_1, v_2, \ldots v_n\}$; then, the combination function applied to $v_1, v_2, \ldots v_n$ should return a smaller value than v. In other words, the decomposition function should warranty that combineActualFreshness ($v_1, v_2, \ldots v_n$) $\leq v$.

The expected values propagated to predecessors (i.e. $v_1, v_2, \ldots v_n$) should be the greatest values that allows achieving freshness expectations (i.e. v). In other words, the combination function applied to smaller values should return a synthesized value that is smaller than freshness expectations. For that reason, we require the combineActualFreshness and decomposeExpectedFreshness functions to be monotonic.

The decomposition function is the solution to the following optimization problem:

$$\begin{aligned} &\text{Maximize:} && (v_1, v_2, \ldots v_n) \\ &\text{subject to:} && \text{combineActualFreshness}\ (v_1, v_2, \ldots v_n) \leq v \end{aligned}$$

Unfortunately, for some combination functions the decomposition function cannot be easily deduced. Furthermore, in some situations, we cannot completely determine expected values and we only obtain equations with some degree of freedom, which are not as useful as those illustrated in previous examples. Next example illustratres one of such cases.

Example 3.12. Consider that the combination function for the quality graph of Figure 3.7a performs an average of both input values, i.e. for activity A_3, combineActualFreshness (18,12) = 15. The freshness expected values for predecessors of activity A_3 should be the greater values that verify that combineActualFreshness (v_1,v_2) ≤ 30-4, i.e. $(v_1+v_2)/2 \leq 26$.

In this case, values v_1 and v_2 are not determined and there is a space of solutions that verifies the condition. Specifically, solutions are of the form (v, 52-v}. □

In cases where the optimal decomposition function cannot be exactly determined, the function should be implemented for returning some values that could be useful for a particular scenario but knowing that it will be more restrictive than necessary, for example, decomposeExpectedFreshness (v) = {v,v…v}.

Once the decomposition function has been identified, for a specific scenario, the propagation of expected values is analogous to that of actual values. It can be sketched as follows (see Figure 3.8): First, for each successor activity (C_1… C_m), we obtain the expected freshness. We take the most restrictive value (the minimum) and we subtract the processing cost. Then, we decompose such value in a set of values, one for each predecessor activity (P_1… P_n), and we subtract, from each value, the inter-process delay with the corresponding predecessor.

```
1.  $u_1$ = ExpectedFreshness(A,$C_1$) …
    $u_m$ = ExpectedFreshness(A,$C_m$)
2.  v = min {$u_1$,…$u_m$} – ProcessingCost(A)
3.  {$v_1$,…$v_n$} = DecomposeExpectedFreshness (v)
4.  ExpectedFreshness ($P_1$,A) = $v_1$ -  InterProcessDelay ($P_1$,A) …
    ExpectedFreshness ($P_n$,A) = $v_n$ -  InterProcessDelay ($P_n$,A)
```

Figure 3.8 – Expected freshness propagation strategy

The previous strategy is implemented in an algorithm for propagating freshness expectations: *ExpectedFreshnessPropagation* (see Algorithm 3.2). The algorithm first spans target nodes, obtaining target data expected freshness and storing the expected freshness for each incoming edge. Then, the algorithm traverses activity nodes, calculating the minimum expected freshness of outgoing edges and subtracting the processing cost. The resulting value is decomposed obtaining an expected value for each predecessor, in a list (valList). Each value of the list is subtracted of the corresponding inter-process delay and stored for the incoming edge. A pseudocode of the algorithm can be sketched as shown in Algorithm 3.2.

The *decomposeExpectedFreshness* function is an abstract function that should be overloaded to implement the adequate decomposition strategy.

```
FUNCTION ExpectedFreshnessPropagation (G: QualityGraph) RETURNS QualityGraph
    INTEGER efreshness, value, m;
    FOR EACH target node T DO
        efreshness = getTargetExpectedFreshness(G,T);
        EDGE e = data edge incoming T in G
        G.addProperty(e,"ExpectedFreshness",efreshness);
    ENDFOR;
    FOR EACH activity node A in inverse topological order of G DO
        m = min ({G.getPropertyValue(e,"ExpectedFreshness") / e is a data edge outgoing A in G})
        value= m – getProcessingCost(G,A);
        HASHTABLE valList = edges incoming A in G;
        decomposeExpectedFreshness (value, valList);
        FOR EACH data edge e incoming A in G DO
            efreshness = valList.get(e) – getInterProcessDelay(G,e);
            G.addProperty(e,"ExpectedFreshness",efreshness);
        ENDFOR;
    ENDFOR;
    RETURN G;
END
```

Algorithm 3.2 – Basic algorithm for propagating freshness expected values

Next sub-section presents some direct applications of both types of freshness propagations.

3.7. Usages of the approach

The algorithms for propagating actual and expected values of data freshness proposed in previous sub-sections can be used in different contexts, either in order to evaluate data freshness in an existing DIS or to determine constraints for DIS development. This sub-section summarizes the freshness evaluation approach by briefly presenting some usages of the approach for different purposes and at different development stages. Some of these usages are developed in the remaining of the chapter (Sections 4 and 5) and in Chapter 5.

3.7.1. Evaluating data freshness at different phases of the DIS lifecycle

The data freshness evaluation approach can be used at different phases of the DIS lifecycle, e.g. at design, production or maintenance phases. Depending on the phase, a quality graph may represent an existing DIS (implemented, operative) or a specification of a DIS (not yet implemented). In the former, precise property values (source data actual freshness, processing costs and inter-process delays) can be obtained during DIS execution. In the latter, property values should be estimated, generally based on cost models. In both cases, property values can be upper bounds (e.g. worst case processing costs) so the evaluation algorithm will obtain upper bounds for data freshness.

Some usages of the approach, in an existing DIS are:

- At query evaluation: We can evaluate data freshness during query evaluation in order to obtain precise freshness values of the conveyed data (possibly labeling data with its freshness values). The actual freshness evaluation algorithm can be integrated to the query evaluation process. Precise property values can be obtained during DIS execution. In that case, we obtain the actual freshness of the conveyed data.
- At query planning: We can evaluate data freshness before executing a query in order to predict the freshness of data that may be returned in response to the query. Data freshness evaluation should be based on estimations of property values (e.g. using cost models or statistics of previous query evaluations). This is the case of virtual DIS (e.g. mediation systems) where the calculation operations (activities) are decided for each user query. Query optimizers use costs models and statistics for predicting operations costs.
- At DIS monitoring: We can evaluate data freshness offline (after executing several queries) in order to monitor if the DIS has quality problems. Data freshness evaluation should be based on estimations of property values (generally based on statistics of DIS executions). The data freshness evaluation algorithm can be integrated to a quality auditing tool.

Some usages of the approach, in a DIS specification are:

- At DIS design: We can evaluate data freshness in a (conceptual or logical) model of the DIS in order to validate the model. Data freshness evaluation should be based on estimations of DIS properties (e.g. using cost models).
- At DIS maintenance: When doing modifications to DIS design (e.g. changing implementation of some activities or adding new components for providing additional processing), we can evaluate data freshness of the new model in order to validate the changes or decide if they are convenient. Statistics or cost models for the new/modified components can be used as estimations of DIS properties.
- At DIS reengineering: When reengineering a DIS (possibly because of quality problems), we can evaluate data freshness in a new version of the DIS (e.g. substituting some activities by more performing ones) in order to validate the modifications. Data freshness evaluation should be based on statistics of DIS executions and estimations of property values for the new components.

Note that for query evaluation and DIS monitoring data freshness evaluation is performed *a posteriori*, i.e. after executing queries, however, for query planning data freshness evaluation is performed *a priori*, i.e. before executing queries. In a DIS specification, data freshness evaluation is also performed *a priori*.

3.7.2. Different applications of the evaluation approach

Both types of propagations (of actual and expected values), either a priori or a posteriori, can be used for different purposes, ranging from simple quality assessment (e.g. for informing users of actual freshness) to quality improvement (e.g. for analyzing improvement actions). In this sub-section we discuss several usages of the approach.

Data freshness assessment

A first use of the approach consists simply on evaluating data freshness and communicating it to the correspondent actors (designers, administrators, users, source providers). The evaluation results can be used for different purposes, for example:

- *Communicating data freshness to users*: Propagating freshness actual values, from sources to targets, we obtain metadata (data freshness) that qualifies conveyed data. Freshness values can be communicated to users either as labels of conveyed data (in response to a query) or as upper bounds (before executing the query). In both cases, providing data freshness values to users represent a value-added to the data.

- *Estimating data freshness*: Using estimations of property values (e.g. based on cost models or statistics) we can estimate the freshness of data that may be returned by a DIS. Results can validate design decisions or motivate the development of more performing components. So, data freshness may have impact in early design steps. In addition, at maintenance phase, freshness estimations may serve for monitoring the DIS.
- *Specifying constraints for source data actual freshness*: Propagating freshness expected values, from targets to sources, we obtain freshness expected values for source data. These values may serve as constraints (upper bounds for source data actual freshness) for warranting the achievement of freshness expectations. They can be communicated to source providers when negotiating the service. In that way, freshness expectations also may have impact in early design steps.
- *Specifying constraints for DIS development*: The comparison among target data expected freshness and source data actual freshness serves to determine constraints for DIS implementation. Their difference, if positive, indicates the longest period of time that DIS processes are allowed to spend in manipulating data, i.e. it is an upper bound for the execution delay of the DIS which includes the processing costs of activities and the inter-process delays among them. Examples of constraints are: maximal processing costs for activities, minimal execution frequencies for activities, minimal refreshment frequencies for materialized data and minimal access frequencies for data sources.

Comparison of different DIS implementations

A direct application of the freshness evaluation approach is the comparison among different quality graphs. The application consists in evaluating data quality in each quality graph and selecting the one that produces data with the highest quality (data freshness can be the unique quality criteria or can be balanced with other ones). The quality graphs may differentiate, for example, in the access to alternative data sources, in the use of different activities and in the interaction among activities (data and control flows).

- *Selecting a quality graph:* Freshness actual values can be propagated, from sources to targets, obtaining measures for comparing among graphs. The comparison can be done at different moments, for example, at design phase, we can compare different (conceptual or logical) models for the DIS, at production phase (query planning), we can compare alternative execution plans, at maintenance phase we can compare different modifications to DIS implementation.
- *Selecting a data source:* Given a model for the DIS, accessing to some generic sources (specification of the data types that must be provided by sources), we can propagate freshness expected values to sources obtaining constrains for source data. Then, candidate sources can be compared in order to select the one that provides data with the highest quality. Analogously, the comparison can be done at the different phases of the DIS lifecycle.

In this line, we used the freshness evaluation approach in a mediation application in the context of data personalization. Complementing a procedure that automatically generates mediation queries for accessing alternative data sources, the evaluation approach was used for estimating data freshness of the generated queries in order to select the best one for a given user [Kostadinov+2004]. This application is described in Chapter 5 (Sub-section 3.1).

Data freshness analysis

The data freshness propagation algorithms can be used as auditing tools for measuring the quality of the data produced by the DIS and analyzing the DIS based on data quality. Analysis includes checking if freshness expectations are satisfied and, if not, determining the critical points of the DIS. Both types of propagations are useful:

- *Checking if freshness expectations are satisfied for targets*: We can compare freshness values obtained with the *ActualFreshnessPropagation* algorithm with those expected by users (target data expected freshness). The comparison allows finding the data targets for which freshness expectations are satisfied and those for which freshness expectations are overdrawn. This result may be important for informing users about the expectations that are not satisfied (inviting them to relax expectations).

- *Checking if freshness expectations are satisfied by sources*: Analogously, we can compare freshness values obtained with the *ExpectedFreshnessPropagation* algorithm with those provided by sources (source data actual freshness). The comparison allows finding the data sources for which freshness expectations are satisfied and those for which freshness expectations are overdrawn. This result may be important for informing source providers about data that is not satisfactory for users (inviting them to provide a better service. i.e. fresher data).

In both cases, we identify portions of the DIS (targets, sources, paths from sources to targets) that may be analyzed in detail and possibly improved. Analysis can be done at design phase in order to validate the model, at production phase in order to monitor the DIS and at maintenance phase in order to suggest modifications. Sub-sections 4.1 to 4.3 discuss data freshness analysis.

Data freshness improvement

If freshness expectations are overdrawn, different improvement actions can be taken in order to enforce data freshness. Certain strategies intent to improve DIS implementation (e.g. substituting an activity by a more performing component, synchronizing activities for reducing delays, powering hardware for accessing more frequently or performingly to data sources) and other ones intent to obtain fresher source data (e.g. substituting a data source, negotiating with source data providers for relaxing access constraints). Sub-section 4.4 discusses improvement actions.

Next section discusses data freshness enforcement. It utilizes both types of quality propagations for analyzing the quality graph, building a diagnostic on the achievement of freshness expectations and highlighting the bottlenecks for data freshness achievement, i.e. the portions of the quality graph that should be improved in order to enforce data freshness. We also present some improvement actions that can be applied to those bottlenecks in order to enforce data freshness.

4. Data freshness enforcement

The DIS should provide, for each target node, the data freshness expected by the users. The freshness evaluation approach, presented in previous section, allows checking if a DIS satisfies freshness expectations. To do such validation, we can calculate the freshness actual values for target nodes and compare them with those expected by users. If freshness actual values are lower than expected values then freshness can be guaranteed. If freshness actual values are greater than expected values, freshness expectations are not achieved and some improvement actions should be followed in order to enforce freshness. Examples of improvement actions are improving the implementation of activities in order to reduce their processing cost and synchronizing activities in order to reduce inter-process delays among them. The identification of critical paths (portions of the quality graph that represent bottlenecks for freshness calculation) allows concentrating improvement actions in the subsets of activities that cause freshness expectations to be overdrawn.

In this section we present an approach for analyzing the quality graph at different levels of abstraction in order to identify critical paths and apply improvement actions to the nodes in the path.

4.1. Top-down analysis of data freshness

In this sub-section we propose an approach for analyzing data freshness in a top-down way, inspired from OLAP browsing mechanisms. Generally, OLAP users start analyzing aggregated data and when abnormal or warming situations are suspected, they perform drill-downs on identified data in order to obtain further information for understanding the phenomena, determining causes and conceiving solutions. Analogously, the proposed approach first analyzes data freshness in a high-level quality graph (macroscopic representation of the DIS) and analyzes more detailed quality graphs (microscopic representations of the DIS) when further details are necessaries for enforcing data freshness. Data freshness analysis consists in propagating data freshness values as discussed in previous section (ActualFreshnessPropagation algorithm), checking if freshness expectations can be achieved. If freshness values satisfy freshness expectations the DIS is appropriate for users freshness needs, however, if freshness values overdraw freshness expectations, detailed information about DIS activities (i.e. a lower level quality graph) may be useful for identifying bottlenecks and proposing improvement actions.

We consider a hierarchy of representations of the DIS processes (quality graphs at different abstraction levels) and two operators: *level-up* and *level-down*, which allow changing from a representation to another one, i.e. changing the level of abstraction. Figure 3.9 shows a hierarchy of quality graphs composed of three levels; the highest-level graph has a unique activity representing the whole DIS, allowing the visualization of the data sources that participate in the calculation of data targets; the intermediate-level graph abstracts macro activities and the lowest-level graph shows details about activity logics (i.e. detailed data and control flows).

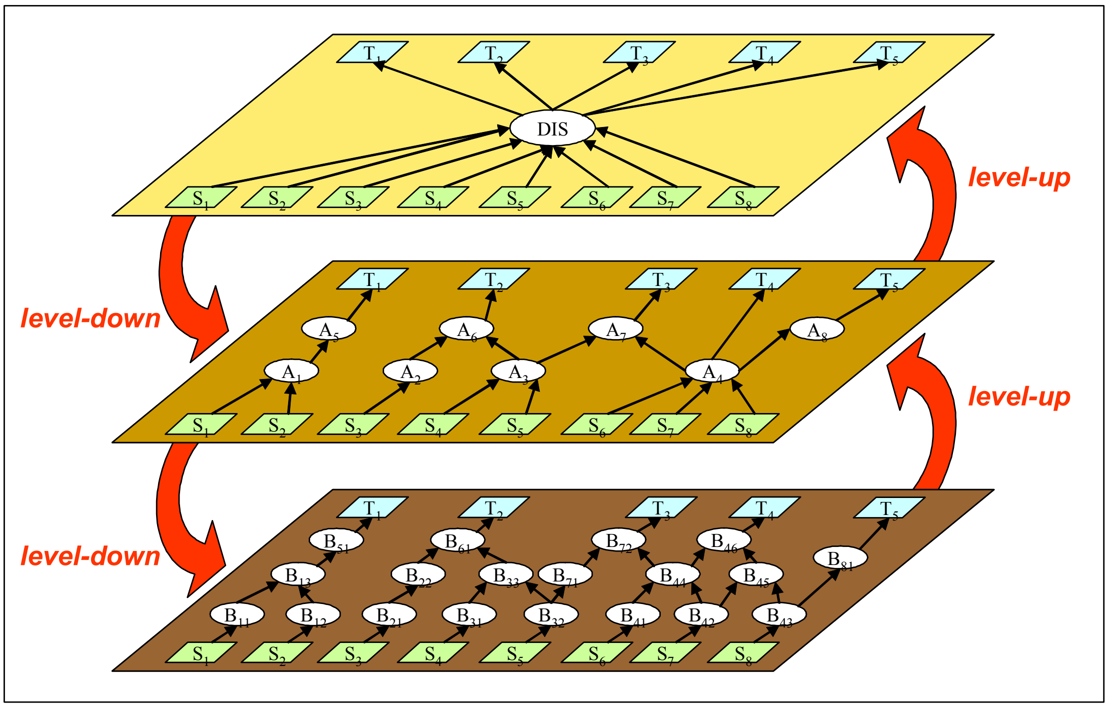

Figure 3.9 – Hierarchy of quality graphs

Next sub-sections model the hierarchy of quality graphs, describe the operations for browsing inside the model and aggregating processing costs and inter-process delays for high-level graphs, and discuss data freshness evaluation at different levels of the hierarchy.

4.1.1. Hierarchy of quality graphs

For representing the hierarchy of quality graphs we need two components: (i) an ordered sequence of quality graphs (each graph increasing the level of abstraction of its predecessor in the sequence), and (ii) a relation among activities of consecutive levels, indicating that they correspond to the same conceptual task.

For representing the latter, we consider a hierarchy of activities where the root represents the whole DIS and each level represents DIS activities at a given level of abstraction. High-level activities abstract high-level tasks while lower-level activities show the processing details of the tasks. Ascending in the hierarchy correspond to abstracting task behaviors while descending in the hierarchy can be interpreted as decomposing an activity in more detailed and precise sub-tasks. For example, certain extraction activity may be decomposed in a sequence of sub-activities that perform the connection to the source, the execution of an extraction query, the filtering of irrelevant or erroneous data and the storage of the resulting data. Figure 3.10 shows the hierarchy of activities for the DISs of Figure 3.9. Note that all branches of the hierarchy tree have the same length.

We represent the hierarchy of quality graphs with the above-mentioned components: a hierarchy of activities and an array of quality graphs:

Definition 3.7 (hierarchy of quality graphs). A *hierarchy of quality graphs* is a pair (AH, GH) where *AH* is a tree of activities representing the hierarchy of activities and *GH* is a list of quality graphs representing the levels of abstraction. □

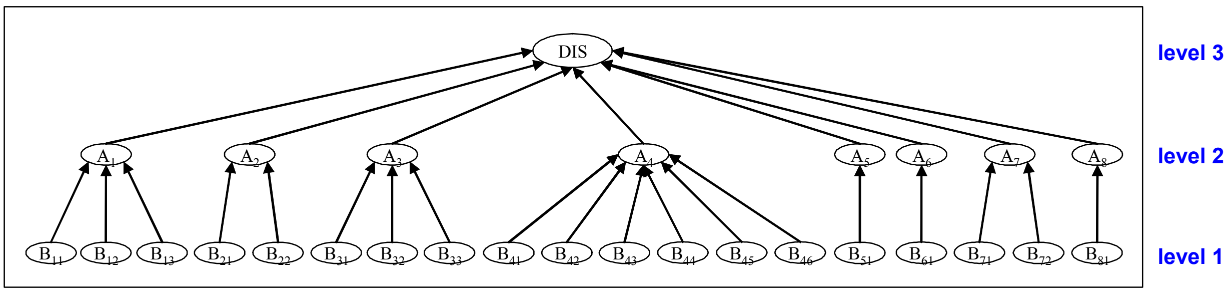

Figure 3.10 – Hierarchy of activities

When referring to two consecutive graphs in the hierarchy, the lowest-level graph is called *detailed graph* and the highest-level graph is called *summarized graph*.

In the following, we show how to build a summarized graph from a detailed one, how to aggregate property values and how to evaluate data freshness in it. To this end, we need to make some hypotheses about the hierarchy of quality graphs and the propagation of data freshness. The first hypothesis concerns the semantics of activities. It states that the execution of sub-activities must have the same effect than executing their parent activity, i.e. take the same inputs and produce the same outputs (both in data and control flows). Consequently, there may exist an edge between two activities (A and B) if and only if there is an edge between a sub-activity of A and a sub-activity of B. The second hypothesis states that data and control flow of the lowest-level quality graph must warranty that there will be no cycles at any abstraction level. In other works, if there is a path among a sub-activity descendent of A and a sub-activity descendent of B (being A and B two activities of the same high-level quality graph), it must not exist a path among any sub-activity descendent of B and any sub-activity descendent of A. The third hypothesis states that sibling sub-activities must be connected in the corresponding quality graph. This hypothesis is necessary to calculate processing costs of high-level activities, as will be explained in next sub-section. The last hypothesis states that the combineActualFreshness function returns the maximum of input values and consequently, the decomposeExpectedFreshness function returns the same value to all predecessors. This hypothesis is necessary to prove that freshness values can be propagated at all abstraction levels. Using different combination and decomposition functions we should make analogous proofs that those that will be shown in Sub-section 4.3. We have chosen these functions because they are the most typically used in several types of DIS, for example [Naumann+1999] [Braumandl 2003].

Given a quality graph of level L and a hierarchy of activities, the quality graph of level L+1 can be built using the *buildLevelGraph* function (see Algorithm 3.3). The method starts adding source and target nodes (which are the same for all levels) and then adds the activity nodes of level L+1. An edge between two nodes is created whether there is an edge between children activities. Given the quality graph of lowest-level and the hierarchy of activities, all higher-level quality graphs can be built invoking successively the *buildLevelGraph* function.

The GraphHierarchy class has methods for managing the array of quality graphs and the tree of activities:

```
FUNCTION getLevel (G: QualityGraph) RETURNS INTEGER
FUNCTION getLevel (A: Activity) RETURNS INTEGER
FUNCTION getLevelGraph (level: INTEGER) RETURNS QualityGraph
FUNCTION getLevelGraph (A: Activity) RETURNS QualityGraph
FUNCTION levelUp (G: QualityGraph) RETURNS QualityGraph
FUNCTION levelDown (G: QualityGraph) RETURNS QualityGraph
FUNCTION getSubActivities (A: Activity) RETURNS ActivitySet
FUNCTION getParentActivity (A: Activity) RETURNS Activity
FUNCTION getSiblingActivities (A: Activity) RETURNS ActivitySet
```

The *getLevel* functions return the level (in the hierarchy) of a quality graph, giving either the graph or one of its activities. The *getLevelGraph* functions return the quality graph of a level, giving either the level or one of its activities. The *levelUp* and *levelDown* functions return the quality graphs that are immediately upper (less detail) and lower (more detail) in the hierarchy than a given quality graph, respectively. The getSubActivities, getParentActivity and getSiblingActivities functions return the corresponding neighbors in the hierarchy of activities.

```
FUNCTION buildLevelGraph (AH: TREE OF Activity, int level, G: QualityGraph)
                                 RETURNS QualityGraph
    QualityGraph Q;
    Q.insertNodes (G.getSourceNodes());
    Q.insertNodes (G.getTargetNodes());
    Q.insertNodes (AH.getLevelNodes(level));
    FOR EACH edge e=(B1,B2) in G DO
        IF (B1 is a source or target node) THEN A1=B1;
        ELSE A1 = AH.getParentActivity(B1);
        IF (B2 is a source or target node) THEN A2=B2;
        ELSE A2 = AH.getParentActivity(B2);
        Q.insertEdge (A1,A2,e.getType());
    ENDFOR;
    RETURN Q;
END
```

Algorithm 3.3 – Method for building the hierarchy of quality graphs

In order to evaluate data freshness in high-level quality graphs, processing costs and inter-process delays should be calculated from the processing costs and inter-process delays of lower-level graphs (which is analogous, following the analogy with OLAP applications, to the aggregation of measures, i.e. the roll-up operation). Next sub-section discusses their calculation.

4.1.2. Labeling of high level quality graphs

In this sub-section we deal with the calculation of processing costs and inter-process delays of high-level quality graphs. In order to simplify the analysis, we label quality graphs with such properties[*]. The labeling of the lowest-level quality graph can be done executing the getSourceActualFreshness, getProcessingCost and getInterProcessDelay functions discussed in Section 3.

Intuitively, the processing cost of an activity in the summarized graph should be the time necessary to execute all its sub-activities in the detailed graph. In other words, the processing cost should equal the extent of time between the start of execution of the initial sub-activity (the first one in starting execution) and the end of execution of the final sub-activity (the last one in finishing execution). In order to find initial and final sub-activities, and consequently calculating the processing cost, sub-activities are scheduled, respecting the processing costs and inter-process delays of the detailed graph, as follows:

> **Definition 3.8 (execution schedule)**. An *execution schedule* is a function that maps sub-activities to time intervals of the form $<t_A,T_A>$, with $0 \leq t_A \leq T_A$; t_A is called the *starting time* and T_A is called the *ending time* of the execution of sub-activity A. A schedule verifies the following conditions: (i) for each sub-activity, the length of its interval equals its processing cost, (ii) for each pair of consecutive sub-activities, the separation between their intervals equals the inter-process delay between them, (iii) for some sub-activity, t_A=0. The sub-activity (or sub-activities) that satisfies the last condition is called *initial sub-activity*. The sub-activity (or sub-activities) that has maximum ending time is called *final sub-activity*. □

Execution schedules can be visualized using Gantt charts as illustrated in Figure 3.11b. Time-intervals are represented by rectangles, which lengths indicate processing costs. Separations among rectangles represent inter-process delays.

[*] The same analysis can be done without labeling quality graphs but overloading the getSourceActualFreshness, getProcessingCost and getInterProcessDelay functions for high-level quality graphs. For simplifying the understanding of the approach, we calculate property values, label quality graphs with them and overload functions for simply reading the property values.

Execution schedules can be computed using the Critical Path Method (CPM) [Hiller+1991]. The method uses a *potential task graph*, which is a directed labeled graph whose nodes represent tasks (activities) and whose edges represent precedence constraints, labeled with time constraints (delays). The potential task graph for a set of sub-activities $\{B_1, \ldots B_n\}$ can be built from the detailed graph. The problem of finding starting times for sub-activities reduces to finding most expensive paths in the potential task graph. The Ford algorithm [Hiller+1991] computes starting times with order $O(m^3)$ being m the number of edges of the graph. Ending times are calculated adding processing costs to starting times.

The following example illustrates the use of the execution schedule for calculating processing costs.

Example 3.13. Consider an activity B with 3 sub-activities: B_1, B_2 and B_3. Figure 3.11a shows a portion of the quality graph that contains the sub-activities. Figure 3.11b shows the execution schedule corresponding to the quality graph. The starting times of B_1, B_2 and B_3 are 1, 0 and 4 respectively, while their ending times are 3, 2 and 8 respectively. The initial sub-activity is B_1 and the final sub-activity is B_3. □

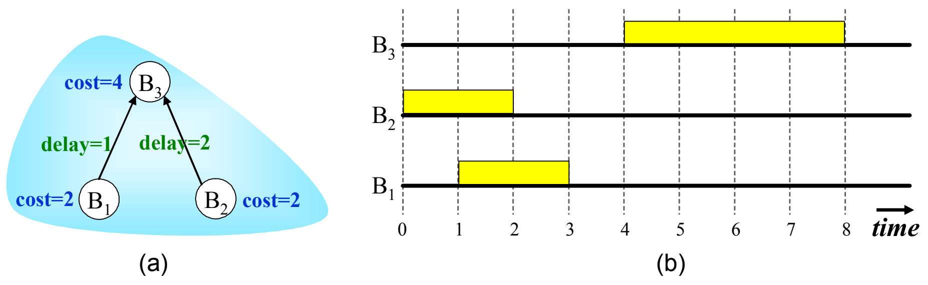

Figure 3.11 – Calculation of starting and ending times of sub-activities: (a) portion of a quality graph, (b) execution schedule

Now, we can formalize the calculation of processing costs in a high-level graph. Let A be an activity of a summarized graph S and let $\{A_1 \ldots A_m\}$ be the sub-activities of A in a detailed graph D. The processing cost of A is calculated as the time passed from the starting time of its initial sub-activity (zero) to the ending time of its final sub-activity. Figure 3.12 shows the calculation formula.

$$\begin{aligned} \text{ProcessingCost (A)} &= \text{EndingTime(FinalSubActivity(A))} \\ &= \max \{\text{EndingTime}(A_1), \ldots \text{EndingTime}(A_m)\} \end{aligned}$$

Figure 3.12 – Calculation of processing costs in high-level graphs

In order to calculate inter-process delays, we should consider delays among sub-activities but also take into account the starting and ending times of sub-activities. Intuitively, the inter-process delay between two activities A and B in a summarized graph should be the difference of time between the executions of their sub-activities in the detailed graph. Specifically, inter-process delay should equal the extent of time between the ending time of the final sub-activity of A and the starting time of the initial sub-activity of B. To this end, the execution schedules of activities A and B can be concatenated, respecting the inter-process delay among sub-activities, as shown in Figure 3.13b.

Example 3.14. Consider two activities A and B of a summarized graph with sub-activities $\{A_1, A_2\}$ and $\{B_1, B_2\}$ respectively. Figure 3.13a shows a portion of the detailed graph that contains the sub-activities; shadow zones highlight siblings in the hierarchy of activities. Figure 3.13b shows the execution schedules (global time is added at the top of the figure for facilitating the visualization of inter-process delays). As the inter-process delay of (B_1, A_2) is 8 units of time, the schedules are shifted in 8 units of time respect to global time. The inter-process delay of (B,A) is easily seen in the graphic: it is 3 units of time. □

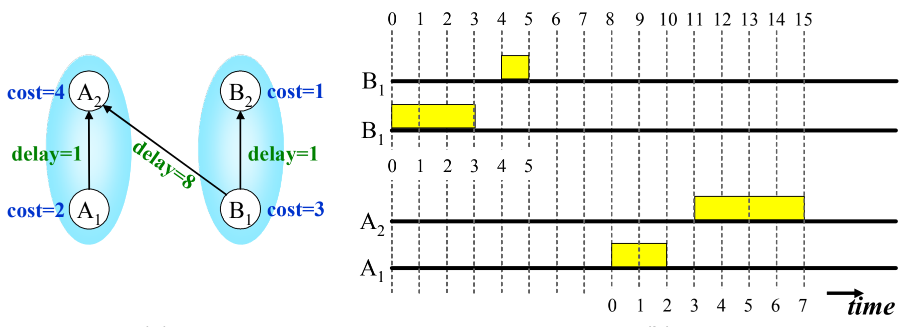

Figure 3.13 – Concatenation of execution schedules: (a) portion of a quality graph, (b) execution schedules

Now, we can formalize the calculation of inter-process delays in a high-level graph. Let A and B be two activities of a summarized graph S and let $\{A_1 \ldots A_m\}$ and $\{B_1 \ldots B_n\}$ be their sub-activities in a detailed graph D, respectively. The inter-process delay between two sub-activities A_i and B_j may be greater or equal to the inter-process delay between A and B. Equality is achieved when A_i is a final sub-activity of A and B_i is an initial sub-activity of B. In the rest of the cases, the difference of time between the ending time of a final sub-activity of A and the ending time of A_i, and the difference of time between the starting time of B_j and the starting time of an initial sub-activity of B (which is time 0), are both subtracted from the inter-process delay of (A_i,B_j). Note that inter-process delays can be negative in a summarized graph if B starts execution before A finishes. Figure 3.14 shows the calculation of inter-process delays.

```
InterProcessDelay (A,B) = max {InterProcessDelay (Aᵢ,Bⱼ) –
          (EndingTime(FinalSubActivity(A)) – EndingTime(Aᵢ)) – StartingTime(Bⱼ)
             / (Aᵢ,Bⱼ) is a data edge of D, 1≤i≤m, 1≤j≤n }
```

Figure 3.14 – Calculation of inter-process delays in high level graphs

The maximum is taken in the case there exist several data edges among sub-activities of A and sub-activities of B, despite the same value should be obtained if inter-process delays are coherently assigned in the detailed graph.

Next example summarizes the calculation of processing costs and inter-process delays.

Example 3.15. Figure 3.15 shows a summarized graph S with three activities (A_3, A_6 and A_7) and a detailed graph D with six sub-activities (B_{31}, B_{32}, B_{33}, B_{61}, B_{71} and B_{72}); both graphs are simplified versions or those of Figure 3.9. Figure 3.15a shows processing costs and inter-process delays of D while Figure 3.11b shows starting and ending times of sub-activities (e.g. the starting time of B_{33} is 4 units of time and its ending time is 8 units of time). The processing cost of A_3 is calculated as the ending time of its final sub-activity (B_{33}), i.e. 8 units of time. The inter-process delay between A_3 and A_7 is calculated as the inter-process delay between B_{32} and B_{71} minus the difference of ending times among B_{33} and B_{32} minus the starting time of B_{71}, obtaining a value of -3 (3 – (8-2) – 0) units of time. The other processing costs and inter-process delays are calculated analogously, following the formulas of Figure 3.12 and Figure 3.14. □

Next sub-section discusses the use of such labels for calculating data freshness in high-level quality graphs.

4.1.3. Data freshness evaluation at different abstraction levels

Having built the hierarchy of quality graphs and labeled the graphs with processing cost, inter-process delay and source data actual freshness properties, we can evaluate data freshness at the different abstraction levels using the ActualFreshnessPropagation algorithm. Lemma 3.1 proves that freshness values calculated in a summarized graph are greater or equal to those calculated in a detailed graph. In other words, the propagation of freshness values in high-level graphs brings upper bounds for data freshness.

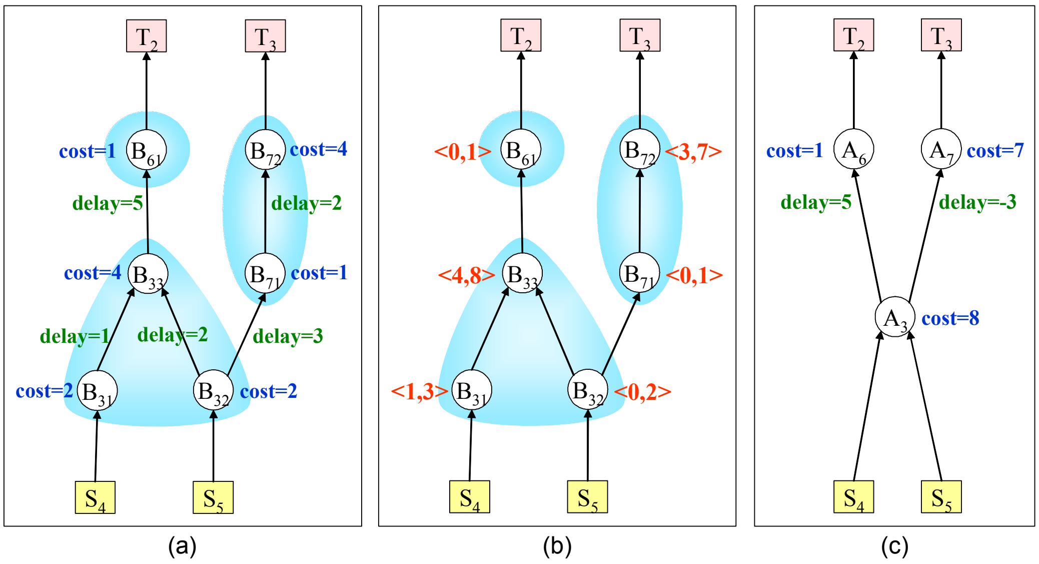

Figure 3.15 – Calculation of processing costs and inter-process delays in high level graphs: (a) detailed graph, (b) starting and ending times in the detailed graph, and (c) summarized graph

---

Lemma 3.1. Given a summarized graph S obtained applying the level-up operator over a detailed graph D, it is verified that:

∀ target node V (being B the predecessor of V in D and being A the parent activity of B in S) . Freshness(A,V) ≥ Freshness(B,V)

The proof is deferred to Sub-section 4.3 because we need to use some results presented in such sub-section.

In most situations, the level-up method preserves freshness actual values propagated to target nodes, i.e. the same freshness values are obtained if evaluation is performed in a summarized or in a detailed graph. Exceptions may occur when an activity has several initial and final sub-activities, as illustrated in next example. In such exceptions, some freshness values of the summarized graph may be greater than those of the detailed graph.

Example 3.16. Consider the propagation of freshness actual values in the summarized and detailed graphs of Figure 3.16 (S and D respectively). Note that actual freshness of data incoming V_2 is preserved, but actual freshness of data incoming V_1 is considerably lower in the detailed graph. The reason is that there is a path from S_2 to T_1 in S but there is no path between them in D. □

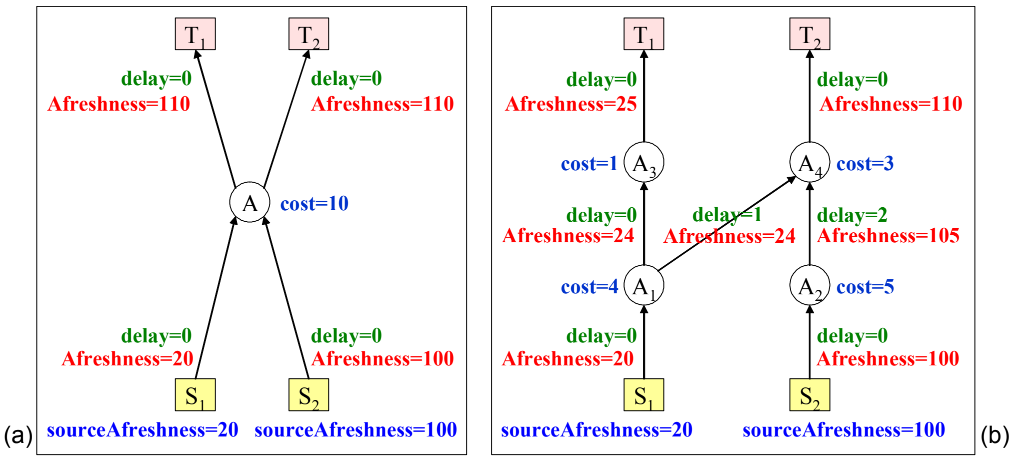

Figure 3.16 – Propagating data freshness: (a) summarized graph, (b) detailed graph

Lemma 3.1 warranties that data freshness analysis realized in high-level graphs are valid in lower-level ones, i.e. if freshness expectations are satisfied in high-level graphs they are achieved in lower-level ones. When expectations are overdrawn, the level-down method allows decomposing activities and analyzing the graph in detail in order to obtain more accurate freshness values. Furthermore, better adapted improvement actions can be applied to detailed graphs in order to enforce data freshness (which will be discussed in Sub-section 4.4).

The *levelUp* and *levelDown* functions allow browsing in the hierarchy of quality graphs, but they return the whole graph of the corresponding level. In order to analyze in detail certain activities (e.g. for trying to optimize them and then enforce data freshness) we need methods for browsing portions of the quality graphs. Next sub-section presents such methods.

4.2. Browsing among quality graphs

There are two basic operations to refine the analysis of an activity: focus+ and zoom+. Focusing on an activity means concentrating the analysis in the activity ignoring the other ones, in other words, keeping only the portion of the quality graph that is relevant for the analysis: the activity and its neighbors. Zooming in an activity means analyzing the sub-activities that compose the given activity, in other words, descending a level in the hierarchy of quality graphs and showing the sub-activities and their neighbors. The focus– and zoom–methods achieve the inverse effects.

Example 3.17. Consider the top-down analysis of Figure 3.17 over the hierarchy of quality graphs of Figure 3.9. We start the analysis at the high-level quality graph (first panel). The second panel results of applying the zoom+ operation to the node representing the DIS; it shows the main DIS activities. Suppose that we want to analyze activity A_4, for example, for trying to reduce its processing cost. To this end, we apply the focus+ operation to A_4 (third panel), showing A_4 and its neighbor nodes. The last panel results of applying the zoom+ operation to A_4; it shows the sub-activities that compose it and their neighbor nodes. Note that after applying some operations, source and target nodes might represent activities. Applying zoom–and focus– operations in reverse order we obtain the highest-level quality graph. □

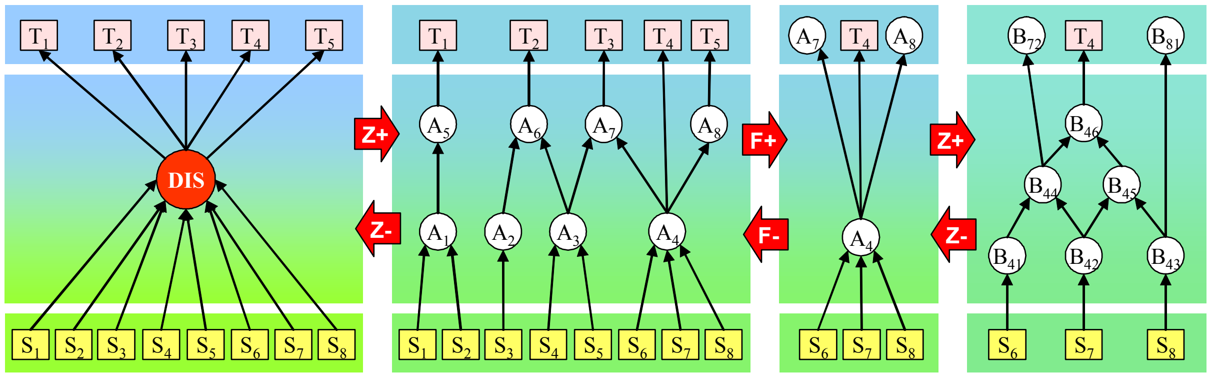

Figure 3.17 – Operations for browsing the hierarchy of quality graphs: zoom+ (Z+), zoom– (Z–), focus+ (F+) and focus– (F–)

Algorithm 3.4 shows the pseudocodes of the focusing methods. The *focus+* method returns a sub-graph of the given quality graph containing the given activity, its predecessors and successors and the edges among them, i.e. it returns the sub-graph induced by an activity and its neighbors. The *neighborSubGraph* method (see Algorithm 3.6) returns the portion of the quality graph of a given level induced by a given set of nodes and their neighbors. The *focus–* method returns a quality graph (of the same abstraction level than the given one) containing the activity, its sibling nodes (in the hierarchy of activities), their predecessors and successors and the edges among them, i.e. it returns the sub-graph of the level graph induced by an activity, its siblings and their neighbors.

Algorithm 3.5 shows the pseudocodes of the zooming methods. The *zoom+* method returns a quality graph (of the immediately inferior abstraction level) containing the sub-activities of the given activity, their predecessors and successors and the edges among them, i.e. it returns the sub-graph of the inferior-level graph induced by sub-activities and their neighbors. The *zoom–* method returns a quality graph (of the immediately superior abstraction level) containing the parent activity of the given one, their predecessors and successors and the edges among them, i.e. it returns the sub-graph of the upper-level graph induced by the parent activity and their neighbors.

```
FUNCTION focus+ (G: QualityGraph, A: Activity) RETURNS QualityGraph
    RETURN neighborSubGraph (G,{A});
END
FUNCTION focus– (GH: GraphHierarchy, A: Activity) RETURNS QualityGraph
    QualityGraph G = GH.getLevelGraph(A);
    ActivitySet SA = GH.getSiblingActivities(A)) ∪ {A};
    RETURN neighborSubGraph (G,SA);
END
```

Algorithm 3.4 – Focusing methods

```
FUNCTION zoom+ (GH: GraphHierarchy, Activity A) RETURNS QualityGraph
    QualityGraph G1 = GH.getLevelGraph(A);
    QualityGraph G2 = GH.levelDown(G1);
    ActivitySet SA = GH.getSubActivities(A);
    RETURN neighborSubGraph (G2,SA);
END
FUNCTION zoom– (GH: GraphHierarchy, Activity B) RETURNS QualityGraph
    QualityGraph G1 = GH.getLevelGraph(A);
    QualityGraph G2 = GH.levelUp(G1);
    Activity A = GH.getParentActivity(B);
    RETURN neighborSubGraph (G2,{A});
END
```

Algorithm 3.5 – Zooming methods

```
FUNCTION neighborSubGraph (G: QualityGraph, SA: ActivitySet) RETURNS QualityGraph
    QualityGraph Q;
    Q.insertNodes (SA);
    FOR EACH node A of SA DO
        Q.insertNodes (G.getPredecessors(A));
        Q.insertNodes (G.getSuccessors(A));
    ENDFOR;
    FOR EACH edge e incoming or outgoing nodes of SA in G DO
        Q.insertEdge (e);
    ENDFOR;
    RETURN Q;
END
```

Algorithm 3.6 – A method for building sub-graphs induced by a set of nodes and their neighbors

The focus+ and zoom+ methods allow refining the analysis of an activity. In order to refine the analysis of a (connected) sub-graph of the quality graph, analogous methods can be defined for focusing and zooming in a set of connected activities. Their signatures are the following:

```
    FUNCTION focus+ (G: QualityGraph, SA: ActivitySet) RETURNS QualityGraph
    FUNCTION focus– (GH: GraphHierarchy, SA: ActivitySet) RETURNS QualityGraph
    FUNCTION zoom+ (GH: GraphHierarchy, SA: ActivitySet) RETURNS QualityGraph
    FUNCTION zoom– (QH: GraphHierarchy, SA: ActivitySet) RETURNS QualityGraph
```

Pseudocodes are analogous.

When expectations are overdrawn, the zoom+ method allows decomposing an activity in order to study it in detail and eventually finding accurate improvement actions (which will be discussed in Sub-section 4.4), for example, substituting a sub-activity for a more performing component. Furthermore, if ending times are decreased in the detailed graph, the processing cost of the activity may be decreased in the summarized graph too. Then, the zoom+ method represents an interesting opportunity for enforcing data freshness. In addition, the focus+ method allows focusing on one activity and ignoring the others, which is more manageably than enormous graphs with details about all activities.

Next sub-section presents a method for finding the activities that constitute bottlenecks for data freshness calculation. The zoom+ and focus+ methods may be applied to these activities, targeting the analysis of data freshness.

4.3. Determination of critical paths

For each target node, it may exist a path (along the data flow), starting at a source node, for which the freshness of delivered data can be obtained adding all the inter-process delays and processing costs of the nodes in the path, to the source data actual freshness of the source node. This path is called the *critical path* of the target node and represents the bottleneck for data freshness. The following example presents the intuition of critical paths.

Example 3.18. Consider the quality graph of Figure 3.18. The freshness of data produced by activity A_6 (delivered to target T_2) can be calculated adding the source data actual freshness of source S_1 (0), plus inter-process delays (0,0,10,20) and processing costs (30,60,30,5) in the path from S_1 passing by activities A_1,A_3,A_5 and A_6, i.e. 0 + (0,0,10,20) + (30,60,30,5) = 155. So, this path is a critical path for T_2. □

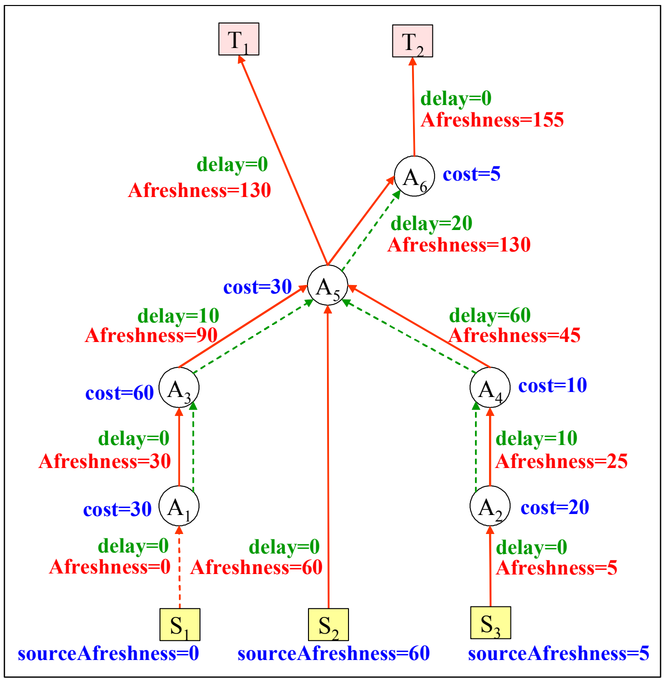

Figure 3.18 – An example of critical paths

As previously argued, the freshness of the data delivered to the user may be improved optimizing the design of the activities in order to reduce their processing costs or synchronizing the activities in order to reduce the inter-process delays among them. Sometimes the changes can be concentrated in the critical path, i.e. reducing source data actual freshness, processing costs and inter-process delays of the nodes of the critical path. This motivates the analysis and determination of critical paths.

The existence of critical paths depends on the definition of the combination function, i.e. for certain functions the critical path may not exist. In this sub-section, we consider that the combination function returns the maximum of input values and we prove that for such function the critical path always exists. Similar analysis can be done for finding bottlenecks for other combination functions.

Before defining critical paths we define the concepts of data path and path freshness.

Definition 3.9 (data path). A *data path* in a quality graph is a sequence of nodes of the graph, where each node is connected to its successor in the sequence by a data edge. We denote a data path, giving the sequence of nodes that compose it, comma separated and between square brackets, for example $[A_0,A_1,A_3,A_4]$. We also use suspension points for omitting intermediate nodes, for example $[A_0,\ldots A_4]$. □

Definition 3.10 (path freshness). Given a data path in a quality graph $[A_0,A_1,\ldots A_p]$, starting at a source node A_0, the *path freshness* is the freshness actual value propagated along the path (ignoring other nodes of the graph), i.e. it is the sum of source data actual freshness of the source node, the processing costs of the nodes in the path and the inter-process delays among the nodes[*]:

$$\text{PathFreshness}([A_0,\ldots A_p]) = \text{SourceActualFreshness}(A_0) + \Sigma_{x=0..p}\ \text{ProcesssingCost}(A_x) + \Sigma_{x=1..p}\ \text{InterProcessDelay}(A_{x-1},A_x) \quad □$$

The critical path for an activity is the data path (from a source node to the activity) that determines the freshness of the activity, i.e. the freshness actual value for the activity is equal to the path freshness. In other words, if we ignore other nodes and we calculate data freshness only using the critical path we obtain the same freshness value. We define a critical path as follows:

Definition 3.11 (critical path). Given an activity node A_p, a *critical path* for A_p is a data path $[A_0,\ldots A_p]$, from a source node A_0, for which the freshness of data produced by node A_p (delivered to each successor, e.g. C) equals the path freshness.

$$\text{Freshness}(A_p,C) = \text{PathFreshness}([A_0,\ldots A_p])$$

Given a target node T_i, a *critical path* for T_i is the critical path of its predecessor activity. □

The following lemma states the existence of at least a critical path for each activity node.

Lemma 3.2. Given an activity node A_p, with successor A_{p+1}, there exists a data path $[A_0,A_1,\ldots A_p]$ from a source node A_0 to A_p that verifies:

$$\text{Freshness}(A_p,A_{p+1}) = \text{PathFreshness}([A_0,\ldots A_p])$$

Proof:

According to the ActualFreshnessPropagation algorithm, freshness of data produced by node A_p (delivered to A_{p+1}) is obtained adding the processing cost of A_p to the combination of freshness values (plus inter-process delays) of predecessors ($P_1\ldots P_n$). The considered combination function returns the maximum input values.

$$\text{Freshness}(A_p,A_{p+1}) = \text{ProcessingCost}(A_p) + \max(\{\text{Freshness}(P_1,A_p) + \text{InterProcessDelay}(P_1,A_p), \ldots \text{Freshness}(P_n,A_p) + \text{InterProcessDelay}(P_n,A_p)\})$$

Let A_{p-1} be the predecessor that achieves the maximum in the previous formula. Then:

$$\text{Freshness}(A_p,A_{p+1}) = \text{ProcessingCost}(A_p) + \text{InterProcessDelay}(A_{p-1},A_p) + \text{Freshness}(A_{p-1},A_p)$$

Applying the same reasoning to A_{p-1}, to its predecessor achieving the maximum (A_{p-2}) and so on, we obtain:

$$\begin{aligned}\text{Freshness}(A_p,A_{p+1}) = {} & \text{ProcessingCost}(A_p) + \text{InterProcessDelay}(A_{p-1},A_p) \\ & + \text{ProcessingCost}(A_{p-1}) + \text{InterProcessDelay}(A_{p-2},A_{p-1}) + \ldots \\ & + \text{ProcessingCost}(A_1) + \text{InterProcessDelay}(A_0,A_1) \\ & + \text{SourceActualFreshness}(A_0)\end{aligned}$$

By definition of path freshness (Definition 3.10) we have:

$$\text{Freshness}(A_p,A_{p+1}) = \text{PathFreshness}([A_0,A_1,\ldots A_p]) \quad ■$$

[*] Remember that, as defined in Sub-section 3.1, processing costs are associated to all nodes (with zero value for source and target nodes) and inter-process delays are associated to all data edges (with zero value for edges outgoing source nodes or incoming target nodes).

The following lemma and its corollary provide a way of finding critical paths by computing the path freshness of all data paths from source nodes. They state that critical paths are those that have the greatest path freshness.

Lemma 3.3. Given an activity node A_p, with successor A_{p+1}, all data paths $[A_0,\ldots A_p]$ from source nodes verify:

$$Freshness(A_p,A_{p+1}) \geq PathFreshness([A_0,\ldots A_p])$$

Proof by induction in the length of the path:

→ *Basis step*: for data paths of length 2 $[A_0,A_1]$ **.** $Freshness(A_1,A_2) \geq PathFreshness([A_0,A_1])$

Proof:

$$Freshness(A_1,A_2) = ProcessingCost\ (A_1) + \max(\{Freshness(X,A_1) + InterProcessDelay\ (X,A_1)\ /\ X \text{ is a predecessor of } A_1\})$$

Taking a particular predecessor achieves a smaller or equal value. Then, for A_0:

$$\begin{aligned} Freshness(A_1,A_2) &\geq ProcessingCost\ (A_1) + Freshness(A_0,A_1) + InterProcessDelay\ (A_0,A_1) \\ &= ProcessingCost\ (A_1) + SourceActualFreshness(A_0) + InterProcessDelay\ (A_0,A_1) \\ &= PathFreshness([A_0,A_1]) \quad \blacksquare \end{aligned}$$

→ *Inductive step:* Assume that for all data paths of length $h\geq2$ $[A_0,\ldots A_{h-1}]$ **.** $Freshness(A_{h-1},A_h) \geq PathFreshness([A_0,\ldots A_{h-1}])$ in order to prove that for data paths of length h+1 $[A_0,\ldots A_{h-1},A_h]$ **.** $Freshness(A_h,A_{h+1}) \geq PathFreshness([A_0,\ldots A_{h-1},A_h])$

Proof:

$$Freshness(A_h,A_{h+1}) = ProcessingCost(A_h) + \max(\{Freshness(X,A_h) + InterProcessDelay(X,A_h)\ /\ X \text{ is a predecessor of } A_h\})$$

Taking a particular predecessor achieves a smaller or equal value. Then, for A_{h-1}:

$$Freshness(A_h,A_{h+1}) \geq ProcessingCost\ (A_h) + Freshness(A_{h-1},A_h) + InterProcessDelay\ (A_{h-1},A_h)$$

Using inductive hypothesis:

$$\begin{aligned} Freshness(A_h,A_{h+1}) &\geq ProcessingCost\ (A_h) + PathFreshness([A_0,\ldots A_{h-1}]) + InterProcessDelay\ (A_{h-1},A_h) \\ &= PathFreshness([A_0,\ldots A_{h-1},A_h]) \quad \blacksquare \end{aligned}$$

Corollary:

Given an activity node A_p, with successor A_{p+1}, the freshness of data produced by A_p is equal to the maximum path freshness of the paths from a source node, i.e.:

$$Freshness(A_p,A_{p+1}) = \max\{PathFreshness([A_0,\ldots A_p])\ /\ [A_0,\ldots A_p] \text{ is a data path from a source node}\}$$

Proof:

By Lemma 3.3 the freshness of data produced by A_p is greater or equal to the path freshness of all data paths from source nodes and by Lemma 3.2 we know that there exists a path that verifies the equality (a critical path). So such path is the one with greatest path freshness. ■

Corollary of Lemma 3.3 suggest an effective method for calculating critical paths: finding the data paths with greatest path freshness. They can be computed using the Critical Path Method (CPM) [Hiller+1991], labeling a *potential task graph* with processing costs, inter-process delays and source data actual freshness (the potential task graph is similar to the quality graph but all labels are associated to edges). The Bellman algorithm [Hiller+1991] computes critical paths with order O(m) being m the number of edges of the graph.

We can now prove Lemma 3.1 using Lemma 3.2 and Lemma 3.3, as follows:

Proof of Lemma 3.1. The lemma stated that given a summarized graph S obtained applying the level-up operator over a detailed graph D, it is verified that:

$\forall$ target node V (being B the predecessor of V in D and being A the parent activity of B in S) . Freshness(A,V) $\geq$ Freshness(B,V)

Proof:

Let $[X, B_{11}, \dots B_{1m_1}, B_{21}, \dots B_{2m_2}, \dots B_{n1}, \dots B_{nm_n}]$ be the critical path built by Lemma 3.2, with $B_{nm_n}=B$, and let A_i be the parent activity of $B_{i1}, \dots B_{im_i}$, $1 \leq i \leq n$, with $A_n=A$. Note that the path $[X, A_1, \dots A_n]$ must exist in S because, by construction (BuildLevelGraph method), an edge $(B_{im_i}, B_{(i+1)1})$, $1 \leq i < n$, cause the edge (A_i, A_{i+1}) to belong to S.

On the one hand, by Lemma 3.2 and definition of path freshness we have:

$$\begin{aligned} \text{Freshness}(B,V) &= \text{PathFreshness}([X, B_{11}, \dots B_{1m_1}, B_{21}, \dots B_{2m_2}, \dots B_{n1}, \dots B_{nm_n}]) \\ &= \text{SourceActualFreshness}(X) + \text{ProcessingCost}(B_{11}) + \text{InterProcessDelay}(B_{11}, B_{12}) \\ &\quad + \dots + \text{ProcessingCost}(B_{nm_n}) \end{aligned} \quad (1)$$

On the other hand, by Lemma 3.3 and definition of path freshness we have:

$$\begin{aligned} \text{Freshness}(A,V) &\geq \text{PathFreshness}([X, A_1, \dots A_n]) \\ &= \text{SourceActualFreshness}(X) + \text{ProcessingCost}(A_1) + \text{InterProcessDelay}(A_1, A_2) \\ &\quad + \dots + \text{ProcessingCost}(A_n) \end{aligned} \quad (2)$$

We define α_i=StartingTime(B_{i1}) and β_i=EndingTime(B_{im_i}), $1 \leq i \leq n$. Note that:

$$\text{ProcessingCost}(B_{i1}) + \text{InterProcessDelay}(B_{i1}, B_{i2}) + \dots + \text{ProcessingCost}(B_{im_i}) = \beta_i - \alpha_i \quad (3)$$

According to the calculation of processing costs and inter-process delays for summarized graphs (Figure 3.12 and Figure 3.14 respectively) we have:

$$\text{ProcessingCost}(A_i) = \text{EndingTime}(\text{FinalSubActivity}(A_i))$$

$$\begin{aligned} \text{InterProcessDelay}(A_i, A_{i+1}) = \max (\{ &\text{InterProcessDelay}(B_{ij}, B_{(i+1)k}) - \text{EndingTime}(\text{FinalSubActivity}(A_i)) + \\ &\text{EndingTime}(B_{ij}) - \text{StartingTime}(B_{(i+1)k}) \,/\, (B_{ij}, B_{(i+1)k}) \text{ is a data edge of D}\}) \end{aligned}$$

In particular, the edge $(B_{im_i}, B_{(i+1)1})$ achieves a smaller or equal value in previous formula. By algebraic manipulation we have:

$$\begin{aligned} \text{InterProcessDelay}(B_{im_i}, B_{(i+1)1}) &\leq \text{EndingTime}(\text{FinalSubActivity}(A_i)) + \text{InterProcessDelay}(A_i, A_{i+1}) - \beta_i + \alpha_{i+1} \\ &= \text{ProcessingCost}(A_i) + \text{InterProcessDelay}(A_i, A_{i+1}) - \beta_i + \alpha_{i+1} \end{aligned} \quad (4)$$

Substituting (3) and (4) in (1) we obtain:

$$\begin{aligned} \text{Freshness}(B,V) \leq\ &\text{SourceActualFreshness}(X) + (\beta_1 - \alpha_1) \\ &+ (\text{ProcessingCost}(A_1) + \text{InterProcessDelay}(A_1, A_2) - \beta_1 + \alpha_2) + (\beta_2 - \alpha_2) + \dots \\ &+ (\text{ProcessingCost}(A_{n-1}) + \text{InterProcessDelay}(A_{n-1}, A_n) - \beta_{n-1} + \alpha_n) + (\beta_n - \alpha_n) \\ =\ &\text{SourceActualFreshness}(X) - \alpha_1 + \text{ProcessingCost}(A_1) + \text{InterProcessDelay}(A_1, A_2) + \dots \\ &+ \text{ProcessingCost}(A_{n-1}) + \text{InterProcessDelay}(A_{n-1}, A_n) + \beta_n \end{aligned}$$

As $\alpha_1 \geq 0$ and $\beta_n \leq$ ProcessingCost(A_n):

$$\begin{aligned} \text{Freshness}(B,V) \leq\ &\text{SourceActualFreshness}(X) + \text{ProcessingCost}(A_1) + \text{InterProcessDelay}(A_1, A_2) + \dots \\ &+ \text{ProcessingCost}(A_{n-1}) + \text{InterProcessDelay}(A_{n-1}, A_n) + \text{ProcessingCost}(A_n) \end{aligned}$$

Finally, using (2):

$$\text{Freshness}(B,V) \leq \text{PathFreshness}([X, A_1, \dots A_n]) \leq \text{Freshness}(A,V) \quad \blacksquare$$

As proved in previous lemma, when calculating freshness in a more detailed graph we may obtain more precise values than in a summarized graph. In addition, some improvement actions can be better applied in a more detailed graph which brings more specific information about processing costs and inter-process delays.

Next sub-section discusses some strategies for enforcing data freshness when user freshness expectations cannot be achieved.

4.4. Improvement actions

If freshness actual values are greater than expected values, freshness expectations are not achieved and some improvement actions should be followed in order to enforce freshness. Freshness actual values can be improved optimizing the design and implementation of the activities in order to reduce their processing cost, synchronizing the activities in order to reduce the inter-process delay among them, or negotiating with source providers in order to obtain fresher source data. In addition, freshness expected values can be relaxed negotiating with users. Consequently, improvement actions can be the response to one of the following objectives: (i) reduce processing costs, (ii) reduce synchronization delays, (iii) reduce source data actual freshness, or (iv) augment target data expected freshness.

The result of applying an improvement action will be changes in the topology or the properties of the quality graph (i.e. changes in nodes, edges and labels). Elementary actions are:

– addNode (N: Node) – This action adds a node to the quality graph. The node is not yet connected to other nodes (it has no incoming nor outgoing edges) and has not labels.

– addEdge (e: Edge) – This action adds an edge to the quality graph. The edge has not labels.

– addProperty (N: Node, P: Property, value: Object) – This action associates a label property=value to a node of the quality graph. If the property is already associated to the node, its value is updated. The data type of the value corresponds to the data type of the property.

– addProperty (e: Edge, P: Property, value: Object) – This action associates a label property=value to an edge of the quality graph. If the property is already associated to the node, its value is updated. The data type of the value corresponds to the data type of the property.

– removeNode (N: Node) – This action removes a node (and all incoming and outgoing edges, as well as associated properties) from a quality graph.

– removeEdge (e: Edge) – This action removes an edge (and all associated properties) from a quality graph.

– removeProperty (N: Node, P: Property) – This action deletes a property from a node of the quality graph.

– removeProperty (e: Edge, P: Property) – This action deletes a property from an edge of the quality graph.

All these elementary actions are member methods of the QualityGraph class.

Elementary actions can be combined, conforming macro actions that solve typical improvement strategies. There is a great variety of macro actions that can be defined. Their feasibility depends on the particular application scenario, i.e. an action that considerably improves freshness in a DIS may have no impact on another one. In this sub-section we illustrate some typical macro actions (without trying to be exhaustive) that can be applied in a great variety of DISs. Examples of macro actions are:

– replaceNode (G: QualityGraph, N: Node, N': Node, NP': NodePropertySet) – This action replaces a node (N) by a new one (N'), keeping the same edges. A set of properties (NP') is associated to the new nodes. The action can be used, for example, for replacing an activity by a more performing one (reducing processing cost) or replacing a source by another one (reducing source data actual freshness).

– replaceSubGraph (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NS': NodeSet, ES': EdgeSet, NP': NodePropertySet, EP': EdgePropertySet) – This action replaces a set of nodes and a set of edges conforming a sub-graph (NS and ES) by new set of nodes (NS'), connected by a set of edges (ES'). Sets of properties (NP' and EP') are associated to the new nodes and edges respectively. The action can be used, for example, for replacing a set of activities by a set of more performing components or replacing a source and its wrapper by new ones providing fresher data.

– decomposeNode (G: QualityGraph, N: Node, NS': NodeSet, ES': EdgeSet, NP': NodePropertySet, EP': EdgePropertySet) – This action replaces a node (N) by a new set of nodes conforming a sub-graph (NS) which represent refined tasks, connected by a set of edges (ES'). Sets of properties (NP' and EP') are associated to the new nodes and edges respectively. The action can be used, for example, for replacing an activity by a set of activities representing refined tasks (which can be optimized or synchronized separately).

- parallelizeNodes (G: QualityGraph, NS: NodeSet, ES': EdgeSet, EP': EdgePropertySet) – This action replaces a set of edges among a path of nodes (NS), corresponding to a sequential execution, by edges (ES') connecting the nodes according to a parallel execution. A set of properties (EP') is associated to the new edges. The action can be used, for example, for parallelizing the execution of certain activities in order to reduce global processing cost.
- changeNodesProperties (G: QualityGraph, NP': NodePropertySet) – This action changes property values of a set nodes. The argument NP' is a set of 3-uples of the form <node,property,value>. The action can be used, for example, for changing the processing cost property after optimizing the implementation activities, changing the source data actual freshness property after negotiation with source providers or changing DIS policies (e.g. refreshment frequencies) that will impact inter-process delays.
- changeEdgesProperties (e: Edge, EP': EdgePropertySet) – This action changes the property values of a set of edges. The argument EP' is a set of 3-uples of the form <edge,property,value>. The action can be used, for example, for changing the inter-process delay property after synchronizing activities.

In order to facilitate the easy implementation of macro actions, we define two additional macro actions (that can be used as templates) which manage (add and remove respectively) sets of nodes, edges and labels, invoking elementary actions. They have the following signatures:

```
PROCEDURE addSets (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NP: NodePropertySet,  EP:
                   EdgePropertySet)

PROCEDURE removeSets (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NP: NodePropertySet,
                      EP: EdgePropertySet)
```

Inputs consists of a quality graph, a set of nodes, a set of edges, a set of triplets <node,property,value> and a set of triplets <edge,property,value>. Pseudocodes are shown in Algorithm 3.7.

```
PROCEDURE addSets (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NP: NodePropertySet,
                   EP: EdgePropertySet)
    FOR EACH node N in NS DO
        G.addNode(N);
    FOR EACH node e in ES DO
        G.addEdge(e);
    FOR EACH triplet (node,prop,value) in NP DO
        G.addProperty(node,prop,value);
    FOR EACH triplet (edge,prop,value) in EP DO
        G.addProperty(edge,prop,value);
END

PROCEDURE removeSets (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NP: NodePropertySet,
                      EP: EdgePropertySet)
    FOR EACH node N in NS DO
        G.removeNode(N);
    FOR EACH edge e in ES DO
        G.removeEdge(e);
    FOR EACH triplet (node,prop,value) in NP DO
        G.removeProperty(node,prop);
    FOR EACH triplet (edge,prop,value) in EP DO
        G.removeProperty(edge,prop);
END
```

Algorithm 3.7 – Template macro actions

Algorithm 3.8 shows the pseudocodes of the replaceNode, replaceSubGraph, decomposeNode, parallelizeNodes, changeNodesProperties and changeEdgesProperties macro actions. They are quite similar: they build the appropriate sets (nodes, edges, labels) and invoke the addSets and removeSets template actions.

```
PROCEDURE replaceNode (G: QualityGraph, N: Node, N': Node, NP': NodePropertySet)
    EdgeSet ES';
    EdgePropertySet EP';
    FOR EACH edge e=(B,N)^T incoming N in G DO
        e' = (B,N')^T;
        (prop,value) = G.getProperties(e);
        ES'.insert(e');
        EP'.insert(e',prop,value);
    FOR EACH edge e=(N,C)^T outgoing N in G DO
        e' = (N',C)^T;
        (prop,value) = G.getProperties(e);
        ES'.insert(e');
        EP'.insert(e',prop,value);
    removeSets(G,{N},{},{},{});
    addSets(G,{N'},ES',NP',EP');
END
```

```
PROCEDURE replaceSubGraph (G: QualityGraph, NS: NodeSet, ES: EdgeSet, NS': NodeSet,
                          ES': EdgeSet, NP': NodePropertySet, EP': EdgePropertySet)
    removeSets(G,NS,ES,{},{});
    addSets(G,NS',ES',NP',EP');
END
```

```
PROCEDURE decomposeNode (G: QualityGraph, N: Node, NS': NodeSet, ES': EdgeSet,
                        NP': NodePropertySet, EP': EdgePropertySet)
    removeSets(G,{N},{},{},{});
    addSets(G,NS',ES',NP',EP');
END
```

```
PROCEDURE parallelizeNodes (G: QualityGraph, NS: NodeSet, ES': EdgeSet, EP':
                           EdgePropertySet)
    EdgeSet ES;
    FOR EACH edge e incoming a node of NS in G DO
        ES.insert(e);
    FOR EACH edge e outgoing a node of NS in G DO
        ES.insert(e);
    removeSets(G,{},ES,{},{});
    addSets(G,{},ES',{},EP');
END
```

```
PROCEDURE changeNodesProperties (G: QualityGraph, NP': NodePropertySet)
    addSets(G,{},{},NP',{});
END
```

```
PROCEDURE changeEdgesProperties (G: QualityGraph, EP': EdgePropertySet)
    addSets(G,{},{},{},EP');
END
```

Algorithm 3.8 – Macro improvement actions

In the remaining of the sub-section, we discuss the use of macro improvement actions according to the four objectives above mentioned.

Reducing processing costs

A typical way of reducing processing costs is replacing the current implementation of an activity by a more performing one. To this end, a new ad-hoc process can be created or an existent component can be reused (e.g. a web service provided by a third party developer). In the latter case, additional activities (adapters) may be necessaries for connecting the existing activities to the new component. The cost of adapters must be taken into account. Complex activities may be decomposed in simpler portions in order to replace only some of them.

Another improvement action consists in parallelizing the execution of some activities. Analogously, additional activities may be necessaries for controlling the execution (e.g. waiting for the end of all the activities before executing successors).

Further actions include running some activities in a more performing server and powering CPU and memory. In this case, the topology of the quality graph does not change but property values are modified to reflex the new scenario (the processing cost may be one of the updated properties). In addition, specific actions can be defined for specific scenarios.

Example 3.19. Consider activity A_6 of the quality graph of Figure 3.19a, which integrates data coming from predecessor activities and also builds aggregates and statistics needed by successor activities. The process that implements A_6 can be decomposed in three routines (Figure 3.19b): activity A_{61} performs data integration, activity A_{62} computes statistics for successor A_7 and activity A_{63} build aggregates for successor A_8. The decomposition allows replacing some routines by more performing ones, e.g. replacing A_{61} by A_{64} (Figure 3.19c) and allocating more resources to it in order to execute it more performingly. Note that the time for building statistics for A_7, is unnecessarily paid for data going to A_8, thus, activities A_{62} and A_{63} can be parallelized (Figure 3.19c).

The applied improvement actions are:

- decomposeNode (G, A_6, {A_{61},A_{62},A_{63}}, {(A_5,A_{61}),(A_4,A_{61}),(A_{61},A_{62}),(A_{62},A_{63}),(A_{63},A_7),(A_{63},A_8)},
  {<A_{61},'ProcessingCost',3>,<A_{62},'ProcessingCost',2>,<A_{63},'ProcessingCost',1>},
  {<(A_5,A_{61}),'InterProcessDelay',12>,<(A_4,A_{61}),'InterProcessDelay',0>,
  <(A_{61},A_{62}),'InterProcessDelay',0>,<(A_{62},A_{63}),'InterProcessDelay',0>,
  <(A_{63},A_7),'InterProcessDelay',7>,<(A_{63},A_8),'InterProcessDelay',7>})
- replaceNode (G, A_{61}, A_{64}, {<A_{64},'ProcessingCost',2>})
- changeNodesProperties (G, {<A_{64},'ProcessingCost',1>})
- parallelizeNodes (G, {A_{62},A_{63}}, {(A_{64},A_{62}),(A_{64},A_{63}),(A_{62},A_7),(A_{63},A_8)},
  {<(A_{64},A_{62}),'InterProcessDelay',0>,<(A_{64},A_{63}),'InterProcessDelay',0>,
  <(A_{62},A_7),'InterProcessDelay',7>,<(A_{63},A_8),'InterProcessDelay',7>}) □

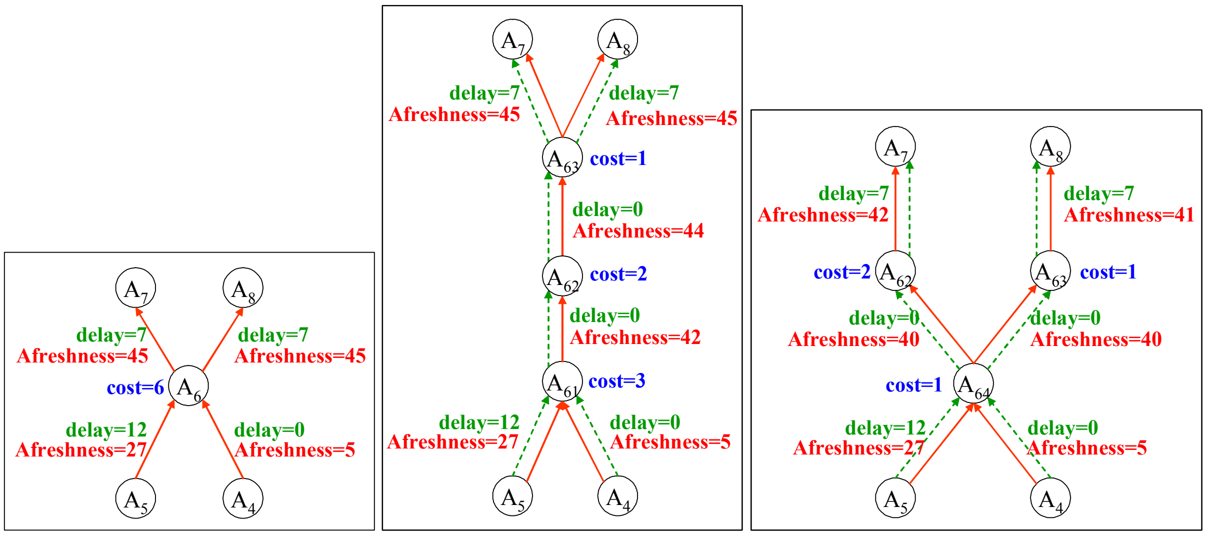

Figure 3.19 – Portions of a quality graph: (a) before performing improvement actions, (b) after decomposing an activity, (c) after replacing and parallelizing activities

Reducing inter-process delays

A typical way of reducing inter-process delays is synchronizing activities, eventually changing their synchronization policies, for executing one after the other. Activities may have synchronous or asynchronous (pull or push) policies, driven by temporal or non-temporal events (see Sub-section 2.3 of Chapter 2) and coexistence of different policies may introduce delays. Even synchronization-related properties (synchronization policies, execution frequencies, synchronization events, control events) are inherent to control flow, they indirectly causes inter-process delays among activities. Synchronizing activities do not necessarily mean that activities must execute one immediately after the other; certain delays may be necessaries due to processing constraints, for example the need of sequentially execute other activities or system routines. Then, improvement actions may reduce delays instead of eliminating them.

Unfortunately, improving the synchronization among some activities may worsen the synchronization with another ones. Consequently, when applying local synchronization techniques to portions of the quality graph, the impact to the whole graph should be studied. Next example illustrates this fact.

Example 3.20. Figure 3.20a shows three activities A_5, A_6 and A_8, the two former with periodic pull policies and the latter with aperiodic pull policy. Activity A_5 executes every 12 units of time and activity A_6 every 7 units of time, both materializing data. As they execute asynchronously, activity A_6 reads data that have been materialized for some time, 12 units of time in the worst case. Activity A_8 executes aperiodically, when users pose queries, but also reads data that has been previously materialized, 7 units of time in the worst case. Figure 3.20b illustrates their execution over time (executions are represented by rectangles, which lengths indicate processing costs) and the materialized data that is used as input (dotted lines).

If activity A_6 is synchronized with activity A_5 (its execution frequency is changed for executing every 12 units of time, just after A_5) inter-process delays are negligible. But note that in this case, the inter-process delay with activity A_8 will be 12 units of time in the worst case (instead of 7).

However, if we change the execution frequency of activity A_6 for executing every 6 units of time (as shown in Figure 3.20c), inter-process delays between activities A_5 and A_6 are either 0 or 6 units of time (6 units of time in the worst case, instead of 12) and delays between activities A_6 and A_8 are at most 6 units of time (instead of 7).

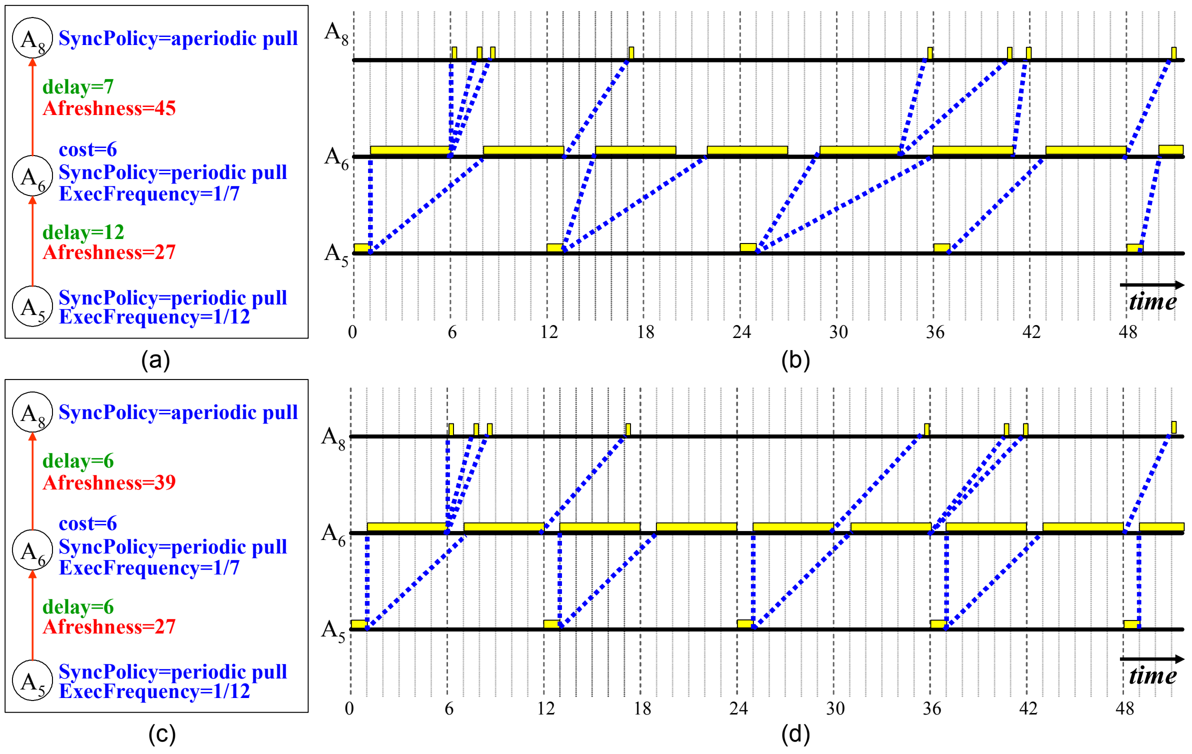

Figure 3.20 – Synchronization of activities: (a) portion of a quality graph, and (b) diagram of activity executions before the improvement action, and (c) portion of the quality graph and (d) diagram of activity executions after the improvement action

The applied improvement actions are:

- changeNodesProperties (G,{<A_6,'ExecFrequency',1/6>})
- changeEdgesProperties (G,{<(A_5,A_6),'InterProcessDelay',6>,<(A_6,A_8),'InterProcessDelay',6>}) □

Increasing the refreshment frequency of materialized data (i.e. executing the activity more frequently) may improve freshness, as shown in previous example. However, note that refreshment frequencies are constrained to allowed source accesses and activity processing times. The former arises when sources can only be queried at certain times or the number of source accesses is limited (for example because of source access price) and requires negotiation with source data providers for relaxing access constraints. The latter is intuitive: if an activity last 5 units of time for executing, it cannot execute every 2 units of time. So, synchronization techniques may be complemented by techniques for reducing processing costs, for example, decomposing an activity in portions with smaller processing costs and then increasing the refreshment frequency of new activities.

In previous example we only had two activities with periodic execution. The problem is more complex when activities have several predecessors and successors, each one with different synchronization policies. Furthermore, the combination of pull and push policies with different types of events makes very difficult the development of general synchronization techniques; specific techniques should be studied for particular application scenarios. In Section 5, we present a detailed analysis for one concrete scenario.

Reducing source actual freshness

If a source provides with data having too high freshness actual values, it can be substituted by another source providing with the same type of data but having lower freshness actual values. Note that the decision of substituting a source may also depend on other quality factors (e.g. accuracy, completeness, availability). The replacement of a source may imply the modification of other DIS components, especially wrapper activities.

Note that the new source may provide with incomplete information, for example, if it does not provide with certain attributes (which may be provided by another source). Then, a source can be replaced by a sub-graph (accessing several sources) that calculates the same data. Note that even source data actual freshness may be reduced, processing costs of new activities may be higher, so they should also be studied. Analogously, certain sub-graphs representing the access to several sources can be replaced by the access to a unique source.

Example 3.21. Figure 3.21a shows a portion of a quality graph accessing to source S_2. Figure 3.21b shows the replacement of source S_2 for sources S_4 and S_5, and the consequent replacement of wrapper activity A_2 for activities A_{21}, A_{22} and A_{23}. The applied improvement action is:

- replaceSubGraph (G, {S_2,A_2}, {}, {S_4,S_5,A_{21},A_{22},A_{23}}, {(S_4,A_{21}),(S_5,A_{22}),(A_{21},A_{23}),(A_{22},A_{23}),(A_{23},A_5)}, {<S_4,'SourceActualFreshness',12>,<S_5,'SourceActualFreshness',6>,<A_{21},'ProcessingCost',2>, <A_{22},'ProcessingCost',1>,<A_{23},'ProcessingCost',2>}, {<(A_{21},A_{22}),'InterProcessDelay',0>,<(A_{22},A_{23}),'InterProcessDelay',0>, <(A_{23},A_5),'InterProcessDelay',0>}) □

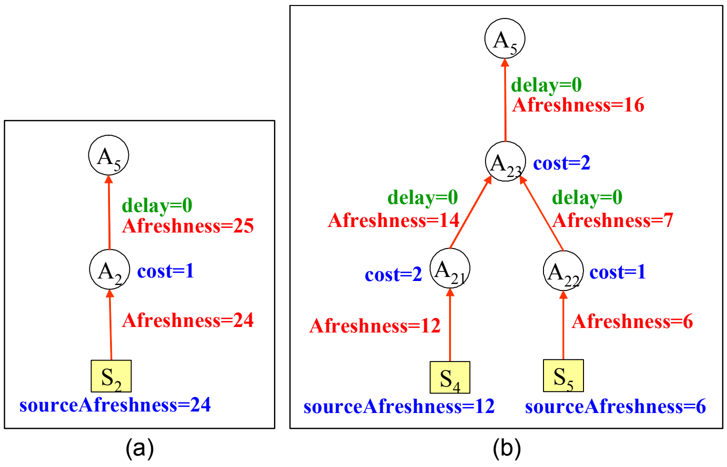

Figure 3.21 – Portions of the example quality graph (a) before and (b) after replacing a source

The comparison of different source data providers allows selecting the sources that provide the freshest data. The propagation of freshness expected values aids in such selection. Negotiating with source providers is also possible, demanding (and eventually paying) for a better service (i.e. better source data actual freshness).

Augmenting target expected freshness

Finally, when data freshness cannot be further improved, i.e. user expectations are too high for the data that can be effectively obtained from data repositories or improvement actions are too expensive (e.g. imply the acquisition of new hardware), we should negotiate with users in order to relax freshness expectations. Furthermore, many users have incremental behaviors, i.e. they ask for certain freshness values (very exigent) and if the DIS cannot provide these values, they try with relaxed values and so on.

Example 3.22. Consider that a user has demanded a freshness expected value of 10 units of time for certain data target T but the DIS cannot convey so fresh data, so the user is notified receiving no data. Then, the user decides to relax freshness expectations demanding a freshness expected value of 15 units of time and then, he tries his query again. The applied improvement action is:

- changeNodesProperties (G, {<T,'TargetExpectedFreshness',15>}) □

Summary

All previously discussed actions are general enough to be applied to different types of DIS but their use for freshness enforcement should be guided by some high-level strategy in order to be effective. On the contrary, the ad-hoc use of improvement actions may be not viable. For example, manually finding optimal refreshment frequencies in order to minimize inter-process delays, in a DIS with several tens of activities is not an easy matter. Our approach consists in studying the DIS in a high-level quality graph and zooming in the critical paths (or portions of the paths) in concentrate improvement actions in the nodes and edges of the path. Clearly, in some scenarios, the actions applied to critical paths are not enough for enforcing data freshness and other portions of the graph should be studied, for example, when the synchronization of some activities degraded the synchronization of other ones. The methods for browsing in the hierarchy of quality graphs are useful to this end.

The set of macro actions described in this sub-section is not complete, in the sense of trying to cover all possible improvement strategies. On the contrary, our approach allows the definition of specialized strategies for concrete application scenarios, as the one that will be presented in Section 5.

Next sub-section summarizes our approach with an example.

4.5. Summarizing example

In the following example, we analyze a quality graph where freshness expectations are not achieved. We firstly compute the critical path and we zoom in activities with higher processing costs and synchronization delays, analyzing possible improvement actions.

Example 3.23. Consider the DIS of Figure 3.22a, which has two data sources (Source$_1$ and Source$_2$) and two data targets (Query$_1$ and Query$_2$). Figure 3.22b shows a first zoom+ operation in order to show the activities that compose the DIS process, which perform the data extraction (Extr$_1$ and Extr$_2$), integration (Integ$_3$) and aggregation (Aggr$_4$ and Aggr$_5$). Considering the properties shown in Figure 3.22b, we achieve freshness values of 68 hours for Query$_1$ and 61 hours for Query$_2$. The former is acceptable, but the latter is too high and must be reduced in order to achieve user expectations.

We analyze the critical path for Query$_2$, i.e. [Source$_2$, Extr$_2$, Integ$_3$, Aggr$_5$]. First of all, we browse in the hierarchy of quality graphs, focusing in each activity of the critical path and zooming in them in order to have an overview of candidate portions to improve, as shown in Figure 3.22 (c, d and e). Extr$_2$ extracts and cleans information from Source$_2$; it is composed of three sub-activities: the wrapping process (Wrap$_{21}$) and two cleaning processes (Clean$_{22}$ and Clean$_{23}$). Integ$_3$ consolidates data extracted from both sources; it is composed of an initial integration activity (Integ$_{31}$), a cleaning activity that corrects some kinds of common errors (Clean$_{32}$) and a final activity that performs complex calculations (Calc$_{33}$). Due to the complexity of operations, both Integ$_{31}$ and Calc$_{33}$ materialize data once a day. Aggr$_5$ builds data needed by Query$_2$; it is composed of an aggregation activity (Aggr$_{51}$) and a posterior cleaning process for grouped data (Clean$_{52}$).

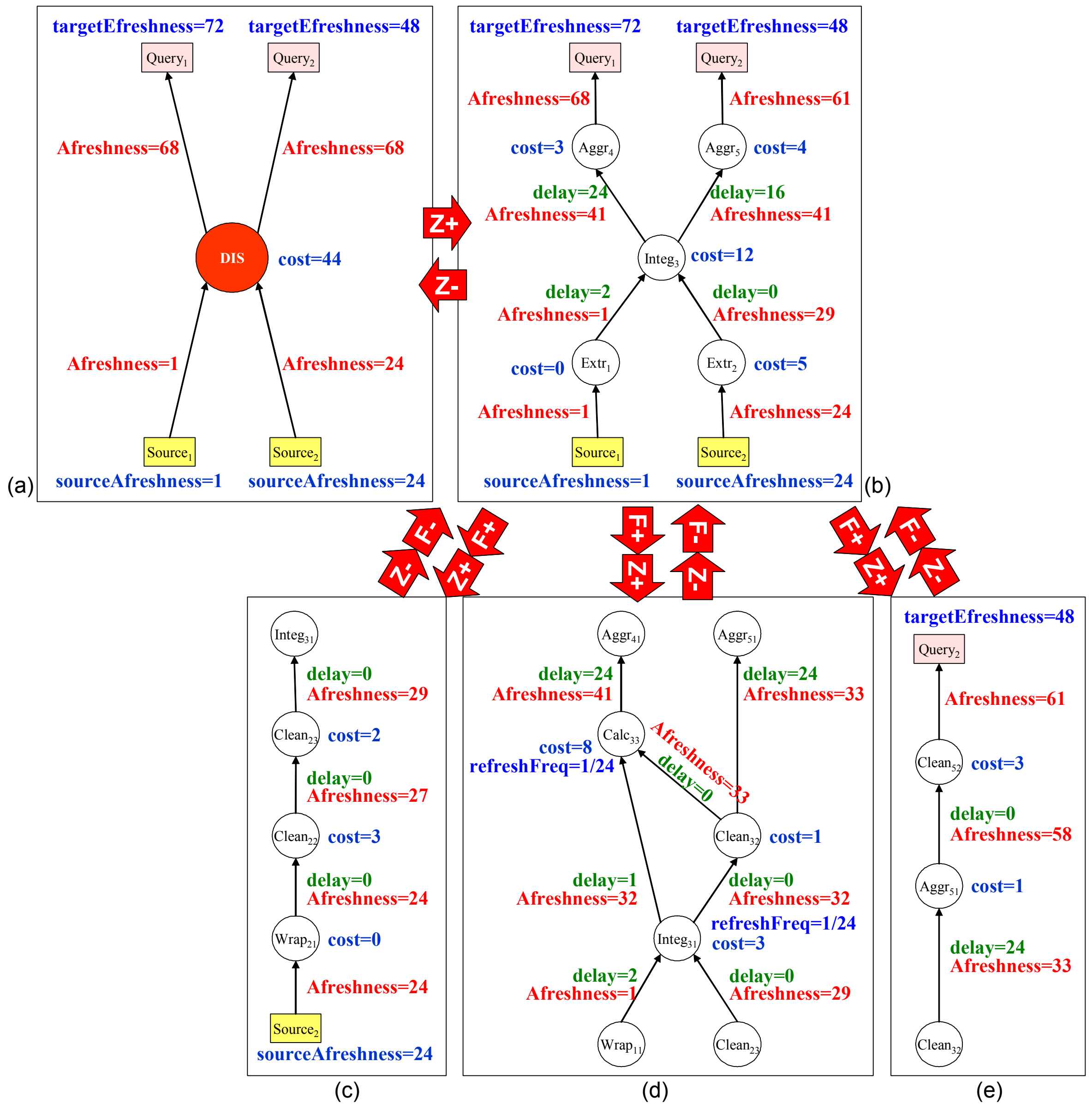

Figure 3.22 – Browsing in the critical path for Query$_2$: (a)highest-level quality graph, (b) zoom+ in the DIS node; and focus+ and zoom+ in activities: (c) Extr$_2$, (d) Integ$_3$, and (e) Aggr$_5$

Let's start analyzing Integ$_3$ and its synchronization with successors. Note that there is a big inter-process delay between Clean$_{32}$ and Aggr$_{51}$ due to the refreshment frequencies of Integ$_{31}$, as shown in Figure 3.22d. A good improvement action consists in increasing the refreshment frequency of Integ$_{31}$ (and thus executing Calc$_{33}$ more frequently). This action should be negotiated with the provider of Source$_2$ (to know if the source can be accessed more frequently) and DIS administrators (to know if the wrapper can be executed more frequently). Note that as activity Calc33 is very costly, it may continue executing once a day. Then, the first improvement actions consists in changing the refreshment frequency of Integ$_{31}$ to 12 hours and consequently changing the inter-process delay between Clean$_{32}$ and Aggr$_{51}$ to 12 hours, i.e.:

- changeNodesProperties (G$_3$,{<Integ$_{31}$,'RefreshFrequency',1/12>})
- changeEdgesProperties (G$_3$,{<(Clean$_{32}$,Aggr$_{51}$),'InterProcessDelay',12>)

Figure 3.23a shows the detailed graph for activity Integ$_3$ after these actions. Returning to the summarized graph, freshness actual value for Query$_2$ is 49 hours, which is still not acceptable.

Another improvement action consists in parallelizing the cleaning sub-activities of Extr$_2$. Both activities act over different data so the output of the wrapper can be decomposed in two disjoint sets (substituting Wrap$_{21}$

by $Wrap_{24}$) and merged at the end (adding activity Mer_{25}), both having negligible cost. As $Clean_{23}$ finishes before $Clean_{22}$, it materializes data. Actions are:

- replaceSubGraph (G, {}, {($Clean_{23}$,$Integ_{31}$)}, {Mer_{25}}, {($Clean_{23}$,Mer_{25}),(Mer_{25},$Integ_{31}$)},
  {<Mer_{25},'ProcessingCost',0>},
  {<($Clean_{23}$,Mer_{25}),'InterProcessDelay',0>,<(Mer_{25},$Integ_{31}$),'InterProcessDelay',0>})
- replaceNode (G, $Wrap_{21}$, $Wrap_{24}$, {<$Wrap_{24}$,'ProcessingCost',0>})
- parallelizeNodes (G_3, {$Clean_{22}$,$Clean_{23}$},
  {($Wrap_{24}$,$Clean_{22}$),($Wrap_{24}$,$Clean_{23}$),($Clean_{22}$,Mer_{25}),($Clean_{23}$,Mer_{25})},
  {<($Wrap_{24}$,$Clean_{22}$),'InterProcessDelay',0>,<($Wrap_{24}$,$Clean_{23}$),'InterProcessDelay',0>,
  <($Clean_{22}$,Mer_{25}),'InterProcessDelay',0>,<($Clean_{23}$,Mer_{25}),'InterProcessDelay',1>})

Figure 3.23b shows the detailed graph for activity $Extr_2$ after these actions. Returning to the summarized graph (Figure 3.23c), freshness actual value for $Query_2$ is 47 hours, which is acceptable. □

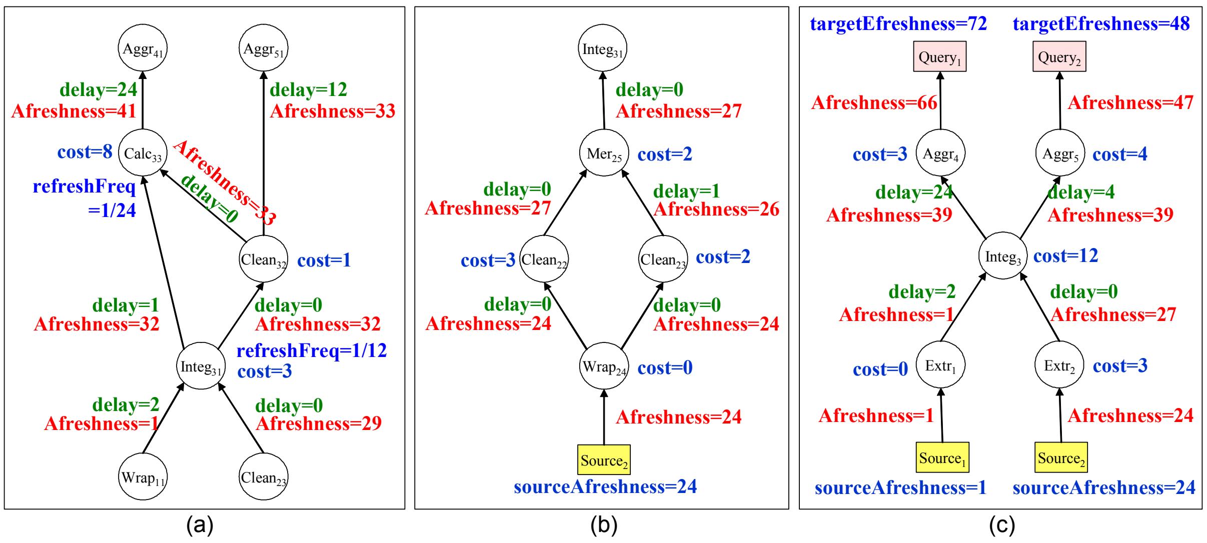

Figure 3.23 – Quality graphs after improvement actions: (a) detailed graph for $Integ_3$, (b) detailed graph for $Extr_2$, and (c) summarized graph (after all actions)

Along this section we have illustrated that data freshness enforcement may consist of several improvement actions (we have given examples of actions), which use for DIS design or reengineering depends on the particularities of concrete application scenarios. To illustrate this, next section fixes an application scenario and analyzes one improvement strategy: the synchronization of activities in order to enforce data freshness. Other strategies can be analyzed analogously for specific application scenarios; the quality evaluation framework and the general strategies discussed in this section (critical path, top-down analysis, actual and expected freshness propagation) may help in the analysis.

5. Synchronization of activities

Previous section proposed a set of elementary and macro improvement actions and argued that improvement strategies for DIS design can be enunciated for concrete DIS scenarios based on improvement actions. The purpose of this section is to illustrate the development of an improvement strategy.

Based on an improvement action (*change of execution frequencies of activities*) suggested in Sub-section 4.4 for reducing inter-process delays, we discuss the determination of appropriate execution frequencies for activities. We consider a concrete application scenario where activities have specific synchronization policies and we deal with their synchronization in order to find the optimal execution frequencies that allow achieving freshness expectations. Next sub-section motivates and states the problem and the rest of the section discusses optimal and heuristic solutions.

5.1. DIS synchronization problem

The comparison among source data actual freshness and target data expected freshness serves to determine constraints for DIS design. Their difference, if positive, indicates the longest period of time that DIS processes are allowed to spend in manipulating data, i.e. it is an upper bound for the execution delay of the DIS, which includes the processing costs of activities and the inter-process delays among them. If processing costs of activities are not too high, i.e. if all activities can execute in less time than the bound for the execution delay, the achievement (or overdraw) of freshness expectations depends on inter-process delays. The idea is to synchronize activities in a way that inter-process delays are sufficiently small to allow achieving freshness expectations.

Example 3.24. Consider the quality graph of Figure 3.24. In the path [S_2,A_2,A_3,A_4,T_1], the difference between freshness expectations and source data actual freshness is 12 units of time. The activities in the path summarize 4 units of time of processing costs, so 8 units of time can be spent in inter-process delays, i.e. greater values will cause freshness actual value to overdraw freshness expected value.

If activity A_3 executes every 3 units of time and activity A_4 executes asynchronously with each user query, the delay among A_3 and A_4 will be 3 units of time in the worst case. Analogously, we can set the execution frequency of A_2 for obtaining a delay of 5 units of time between A_2 and A_3. These delays of 3 and 5 units of time can be tolerated. Of course, delays in the path [S_1,A_1,A_3,A_4,T_1] must also be analyzed. □

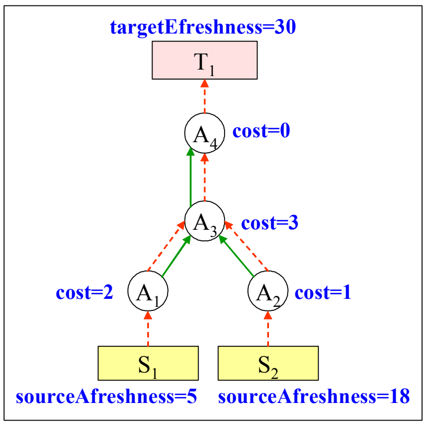

Figure 3.24 – Determination of execution policies

Before enunciating the problem, we state some hypotheses that fix the application scenario, specially, the DIS synchronization policies:

- Activities providing data to targets (called conveyance activities) execute when users ask for data (pulled by data targets) and do not materialize data; the other activities (called intermediate activities) execute asynchronously, guided by periodic pull events and may materialize data.
- Activities must finish execution before starting a new execution. Consequently, the execution period[*] of an activity should be greater or equal to its processing cost.
- Inter-process delays are determined exclusively by the execution frequencies of activities, i.e. they are not influenced by other DIS properties (e.g. hardware constraints, scheduling restrictions, communication delays). Activities can be executed in parallel. Control flow coincides with data flow.
- Sources present no access constraints, i.e. they can be accessed as frequently as needed.
- Processing costs, source data actual freshness and target data expected freshness, as well as execution periods, are all integer values, multiples of certain unit of time *T* (e.g. a day, an hour, fifteen minutes). The combineActualFreshness function returns the maximum of input values.

Synchronizing activities consist in finding an appropriate execution frequency for each intermediate activity in order to coordinate the whole DIS and obtain inter-process delays that allow achieving freshness expectations. Note that different synchronization policies (providing the execution frequency for each activity) may allow

[*] The execution period is the inverse of the execution frequency, i.e. if an activity executes every *P* units of time, execution frequency is *1/P* and execution period is *P*.

achieving freshness expectations. We need a method for finding feasible policies but we also need to decide which one to choose. Several criteria for selecting when a policy is better than another can be proposed. An example of this kind of criteria is minimizing the overall maintenance cost, which can be calculated as the processing cost of each activity, weighted par its execution frequency [Theodoratos+1999]. Finding the best synchronization of activities is an optimization problem. It can be stated as follows:

Definition 3.12 Given a quality graph G, labeled with source data actual freshness, target data expected freshness and processing cost properties, the *DIS synchronization problem* consists in finding the most appropriate execution frequencies for intermediate activities in a way that inter-process delays allow achieving freshness expectations, minimizing maintenance cost. □

In the following sub-sections we characterize the solution space and propose some algorithms to solve the problem.

5.2. Characterization of the solution space

A solution to the DIS synchronization problem is a set of execution periods (or execution frequencies), one for each intermediate activity. In this sub-section we characterize the solution space, determining the conditions that a solution must verify.

Basically, the idea is to bound the amount of time that can be consumed in synchronization, studying the difference between actual and expected freshness. We firstly obtain upper and lower bounds for the freshness of the data produced by each activity, which are called *uppermost* and *lowest* freshness values respectively. The lowest freshness value is calculated propagating source data actual freshness, from sources to targets, adding the processing costs of activities. This is the lowest freshness value that the DIS might obtain, which is achieved if all activities are synchronized for starting as soon as the predecessor activities finish, without data materialization and consequently, without inter-process delays. Note that this kind of synchronization is not always possible. The uppermost freshness value is calculated propagating target data expected freshness, from targets to sources, subtracting the processing costs of activities. This is the greatest freshness value that can be supported by the DIS in order to achieve freshness expectations for data targets. The lowest and uppermost freshness values can be calculated using the propagation algorithms described in Section 3 (see Algorithm 3.1 and Algorithm 3.2) overloading the getInterProcessDelay function in order to return zero.

If for some activity, the lowest freshness value is greater than the uppermost freshness value, freshness cannot be assured no matter the synchronization of activities and other improvement actions should be followed, for example, for reducing processing costs. On the contrary, if the uppermost freshness value is greater than the lowest freshness value, their difference is an upper bound for the execution period of the activity. We define the *greatest execution period* for the activity as such difference.

Definition 3.13 The *greatest execution period* of an activity is calculated as the difference between uppermost and lowest freshness values. □

The valid execution periods for an activity are those comprised among the processing cost (smallest execution period that can be implemented) and the greatest execution period (calculated as explained before). Obviously, if for some activity, the processing cost is greater than the greatest execution period, freshness cannot be assured no matter the synchronization of activities and consequently, other improvement actions should be followed.

We can now characterize the solution space.

Definition 3.14 Given a quality graph G, with k intermediate activities $\{A_1 \ldots A_k\}$ that produce intermediate data consumed by other activities and n-k conveyance activities $\{A_{k+1} \ldots A_n\}$ that deliver data to some target node, *the solution space of the DIS synchronization problem* consists of two conditions:

(i) $\text{ProcessingCost}(A_i) \leq \text{ExecutionPeriod}(A_i) \leq \text{GreatestExecutionPeriod}(A_i)$, $1 \leq i \leq k$

(ii) $\text{ActualFreshness}(A_i,T_i) \leq \text{ExpectedFreshness}(A_i,T_i)$, $k+1 \leq i \leq ..n$, for T_i being the successor of A_i

The former condition ranges the execution period of intermediate activities (which are the variables of the problem) between the processing cost and the greatest execution period of the activity. The latter assures that actual freshness is not greater than expected freshness for all conveyance activities. □

For calculating data freshness (in order to check the second condition), inter-process delays must be calculated. Let's start defining the notion of synchronism.

Definition 3.15 Two activities are *synchronized* with a shift time of K (*K-synchronized* for short), if for each execution of the latter activity there is an execution of the former exactly K units of time before. □

Example 3.25. Consider two activities A and B, where A executes every 4 hours. If B also executes every 4 hours, just 1 hour after A, then they are 1-synchronized. If B executes every 8 hours, immediately after an execution of A, they are 0-synchronized. But if B executes every 3 hours, it is shifted with A in a variable number of hours, between 0 and 3 hours, so they are not synchronized. □

If two activities A and B are 0-synchronized, B can start executing as soon as A finishes, which will minimize the waste of time between them. The inter-process delay among them is negligible. If two activities A and B are not 0-synchronized, A must materialize data, which will be asynchronously read by B. In this case, there is a positive inter-process delay between A and B. In order to estimate the inter-process delays, we analyze the execution periods of activities. If A and B are both periodically executed, the inter-process delay among them can be calculated, in the worst case, with the following formula:

InterProcessDelay(A,B) = ExecutionPeriod(A) – GCD(ExecutionPeriod(A),ExecutionPeriod(B))

where GCD is the greatest common divisor function. Note than when A and B are 0-synchronized, the GCD function equals the execution period of A, so the inter-process delay is zero. Also note that the inter-process delay obtained with this formula should be increased if the execution of activities is shifted, for example, for executing B one hour after. If B is not periodically executed (which is the case of conveyance activities) the inter-process delay between A and B can be calculated, in the worst case, with the following formula:

InterProcessDelay(A,B) = ExecutionPeriod(A)

Example 3.26. Figure 3.25 shows the three synchronization cases discussed in Example 3.25. In case (b), GCD(4,8)=4 so inter-process delay is 4–4=0. In case (c), GCD(4,3)=1, so inter-process delay is 4-1=3 in the worst case; Figure 3.25c illustrates the cases where inter-process delay takes the maximum value. In case (a), GCD(4,4)=4, so inter-process delay is 4–4=0; however, for external reasons activity B is configured for executing one hour after A, 1-synchronized, so delay is 1. □

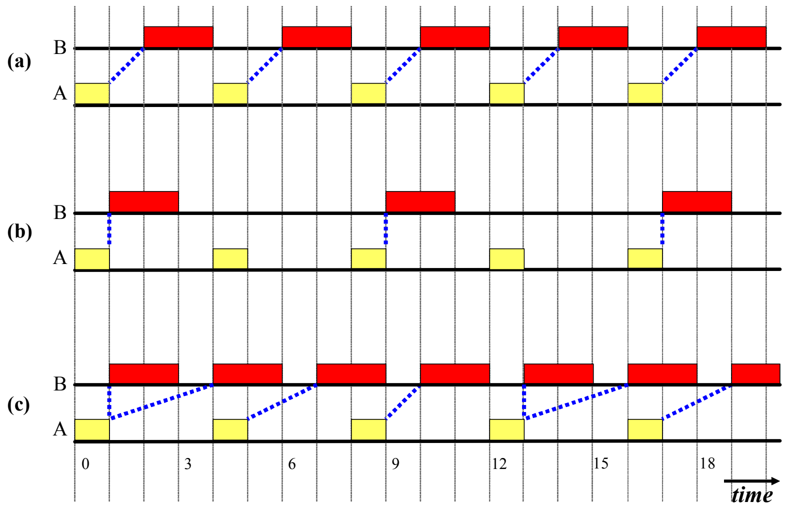

Figure 3.25 – Determination of inter-process delays

Having a calculation method for inter-process delays, the getInterProcessDelay function can be conveniently overloaded and actual freshness can be evaluated using the ActualFreshessPropagation algorithm, which allows to practically check the second condition of the solution space (see Definition 3.14). However, in order to formalize the problem, the second condition should be expressed in terms of the variables (execution periods). As stated in the corollary of Lemma 3.3, the freshness of the data produced by an activity node coincides with the path freshness of its critical path. If we calculate all paths from a source to a conveyance activity, we can decompose the second condition in a set of conditions, one for each path, stating that the path freshness must be lower or equal to expected freshness.

We formalize the problem as a nonlinear integer programming (NLIP) problem (see Definition 3.16). The variables (x_i) represent the execution periods of intermediate activities. The objective function corresponds to the overall maintenance cost (sum of processing costs weighted by execution frequencies), which must be minimized. The constraints delimit the solution space. The former ranges variables between the processing cost and the greatest execution period of the activity. The latter states that for each path going from a source node to a conveyance activity, the path freshness must be lower or equal to the expected freshness of the activity. This assures that actual freshness is not greater than expected freshness for all conveyance activities. Note that neither the objective function nor the second constraint (due to GCD) are linear.

Definition 3.16 Given a quality graph G, with m sources $\{S_1 \ldots S_m\}$, k intermediate activities $\{A_1 \ldots A_k\}$ and n-k conveyance activities $\{A_{k+1} \ldots A_n\}$, let c_i, z_i, af_i, ef_i be non-negative integers corresponding to:

– c_i: the processing cost of activity A_i, $1 \leq i \leq n$,

– z_i: the greatest execution period of activity A_i, $1 \leq i \leq k$.

– af_i: the source data actual freshness of source node S_i, $1 \leq i \leq m$,

– ef_i: the expected freshness for data produced by activity A_i, $k < i \leq n$.

Let $\{P_1 \ldots P_r\}$ be the set of paths from a source node to a conveyance activity. P_j has the form $[P_{j0}, P_{j1}, \ldots P_{jw_j}, P_{j(w_j+1)}]$, $1 \leq j \leq r$, $w_j \leq k$, where:

– P_{j0} is the index of a source relation, i.e. $1 \leq P_{j0} \leq m$,

– P_{ji} is the index of a intermediate activity, i.e. $1 \leq P_{ji} \leq k$, $1 \leq i \leq w_j$,

– $P_{j(w_j+1)}$ is the index of a conveyance activity, i.e. $k < P_{j(w_j+1)} \leq n$.

The *DIS synchronization problem* is formalized as follows:

Minimize: $\sum_{i=1..k} (c_i / x_i)$
subject to:

$$c_i \leq x_i \leq z_i, \; i{:}1..k$$

$$af_{P_{j0}} + \sum_{i=1..(w_j+1)} (c_{P_{ji}}) + \sum_{i=1..(w_j-1)} (x_{P_{ji}} - GCD(x_{P_{ji}}, x_{P_{j(i+1)}})) + x_{P_{jw_j}} \leq ef_{P_{j(w_j+1)}}, \; i{:}1..k, \; j{:}1..r \quad \square$$

Example 3.27. Consider the quality graph of Figure 3.26. Maximal execution periods are calculated propagating uppermost and lowest freshness values. There are two paths from sources to conveyance activities: $[S_1,A_1,A_3,A_4]$ and $[S_2,A_2,A_3,A_4]$. The NLIP problem for this quality graph is:

Minimize: $2/x_1 + 1/x_2 + 3/x_3$
subject to:

$2 \leq x_1 \leq 20$
$1 \leq x_2 \leq 8$
$3 \leq x_3 \leq 8$
$x_1 + x_3 - GCD(x_1,x_3) \leq 20$
$x_2 + x_3 - GCD(x_2,x_3) \leq 8$

The two latter conditions have been simplified from:

$5 + 2 + 3 + 0 + x_1 - GCD(x_1,x_3) + x_3 \leq 30$
$18 + 1 + 3 + 0 + x_2 - GCD(x_2,x_3) + x_3 \leq 30 \quad \square$

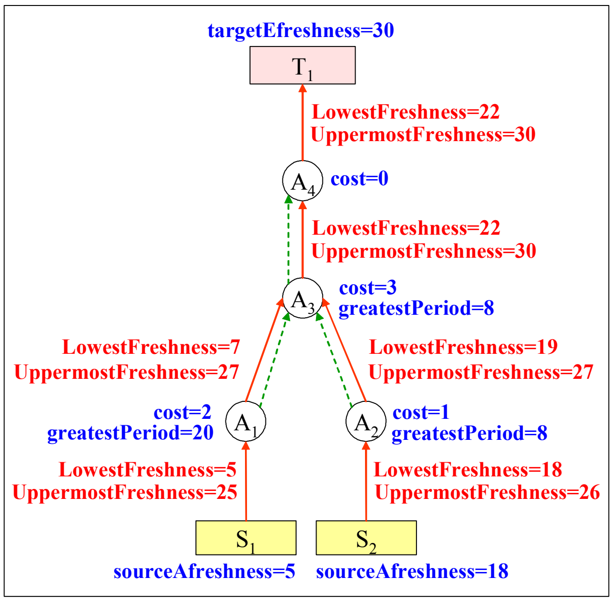

Figure 3.26 – Quality graph labeled with uppermost and lowest freshness values and greatest execution periods

In practice, using the ActualFreshnessPropagation algorithm for obtaining freshness actual values is better than generating all possible paths from sources to conveyance activities. The following sub-section discusses different algorithms for solving the problem.

5.3. Solutions to the DIS synchronization problem

In this sub-section we discuss different solutions to the DIS synchronization problem. We first present a naïve method for rapidly finding a solution (generally non optimal). Then, we discuss a branch-and-bound[*] algorithm for finding the optimal solution. The solution built with the naïve method is used for pruning the solution space. Other properties are analyzed for providing further pruning. However, as most branch-and-bound methods, the algorithm only can be executed with small size graphs due to its high complexity. We then discuss heuristics for finding non-optimal but good-enough solutions.

All algorithms return a tuple (an array of execution periods, one for each intermediate activity) that belongs to the solution space, i.e. verifies the two constraints of the problem defined in previous sub-section. Each tuple has associated a maintenance cost, i.e. its value for the objective function. Tuples are called *candidate tuples* when feasibility is not yet checked and *solutions* when they are a feasible solution to the problem.

We define the following type to manage tuples and their costs:

```
TYPE Tuple = RECORD (periods: ARRAY OF INTEGER, cost: FLOAT)
```

Next sub-sections describe the algorithms.

Naïve algorithm

The naïve algorithm builds a candidate tuple, assigning the same execution period to all the intermediate activities, in a way that all intermediate activities are 0-synchronized without inter-process delay among them. The unique inter-process delays are those with conveyance activities. The value assigned to all variables is the smallest of the greatest execution periods of activities. Observe that if for an activity, its processing cost is greater than the assigned execution period, then, the candidate tuple is not a feasible solution (it does not verify the first problem constraint). So, the naïve algorithm does not always find a solution.

A pseudocode of the naïve algorithm can be sketched as shown in Algorithm 3.9 (*BaseSolution* function). It first obtains the minimum of the greatest execution periods and then assigns it as execution period of all intermediate activities. If the candidate tuple is not a feasible solution, it returns an infinite cost tuple. The MaintenanceCost function calculates the objective function.

```
FUNCTION BaseSolution (G: QualityGraph) RETURNS Tuple
    Tuple T;
    P = min {G.getPropertyValue(A,"GreatestExecutionPeriod") / A is an intermediate activity}
    FOR EACH intermediate activity A in G DO
        IF (getProcessingCost(G,A) > P) THEN
            T.cost = infinite;
            RETURN T;
        ELSE
            T.periods[A] = P;
    ENDFOR;
    T.cost = MaintenanceCost (G,T);
    RETURN T;
END
```

Algorithm 3.9 – Naïve algorithm for building a base solution

[*] Branch-and-bound is a classical method for solving NLIP problems [Cooper+1981] [Li+2003].

If the naïve algorithm finds a solution, such solution can be used as first solution for an exhaustive branch-and-bound algorithm (for pruning some tuples), if not, we take the infinite cost tuple (that obviously will not prune any tuple). Next sub-section presents a branch-and-bound algorithm for finding an optimal solution.

Optimal algorithm

The optimal algorithm will traverse the solution space, exhaustively testing all possible values for variables (comprised between the range imposed by the first problem constraint, i.e. the processing cost and the greatest execution period). The traversal starts with a tuple composed of the greatest execution periods and proceeds, in each iteration, decreasing one of the variables in a unit of time, backtracking when we can assure that the optimal solution cannot be found decreasing more variables (pruning criteria).

```
FUNCTION OptimalSolution (G: QualityGraph) RETURNS Tuple
    Tuple Best = BaseSolution (G);
    Tuple T;
    FOR EACH intermediate activity A in G DO
        INTEGER P = G.getPropertyValue(A,"GreatestExecutionPeriod");
        G.setPropertyValue(A,"ExecutionPeriod",P);
        T.periods[A] = P;
    ENDFOR;
    T.cost = MaintenanceCost (G,T);
    RETURN BacktrackingIteration (G,T,Best);
END

FUNCTION BacktrackingIteration (G: QualityGraph, T: Tuple, Best: Tuple) RETURNS Tuple
    IF (T.cost < Best.cost) THEN
        G = ActualFreshnessPropagation (G);
        IF IsFeasibleSolution (G,T) THEN RETURN T;
        ELSE
            FOR EACH intermediate activity A in G DO
                Tuple S = T;
                IF (S.periods[A] > getProcessingCost(G,A)) THEN
                    S.periods[A] --;
                    G.setPropertyValue(A,"ExecutionPeriod",S.periods[A]);
                    S.cost = MaintenanceCost (G,S);
                    Best = BacktrackingIteration (G, S, Best);
            ENDFOR
        ENDIF
    ENDIF
    RETURN Best;
END

FUNCTION IsFeasibleSolution (G: QualityGraph, T: Tuple) RETURNS BOOLEAN
    FOR EACH intermediate activity A in G DO
        If (G.getPropertyValue(A,"ActualFreshness") > G.getPropertyValue(A,"ExpectedFreshness"))
            RETURN false;
    ENDFOR
    RETURN true;
END
```

Algorithm 3.10 – Exhaustive backtracking algorithm for finding the optimal solution

In order to define pruning criteria, let's start observing an important property of the problem: the objective function (maintenance cost) is monotonic decreasing; it takes smaller values when the variables take greater values. This means that if we find a feasible solution (x_1, x_2,… x_k), all candidates tuples (y_1, y_2,… y_k) with $y_i \leq x_i$, $1 \leq i \leq k$, will have a greater or equal maintenance cost and thus, they can be pruned. This property also suggest that it is better to evaluate tuples with bigger values first; for that reason, the traversal starts with the greatest execution periods. Analogously, when the maintenance cost of a tuple is greater than that of a known solution, the branch can be pruned. Conversely, the second problem constraint is not monotonic due to the GCD function, which oscillates. This means that evaluating the constraint for a candidate tuple (x_1, x_2,… x_k) does not allow to infer its value for neighbor tuples. It makes difficult the expression of pruning criteria for this constraint.

A pseudocode of the algorithm is shown in Algorithm 3.10 (*OptimalSolution* function). It builds a candidate tuple with the greatest execution period for each activity and invokes the BacktrackingIteration function, which traverses the solution space. The base solution built by the naïve algorithm is used as current best solution. In each iteration of the *BacktrackingIteration* function, the cost of the candidate tuple is compared with the cost of the current best solution and if it is smaller, actual freshness is evaluated, invoking the ActualFreshnessPropagation algorithm (note that graph G is labeled with the execution periods of T, for allowing the evaluation of data freshness with these periods). Freshness actual values are compared with freshness expected values for each conveyance activity (IsFeasibleSolution function). If the comparison is successful, the tuple becomes the new best solution and the current branch is pruned. If not, the function iterates descending each variable in a unit of time. The execution period of the corresponding activity is updated in the graph and the maintenance cost is recalculated, then, the BacktrackingIteration function is recursively called for the new tuple.

As most branch-and-bound methods, the algorithm has combinatory complexity, and consequently it can only be executed with small size graphs. The GCD function, which is called several times per iteration during data freshness evaluation (for each data edge between intermediate activities), has also exponential order. However, as GCD arguments are bounded, the function results can be pre-calculated and stored in a matrix, which can be accessed with order 1. Next sub-sections discuss heuristics for reducing problem size.

K-Path heuristic

One of the difficulties of the problem is the synchronization of activities having several predecessors and/or successors, because improving the synchronization with one of them may degrade the synchronization with another one. However, activities that have one predecessor and one successor can be easily synchronized with their predecessors or successors without affecting other nodes. Executing these activities at different frequencies has no sense and causes the degradation of data freshness.

A first idea for reducing problem size is 0-synchronizing activities that have one predecessor and one successor, i.e. activities belonging to a K-path. K-paths are defined as follows:

> **Definition 3.17** Given a quality graph G, a *K-path* in G is a path of intermediate activities [A_1,A_2,…A_u], where the activities in the path (excepting the initial one) have only one incoming edge in G and the activities in the path (excepting the final one) have only one outgoing edge in G. □

> **Example 3.28.** In the quality graph of Figure 3.27 there are 5 K-paths, which are highlighted with shadow boxes. □

A heuristic for improving the optimal algorithm presented in previous sub-section (*K-path heuristic*), consists in assigning the same execution periods to the activities belonging to a K-path. This reduces the number of variables of the problem and therefore reduces the problem size. For example, in the quality graph of Figure 3.27, the original problem has 10 variables (x_1…x_{10}) while the heuristic problem has only 5 variables (x_1, x_2, x_4, x_6 and x_9). As the optimal algorithm has combinatorial complexity, a reduction of the problem size considerably impacts its performance. Furthermore, the bounds for the execution period of the activities in a K-path are more restrictive. The lower bound must be the greatest of processing costs (in order to can execute all activities) and the upper bound must be the least of greatest execution periods. However, it can be proved that the greatest execution period is the same for all activities in a K-path (because of the way actual and expected freshness are propagated).

The BacktrackingIteration function (see Algorithm 3.10) should be lightly modified for iterating in the K-paths instead of on intermediate activities (FOR clause) but setting execution periods of all the activities in the K-path.

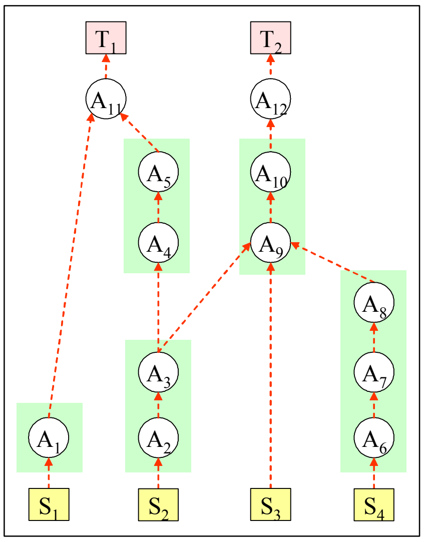

Figure 3.27 – Identification of K-paths

Random heuristic

Another idea is to build a candidate tuple with random execution periods (among solution space bounds) and then check its feasibility (using the AcutalFreshnessPropagation function). The random selection can be done for each intermediate activity or for each K-path, i.e. combined with the K-path heuristic.

The repeatedly execution of the random heuristic, a certain number of times, allows the comparison among the found solutions and the selection of the better one. Evidently, this method does not assure to find the optimal solution, however better solutions are found if we increase the number of executions. If we have a bound for the maintenance cost, we can stop when finding a good enough solution. Note that the bound may be not feasible (even for the optimal solution), so the method should also have a stop condition in the number of iterations.

If the better random solution is not good enough, it can be used as a base solution (instead of the naïve one) providing further pruning to the solution space. Furthermore, the most solutions we find, the most the solution space can be reduced. For understanding this idea, remember that the objective function is monotonic, so each time we find a feasible solution we can prune the solution space, eliminating all tuples that are smaller than the found solution.

Example 3.29. Consider the two-variable solution space (x_1, x_2) shown in Figure 3.28a, for the synchronization of two intermediate activities (or two K-paths). Bounds for variables are [a_1,b_1] and [a_2,b_2] respectively. If we found a feasible solution (s_1,s_2), we know that all smaller tuples will not improve the objective function, so we can reduce the solution space deleting the region under (s_1,s_2), i.e. the shadow region of Figure 3.28a. Considering other feasible solutions, the solution space is yet reduced, as shown in Figure 3.28b.

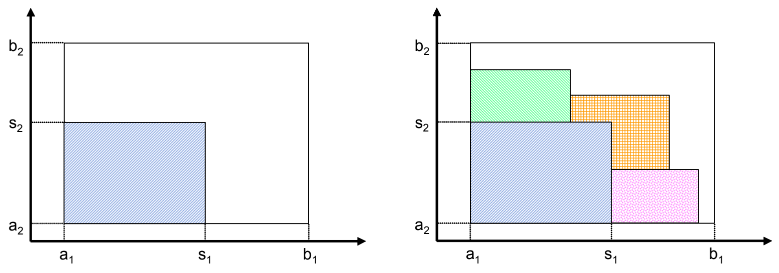

Figure 3.28 – Reduction of the solution space: (a) with one feasible solution, and (b) with several feasible solutions

The solution space can be stored in a boolean k-dimensional matrix corresponding to variables (x_1, x_2,… x_k), where a false value means that the tuple has been pruned of the solution space*. The BacktrackingIteration function can be improved checking the belonging to the matrix before iterating (instead of simply compare with processing costs; see Algorithm 3.10).

The oscillatory nature of the GCD function makes difficult the definition of local searches for optimizing a feasible solution, as in the Greedy Randomized Adaptive Search Procedure (GRASP) [Pitsoulis+2001]. However, we think that advances in operations research methods can be applied in order to find more appropriate heuristics, which is out of the scope of this thesis.

6. Conclusion

In this chapter we dealt we data freshness evaluation and enforcement topics.

We proposed a quality evaluation framework that is a first attempt to formalize the elements involved in data quality evaluation. In the framework, the DIS is modeled as a directed acyclic graph, called quality graph, which reflex the workflow structure of the DIS and contains (as labels) the DIS properties that are relevant for quality evaluation. Quality evaluation is performed by evaluation algorithms that calculate data quality traversing the quality graph.

We presented a basic algorithm for data freshness evaluation. Compared to existing evaluation proposals that only combine freshness values of source data, our algorithm takes into account two DIS properties that have impact in data freshness: the processing cost of activities and the inter-process delay among them. The algorithm can be instantiated for different application scenarios by analyzing the properties that influence the processing costs, inter-process delays and source data actual freshness in specific scenarios.

We also presented an enforcement approach for analyzing the DIS at different abstraction levels, identifying the portions that cause the non-achieved of freshness expectations. We suggested some basic improvement actions, which can be used as building-blocks for specifying macro improvement actions adapted to specific scenarios. As an application, we studied the development of an improvement strategy that follows an improvement action for a concrete application scenario. Other improvement strategies can be analyzed analogously; the quality evaluation framework and the general strategies discussed in this section (critical path, top-down analysis, actual and expected freshness propagation) may help in the analysis.

The proposal can be used at different phases of the DIS lifecycle (e.g. at design, production or maintenance phases), either for communicating data freshness to users, specifying constraints for source data or DIS development, comparing different DIS implementations accessing to alternative sources, checking the satisfaction of user freshness expectations or analyzing improvement actions for enforcing data freshness. Chapter 5 presents some applications that illustrate some of these usages.

Although we have shown that out approach can be used for DIS maintenance and evolution, we don't treat this subject in this thesis. The work of Marotta [Marotta+2006] based in our framework, treats the problem of detecting changes in source data quality and propagating changes to the DIS. They also apply improvement actions to enforce data freshness after changes. The work of Kostadinov [Mostadinov+2006] treats the expression of user preferences and then, the changes in user quality expectations. In Chapter 6, we discuss the relationship with such works as perspectives of research in these areas.

* Specific data structures for storing sparse matrices or geo-spatial data can be used.