Mälardalen University Licentiate Thesis
No.258

Systematic Design of Data Management for Real-Time Data-Intensive Applications

Simin Cai

April 2017

Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden

ISSN 1651-9256
ISBN 978-91-7485-334-6
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

Abstract

Modern real-time data-intensive systems generate large amounts of data that are processed using complex data-related computations such as data aggregation. In order to maintain the consistency of data, such computations must be both logically correct (producing correct and consistent results) and temporally correct (completing before specified deadlines). One solution to ensure logical and temporal correctness is to model these computations as transactions and manage them using a Real-Time Database Management System (RTDBMS). Ideally, depending on the particular system, the transactions are customized with the desired logical and temporal correctness properties, which are achieved by the customized RTDBMS with appropriate run-time mechanisms. However, developing such a data management solution with provided guarantees is not easy, partly due to inadequate support for systematic analysis during the design. Firstly, designers do not have means to identify the characteristics of the computations, especially data aggregation, and to reason about their implications. Design flaws might not be discovered, and thus they may be propagated to the implementation. Secondly, trade-off analysis of conflicting properties, such as conflicts between transaction isolation and temporal correctness, is mainly performed ad-hoc, which increases the risk of unpredictable behavior.

In this thesis, we propose a systematic approach to develop transaction-based data management with data aggregation support for real-time systems. Our approach includes the following contributions: (i) a taxonomy of data aggregation, (ii) a process for customizing transaction models and RTDBMS, and (iii) a pattern-based method of modeling transactions in the timed automata framework, which we show how to verify with respect to transaction isolation and temporal correctness. Our proposed taxonomy of data aggregation processes helps in identifying their common and variable characteristics, based on which their implications can be reasoned about. Our proposed process

allows designers to derive transaction models with desired properties for the data-related computations from system requirements, and decide the appropriate run-time mechanisms for the customized RTDBMS to achieve the desired properties. To perform systematic trade-off analysis between transaction isolation and temporal correctness specifically, we propose a method to create formal models of transactions with concurrency control, based on which the isolation and temporal correctness properties can be verified by model checking, using the UPPAAL tool. By applying the proposed approach to the development of an industrial demonstrator, we validate the applicability of our approach.

To my parents

Acknowledgments

I would like to thank, first and foremost, my supervisors Associate Professor Cristina Seceleanu, Dr. Dag Nyström and Associate Professor Barbara Gallina. Thank you for your patience, guidance and encouragement to my research, as well as your optimism and perseverance that have inspired me during all these years. I also wish to express my gratitude to Alf Larsson from Ericsson, Bengt Gunne from Mimer, Detlef Scholle from Alten, as well as your colleagues, for the precious support and feedback to the DAGGERS project.

I also would like to thank my grading committee members Professor Sten F Andler and Professor Mikael Sjödin, especially my faculty examiner Associate Professor Luís Almeida. Thank you for your time and effort to review my thesis. It is an honor for me to have you in my grading committee.

Big hugs to my friends and colleagues at MDH (which will be a long list :-)), not only for the discussions and cooperations, but also for making these years a pleasant journey in my life. Many thanks to the (guest) professors and lecturers at MDH, who have provided me knowledge and inspiration to continue my research.

Many hugs to my friends, Yemao, Xueming, Nico, Wei, Tengjiao, Anders, Yanzhu, and more. Thank you for your company and friendship through thick and thin. Special thanks to Fredrik, for bringing me a more beautiful world since we met.

Last but not least, it is never enough to express my gratitude to my parents for their selfless love. Thank you for letting me choose my way, and supporting me with everything.

Simin Cai
Västerås, May, 2017

List of Publications

Papers Included in the Licentiate Thesis[1]

Paper A *DAGGTAX: A Taxonomy of Data Aggregation Processes.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. Technical Report. Mälardalen Real-Time Research Center, Mälardalen University, Sweden, May 2017. A shorter version has been submitted to the 7th International Conference on Model & Data Engineering (MEDI), Springer.

Paper B *A Formal Approach for Flexible Modeling and Analysis of Transaction Timeliness and Isolation.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. In Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS), Brest, France, 19-21st October 2016. ACM.

Paper C *Towards the Verification of Temporal Data Consistency in Real-Time Data Management.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. In Proceedings of the 2nd International Workshop on modeling, analysis and control of complex Cyber-Physical Systems (CPSDATA), Vienna, Austria, April 2016. IEEE.

Paper D *Design of Cloud Monitoring Systems via DAGGTAX: a Case Study.* Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu, and Alf Larsson. In Proceedings of the 8th International Conference on Ambient Systems, Networks and Technologies (ANT), Madeira, Portugal, May 2017. Elsevier.

[1] The included papers have been reformatted to comply with the thesis layout

Contents

I

Thesis

Chapter 1

Introduction

Traditionally, real-time systems used to be closed systems with static well-defined functionalities managing small amounts of data. In recent years, however, real-time systems are designed to provide more advanced functionalities, with higher degrees of openness to other systems, and consequently become more data intensive. For instance, the amount of data managed by software is increasing in modern automotive systems. In some recent models, over 2500 signals are generated and processed in real-time [1]. In factory automation systems, hundreds of sensors are deployed to monitor the states of the working environment and the system, based on which time-constrained actions are taken to complete production work [2]. Managing large amounts of data has become an emerging challenge that the designers of real-time systems are facing.

Not only the amounts of data are growing, but also the data-related computations are becoming more complex in data-intensive real-time systems, since the latter needs to meet both temporal and logical constraints [3, 4]. On one hand, just as in traditional real-time systems, these computations need to satisfy the timeliness requirement, that is, to meet their deadlines. On the other hand, data-intensive real-time systems often bear a higher concern with respect to logical data consistency compared with the traditional ones. Concurrent access of data is common in data-intensive systems, which may introduce unwanted interference that harms logical data consistency. In addition, application semantics may entail various requirements, such as failure recovery and persistence of computation results, which also increases the complexity of the computations.

One common type of complex data-centric computation is data aggrega-

tion, which is defined as the process of producing a synthesized form of data from multiple data items using an aggregate function [5]. Compared to the raw data before aggregation, the aggregated data is usually smaller in size, often meaning reduced data storage and transmission costs, while the key information is still preserved. Therefore, data aggregation has been extensively applied in a variety of real-time applications, such as automotive [6] and avionic systems [7], in which data are aggregated from various sensors and electronic units. In many data-intensive systems, the aggregated data of one aggregation process could serve as the raw data of another, hence forming a multiple-level aggregation design. In a factory automation system, for instance, a multi-level aggregation design can be adopted for production monitoring [2]. Condition data are collected from field devices such as different sensors, where the first-level data aggregation is performed. The aggregated results are transmitted to, and further aggregated in, the production cell controllers, the factory monitoring system, and even in the cloud. In such a system, each level of aggregation may have its unique characteristics, not only in the functional aspects implemented by specific aggregate functions, but also in non-functional properties [8] including logical and temporal correctness.

A promising solution to manage the increasing amounts of data and the complex computations is to use Real-Time DataBase Management Systems (RTDBMS) for structured data management. For instance, Almeida et al. [9] adopt an RTDBMS to manage real-time sensor data and coordinate the actions of autonomous agents. In an RTDBMS, data-related computations are modeled as *transactions*, which are collections of logically related operations on data, which maintain both *logical data consistency* and *temporal correctness* [3]. Among them, logical data consistency is maintained by ensuring the so-called *ACID* properties, which refer to, respectively: *Atomicity* (a transaction either runs completely or rollbacks all changes), *Consistency* (a transaction executing by itself does not violate logical constraints), *Isolation* (uncommitted changes of one transaction shall not be seen by concurrent transactions), and *Durability* (committed changes are made permanent) [10, 11]. Temporal correctness includes two aspects: *timeliness* and *temporal data consistency*. Timeliness refers to the property that the transaction should complete its computation by the specified deadline, while temporal data consistency requires that the data used for the computation should represent a fresh and consistent view of the system and the environment [3, 12]. Ideally, both logical data consistency and temporal correctness should be guaranteed by the RTDBMS. However, conflicts could arise when temporal correctness is breached due to the unpredictability introduced by the transaction management mechanisms for ACID

such as concurrency control. In such situations, ACID assurance is often relaxed in favor of temporal correctness [13]. Depending on the specific application semantics, the relaxation of ACID could vary, in the spectrum of one or more properties, that is, A, C, I, and D [14].

In order to design a transaction-based data management solution for a data-intensive real-time system, the system designer needs to model the data-related computations as transactions, and design an RTDBMS to manage these transactions so that the desired correctness requirements are met. To our knowledge, existing DBMS design methodologies [15, 16, 17, 18, 14, 19, 20] do not provide support for the systematic analysis of trade-offs between logical data consistency and temporal correctness. Following these methodologies, designers may either design data management solutions without being aware of their impact on logical data consistency and temporal correctness, or choose an inappropriate relaxation of ACID properties and the transaction management mechanisms without sound analysis. Consequently, the risks of failing to identify conflicting properties in the design increase, and hence such conflicts could propagate to the implementation phase, leading to system-level failures during the execution.

In this thesis, we investigate how to design data aggregation and management systematically for data-intensive real-time systems. We propose an engineering process called DAGGERS (Data AGGregation for Embedded Real-time Systems) [21] to systematically develop RTDBMS customized for systems that need to trade-off between logical data consistency and temporal correctness. The DAGGERS process consists of the following steps: (i) Specifying the data-related computations, as well as the logical data consistency and temporal correctness properties, from system requirements; (ii) Selecting the appropriate transaction models to model the computations, and deciding the corresponding transaction management mechanisms that can guarantee the properties; (iii) Generating the RTDBMS using the selected transaction model and the mechanisms.

We focus on techniques to facilitate step (i) and (ii) in this thesis. Concretely, for step (i) we propose a taxonomy for data aggregation and its properties. We do this since this type of data-related computation is essential in many applications, yet a structured knowledge base for data aggregation processes is missing, which hinders applying systematic analysis in the design. For step (ii), we propose a timed-automata-based approach to model data-related computations, and analyze the trade-offs between logical data consistency and temporal correctness using model-checking techniques. We especially focus on the trade-offs between isolation and temporal correctness, and the selection

of the appropriate concurrency control algorithm. However, the approach can be extended for the analysis of other transactional properties.

To ensure applicability, we validate the proposed solutions on two industrial projects, with the engineers from industry in the loop. The validation shows that our proposed taxonomy can help to ease the effort in designing data aggregation for real-time data-intensive systems, and prevent software design flaws prior to implementation. The validation of the entire approach will be performed in our future work.

1.1 Thesis Overview

This thesis is divided into two parts. The first part is a summary of our research, including the preliminaries of this thesis (Chapter 2), a brief description of our research goals, methods and contributions (Chapter 3), a discussion on the related work (Chapter 4), and conclusions and future work (Chapter 5).

The second part is a collection of papers included in this thesis, listed as follows:

Paper A *DAGGTAX: A Taxonomy of Data Aggregation Processes.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. Technical Report. Mälardalen Real-Time Research Center, Mälardalen University, Sweden, May 2017. A shorter version has been submitted to the 7th International Conference on Model & Data Engineering (MEDI).

Abstract: Data aggregation processes are essential constituents for data management in modern computer systems, such as decision support systems and Internet of Things (IoT) systems. Due to the heterogeneity and real-time constraints in such systems, designing appropriate data aggregation processes often demands considerable efforts. A study on the characteristics of data aggregation processes will provide a comprehensive view for the designers, and facilitate potential tool support to ease the design process. In this paper, we propose a taxonomy called DAGGTAX, which is a feature diagram that models the common and variable characteristics of data aggregation processes, especially focusing on the real-time aspect. The taxonomy can serve as the foundation of a design tool that enables designers to build an aggregation process by selecting and composing desired features, and to reason about the feasibility of the design. We also provide a set of design heuristics that could help designers to decide the appropriate mechanisms for achieving the selected features. Our

industrial case study demonstrates that DAGGTAX not only strengthens the understanding, but also facilitates the model-driven design of data aggregation processes.

Paper contribution: I was the main driver of the paper. I performed the survey on data aggregation processes and proposed the taxonomy. I also conducted the industrial case study and wrote the paper. The other authors contributed with important ideas and comments.

Paper B *A Formal Approach for Flexible Modeling and Analysis of Transaction Timeliness and Isolation.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. In Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS), Brest, France, 19-21st October 2016. ACM.

Abstract: Traditional Concurrency Control (CC) mechanisms ensure absence of undesired interference in transaction-based systems and enforce isolation. However, CC may introduce unpredictable delays that could lead to breached timeliness, which is unwanted for real-time transactions. To avoid deadline misses, some CC algorithms relax isolation in favor of timeliness, whereas others limit possible interleavings by leveraging real-time constraints and preserve isolation. Selecting an appropriate CC algorithm that can guarantee timeliness at an acceptable level of isolation thus becomes an essential concern for system designers. However, trading-off isolation for timeliness is not easy with existing analysis techniques in database and real-time communities. In this paper, we propose to use model checking of a timed automata model of the transaction system, in order to check the traded-off timeliness and isolation. Our solution provides modularization for the basic transactional constituents, which enables flexible modeling and composition of various candidate CC algorithms, and thus reduces the effort of selecting the appropriate CC algorithm.

Paper contribution: I was the main driver of the paper. I proposed the modeling and verification approach presented in the paper and wrote the paper. The other authors contributed with important ideas and comments.

Paper C *Towards the Verification of Temporal Data Consistency in Real-Time Data Management.* Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. In Proceedings of the 2nd International Workshop on modeling, analysis and control of complex Cyber-Physical Systems (CPSDATA),

Vienna, Austria, April 2016. IEEE.

Abstract: Many Cyber-Physical Systems (CPSs) require both timeliness of computation and temporal consistency of their data. Therefore, when using real-time databases in a real-time CPS application, the Real-Time Database Management Systems (RTDBMS) must ensure both transaction timeliness and temporal data consistency. RTDBMS prevent unwanted interferences of concurrent transactions via concurrency control, which in turn has a significant impact on the timeliness and temporal consistency of data. Therefore it is important to verify, already at early design stages that these properties are not breached by the concurrency control. However, most often such early on guarantees of properties under concurrency control are missing. In this paper we show how to verify transaction timeliness and temporal data consistency using model checking. We model the transaction work units, the data and the concurrency control mechanism as a network of timed automata, and specify the properties in TCTL. The properties are then checked exhaustively and automatically using the UPPAAL model checker.

Paper contribution: I was the main driver of the paper. I developed the formal models and performed the verification, and wrote the paper. The other authors contributed with important ideas and comments.

Paper D *Design of Cloud Monitoring Systems via DAGGTAX: a Case Study.* Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu, and Alf Larsson. In Proceedings of the 8th International Conference on Ambient Systems, Networks and Technologies (ANT), Madeira, Portugal, May 2017. Elsevier.

Abstract: Efficient auto-scaling of cloud resources relies on the monitoring of the cloud, which involves multiple aggregation processes and large amounts of data with various and interdependent requirements. A systematic way of describing the data together with the possible aggregations is beneficial for designers to reason about the properties of these aspects as well as their implications on the design, thus improving quality and lowering development costs. In this paper, we propose to apply DAGGTAX, a feature-oriented taxonomy for organizing common and variable data and aggregation process properties, to the design of cloud monitoring systems. We demonstrate the effectiveness of DAGGTAX via a case study provided by industry, which aims to design a cloud monitoring system that serves auto-scaling for a video streaming system. We design the cloud monitoring system by selecting and composing DAGGTAX

features, and reason about the feasibility of the selected features. The case study shows that the application of DAGGTAX can help designers to identify reusable features, analyze trade-offs between selected features, and derive crucial system parameters.

Paper contribution: I was the main driver of the paper. I applied the taxonomy, designed the system, implemented a prototype, and wrote the paper. Alf Larsson provided the industrial use case, and useful comments. The other authors contributed with important ideas and comments.

Chapter 2

Preliminaries

In this chapter we present the needed preliminaries of this thesis. We first present the background knowledge about data aggregation, a common type of data-related computations considered in this thesis. We then recall the concept of transaction, including the relaxation of ACID in RTDBMS. After that, we briefly present the basics of model checking with UPPAAL, which is the formal analysis technique used in this thesis.

2.1 Data Aggregation

Data aggregation is the process of producing a synthesized form of data from multiple data items using an aggregate function [5]. It is applied extensively in information systems [5, 22, 23]. For instance, in database systems, data tuples are aggregated to compute statistical values; in resource-constrained systems, large amounts of data are aggregated to save storage or transmission resources; in systems concerning privacy and security, aggregation of details prevents information exposure.

In complex information systems, the aggregated data of one aggregation process could serve as the raw data of another process, forming a multi-level aggregation architecture. For instance, a cooperative autonomous robot aggregates the states of its companions to make a decision, which could again be transmitted to other robots as the raw data for their aggregation [24]. VigilNet exploits four levels of aggregation to perform real-time surveillance [25], as shown in Figure 2.1. The first-level aggregation takes place in the sensor layer,

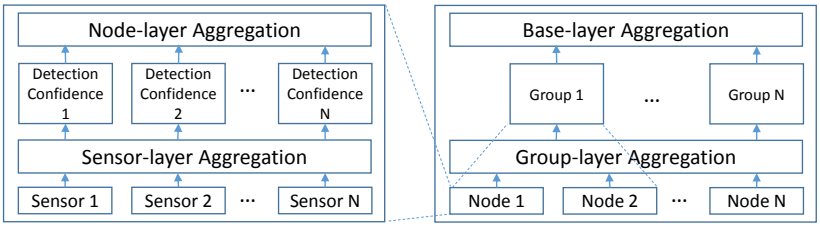


Figure 2.1: Data aggregation architecture of VigilNet [25]

in which surveillance sensor data are aggregated to form detection confidence vectors. These confidence vectors are used as raw data for the second-level aggregation in the node layer, to produce a report of the tracked targets. Using the reports from the nodes, the third-level aggregation creates aggregated reports for each group of nodes. At last, in the base layer, reports from the groups of nodes are aggregated together with historical data to make the estimation of the targets. Both the data and the aggregation processes in each level could have their unique characteristics. In VigilNet, the values of the sensor data become obsolete much faster than the historical data. Each type of sensor data also differs in when and how the data are collected and used. The aggregation processes also have different characteristics. In these four levels, the processes are triggered by different conditions, and apply various functions to perform aggregation. Although all these processes have to meet real-time requirements, the aggregation processes in the first and second levels have more strict time constraints than the other aggregation processes.

In this thesis, we have surveyed how data aggregation processes are designed in modern information systems, and studied their common and variable characteristics. Based on these studies we propose our taxonomy of data aggregation processes in Chapter 6.

2.2 The Concept of Transaction

A *transaction* is a partially-ordered set of logically-related operations on the database, which as a whole guarantee the logical data consistency [10], that is, satisfying a set of integrity constraints imposed on the database [26]. The partially-ordered set of operations is called a *work unit* [14], which may include read operations that read data from the database, write operations that modify the data in the database, and be extended with other operations that do

not interact with the database directly. Initially, a transaction maintains logical data consistency by ensuring the so-called ACID (Atomicity, Consistency, Isolation, and Durability) properties during the execution [10, 11]. Atomicity refers to the "all-or-nothing" semantics, meaning that if a transaction fails before completion, all its changes should be rolled back. Consistency requires that a transaction executed alone should not violate any logical constraints. Isolation refers to the property that no uncommitted changes within a transaction should be seen by any other, in a concurrent execution. If a transaction is committed, durability requires that its changes should be permanent and able to survive system failures. A database management system enforces these properties by applying various mechanisms to the transaction management. For instance, various logging and recovery mechanisms are among the choices for maintaining atomicity and durability, while in modern DBMS, concurrency control techniques are applied to achieve isolation [27].

Program 2.1: Transaction T that transfers 100 from A to B

```
Begin
        read A
        subtract(A, 100)
        write A
        if A<0, Abort
        read B
        add(B, 100)
        write B
Commit
```

Let us consider two bank accounts, A and B, each having an initial balance of 150. Program 2.1 shows a transaction T that transfers 100 from account A to B. In this transaction, "Begin" indicates the start of the transaction, while "Commit" and "Abort" indicate the successful termination and failure, respectively. The transaction first reads the value of A from the database to a local variable, subtracts 100 from A, and writes the new value back to the database. Similarly, it then adds 100 to B in the database. When T is executed alone and commits, the values of A and B are 50 and 250, respectively. Let us assume that full ACID is ensured by the DBMS. In case T is aborted during the execution, the values of A and B are exactly the same as the values when T is started, that is, 150 for both (atomicity assurance). An integrity constraint is implemented to make sure that the balance of A is never negative, if T is executed alone (consistency assurance). If two instances of T are executed

concurrently, their effects on the database are as if they are executed one after another, that is, one transaction changes A to 50 and B to 250, while the other one gets aborted (isolation assurance). Once T has committed, the new values are permanently stored in the disk, and can be recovered if the DBMS crashes (durability assurance).

2.2.1 Relaxed ACID Properties

Although full ACID assurance achieves a high level of logical data consistency and has thus witnessed success in many applications, it is not a "one-size-fits-all" solution for all applications [28, 14]. First, full ACID assurance might not be necessary, or desired, depending on the application semantics. For instance, in Computer Supported Cooperation Work (CSCW) systems, a transaction may need to access partial results of another concurrent transaction, which is however prohibited by full isolation [19]. Second, full ACID assurance may not be possible under the particular system constraints. As stated in the CAP theorem [29], in distributed database systems with network Partitions (P), trade-offs always occur between logical data Consistency (C) and Availability (A). Further, the PACELC theorem [30] states that even when partitions do not exist, the database system always needs to trade off between logical data consistency and latency. Therefore, in scenarios such as cloud computing and high-volume data stream management, full ACID is relaxed for availability and low latency.

As an example, let us consider a travel agency that provides reservation services. A typical trip reservation transaction could involve the following series of activities: booking a flight, booking a hotel, and paying the bill. If full ACID is ensured, when a customer books a trip, the updated information of available flight tickets is not visible to another customer until the payment has succeeded due to full isolation. This results in long waiting time for the other customer to get updated information. In addition, due to full atomicity, failing to complete the hotel reservation will lead to the rollback of the entire transaction, including the flight reservation. The customer will need to book the flight again, but the tickets may have already been sold out by that time. In order to provide better service, the traveling agency may desire another transaction model, with relaxed ACID properties.

The relaxation could be carried out in one or several of the ACID properties, depending on the requirements of the developed system. Decades of research have proposed a rich spectrum of transaction models, each consisting of a particular level of A, C, I, and D [14]. For instance, in the nested transaction model [31], if a sub-transaction fails, its parent can decide whether to ignore

the failure, or to restart the failed sub-transaction, rather than to abort the entire transaction as required by full atomicity. By applying the nested transaction model in the travel agency example, the trip reservation transaction can choose to continue when a failure occurs in the hotel booking sub-transaction. Another example of transaction models with relaxed ACID is the SAGAS model [32], in which a long-running transaction can be divided into steps. As a relaxation of full isolation, the results of these internal steps are visible to other transactions before the long-running transaction is committed. By using this model in the travel agency system, the updated tickets information is allowed to be seen by other customers before the payment is finalized. For more information about transaction models and relaxation variants of ACID, we refer the readers to literature [14].

In this thesis, we are particularly interested in the *Isolation levels*, which represent a well-accepted framework for relaxing isolation, and are implemented by most commercial DBMS. The isolation levels are introduced in the ANSI/ISO SQL-92 standard [33], and later extended and generalized by Berenson et al. [34] and Adya et al. [35]. An isolation level is defined as the property of avoiding a particular subset of phenomena (or anomalies), that is, the interferences caused by concurrent execution [35, 33]. Assuming that $T1$ and $T2$ are two transactions as defined previously, we describe the phenomena introduced by the SQL-92 standard as follows:

- **Dirty Read.** Transaction $T2$ reads a data item that was modified by transaction $T1$ before $T1$ commits. If $T1$ is rolled back, the data read by $T2$ is not valid.

- **Non-repeatable Read.** Transaction $T1$ reads a data item. Before $T1$ commits, $T2$ modifies this data item and commits. If $T1$ reads the same data again, it will receive a different value, and thus the data used by $T1$ become inconsistent.

- **Phantom.** Transaction $T1$ reads a set of data items that satisfy a search condition. Before $T1$ commits, $T2$ modifies a data item that affects the result of the search condition and commits. If $T1$ reads data with the same condition again, it will receive a different set of items, and thus the data used by $T1$ become inconsistent.

Four isolation levels are defined in the SQL-92 standards, which are READ UNCOMMITTED (the most relaxed isolation), READ COMMITTED, REPEATABLE READS, and SERIALIZABILITY (the most strict isolation). As

Table 2.1: Isolation levels in the ANSI/ISO SQL-92 standard [33]

Isolation level	Dirty read	Non-repeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READS	Not Possible	Not Possible	Possible
SERIALIZABILITY	Not Possible	Not Possible	Not Possible

listed in Table 2.1, the SERIALIZABLE level precludes all types of phenomena, whereas other levels can be defined to preclude a selected set of phenomena.

2.2.2 Pessimistic Concurrency Control

In order to achieve isolation, a DBMS applies *concurrency control* that regulates the execution of concurrent transactions and prevents unwanted interferences. Among various types of concurrency control applied in DBMS, in this thesis we focus on one of the most common type called Pessimistic Concurrency Control (PCC), which employs locking techniques to prevent interferences [27]. In PCC, a transaction needs to acquire a lock before it accesses the data, and release the lock after using the data. The DBMS decides which transactions should be granted the lock, wait, or be aborted, when lock conflicts occur [27].

A wide range of PCC algorithms have been proposed in literature [27]. They differ from each other in the types of locks, the locking durations, as well as the conflict resolution policies. As a result, these algorithms rule out various types of phenomena, and achieve different levels of isolation. For instance, as explained by Gray et al. [36] and Berenson et al. [34], one can achieve the different SQL-92 isolation levels by adjusting the lock types and locking durations (Table 2.2). In this table, a lock on a data item refers to the fact that a lock is required before reading/writing the data item, while a lock on phantom refers to the fact that a lock on the set of data items satisfying a search condition is required. A short read/write lock means that the lock is released immediately after the read/write is performed, while a long read/write lock means that the lock is released only when the transaction is committed.

Table 2.2: SQL-92 isolation levels achieved by adjusting locks [36, 34]

Isolation level	Read locks on data	Write locks on data
READ UNCOMMITTED	no locks	long write locks
READ COMMITTED	short read locks on data item	long write locks on data item
REPEATABLE READS	long read locks on data item, short read locks on phantom	long write locks on data item
SERIALIZABILITY	long read locks on both data item and phantom	long write locks on data item

2.2.3 Real-time Transactions and Temporal Correctness

In real-time database systems, a *real-time transaction* is one whose correctness depends not only on the logical data consistency, but also on the temporal correctness, which is imposed from both the transaction computation and the data [3]. As any other real-time computation, a real-time transaction should complete its work by its specified deadline. This property is referred to as *timeliness*. In addition, the data involved in the computation should be temporally consistent, including two aspects: *absolute temporal validity* and *relative temporal validity* [12]. A data instance is absolutely valid if its age from being sampled is less than a specified absolute validity interval. A data instance derived from other real-time data (base data) is relatively valid, if the base data are sampled within a specified relative validity interval.

In RTDBMS with hard real-time constraints, since the mechanisms for full ACID may introduce unacceptable latency and unpredictability, ACID may need to be relaxed in order to ensure temporal correctness [13]. For instance, it is common to relax durability, since disk I/O for storing/accessing persistent data is often considered too unpredictable for RTDBMS. Concurrency control algorithms ensuring full isolation have also been considered as a bottleneck to achieve temporal correctness, as they may cause unpredictable delays introduced by long blocking, arbitrary aborting and restarting, which could lead to deadline misses. Therefore, RTBDMS may choose a concurrency control algorithm that achieves relaxed isolation, and improves timeliness [37, 38, 39].

In this thesis, we mainly focus on the formal verification of isolation and temporal correctness of real-time transactions, to ensure that a chosen concurrency control algorithm achieves the desired relaxation level while preserving temporal correctness. The proposed framework is presented in Chapter 7 and 8.

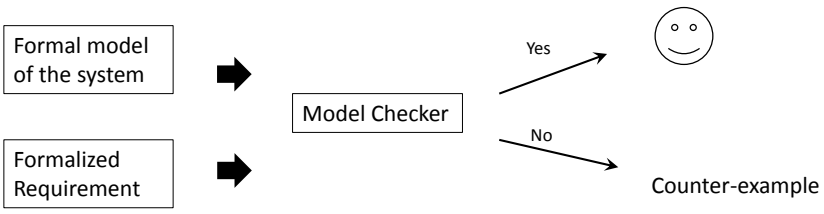


Figure 2.2: Model-checking technique

2.3 Model Checking Using UPPAAL

Model checking is a formal analysis technique that rigorously checks the correctness of a given model of the analyzed system, by exhaustively and automatically exploring all possible states of the model [40]. An overview of the model-checking technique is shown in Figure 2.2. The formal model of the system is described in a language such as UPPAAL Timed Automata [41] presented in Chapter 2.3.1. The properties to be verified are formalized in some logic, in our case as temporal logic (Timed Computation Tree Logic [42]) formulas. The model checker implementing a model-checking algorithm can then automatically verify whether the properties are satisfied by the system.

The properties verified by a model checker are of two main types: (i) *safety* properties, of the form "something (bad) will never happen", and (ii) *liveness* properties, of the form "something (good) will eventually happen". In this thesis, we focus on verifying only safety properties, as exemplified in Chapter 2.3.1. The result of the verification given by the model checker is a "yes/no" answer, indicating that the verified property is satisfied/violated, respectively. For safety properties, if a "no" answer is given, a model execution trace could be returned that acts as a counterexample to the safety property, as shown in Fig 2.2.

2.3.1 UPPAAL Timed Automata

In this thesis, we use the Timed Automata (TA) formal framework [43] to model real-time transactions, and the UPPAAL model checker [41] to verify their correctness (the relaxed ACID properties and temporal correctness). Our choice is justified by the fact that timed automata is an expressive formalism intended to describe the behavior of timed systems in a continuous-time domain. Moreover, the framework is supported by the UPPAAL tool, the state-of-the-

art model checker for real-time systems, which uses an extended version of TA for modeling, called UPPAAL TA in this thesis.

Timed automata [43] are finite-state automata extended with real-valued clock variables. As mentioned previously, UPPAAL TA [41] extends TA with discrete variables as well as other modeling features, like urgent and committed locations, synchronization channels, etc. A real-time system can be modeled as a network of TA composed via the parallel composition operator ("||"), which allows an individual automaton to carry out internal actions, while pairs of automata can perform handshake synchronization. The locations of all automata, together with the clock valuations, define the state of a TA.

We illustrate the basics of UPPAAL TA via a simple example. For more details, we refer the readers to the literature [41]. Figure 2.3 shows a simple network of UPPAAL TA composed of automata A1 and A2. In the figure, a clock variable *cl* is defined in A1 to measure the elapse of time, and progresses continuously. A discrete variable *a* is defined globally, and shared by A1 and A2. A1 consists of locations L1, L2 and L3, out of which L1 is the initial location. At each location, an automaton may non-deterministically choose to: (i) delay as long as the invariant, which is a conjunction of boolean conditions expressed as clock constraints associated to the location, is satisfied; (ii) take a transition along an edge from this location, as long as the specified guard, which is a conjunction of constraints on discrete variables or clock variables, is satisfied. In Figure 2.3a, A1 may stay at L2 until the value of *cl* reaches 3, or move to L2 when the value of *c1* is greater than 1. While moving from L2 to L3, A1 synchronizes with automaton A2 via handshake synchronization, by using a synchronization channel *ch*. An exclamation mark "!" following the channel name denotes the sender, and a question mark "?" denotes the receiver. An assignment resets the clock or sets a discrete variable when an edge is traversed. Guards and assignments can be user-defined functions. In our example, when A1 moves from L2 to L3, the value of *a* is incremented by the function *inc(a)*.

A location can be **urgent** or **committed**. When an automaton reaches an urgent location, marked as "U", it must take the next transition without any delay in time. Another automaton may take transitions at the time, as long as the time does not progress. In our example, L5 is an urgent location. A committed location, marked as "C", indicates that no delay occurs on this location and the following transitions from this location will be taken immediately. When an automaton is at a committed location, another automaton may NOT take any transitions, unless it is also at a committed location. L3 is a committed location.

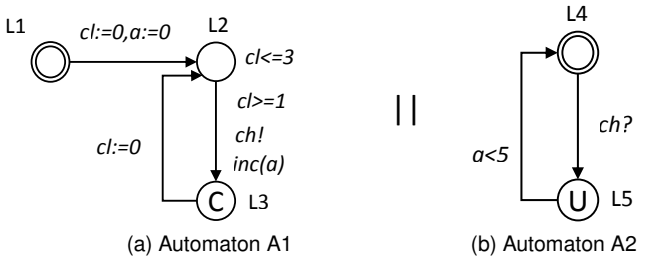


Figure 2.3: Example of a network of UPPAAL TA: A1 || A2

The properties to be verified by model checking the resulting network of timed automata are specified in a decidable subset of (Timed) Computation Tree Logic ((T)CTL) [42], and checked by the UPPAAL model checker. UPPAAL supports verification of liveness and safety properties [41]. As mentioned, in this thesis, we focus on verifying only safety properties. For instance, one can specify the safety property "A1 never reaches location L2" as "$A[\,]\ not\ A1.L2$", in which "A" is a path quantifier and reads "for all paths", whereas "$[\,]$" is the "always" path-specific temporal operator. If a safety property is not satisfied, a counterexample will be provided by UPPAAL. We refer the readers to literature [41] for more information about UPPAAL.

Chapter 3

Research Summary

In this chapter, we present a summary of our research. We formulate the research problem and research goals, describe the research method applied in our research, and present the contributions of the thesis.

3.1 Problem Description

As the amount of data and the complexity of computations are growing, RTDBMS with data aggregation support can be promising for managing the logical data consistency and temporal correctness in real-time data-intensive systems. Designing an RTDBMS is not a trivial task, as full ACID assurance for logical data consistency may need to be relaxed in order to satisfy temporal correctness. Due to a lack of support for systematic analysis, the relaxations are often decided by designers in an ad-hoc manner, which could lead to inappropriate designs that fail to satisfy the desired properties.

To overcome the drawback of ad-hoc design, systematic analysis support is needed for the design of RTDBMS with data aggregation. To achieve this, several issues need to be addressed. First, a methodology is lacking that guides the designer to systematically decide an appropriate ACID relaxation from a rich spectrum of possible choices. Second, the characteristics of the data aggregation computations, as well as their implications with respect to logical data consistency and temporal correctness, are essential to systematic design, but they are not well-understood. Last but not least, existing techniques cannot provide assurance that the selected run-time mechanisms for the RTDBMS can

guarantee the decided trade-offs.

3.2 Research Goals

Given the aforementioned problems, we present our overall research goal and the concrete subgoals in this section. Our overall research goal is formulated as follows:

Overall Research Goal. *Enable the systematic design of transaction-based data management with data aggregation support for real-time systems, so that the desired ACID properties and temporal correctness are guaranteed.*

3.2.1 Research Subgoals

In order to address the overall research goal, we define concrete subgoals that need to be tackled in order to fulfill the former. Designing appropriate real-time data management with data aggregation support requires a profound understanding of data aggregation, as well as means to organize the characteristics and reason about their implications with respect to logical data consistency and temporal correctness. Therefore, we formulate the first subgoal as follows:

Subgoal 1. *Identify and classify the common and variable characteristics of data aggregation such that they can be systematically reasoned about.*

When designing RTDBMS, the challenge is how to derive a transaction model with the appropriate transactional properties, and decide the appropriate mechanisms from a set of candidates, based on systematic analysis. In particular, in this thesis we focus on the selection of concurrency control algorithms based on analysis of trade offs between isolation and temporal correctness. Due to the large number of candidate concurrency control algorithms, we need to find a way to provide flexibility in the modeling and the analysis of the algorithms together with the transactions. Based on this, we formulate our second research subgoal as follows:

Subgoal 2. *Design a method that allows for flexible modeling of real-time transactions and concurrency control, and verification of isolation and temporal correctness.*

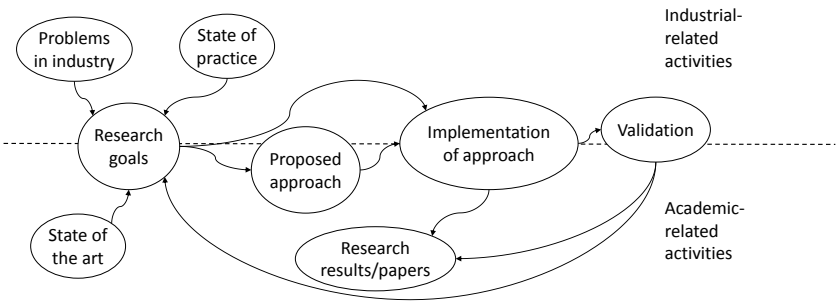


Figure 3.1: The research process of the thesis

The next issue to research into after meeting subgoal 2 is the applicability and usefulness of the proposed approach. Therefore, our third subgoal is presented as follows:

Subgoal 3. *Validate the applicability and usefulness of the proposed approach on an industrial use case.*

3.3 Research Method

In this section we introduce the methods that we use to conduct our research in order to address the research goals. We first describe the general process that we follow in our research, after which we explain the concrete methods used in this thesis.

Our research process is shown in Figure 3.1. This research is initiated by industrial problems that have not been solved by industrial solutions nor thoroughly studied by academic researchers. Based on the industrial problems, the state of practice and the state of the art, we formulate the research goals. To address these goals, we propose a systematic approach, and implement techniques to facilitate the approach, which could be applied to industrial applications. Finally, we validate the approach by applying it to the development of an industrial application. Our proposed approach, as well as the validation process and results, are documented in a series of research papers and reports.

We apply a set of research methods during the activities of the aforementioned process. For the purpose of identifying the gaps between the problems and existing approaches and formulating the research goals, we apply the "crit-

ical analysis of literature" method [44] to study the state of the art and state of practice of the researched area. We gather literature in areas including data aggregation, real-time data management, transaction modeling, etc., and critically analyze the challenges, approaches and solutions related to our research goals. During the implementation of our approach, we apply the "proof of concepts" method [44] to show the correctness and applicability of our proposed approach. When validating the research with industry, we apply the "proof by demonstration" method [44], by developing a demonstrator in an industrial setting using our proposed approach. The developed demonstrator, as well as the development process, are eventually evaluated with respect to our research goals by both the researchers and the industrial partners.

3.4 Thesis Contributions

In this section, we present the technical contributions of this thesis, which address the aforementioned research goals.

3.4.1 The DAGGERS Process

As a first step towards reaching our overall research goal defined previously, we propose, at a conceptual level, a development process called DAGGERS (Data AGGregation for Embedded Real-time Systems), as the methodology to design customized real-time data management solutions in a systematic manner. This process allows designers to identify work units of data aggregation and other data-related computations, as well as the desired properties from system requirements, based on which to derive the appropriate transaction models and the transaction management mechanisms via model checking techniques. The DAGGERS process is a methodology intended to tackle the overall research goal.

An overview of the DAGGERS process is presented in Fig 3.2, including three main steps as follows.

Step I: Specification of initial work units and requirements. The process starts with analyzing the data-related computations, including data aggregations, in the system requirements. The analysis should identify the work units as well as the logical and temporal constraints that need to be fulfilled. Based on these work units and constraints, the system designer can propose the initial transaction models, including the specification of the relationships between

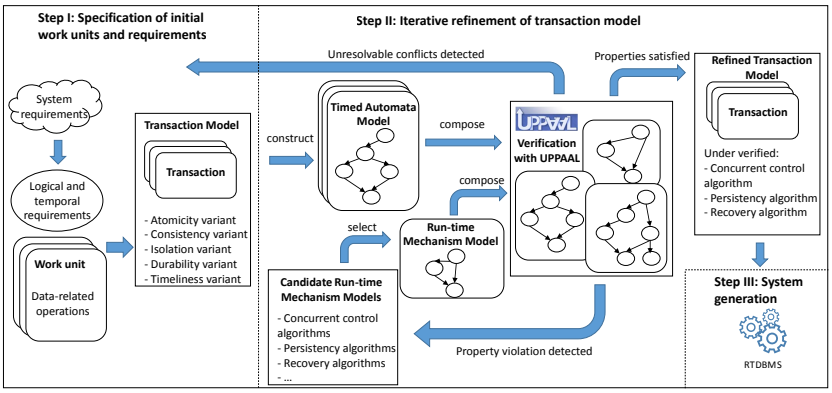


Figure 3.2: The DAGGERS process

the transactions, as well as the ACID and temporal correctness variants to be ensured.

Step II: Iterative refinement of transaction model. In this step, we apply formal modeling and model-checking techniques to derive the refined transaction models, and select the appropriate run-time mechanisms that ensure the desired ACID and temporal correctness properties. We model the work units as a set of timed automata [43], on which the transactional properties are specified formally, and can be checked by the UPPAAL model checker. We also assume that a repository of timed-automata models of commonly used run-time mechanisms has been prepared, which can be reused and composed with the timed automata of the work units. The models are checked iteratively. If model-checking shows that unsolvable conflicts occur with a particular candidate run-time mechanism, this mechanism is replaced by another candidate, and the models are verified again. This iterative process continues until all properties are satisfied by some selected mechanisms.

The refinement is the iterative "select-check" process as follows. First, we select one candidate mechanism from the repository and form a network of timed automata with the work unit models. Second, we model check the automata network against the specified properties. If any property violation is detected, which indicates that the selected mechanism fails to meet the requirement, a new candidate mechanism is selected to replace the current one, and the model checking is restarted. This iterative process continues until all

properties are satisfied by some selected mechanisms.

In case that none of the run-time mechanisms in the repository can ensure the specified properties, the designer needs to adjust the initial transaction models, that is, by adjusting the ACID and temporal correctness variants. If the conflicts cannot be resolved by any transaction model, the designer needs to adjust the requirements as they are proven infeasible under the assumed DBMS platform. As soon as the requirements are adjusted, the entire DAGGERS process is restarted.

The outcome of this step is the refined transaction models that are proved to achieve the appropriate ACID and temporal correctness variants under the selected run-time mechanisms.

Step III: System generation. With the verified transaction models, the designer can implement the transactions in SQL or other programming languages. In addition, a customized RTDBMS can be generated by composing or configuring the verified run-time mechanisms. In this thesis, we only focus on the RTDBMS design, while leaving the system generation as future work.

3.4.2 DAGGTAX: A Taxonomy of Data Aggregation Processes

In order to gain the knowledge for the systematic analysis of data aggregation, we have extensively surveyed data aggregation processes as proposed in theory and used in practice, and investigated their common and variable characteristics. Based on the survey results, we propose a taxonomy of data aggregation processes, called DAGGTAX (Data AGGregation TAXonomy).

The proposed taxonomy is presented as a feature diagram [45], in which each characteristic is modeled as a feature. It covers the common and variable characteristics of the main constituents of an aggregation process, which are the *raw data*, the *aggregate function* and the *aggregated data*. It also covers the features of the *triggering patterns* of the process, as well as the *real-time properties*.

Figure 3.3 presents the overview of DAGGTAX. In this diagram, features presented with solid dots are mandatory features. For instance, "aggregate function" is mandatory for any data aggregation process. Optional features are denoted by circles, such as "real-time (P)", which means that a data aggregation may have real-time constraints. Several features associated with a spanning curve form a group of alternative features, from which one feature

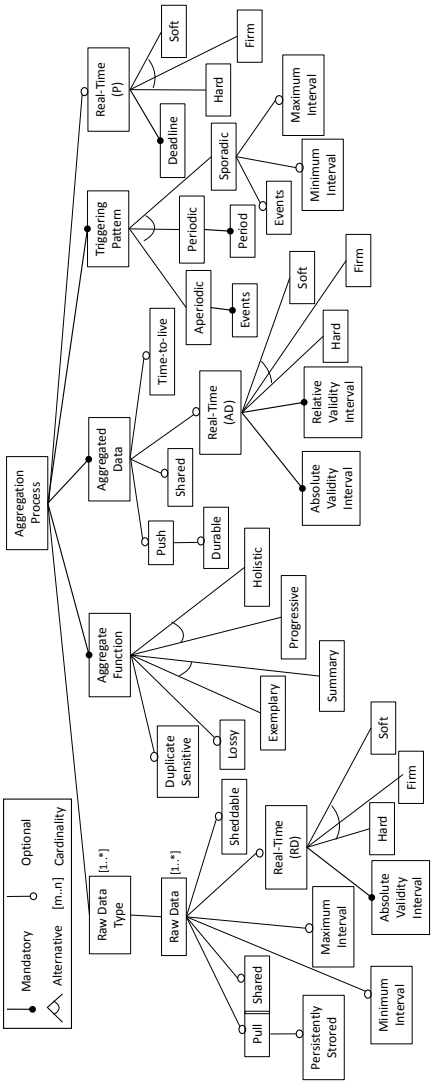


Figure 3.3: Our proposed taxonomy of data aggregation processes

must be selected by a particular aggregation process. As an example, the "triggering pattern" of a data aggregation process must be one of the following: "aperiodic", "periodic", or "sporadic". The cardinality [m..n] ($n \geq m \geq 0$) annotated with a feature denotes how many instances of the feature, including the entire sub-tree, can be contained as children of the feature's parent. We use a star symbol "*" to denote if the bounds of the cardinality are undecided. For instance, in Figure 3.3, a data aggregation process may have at least one "raw data type".

Our taxonomy provides a comprehensive view of data aggregation processes for the designers. Using the taxonomy, a data aggregation process can be constructed via the selection of desired features and their combination. Based on the taxonomy, we have introduced three design rules that eliminate some of the infeasible combinations of features during the design. For instance, a data aggregation process designed with both "soft" "real-time (P)" and "hard" "real-time (AD)" features are considered infeasible, as a soft real-time process may miss its deadline, and hence cannot guarantee hard real-time aggregated data produced by the former. We have also proposed a set of design heuristics to help the designer to decide the necessary mechanisms for achieving the selected features and other system properties. An example of such heuristics is that, if the data has a "shared" feature, the designer may need to consider concurrency control in the system design in order to maintain logical data consistency.

This contribution is proposed in Paper A. It addresses subgoal 1 by providing a structured way of representing knowledge encompassing the common and variable characteristics of data aggregation processes. The feature-oriented representation of the taxonomy allows for potential systematic analysis via tool support. DAGGTAX raises the awareness of the dependencies between the data and the aggregation process, as well as their implications to system properties, which helps designers to reason about the designs systematically, and eliminate infeasible designs prior to implementation.

3.4.3 A Timed-Automata-based Approach for Flexible Modeling and Verification of Isolation and Temporal Correctness

As another contribution to address our research goals, we propose a timed-automata-based approach for modeling real-time concurrent transaction systems, and model checking isolation and temporal correctness under various concurrency control algorithms.

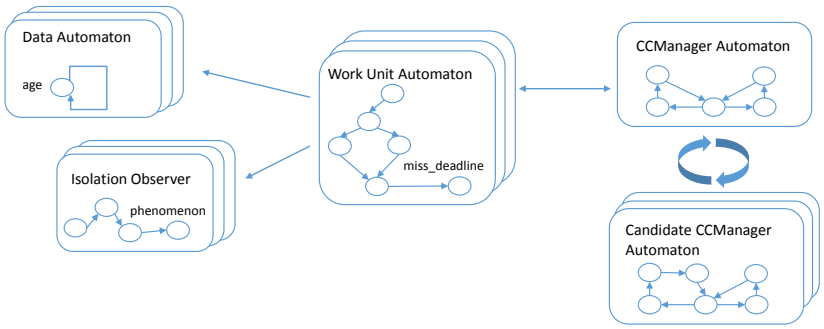


Figure 3.4: Our modeling framework

Modeling. Figure 3.4 shows the framework for modeling a real-time concurrent transaction system. We model the entire system as a timed-automata network, which consists of four types of automata: a set of work unit automata, a set of IsolationObserver automata, a set of Data automata, and a CCManager (Concurrency Control Manager) automaton.

We propose a set of automata skeletons and parameterized patterns as basic modeling blocks to reduce the modeling effort. Such skeletons model the basic structures of transactional constituents and common concurrency control algorithms, while the parameterized patterns model finer-grained recurring database operations, such as reads and writes. In the following text we explain our modeling framework in detail.

A work unit automaton models the work unit of a transaction as well as the interactions with the concurrency control manager. For each work unit automaton, we define a clock variable to trace the time spent by the transaction, and a location *miss_deadline* to represent the status of timeliness being breached. This location is reached only if the clock value exceeds a predefined deadline. Figure 3.5 presents the skeleton for a basic work unit with the locations *begin* and *end*, representing the boundary of the work unit, and a set of operation patterns modeling the data-related operations. This basic work unit skeleton can be extended with *commit_trans* and *abort* to represent successful and failed termination under the full atomicity and durability assumption, as shown in Figure 3.6. In this figure, a clock variable *tc* is defined to trace the elapsed time, and the *miss_deadline* location represents the violation of timeliness.

The work unit skeletons are enriched with instantiated parameterized patterns for data-related operations. An example of the parameterized pattern for

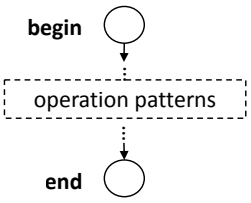


Figure 3.5: Timed automaton skeleton for a work unit

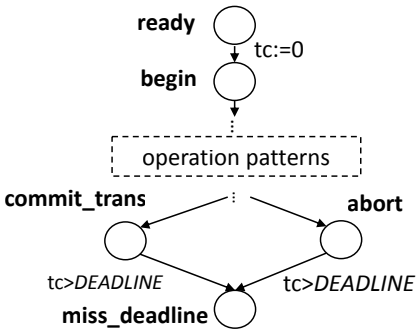


Figure 3.6: Work unit skeleton with atomicity and durability

read/write operations is presented in Figure 3.7. In this example, *tp* is a clock variable modeling the time, while *cs* is a discrete variable modeling the CPU resource. If the CPU is taken (indicated by the guard *cs==1*), the work unit automaton moves to the *wait* location. Otherwise, the automaton moves to the *operation* location. The invariant in this location (*tp<=WCRT*) constrains the automaton to stay at this location for at most *WCRT* time units, which is the worst-case response time of this operation. The guard *tp>=BCRT* constrains the automaton to stay at *operation* for at least *BCRT* time units, which is the best-case response time of this operation. As long as both constraints are satisfied, the automaton can move to *operation_done*, and sets the CPU free.

An IsolationObserver automaton is created to monitor a concurrency phenomenon that should be precluded by a particular isolation level. If a monitored phenomenon occurs, the IsolationObserver will reach the location representing the phenomenon, indicating that isolation is breached. The automaton skeleton for an IsolationObserver is described in Figure 3.8. Since a phenomenon is defined as a particular sequence of operations, we let the IsolationObserver re-

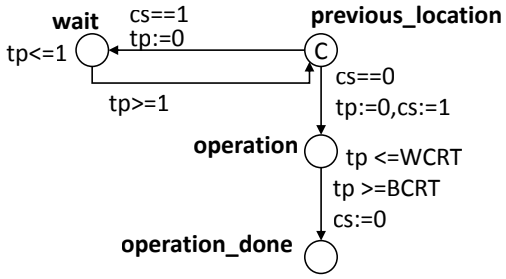


Figure 3.7: Read/write operation pattern

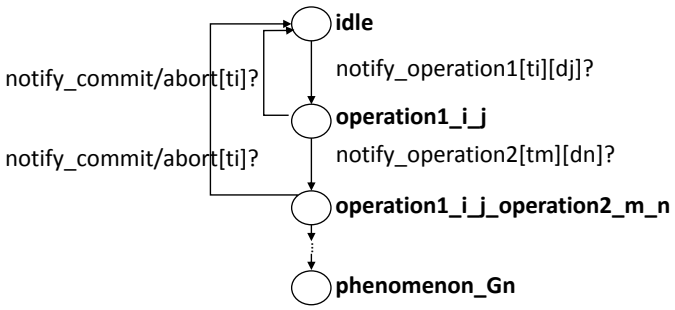


Figure 3.8: Automaton skeleton for an IsolationObserver

ceive the synchronization signals from the work units when such operations are performed. For instance, when work unit T_i successfully completes an operation on data D_j, it broadcasts the signal via channel *notify_operation1[ti][dj]*. When the IsolationObserver receives this signal, it moves from the *idle* location to the *operation1_i_j* location. If the sequence of operations that defines phenomenon Gn does occur, the IsolationObserver will eventually reach the *phenomenon_Gn* location.

A Data automaton models a data instance accessed by transactions. We define a clock variable "age" to trace the age of the data instance, which is reset when the data is updated. The skeleton for a data instance is shown in Figure 3.9.

The CCManager automaton models the concurrency control manager that applies a selected concurrency control algorithm. In Figure 3.10, we present the automaton skeleton for a PCC manager as an example. When the automa-

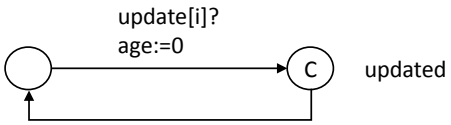


Figure 3.9: Automaton skeleton for a data instance

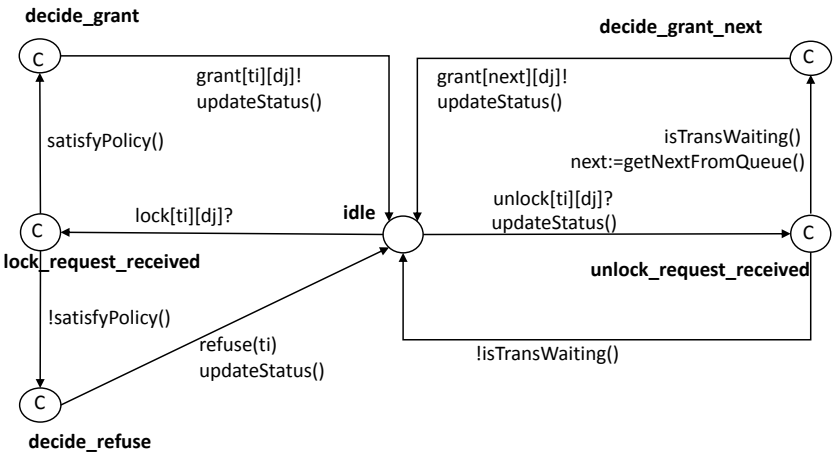


Figure 3.10: Automaton skeleton for a PCC manager

ton receives a locking request via the channel *lock[ti][dj]?*, it takes the transition from the initial location *idle* to *lock_request_received*. A user-defined function called *satisfyPolicy()*, which implements the lock request resolution of the modeled concurrency control algorithm, is defined as a guard on the edges from *lock_request_received*. Taking Two-Phase Locking (2PL [27]) as an example, *satisfyPolicy()* evaluates to false if the data required by the transaction has already been locked by another transaction. If *satisfyPolicy()* returns true, the automaton moves to *decide_grant*. It then immediately sends the signal *grant[ti][dj]!* to transaction T_i, and updates the status of the transactions and the locks, using a user-defined function *updateStatus()*. If *satisfyPolicy()* returns false, the CCManager moves to *decide_deny*, and takes actions as implemented in function *deny()*, before it moves back to *idle*. Since the CCManager has the highest priority, and the time on lock resolution is negligible, all locations in this model are committed locations. When receiving an unlocking

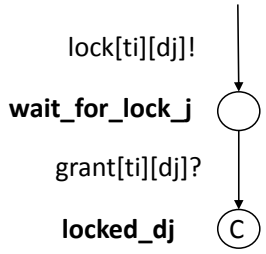


Figure 3.11: Locking pattern

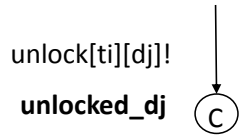


Figure 3.12: Unlocking pattern

request via *unlock[ti][dj]?*, CCManager updates the status, and moves to *unlock_request_received*. On the transitions from this location, the guards check if any transaction is waiting for locking the data, by a user-defined function *isTransWaiting()*. If this function returns true, the automaton sends a signal via *grant[next][dj]!*, to the next transaction obtained by the *getNextFromQueue()* function, and updates the status accordingly.

We also propose patterns for locking and unlocking operations, as presented in Figure 3.11 and Figure 3.12 respectively. After the transaction sends a message via *lock[ti][dj]!*, it waits at location *wait_for_lock_j*, until it receives the message *grant[ti][dj]?*. The patterns can be inserted into the work unit automata at particular positions depending on the selected PCC algorithm.

Using these skeletons and patterns, one can easily compose a network of automata to model a real-time concurrent transaction-based system. Our framework allows different PCC algorithms to be modeled and composed with the rest of the system flexibly. To model the different types of locks and durations, the designer only needs to adjust the types and locations of the instantiated locking/unlocking patterns. The resolution policies for a different PCC algorithm can also be implemented easily since it is well-encapsulated in user-defined functions.

Formal Verification. The properties that we try to verify can be checked against the timed-automata network model constructed using our framework. We specify the properties to be verified as (Timed) Computation Tree Logic ((T)CTL) formulas, as presented previously in Chapter 2.3.1, and use the UPPAAL tool to model check the formalized properties.

To verify timeliness of a transaction, we verify that the *miss_deadline* location is not reachable. For instance, the timeliness of T_i can be specified as:

$$A[\,]\, not\, Ti.miss\_deadline.$$

The absolute validity of data D_i, which requires that the age of D_i is always smaller than or equal to its specified absolute validity interval *AVI(i)*, can be specified as:

$$A[\,]\, Di.age <= AVI(i).$$

The relative validity, referring to the property that the age differences of D_i and D_j should be smaller than or equal to the specified relative validity interval *RVI(i,j)*, can be specified as the following formula:

$$A[\,]\, ((Di.updated\, imply\, Dj.age <= RVI(i,j))\\ and\, (Dj.updated\, imply\, Di.age <= RVI(i,j))).$$

Similarly, verifying a specified isolation level equals to proving that all locations representing the phenomena to be precluded are not reachable. Verifying temporal data consistency equals to checking the age of data against the property. For instance, to verify that T_i and T_j achieve SERIALIZABLE isolation, one must prove that none of the precluded phenomena Gn could occur. This is equivalent to proving that the *phenomenon_Gn* location of IsolationObserver O_n is not reachable, which can be specified as:

$$A[\,]\, not\, On.phenomenon\_Gn.$$

This contribution addresses subgoal 2, and is proposed in Paper B and C. In Paper B we propose the approach for model checking isolation and timeliness. Paper C extends the approach with observer automata for the age of data and temporal data consistency. Our framework enables flexible modeling of real-time transaction-based systems with various concurrency control algorithms, and facilitates formal verification of isolation and temporal correctness, as targeted by subgoal 2.

3.4.4 Validation on Industrial Use Cases

In order to validate the applicability and usefulness of our proposed approach, we apply them to various industrial, or industrially-relevant, use cases. This contribution addresses subgoal 3.

In Paper A we apply the proposed taxonomy to the analysis of an industrial project, the Hardware Assisted Trace (HAT) framework [46], together with its proposers from Ericsson. HAT, as shown in Figure 3.13, is a framework for

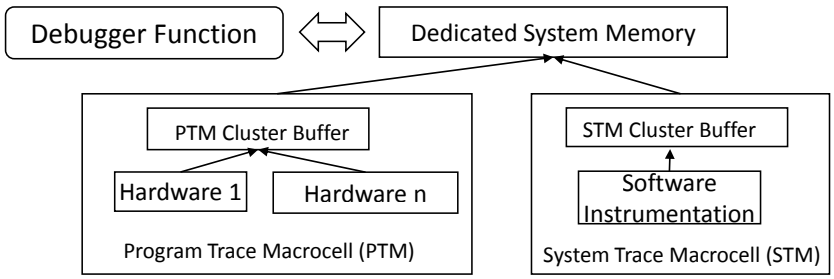


Figure 3.13: General architecture of the Hardware Assisted Trace system

debugging functional errors in an embedded system. In this framework, a debugger function runs in the same system as the debugged program, and collects both hardware and software run-time traces continuously. Two data aggregation processes are identified in the current design. At a lower level, a Program Trace Macrocell (PTM) aggregation process aggregates traces from hardware. These aggregated PTM traces, together with software instrumentation traces from the System Trace Macrocell (STM), are then aggregated by a higher level ApplicationTrace aggregation process, to create an informative trace for the debugged application.

With the DAGGTAX diagrams showing the features of the aggregation processes, the engineers could immediately identify a problem in the PTM buffer management. The problem is that the data in the buffer may be overwritten before they are aggregated. It arises due to the lack of a holistic consideration on the PTM aggregation process and the ApplicationTrace aggregation process at design time. Triggered by aperiodic external events, the PTM process could produce a large number of traces within a short period and fill up the PTM buffer. The ApplicationTrace process, on the other hand, is triggered with a minimum inter-arrival time, and consumes the PTM traces as unsheddable raw data, meaning that each PTM trace should be aggregated by the former. When the inter-arrival time of the PTM triggering events is shorter than the minimum inter-arrival time of the ApplicationTrace process, the PTM traces in the buffer may be overwritten before they could be aggregated by the ApplicationTrace process. This problem has been observed on Ericsson's implemented system, and awaits a solution. However, if the taxonomy was applied on the system design, this problem could have been identified before it was propagated to implementation. We have provided two solutions at design level to solve the identified problem.

This evaluation shows that the taxonomy enhances the understanding of the system by the designers. By applying analysis based on our taxonomy, design flaws can be identified and fixed prior to implementation. Design solutions can be constructed by composing reusable features, and reasoned about based on the taxonomy. As a result, the design space of the solutions could be reduced.

In Paper D we apply the taxonomy to design the monitoring subsystem for achieving auto-scaling functionality of a cloud system, with Ericsson engineers in the loop. We apply DAGGTAX to both the current OpenStack framework for auto-scaling, and a new design that extends the current framework. Our experience shows that our taxonomy promotes a deeper understanding of the systems behavior, and raises awareness about characteristics that need to be considered as well as issues that need to be solved during the design. It helps designers to perform better analysis than otherwise, such as to identify reusable design solutions, make data management decisions, eliminate infeasible feature combinations, and calculate time-related parameters.

This contribution addresses subgoal 3. It demonstrates the applicability of DAGGTAX, and shows that DAGGTAX can help to ease the design effort and reduce software flaws prior to implementation.

3.5 Research Goals Revisited

In this section we present the contributions of this thesis by presenting the relationship between the included papers and the research subgoals. The DAGGERS process proposed in Chapter 3.4.1 addresses our overall research goal directly, by providing a systematic process as our methodology. Each research subgoal is targeted by one or two papers, as illustrated in Table 3.1. The addressed subgoals, together with the DAGGERS process, provide a solution for designing real-time data management solutions based on systematic analysis of transactional properties, and address our overall research goal.

Table 3.1: Contribution of included papers with respect to research subgoals

	Overall goal	Subgoal 1	Subgoal 2	Subgoal 3
DAGGERS	X			
Paper A		X		X
Paper B			X	
Paper C			X	
Paper D			X	

Chapter 4

Related Work

In this chapter, we discuss the related work with respect to data aggregation, as well as the development of customized transaction management for (RT)DBMS.

4.1 Taxonomies of Data Aggregation

Many researchers have promoted the understanding of data aggregation on various aspects. Among these works, considerable efforts have been made on the study of aggregate functions. Mesiar et al. [47], Marichal [48], and Rudas et al. [5] have studied the mathematical properties of aggregate functions, such as continuity and stability, and discussed these properties of common aggregate functions in detail. A procedure for the construction of an appropriate aggregate function is also proposed by Rudas et al. [5]. In order to design a software system that computes aggregation efficiently, Gray et al. [49] have classified aggregate functions into distributive, algebraic and holistic, depending on the amount of intermediate states required for partial aggregates. Later, in order to study the influence of aggregate functions on the performance of sensor data aggregation, Madden et al. [50] have extended Gray's taxonomy, and classified aggregate functions according to their state requirements, tolerance of loss, duplicate sensitivity, and monotonicity. Fasolo et al. [23] classify aggregate functions with respect to four dimensions, which are lossy aggregation, duplicate sensitivity, resilience to losses/failures and correlation awareness. All these works above only address the characteristics of aggregate functions. Our

taxonomy, although inspired by these works in the classification of aggregate functions, includes all constituents of data aggregation processes and the time constraints, and provides a feature-diagram-based representation to organize the commonalities and variabilities of data aggregation processes.

A large proportion of existing works have their focus on in-network data aggregation, which is commonly used in sensor networks. In-network aggregation is the process of processing and aggregating data at intermediate nodes when data are transmitted from sensor nodes to sinks through the network [23]. Besides a classification of aggregate functions that we have discussed in the previous paragraph, Fasolo et al. [23] classify the existing routing protocols according to the aggregation method, resilience to link failures, overhead to setup/maintain aggregation structure, scalability, resilience to node mobility, energy saving method and timing strategy. Their work, however, does not address the characteristics of the data, and does not provide a structured representation for the common and variable characteristics. The aggregation protocols are also classified by Solis et al. [51], Makhloufi et al. [52], and Rajagopalan [53], with respect to different classification criteria. In contrast to the above works focusing mainly on aggregation protocols, Alzaid et al. [54] have proposed a taxonomy of secure aggregation schemes that classifies them into different models. All these works differ from our taxonomy in that they provide taxonomies from a different perspective, such as network topology for instance. Instead, our work strives to understand the features and their implications of data aggregation processes and its constituents in design.

4.2 Customized Transaction Management

Designing effective and efficient transaction management to suit applications' need has been a hot topic in the development of DBMS. In this section we discuss the related work regarding the design methodologies, and the formalization and analysis of transaction models.

4.2.1 Design methodologies for developing customized transaction management

Various design methodologies have been proposed in literature for developing transaction management for DBMS. KIDS [15], an early effort to construct a DBMS, identifies transaction management as an aspect, and decomposes it into sub-aspects representing functionalities such as concurrency control and

recovery. During the process proposed in KIDS, the database system designer has to decide the selection of run-time implementation techniques, based on the analysis of requirements and a classification of the implementation algorithms. AspectOPTIMA [18] proposes a reusable aspect-oriented framework for constructing customized transaction management. The designer selects particular aspects, such as a particular concurrency control algorithm and a recovery algorithm, which are composed to provide ACID assurance. These methodologies mainly focus on DBMS without real-time constraints. Moreover, formal methods are not used to guide the design decisions.

PRISMA [14] is a software product-line-oriented process for requirement engineering of flexible transaction models, which supports identification, reasoning and composition of ACID variants and their sub-features in the requirement phase. Formal methods are applied to reason about the consistency of the selected ACID variants, but not to the analysis of the transactional behaviors under various selected transaction management mechanisms. Temporal correctness of transactions is not considered in PRISMA.

Mentis et al. [20] propose a model-driven approach for generating the implementation of selected transaction models. The implementation is modeled in state machines and verified against the ACID properties, after which the code of the transaction management can be generated. Temporal correctness is not considered in this approach.

Khachana et al. [19] propose an approach to produce a monolithic transaction processing system able to adjust the relaxation of ACID properties at runtime, according to business requirements through user interaction. This approach, however, does not perform design-time analysis on the transactional properties, and does not target real-time applications.

COMET [16] combines a component-based approach and aspect-oriented programing to build tailored RTDBMS. Encapsulating database functionalities as components, and crosscutting features such as transaction management as aspects, COMET generates a tailored RTDBMS by weaving the selected components and aspects together. In FAME-DBMS [17], functional requirements on a DBMS are represented as features, which are composed to construct DBMS variants. Built on top of FAME-DBMS, AUTODAMA [55] generates tailorable DBMS specially for automotive systems. In these methodologies, the selection of building modules is based on functional requirements analysis, as well as constraints on code-size and performance. They mainly address resource consumption and footprint issues for embedded systems, rather than temporal correctness and the possible conflicts with ACID assurance.

Compared with these aforementioned works, our DAGGERS process starts

from deriving the real-time transaction models accounting for both temporal correctness and ACID properties. More importantly, we emphasize the analysis of trade-offs between these properties, and focus on a systematic approach to analyze different decisions. By iteratively applying formal modeling and verification of the transaction models, we are able to select the appropriate transaction management mechanisms that are proved to guarantee the desired transactional properties.

4.2.2 Formalization and analysis of transaction models

In the real-time community, the common technique to formally analyze transaction performance is schedulability analysis [56]. However, this technique only analyzes the temporal correctness of transactions. To our knowledge, less attention has been devoted to studying to which extent logical data consistency can be achieved in the real-time paradigm.

Formal verification of transaction models plays an important role in the design of customized transaction management. Some substantial work has already been carried out to specify transaction models and reason about their properties. One group of work is represented by the ACTA framework [69] and its successors. ACTA provides a first-order logic formalization to specify the transactional effects of data objects and the interaction between transactions, facilitating reasoning about transaction properties, as well as flexible synthesis of transaction models. Real-Time ACTA [70] extends ACTA with formalization of real-time constraints on transactions and data. However, the formal syntax and semantics for the specification of ACID variants provided by ACTA and Real-Time ACTA are limited, and tool support for verification is missing. SPECTRA [71], which improves the formal syntax of ACTA, and is used in KIDS for specifying transaction models, does not focus on the ACID and timeliness variants. GOLOG [72] improves ACTA by providing formal semantics for the building blocks, using situation calculus and tool support for simulation. However, organizing the building blocks with respect to ACID properties is not in their focus, and real-time properties are not supported. Used in the PRISMA process, SPLACID [73] improves ACTA by providing a more expressive and structured language support for ACID variants and their sub-features, however real-time properties are not considered.

In addition to the thread of ACTA, a variety of methods have been proposed to formalize transactions. For instance, Mentis et al. [20] model transaction models as state machines, and verify that the ACID properties can be satisfied by an implementation using model checking. Other works are intended

Table 4.1: Summary of non-ACTA related work

Related work	**Properties**	**Formalism**	**Analysis Technique**
Mentis et al. [20]	ACID	state machines	model checking
Cerone et al. [57]	logical consistency models, no explicit ACID	axioms and rules	theorem proving
Bocchi et al. [58]	consistency	process calculi	model checking
Kokash et al. [59]	atomicity	Reo, constraint automata	model checking
Grov et al. [60] and Liu et al. [61]	atomicity, isolation	Real-Time Maude	model checking
Suryavanshi et al. [62]	various properties, no explicit ACID	Event-B	theorem proving
Kirchberg [63]	various properties, no explicit ACID	transition rules	model checking
Makni et al. [64]	various properties, no explicit ACID	SPIN	model checking
Gaaloul et al. [65]	atomicity	event calculus	model checking
Bourne [66]	atomicity	temporal logics	model checking
Lanotte et al. [67]	atomicity and timeliness	timed automata	model checking
Kot [68]	various properties, timeliness, no explicit ACID	timed automata	model checking

for analyzing a selected subset of properties in particular scenarios. A list of non-ACTA related works are presented in Table 4.1. Compared with these works, our work is different in several aspects. First, our process strives to analyze both ACID and temporal correctness, which are not entirely covered by the aforementioned works. Second, we contribute to a general, reusable and flexible modeling approach for modeling ACID and temporal correctness properties, as well as the supporting transaction management mechanisms. In particular, our timed-automata framework provides flexible modeling capability for a rage of concurrency control mechanisms by using skeletons and patterns.

Chapter 5

Conclusions and Future Work

In this thesis we propose a systematic approach for designing data management with data aggregation support for real-time applications, in which data aggregation, as well as other data accessing and manipulating computations, are modeled as transactions managed by an RTDBMS. Our approach is the DAGGERS process that systematically derives customized transaction models incorporating the desired trade offs between ACID and temporal correctness properties, and decides the appropriate transaction management mechanisms to generate the RTDBMS. The trade-off decisions are made via formal verification of the desired properties. Concretely, we have proposed the following contributions to facilitate the process:

- A taxonomy of data aggregation processes, depicted as a feature diagram, for analyzing the characteristics of data aggregation and their implications. Three design rules and a set of design heuristics are proposed based on the taxonomy to provide guidance for the design of data aggregation processes.

- A pattern-based formal approach, within the timed automata framework, which facilitates flexible modeling of transactions with various concurrency control algorithms, and model checking isolation and temporal correctness of the modeled transactions.

By the DAGGERS process the system designer can systematically analyze the data-related computations in the real-time application, and trade off the conflicting properties. An RTDBMS can then be generated with run-time mechanisms that are verified to achieve the desired properties.

The validation on two real-world industrial case studies shows that our taxonomy enhances the understanding of the designed system, and raises awareness about the characteristics that need to be considered during the design. By applying the taxonomy, design flaws could be spotted and fixed prior to implementation. The underlying feature model allows data aggregation processes to be constructed by composing reusable features, whose feasibility can be reasoned about while time-related parameters can be derived. Our timed-automata-based approach allows designers to model transactions with various concurrency control algorithms, and formally analyze the satisfiability of isolation and temporal correctness. Based on the analysis the designer can choose a desired trade-off and select the appropriate concurrency control algorithm that has been proved to achieve the selected trade off.

5.1 Future Work

The work of this thesis opens several future research directions. One possible future work involves the integration of DAGGTAX with state-of-art architectural and process modeling frameworks, such that the feature selection of data aggregation processes can be reasoned about together with the choices of deployment and business process models.

Another future work may involve the trading-off methods for other transaction properties. In this thesis we have proposed the method of modeling transactions with concurrency control, with the aim of analyzing isolation and temporal correctness. In the future work we need to develop methods to address the analysis of atomicity, consistency and durability, together with the corresponding DBMS mechanisms.

Integration of other formal techniques may greatly enhance the applicability and scalability of DAGGERS. Since real-time data-intensive applications are often heterogeneous in the amount of data, complexity of computation, and the desired properties, different analysis techniques may need to be applied depending on the particular system or module. For instance, for a large system, Statistical Model Checking [74] could be a better choice than exhaustive model checking in terms of scalability. For quantitative analysis of data production and consumption in aggregation, dataflow models [75] could be useful. Our

future work will investigate which formal methods could be exploited to analyze different situations, and how they can be integrated into the big picture of system design.

Last but not least, a larger scale evaluation of the entire DAGGERS process on the design of an industrial system, with respect to its scalability and efficiency, is also an interesting future work.

Bibliography

[1] R. Hegde, G. Mishra, and K. Gurumurthy. *An insight into the hardware and software complexity of ecus in vehicles*, pages 99–106. Springer, 2011.

[2] A. N. Lee and J. L. M. Lastra. Data aggregation at field device level for industrial ambient monitoring using web services. In *Proceedings of the 9th IEEE International Conference on Industrial Informatics*, pages 491–496, July 2011.

[3] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[4] S. F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. Deeds towards a distributed and active real-time database system. *ACM Sigmod Record*, 25(1):38–51, March 1996.

[5] I. J. Rudas, E. Pap, and J. Fodor. Information aggregation in intelligent systems: An application oriented approach. *Knowledge-Based Systems*, 38:3–13, 2013.

[6] G.R. Goud, N. Sharma, K. Ramamritham, and S. Malewar. Efficient real-time support for automotive applications: A case study. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 335–341, 2006.

[7] K. Bür, P. Omiyi, and Y. Yang. Wireless sensor and actuator networks: Enabling the nervous system of the active aircraft. *IEEE Communications Magazine*, 48(7):118–125, 2010.

[8] D. Mairiza, D. Zowghi, and N. Nurmuliani. An investigation into the notion of non-functional requirements. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 311–317, 2010.

[9] L. Almeida, F. Santos, T. Facchinetti, P. Pedreiras, V. Silva, and L. S. Lopes. Coordinating distributed autonomous agents with a real-time database: The cambada project. In *Proceedings of the 19th International Symposium of Computer and Information Sciences*, 2004.

[10] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers Inc., 1992.

[11] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[12] X. Song and J.W.S. Liu. How well can data temporal consistency be maintained? In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design (CACSD)*, pages 275–284, 1992.

[13] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *Computer*, 32(6):29–36, 1999.

[14] B. Gallina. *PRISMA: a software product line-oriented process for the requirements engineering of flexible transaction models*. PhD thesis, University of Luxembourg, Luxembourg, 2010.

[15] A. Geppert, S. Scherrer, and K.R. Dittrich. Kids: Construction of database management systems based on reuse. *University of Zurich, Switzerland*, 1997.

[16] D. Nyström, A. Tešanovic, M. Nolin, C. Norström, and J. Hansson. Comet: a component-based real-time database for automotive systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems*, pages 1–8. IET, 2004.

[17] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. Fame-dbms: tailor-made data management solutions for embedded systems. In *Proceedings of the 2008 EDBT workshop on Software engineering for tailor-made data management*, pages 1–6. ACM, 2008.

[18] J. Kienzle, E. Duala-Ekoko, and S. Gélineau. Aspectoptima: A case study on aspect dependencies and interactions. In *Transactions on Aspect-Oriented Software Development V*, pages 187–234. Springer, 2009.

[19] R. T. Khachana, A. James, and R. Iqbal. Relaxation of acid properties in autra, the adaptive user-defined transaction relaxing approach. *Future Generation Computer Systems*, 27(1):58–66, 2011.

[20] A. Mentis and P. Katsaros. Model checking and code generation for transaction processing software. *Concurrency and Computation: Practice and Experience*, 24(7):711–722, 2012.

[21] S. Cai, B. Gallina, D. Nyström, and C. Seceleanu. Trading-off data consistency for timeliness in real-time database systems. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems, Work-in-Progress*, pages 13–16, 2015.

[22] S. F. Andler, L. Niklasson, B. Olsson, A. Persson, L. J. de Vin, B. Wangler, T. Ziemke, and T. Planstedt. Information fusion from databases, sensors and simulations: A collaborative research program. In *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop*, pages 234–244, April 2005.

[23] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *IEEE Wireless Communications*, 14(2):70–87, 2007.

[24] J. L. Azevedo, B. Cunha, and L. Almeida. Hierarchical distributed architectures for autonomous mobile robots: A case study. In *Proceedings of the 2007 IEEE Conference on Emerging Technologies and Factory Automation*, pages 973–980, Sept 2007.

[25] T. He, L. Gu, L. Luo, T. Yan, J.A. Stankovic, and S.H. Son. An overview of data aggregation architecture for real-time tracking with sensor networks. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2006.

[26] K. Ramamritham and P. K. Chrysanthis. A taxonomy of correctness criteria in database applications. *The International Journal on Very Large Data Bases*, 5(1):085–097, January 1996.

[27] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2004.

[28] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[29] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[30] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[31] E. B. Moss. *NESTED TRANSACTIONS: AN APPROACH TO RELIABLE DISTRIBUTED COMPUTING*. PhD thesis, Massachusetts Institute of Technology, USA, 1981.

[32] H. Garcia-Molina and K. Salem. Sagas. *ACM SIGMOD Record*, 16(3):249–259, December 1987.

[33] ISO/IEC 9075:1992 Database Language SQL. Standard, International Organization for Standardization.

[34] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, May 1995.

[35] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 67–78, 2000.

[36] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in database systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 175–193. Morgan Kaufmann Publishers Inc., 1998.

[37] P. S. Yu, K. L. Wu, K. J. Lin, and S. H. Son. On real-time databases: concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, Jan 1994.

[38] T. W. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-intensive applications. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 35–45, Dec 1992.

[39] L. B. C. DiPippo and V. F. Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 87–96, 1993.

[40] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[41] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[42] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[43] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[44] H. J. Holz, A. Applin, B. Haberman, D. Joyce, H. Purchase, and C. Reed. Research methods in computing: What are they, and how should we teach them? *ACM SIGCSE Bulletin*, 38(4):96–114, June 2006.

[45] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, USA, 1990.

[46] C. Vitucci and A. Larsson. Hat, hardware assisted trace: Performance oriented trace & debug system. In *Proceedings of 26th International Conference on Software & Systems Engineering and their Applications*, 2015.

[47] R. Mesiar, A. Kolesárová, T. Calvo, and M. Komorníková. A review of aggregation functions. In *Fuzzy Sets and Their Extensions: Representation, Aggregation and Models*, volume 220, pages 121–144. Springer Berlin Heidelberg, 2008.

[48] J. Marichal. *Aggregation Functions for Decision Making*, pages 673–721. ISTE, 2010.

[49] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[50] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[51] I. Solis and K. Obraczka. In-network aggregation trade-offs for data collection in wireless sensor networks. *International Journal of Sensor Networks*, 1(3-4):200–212, 2006.

[52] R. Makhloufi, D. Guillaume, B. Grégory, and D. Gaïti. A survey and performance evaluation of decentralized aggregation schemes for autonomic management. *International Journal of Network Management*, 24(6):469–498, 2014.

[53] R. Rajagopalan and P.K. Varshney. Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8(4):48–63, 2006.

[54] H. Alzaid, F. Ernest, J. Manuel, G. Nieto, and D. Park. A taxonomy of secure data aggregation in wireless sensor networks. *International Journal of Communication Networks and Distributed Systems*, 8(1-2):101–148, 2012.

[55] T. Thüm, S. Schulze, M. Pukall, G. Saake, and S. Günther. Secure and customizable data management for automotive systems: A feasibility study. *ISRN Software Engineering*, 2012.

[56] S. Han, K. Lam, J. Wang, K. Ramamritham, and A.K. Mok. On co-scheduling of update and control transactions in real-time sensing and control systems: Algorithms, analysis, and performance. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2325–2342, 2013.

[57] A. Cerone, G. Bernardi, and A. Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In *Proceedings of the 26th International Conference on Concurrency Theory*, volume 42, pages 58–71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015.

[58] L. Bocchi and H. Melgratti. On the behaviour of general purpose applications on cloud storages. *Service Oriented Computing and Applications*, 9(3):213–227, 2015.

[59] N. Kokash and F. Arbab. Formal design and verification of long-running transactions with extensible coordination tools. *IEEE Transactions on Services Computing*, 6(2):186–200, April 2013.

[60] J. Grov and P. C. Ölveczky. *Increasing Consistency in Multi-site Data Stores: Megastore-CGC and Its Formal Analysis*, pages 159–174. Springer International Publishing, 2014.

[61] S. Liu, P. C. Ölveczky, M. R. Rahman, J. Ganhotra, I. Gupta, and J. Meseguer. Formal modeling and analysis of ramp transaction systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1700–1707. ACM, 2016.

[62] R. Suryavanshi and D. Yadav. Modeling of multiversion concurrency control system using event-b. In *Proceedings of the 2012 Federated Conference on Computer Science and Information Systems*, pages 1397–1401, Sept 2012.

[63] M. Kirchberg. Using abstract state machines to model aries-based transaction processing. *Journal of Universal Computer Science*, pages 157–194, 2009.

[64] A. Makni, R. Bouaziz, and F. Gargouri. Formal verification of an optimistic concurrency control algorithm using spin. In *Proceedings of the 13th International Symposium on Temporal Representation and Reasoning*, pages 160–167, June 2006.

[65] W. Gaaloul, M. Rouached, C. Godart, and M. Hauswirth. *Verifying Composite Service Transactional Behavior Using Event Calculus*, pages 353–370. Springer Berlin Heidelberg, 2007.

[66] S. Bourne. *Formal Verification of Transactional and Configurable Service-Oriented Processes*. PhD thesis, UNIVERSITY OF ADELAIDE, Australia, 2016.

[67] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. Modeling long-running transactions with communicating hierarchical timed automata. In *Formal Methods for Open Object-Based Distributed Systems*, pages 108–122. Springer, 2006.

[68] M. Kot. Modeling selected real-time database concurrency control protocols in uppaal. *Innovations in Systems and Software Engineering*, 5(2):129–138, 2009.

[69] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[70] M. Xiong and K. Ramamritham. Specification and analysis of transactions in real-time active databases. In *Real-Time Database and Information Systems: Research Advances*, volume 420, pages 327–351. Springer, 1997.

[71] A. Geppert. *Methodical construction of database management systems*. PhD thesis, University of Zuerich, Switzerland, 1994.

[72] I. Kiringa. Simulation of advanced transaction models using golog. In *Database Programming Languages*, volume 2397, pages 318–341. Springer Berlin Heidelberg, 2002.

[73] B. Gallina and N. Guelfi. Splacid: An spl-oriented, acta-based, language for reusing (varying) acid properties. In *Proceedings of the 32nd Annual IEEE Software Engineering Workshop*, pages 115–124, 2008.

[74] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 80–96. Springer, 2011.

[75] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

II

Included Papers

Chapter 6

Paper A: DAGGTAX: A Taxonomy of Data Aggregation Processes

Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu.
MRTC Report, Mälardalen University, May 2017.
A shorter version has been submitted to the 7th International Conference on Model & Data Engineering (MEDI).

Abstract

Data aggregation processes are essential constituents for data management in modern computer systems, such as decision support systems and Internet of Things (IoT) systems. Due to the heterogeneity and real-time constraints in such systems, designing appropriate data aggregation processes often demands considerable efforts. A study on the characteristics of data aggregation processes will provide a comprehensive view for the designers, and facilitate potential tool support to ease the design process. In this paper, we propose a taxonomy called DAGGTAX, which is a feature diagram that models the common and variable characteristics of data aggregation processes, especially focusing on the real-time aspect. The taxonomy can serve as the foundation of a design tool that enables designers to build an aggregation process by selecting and composing desired features, and to reason about the feasibility of the design. We also provide a set of design heuristics that could help designers to decide the appropriate mechanisms for achieving the selected features. Our industrial case study demonstrates that DAGGTAX not only strengthens the understanding, but also facilitates the model-driven design of data aggregation processes.

6.1 Introduction

In modern information systems, data aggregation, defined as the process of producing a synthesized form from multiple data items [1], is commonly applied for data processing and management. For example, in order to discover unusual patterns and infer information, a data analysis application often computes a synthesized value from a subset of the database for statistical analysis [2]; in systems dealing with large amounts of data with limited storage, the data are often aggregated to save space [3]; in a sensor network, sensor data are aggregated, and only the aggregated data are transmitted so as to save bandwidth and energy [4]. Since data aggregation plays a key role in many applications, considerable research efforts have been dedicated to this topic. A number of taxonomies have been proposed to provide a comprehensive understanding on various aspects of data aggregation, such as aggregate functions ([2, 5, 1]), aggregation protocols ([6, 4, 7]) and security models ([8]).

The focus of this paper is instead on another important aspect: the data aggregation process (or DAP for short) itself. We consider a DAP as three ordered activities that allow raw data to be transformed into aggregated data via an aggregate function. First, a DAP starts with preparing the raw data needed for the aggregation from the data source into the aggregation unit called the aggregator. Next, an aggregate function is applied by the aggregator on the raw data, and produces the aggregated data. Finally, the aggregated data may be further handled by the aggregator, for example, to be saved into storage or provided to other processes. The main constituents of these activities are the raw data, the aggregate function and the aggregated data.

The main contribution of this paper is a global, high-level characterization of data aggregation processes. We justify our study of the DAP by the fact that it represents a pillar of an aggregation application's workflow, no matter if it is a centralized database management system or a highly distributed sensor network. Understanding DAP is essential to a correct design of the overall application. For instance, a sensor data gathering process, a data aggregation process and an analytic process form the basic workflow of a surveillance application. Multiple DAP can also work together, one's aggregated data being another's raw data, to form a more complex, hierarchical aggregation process. To design a DAP, we must understand the desired features of its main constituents, that is, the raw data, the aggregate function and the aggregated data, as well as those of the DAP itself. Such features, ranging from functional features (such as data sharing) to extra-functional features (such as timeliness), are varying depending on different applications. One aspect of the understand-

ing is to distinguish the mandatory features from the optional ones, so that the application designer is able to sort out the design priorities. Another aspect is to comprehend the implications of the features, and to reason about the (possible) impact on one another. Conflicts may arise among features, in that the existence of one feature may prohibit another one. Trade-offs should be taken into consideration at design time, so that infeasible designs can be ruled out at an early stage.

Among all features, we are particularly interested in the time-related properties of the DAP, since data aggregation is extensively applied in many real-time systems, such as automotive systems [9], avionic systems [10] and industrial automation [11]. In real-time systems, the correctness of a process depends on whether it completes on time, and validity of data depends on the time they are collected and accessed. These real-time properties are expected on raw data, aggregate function and the aggregated data, and impose constraints that cross-cut all three activities of a DAP. Therefore, we will especially emphasize the real-time related features and their implications.

In this paper we therefore propose a taxonomy of data aggregation processes, called DAGGTAX (Data AGGregation TAXonomy), with a focus on their features and consequent implications, from the perspective of the aggregation process itself. The proposed taxonomy is presented as a feature diagram [12]. The aim of our taxonomy is to ease the design of aggregation processes, by providing a comprehensive view on the features and cross-cutting constraints, with a systematic representation. The latter can serve as the basis of a design tool, which enables selecting the desired features, reasoning about possible trade-offs, reducing the design space of the application, and composing the features to build the desired aggregation processes.

The remaining part of the paper is organized as follows. In Section 6.2 we discuss the existing taxonomies of data aggregation. In Section 6.3 we present the preliminaries, followed by a survey of data aggregation processes in scientific literatures in Section 6.4. Section 6.5 presents the proposed taxonomy, and in Section 6.6 we introduce the design rules and heuristics based on the implications of the features presented in the taxonomy. In Section 6.7 we validate the taxonomy by a case study from industry. Section 6.8 gives a further discussion of the implications of the real-time features, before concluding the paper in Section 6.9.

6.2 Related Work

Many researchers have promoted the understanding of data aggregation on various aspects. Among these works, considerable efforts have been made on the study of aggregate functions. Mesiar et al. [13], Marichal [14], and Rudas et al. [1] have studied the mathematical properties of aggregate functions, such as continuity and stability, and discussed these properties of common aggregate functions in detail. A procedure for the construction of an appropriate aggregate function is also proposed by Rudas et al. [1]. In order to design a software system that computes aggregation efficiently, Gray et al. [2] have classified aggregate functions into distributive, algebraic and holistic, depending on the amount of intermediate states required for partial aggregates. Later, in order to study the influence of aggregate functions on the performance of sensor data aggregation, Madden et al. [5] have extended Gray's taxonomy, and classified aggregate functions according to their state requirements, tolerance of loss, duplicate sensitivity, and monotonicity. Fasolo et al. [4] classify aggregate functions with respect to four dimensions, which are lossy aggregation, duplicate sensitivity, resilience to losses/failures and correlation awareness. Our taxonomy builds on these works that focus on the aggregate functions mainly, and provide a comprehensive view of the entire aggregate processes instead.

A large proportion of existing works have their focus on in-network data aggregation, which is commonly used in sensor networks. In-network aggregation is the process of processing and aggregating data at intermediate nodes when data are transmitted from sensor nodes to sinks through the network [4]. Besides a classification of aggregate functions that we have discussed in the previous paragraph, Fasolo et al. [4] classify the existing routing protocols according to the aggregation method, resilience to link failures, overhead to setup/maintain aggregation structure, scalability, resilience to node mobility, energy saving method and timing strategy. The aggregation protocols are also classified by Solis et al. [7], Makhloufi et al. [6], and Rajagopalan [15], with respect to different classification criteria. In contrast to the above works focusing mainly on aggregation protocols, Alzaid et al. [8] have proposed a taxonomy of secure aggregation schemes that classifies them into different models. All these works differ from our taxonomy in that they provide taxonomies from a different perspective, such as network topology for instance. Instead, our work strives to understand the features and their implications of DAP and its constituents in design.

6.3 Preliminaries

In this section, we first recall the concepts of timeliness and temporal data consistency in real-time systems, after which we introduce feature models and feature diagrams that are used to present our taxonomy.

6.3.1 Timeliness and Temporal Data Consistency

In a real-time system, the correctness of a computation depends on both the logical correctness of the results, and the time at which the computation completes [16]. The property of completing the computation by a given deadline is referred to as *timeliness*. A real-time task can be classified as *hard*, *firm* or *soft* real-time, depending on the consequence of a deadline miss [16]. If a hard real-time task misses its deadline, the consequence will be catastrophic, e.g., loss of life or significant amounts of money. Therefore the timeliness of hard real-time tasks must always be guaranteed. For a firm real-time task, such as a task detecting vacant parking places, missing deadlines will render the results useless. For a soft real-time task, missing deadlines will reduce the value of the results. An example of soft real-time task is the signal processing task of a video meeting application, whose quality of service will degrade if the task misses its deadline.

Depending on the regularity of activation, real-time tasks can be classified as *periodic*, *sporadic* or *aperiodic* [16]. A periodic task is activated at a constant rate. The interval between two activations of a periodic task, called its *period*, remains unchanged. A sporadic task is activated with a *Minimum INter-arrival Time (MINT)*, that is, the minimum interval between two consecutive activations. During the design of a real-time system, a sporadic task is often modeled as a periodic task with a period equal to the MINT. An aperiodic task is activated with an unpredictable interval between two consecutive activations. A task triggered by an external event with unknown occurrence pattern can be seen as aperiodic.

Real-time applications often monitor the state of the environment and react to changes accordingly and timely. The environment state is represented as data in the system, which must be updated according to the actual environment state. The coherency between the value of the data in the system and its corresponding environment state is referred to as *temporal data consistency*, which includes two aspects, the *absolute temporal validity* and *relative temporal validity* [17]. A data instance is absolute valid, if the timespan between the time of sampling its corresponding real-world value, and the current time, is

less than a specified *absolute validity interval*. A data instance derived from a set of data instances (base data) is absolute valid if all participating base data are absolute valid. A derived data instance is relative valid, if the base data are sampled within a specified interval, called *relative validity interval*.

Data instances that are not temporally consistent may lead to different consequences. Different levels of strictness with respect to temporal consistency thus exist, which are *hard*, *firm* and *soft* real-time, in a decreasing order of strictness. Using outdated hard real-time data could cause disastrous consequences, and therefore this should not appear. Firm real-time data are useless if they are outdated, whereas outdated soft real-time data can still be used, but will yield degraded usefulness.

6.3.2 Feature Model and Feature Diagram

The notion of *feature* was first introduced by Kang et al. in the Feature-Oriented Domain Analysis (FODA) method [12], in order to capture both the common characteristics of a family of systems as well as the differences between individual systems. Kang et al. define a feature as a prominent or distinctive system characteristic visible to end-users. Czarnecki and Eisenecker extend the definition of a feature to be any functional or extra-functional characteristic at the requirement, architecture, component, or any other level [18]. This definition allows us to model the characteristics of data aggregation processes as features. A *feature model* is a hierarchically organized set of features, representing all possible characteristics of a family of software products. A particular product can be formed by a combination of features, often called a configuration, selected from the feature model of its family.

A feature model is usually represented as a *feature diagram* [12], which is often depicted as a multilevel tree, whose nodes represent features and edges represent decomposition of features. In a feature diagram, a node with a solid dot represents a common feature (as shown in Figure 6.1a), which is mandatory in every configuration. A node with a circle represents an optional feature (Figure 6.1b), which may be selected by a particular configuration. Several nodes associated with a spanning curve represent a group of alternative features (Figure 6.1c), from which one feature must be selected by a particular configuration. The cardinality [m..n] ($n \geq m \geq 0$) annotated with a node in Figure 6.1d denotes how many instances of the feature, including the entire sub-tree, can be considered as children of the feature's parent in a concrete configuration. If $m \geq 1$, a configuration must include at least one instance of the feature, e.g., a feature with [1..1] is then a mandatory feature. If $m = 0$, the

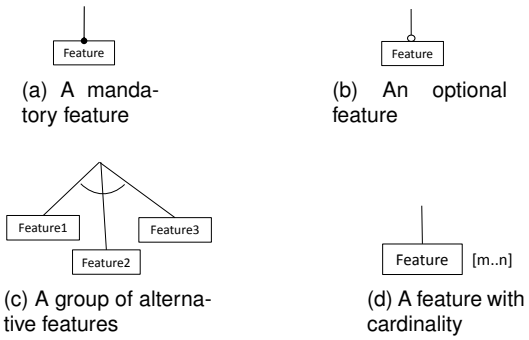


Figure 6.1: Notations of a feature diagram

feature is optional for a configuration.

A valid configuration is a combination of features that meets all specified constraints, which can be dependencies among features within the same model, or dependencies among different models. An example of such a constraint is that the selection of one feature requires the selection of another feature. Researchers in the software product line community have developed a number of tools, providing extensive support for feature modeling and the verification of constraints. For instance, in FeatureIDE [19], software designers can create feature diagrams using a rich graphic interface. Designers can specify constraints across features as well as models, to ensure that only valid configurations are generated from the feature diagram.

6.4 A Survey of Data Aggregation Processes

Serving as an important information processing and analysis technique, data aggregation has been widely applied in a variety of information management systems. Based on scientific literature, in this section, we present a limited survey of application examples that implement data aggregation processes. In order to extract heuristics that help us generate our taxonomy, we select the examples from a wide variety of application domains, and investigate the common and different characteristics of aggregation processes. Some of these examples are general-purpose infrastructures that implement aggregation as a basic service. The other examples develop data aggregation as ad hoc solutions suitable for the particular application scenarios.

In the following subsections, we first present how aggregation is supported in different general-purpose infrastructures that provide data processing and management. Next, a number of ad hoc applications are presented, focusing on the requirements that the aggregation processes implemented in such applications must meet. Finally, we discuss the characteristics of aggregation processes exposed in the surveyed systems and applications.

6.4.1 General-purpose Infrastructures

In this subsection, we investigate the design of aggregation processes in general-purpose systems from the following domains: database management systems, data warehouses, data stream management systems and wireless sensor networks.

Database Management Systems and Data Warehouses Many information management systems adopt a general-purpose relational Database Management System (DBMS) or a Data Warehouse (DW) [20] as a back-end for centralized data management, which have common aggregate functions implemented, and exposed as interfaces for users or programmers. Internally, aggregation is supported by a number of infrastructural services, including query evaluation, data storage and accessing, trigger mechanism, and transaction management. In a typical disk-based relational DBMS, as illustrated in Figure 6.2, data are stored as tuples in the disk. An aggregation process is started by a query issued by a client. The DBMS then evaluates the query and loads the relevant tuples from disk into the main memory. An aggregate function is performed on the tuples and computes the aggregated value, which is then returned to the query issuer, cached in main memory or stored in the disk. An aggregation process can also be triggered by a state change in the database. Both raw data and aggregated data can be accessed by other processes. In order to maintain logical data consistency, such processes, including the aggregation process, are treated as transactions and governed by the transaction management system, which ensures the so-called ACID (Atomicity, Consistency, Isolation, and Durability) properties [21] during their executions.

Data can be aggregated by categories, usually specified in the ”group-by” clause of a query. These categories may have a hierarchical relationship and thus represent the granularities of aggregation. For example, in a temporal database, users may choose to aggregate data by day, week or month, with a coarser granularity; in a spatial database, the aggregation can be based on

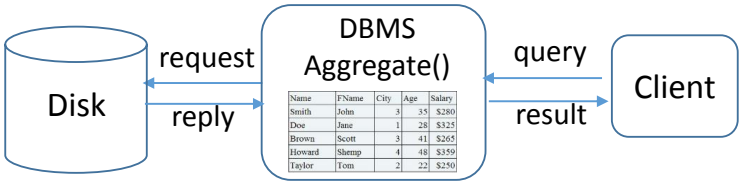


Figure 6.2: Illustration of aggregation in a disk-based DBMS

streets, cities and provinces [22]. In a data warehouse, the stored data usually have many dimensions, and the aggregation may need to be performed on multiple dimensions [20].

The aggregated value may be returned to the query issuer directly, or may be stored persistently in the database as a normal tuple. Alternatively, the aggregated values are cached in materialized views, so that other processes can make use of them [23]. It is common to store the aggregated values as materialized views in data warehouse since these results will be frequently used by analysis processes [20].

A number of aggregate functions are included in the SQL standard and are commonly supported by general-purpose DBMS. Other aggregate functions can be defined as user-defined functions. The aggregation can be triggered by an explicit query issued by the client, or by a trigger that reacts to the change of the database. In a data warehouse, aggregation is often planned periodically to refresh the materialized views using the updated base data. In case a query needs to access current data between the planned aggregation processes, extra aggregation processes may also be started to refresh the views [20].

Online Aggregation in Data Stream Management Systems Data aggregation in traditional DBMS and DWs is performed like batch-processing: on a large number of tuples and in considerable time before returning the aggregated value. To improve performance and user experience, Hellerstein et al. propose "online aggregation" [24], which allows tuples to be aggregated incrementally. Tuples are selected from a base table by a sampling process, and aggregated with the cached partial aggregated result from previously sampled tuples. The partially aggregated value is available, which refers to the user as an approximate aggregated result. The aggregation process is defined with a stopping interface, through which the aggregation can be stopped, giving the approximate result as the final result.

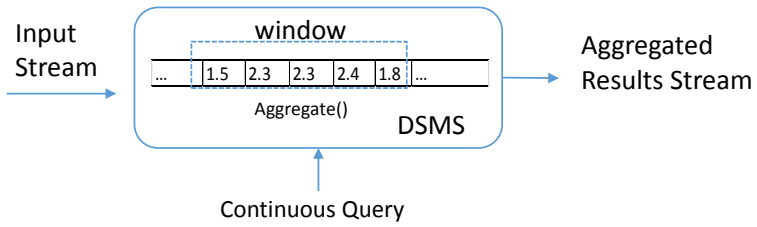


Figure 6.3: Illustration of aggregation in a data stream management system

Online aggregation is often supported by Data Stream Management Systems (DSMS), which provide aggregation for continuous data streams. In Figure 6.3, we illustrate the aggregation in a typical DSMS scenario. Usually, stream data are pushed into the DSMS continuously, often at a high frequency. Individual data instances are not significant, become stale as time passes, and do not need to be stored persistently. Finite subsets of the most recent incoming stream ("windows") are cached in the system. Aggregate functions can be defined by users and are applied on the windows. In the Aurora data stream management system [25], the aggregate function can be associated with a "timeout" parameter, indicating the deadline of the computation of the function. A function should return before it times out, even if some raw data instances are missing or delayed, so as to provide timely response required by many real-time applications. Aurora has implemented a load shedding mechanism, which drops data instances when the system is overloaded. The aggregation is triggered either by continuous queries with specified periods, or by ad hoc queries which are issued by clients. The aggregated results are passed to the receiving application as an outgoing stream. To provide historical data, the aggregated data may also be kept persistently for a specified period of time.

Multiple aggregation processes can be run concurrently, performing aggregation on the same data stream [26]. Oyamada et al. [27] point out that the aggregation in a DSMS may also involve non-streaming data, which can be shared and updated by other processes, causing potential data inconsistency. The authors propose a concurrency control mechanism to prevent the inconsistency.

In-network Aggregation in Wireless Sensor Networks Data aggregation plays an essential role in Wireless Sensor Network (WSN) applications. In these applications, numerous data are gathered from resource-constrained sen-

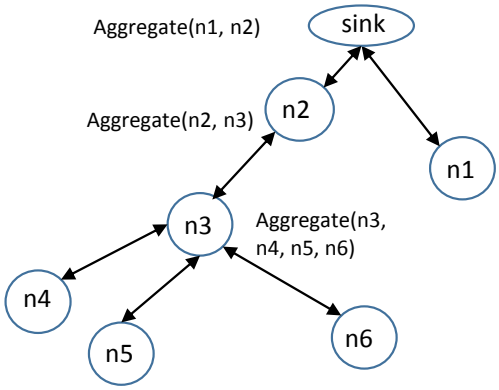


Figure 6.4: Illustration of aggregation in a wireless sensor network

sor nodes that are deployed to monitor the environment. The gathered data are transmitted through a network to sink nodes, which are equipped with more resource for advanced computation and analysis. Along the transmission, data are aggregated in the intermediate sensor nodes or special aggregate nodes, in a decentralized topology. This aggregation technique is also called "in-network aggregation" [4]. In contrast, a sensor network can also apply centralized aggregation if the data of all sensors are transmitted to and aggregated in one single node. Figure 6.4 gives an example of data aggregation in a sensor network. In this example, data from nodes n4, n5 and n6 are aggregated in node n3. This aggregated result is then transmitted to n2, and aggregated with the data of n2. Finally, the data from n2 and n1 are aggregated in the sink node.

Madden et.al [5] propose Tiny AGgregation (TAG), a generic aggregation service for ad hoc networks. In TAG, the user poses aggregation queries from a base station, which are distributed to the nodes in the network. Sensors collect data and route data back to the base station through a routing tree. As the data flow up the tree, it is aggregated by an aggregation function and value-based partitioning according to the query, level by level. At each level, a node awakens when it receives the aggregate request, together with a deadline when it should reply to its parent, and propagates the request to its children with an earlier deadline. Each node then listens to its children, aggregates the data transmitted from the children and the reading of itself, and then replies the aggregated result to its parent. If any node replies after its specified deadline, its value will not be aggregated by its parent, which means that the final aggre-

gated result is actually an approximation. The aggregated results are cached by the nodes, and can be used for fault tolerance reason, e.g., loss of messages from a child. TAG has also classified aggregate functions into distributive, algebraic, holistic, unique and content-sensitive. Decentralized in-network aggregation is only appropriate for distributive and algebraic aggregate functions, since they can be decomposed into sub-aggregates. For other functions, all sensor data have to be collected to one node and aggregated together.

TAG is later implemented in the TinyDB [28], which supports SQL-style queries. Aggregation can be triggered periodically by continuous queries, or at once by a state change or an ad hoc query. Aggregated results can be stored persistently as storage points, which may be accessed by other processes.

6.4.2 Ad Hoc Applications

Many applications have unique requirements, and consequently use their ad hoc aggregation processes to fulfill their requirements. Examples of such applications are presented in the following paragraphs.

He et al. present the VigilNet for real-time surveillance with a tiered architecture [29]. Four layers are implemented in this system and each layer has its data aggregation requirements. The data aggregation architecture of VigilNet is illustrated in Figure 6.5. The first layer is the sensor layer in which data inputs are pushed from individual sensors at specific rates, and aggregated as detection confidence vectors. In this layer the aggregation needs to meet stringent real-time constraints since the sensors send signals about fast-moving targets. The results of sensor-layer aggregation are sent to the node for node-layer aggregation. Each sensor node includes several sensors, and computes the average of sensor confidence vectors incrementally when a new sensor confidence vector arrives. If the aggregated results show the existence of a tracking target, the node estimates the position of the target, and sends a report to the leading node of the local group. The leader buffers the reports from members, until the number reaches a predefined aggregation degree. Then, it aggregates all the reports, estimates the current position of the target, and sends the aggregated report to the base station. The base station aggregates the new report with historical positions of the target, and calculates the velocity using a linear regression procedure.

Defude et al. propose the VESPA (Vehicular Event Sharing with a mobile P2P Architecture) approach [30] for the Vehicular Ad hoc NETwork (VANET), to aggregate traffic information events, such as parking places, accidents and road obstacles, pushed from neighbor vehicles. The events are aggregated by

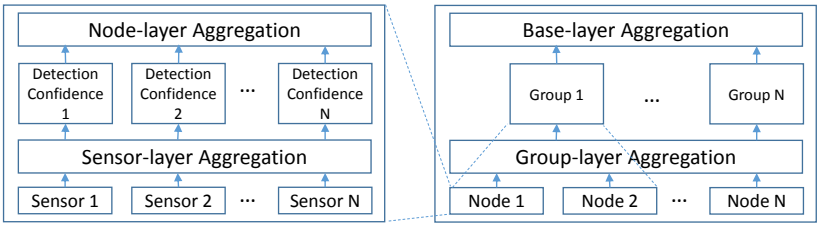


Figure 6.5: Data Aggregation Architecture of VigilNet [29]

times, areas and event types. The aggregated values are stored and accessed for further analysis.

Goud et al. [9] propose a real-time data repository for automotive adaptive cruise control systems. It includes an Environment Data Repository (EDR) and a Derived Data Repository (DDR). The EDR periodically reads sensor readings, aggregates them, and keeps the aggregated value in the repository. The DDR then reads and aggregates the values from EDR, only when the changes of readings from some sensors exceed a threshold. The sensor data are real-time and have their validity intervals. The aggregate processes must complete before the data become invalid, and produce the results for other processes with stringent deadlines.

Arai et al. propose an adaptive two phase approach for approximate ad hoc aggregation in unstructured peer-to-peer (P2P) systems [31]. When an ad hoc aggregate query is issued, in the first phase, sample peers are visited by a random walk from the sink, with a predefined depth. Information of the visited peers are collected to the sink, and analyzed to decide the peers to be aggregated. These peers are then visited in the second phase. For some aggregate functions such as COUNT and AVERAGE, partial aggregate results are computed in the local peer, and returned to the sink. For other aggregate functions, raw data are returned to the sink and aggregated in the sink.

Baulier et al. [32] propose a database system for real-time event aggregation in telecommunication systems. Events generated by phone calls are pushed into the system, which should be aggregated within specific response times. The aggregated results are kept in a main-memory database as views for other time-critical processes. When a new event arrives, it triggers the aggregate process to update the aggregate view. The event record itself is stored into a data warehouse persistently, which is not time-critical.

Bar et al. [33] propose an online aggregation system for network traffic

Table 6.1: Characteristics of Data Aggregation Processes in the Surveyed Applications

Sample	Raw Data	Aggregate Function	Aggregated Data	Triggering Pattern	Real-time Characteristics
relational disk-based DBMS/DW [23, 22, 20]	pulled from data sources; persistently stored; possibly shared by other processes	various functions	possibly durable; possibly shared by other processes	activated by events (queries or database triggers), or activated periodically	usually no deadlines
DSMS (AURORA [25], Oyamada et al. [27], Krishnamurthy et al. [26])	pushed by data sources; possibly pushed periodically; not persistently stored; cached for a particular period; real-time; possibly shared by other processes; can be shedded	various functions	pushed to receiver; possibly durable; may be stored for a period of time	activated by events (ad hoc queries), or activated periodically (periodic continuous queries)	deadlines depending on the application
WSN (TAG [5], TinyDB [28])	pulled from data sources; not persistently stored; possibly be skipped	various functions	cached for a particular period; possibly durable; real-time; possibly shared by other processes	activated by events, or activated periodically	deadlines depending on the application
VigilNet [29], sensor layer	pushed by data sources; not persistently stored; real-time; pushed periodically	detection confidence function	pushed to receiver; not durable	activated periodically	hard deadlines
VigilNet [29], node layer	pushed by data sources; not persistently stored	average	pushed to receiver; not durable	activated by event	soft deadlines
VigilNet [29], group layer	pushed by data sources; cached for a particular period	ad hoc calculation	pushed to receiver; not durable	activated by event	soft deadlines

VigilNet [29], base layer	pushed by data sources; persistently stored	regression	shared by other processes	activated by event	soft deadlines
VESPA [30]	pushed by data sources	various functions	durable; shared by other processes	activated by events	soft deadlines
Goud et al. [9], EDR	pulled from data sources; pulled periodically; real-time; not persistently stored	various functions	not durable; real-time; shared by other processes	activated periodically	hard deadlines
Goud et al. [9], DDR	pulled from data sources; real-time; not persistently stored	various functions	durable; real-time	activated by events	hard deadlines
Arai et al. [31]	pulled from data sources; not persistently stored	various functions	possibly durable	activated by events	no deadlines
Baulier et al. [32]	pushed by data sources; persistently stored	various functions	real-time; not durable; shared by other processes	activated by events	hard deadlines
Bar et al. [33]	pushed by data sources; persistently stored; possibly real-time	various functions	durable	activated by events, or activated periodically	soft deadlines
Bür et al. [10]	pushed by data sources; not persistently stored; real-time;	various functions	not durable; real-time	activated periodically	hard deadlines
Lee et al. [11], device level	pushed by data sources; real-time	various functions	pushed to receiver; not durable	activated by events, or activated periodically	soft deadlines
Lee et al. [11], control system	pulled from data sources; persistently stored	various functions	possibly durable	activated periodically	soft deadlines
Lee et al. [11], remote monitoring system	pulled from data sources; persistently stored	various functions	possibly durable	activated periodically	soft deadlines

Iftikhar [3]	pulled from data sources; persistently stored; stored for a particular period; possibly shared by other processes	various functions	durable; stored for a particular period; possibly shared by other processes	activated periodically	soft deadlines
DataDepot [34]	pulled from data sources; not persistently stored; possibly shared by other processes; real-time	various functions	durable; real-time	activated by events	deadlines depending on the application

monitoring where large volumes of heterogeneous data streams are processed with different time constraints. Arriving stream data instances, as well as non-stream data, are stored persistently in the system. Aggregation can be triggered by ad hoc queries, or triggered periodically by continuous queries. The aggregate results are stored persistently in materialized views. Aggregate functions are computed incrementally, by combining the newly arrived instance with cached aggregated results.

Bür et al. describe an online active control system for aircrafts which employs data aggregation [10]. In this application, real-time data are gathered periodically from sensors deployed in the aircraft, and aggregated periodically. Since the aircraft system is time-critical, the freshness of data and timely processing of aggregation are crucial.

Lee et al. propose an approach for aggregating data in an industrial manufacturing system [11]. Three types of aggregation are described, which are aggregation at device level, aggregation in control system, and aggregation in remote monitoring system. At device level, real-time raw data are produced by sensors and controllers, and are aggregated in the devices. The aggregation is triggered hourly, or by state changes in the device. The aggregation functions are simple calculations for hourly throughput, error count, etc. The aggregated values are sent to subscribing clients, namely the control system and the remote monitoring system. The control system receives the data from devices and store them into a database. Every hour, these data, together with other events, are aggregated to produce error times, throughput, etc. The remote monitoring system also stores the data from devices and performs aggregation. Delay could occur in aggregation in the remote system.

Iftikhar applies data aggregation on integration of data in farming systems [3]. Data are collected from different devices, and stored permanently in a rela-

tional database. A gradual granular data aggregation strategy is then applied on the stored data. Basically, older data should be aggregated in a coarse-grained granularity while newer data are aggregated in a finer granularity. For different granularities, aggregation is triggered in different periods. The aggregated results are kept in the database while the raw data are deleted to save space.

Golab et al. propose a tool called DataDepot for generating data warehouses from streaming data feeds [34], focusing on the real-time quality of the data. Raw data are modeled as tables, which are not persistent and have a freshness property. Raw data are generated from different sources, with various properties such as rate and freshness. Raw tables are aggregated and stored in persistent derived tables which must also be fresh. Updates in the raw tables are propagated to the derived tables.

6.4.3 Survey Results

More than 13,000 research works are indexed in the SCOPUS search engine using "data aggregation" as a search key for title, abstract and keywords in computer science and engineering. Although only a small proportion of related works are examined here, our survey covers a relevant set of systems and application domains, which exposes the common and variable characteristics of the raw data, aggregated data, the aggregate functions, as well as the entire data aggregation processes.

In Table 6.1, we summarize the previous review by listing characteristics of the DAP in the surveyed systems and applications. Clearly, each aggregation process must have raw data, an aggregation function and the aggregated results. However, other characteristics have shown great variety. For instance, aggregation processes prepare the raw data ready for aggregation, by different data acquisition schemes. In some applications the aggregation process needs to pull the raw data from the persistent storage of the data source. Therefore the designer of an aggregation process must take this interaction into consideration. In other applications, however, raw data are pushed by the data source, so fetching raw data is not the concern of the aggregation process. The aggregated data may be stored persistently in some scenarios and are expected to survive system failures, while in other scenarios they can only reside in the volatile memory. As one can see in Table 6.1, the consistency of the data may depend on the time in some DAP, while in others the data are static. A large variety of aggregate functions have been applied in aggregation processes, depending on the requirements of the particular application. The aggregation process itself may be scheduled periodically, or triggered by ad hoc events. In time-critical

systems, the aggregation processes have strict timeliness requirements, while in some analytical systems with large amount of data the delays of the aggregation processes are tolerable. To design an appropriate aggregation process, it follows that one must take these characteristics, as well as their nature (necessity, optionality, etc) and their cross-cutting constraints, into consideration. A designer could benefit from having a systematic representation of these characteristics to ease the design, as well as support for facilitating feasible choices of the involved characteristics.

6.5 Our Proposed Taxonomy

The survey presented in Section 6.4 has revealed a number of characteristics of aggregation processes, including the raw data, the aggregated data, the aggregate functions, as well as the triggering patterns and the timeliness of the processes. Some of these characteristics are common for all aggregation processes, while others are distinct from case to case. In this section we propose a taxonomy of data aggregation processes, as an ordered arrangement of features revealed by the survey. The taxonomy for these common and variable characteristics not only leads to a clear understanding of the aggregation process, but also lays a solid foundation for an eventual tool support for reasoning about the impact of different features on the design.

We choose feature diagram as the presentation of our taxonomy, mainly due to two reasons. First, features may be used to model both functional and extra-functional characteristics of systems. This allows us to capture cross-cutting aspects that have on multiple software modules related to different concerns. Second, the notation of feature diagrams is simple to construct, powerful to capture the common and variable characteristics of different data aggregation processes, and intuitive to provide an organizational view of the processes. The taxonomy is shown in Figure 6.6.

In the following subsections, these features are discussed in details with concrete examples. More precisely, the discussion is organized in order to reflect the logical separation of features. We explain Figure 6.6 from the top level features under “Aggregation Process”, and iterate through all sub-features in a depth-first way. The top level features include “Raw Data”, “Aggregate Function” and “Aggregated Data”, which are the main constituents of an aggregation process. Features that characterize the entire DAP are also top level features, including the “Triggering Pattern” of the process, and “Real-Time (P)”, which refers to the timeliness of the entire process.

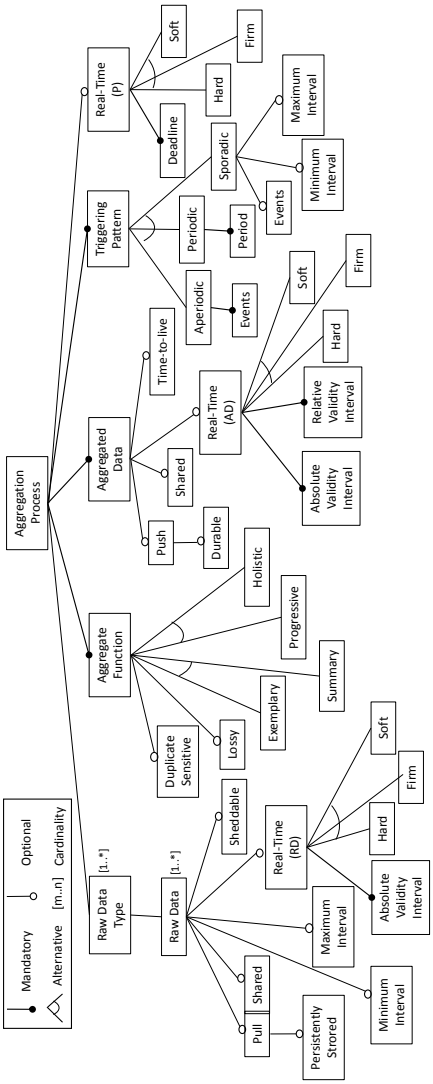


Figure 6.6: The taxonomy of data aggregation processes

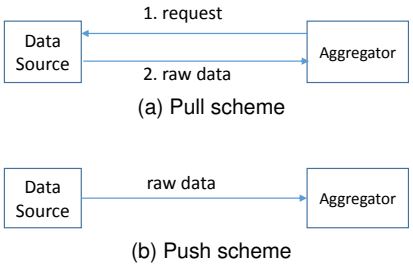


Figure 6.7: Raw data acqusition schemes

6.5.1 Raw Data

One of the mandatory features of real-time data aggregation is the raw data involved in the process. Raw data are the data provided by the DAP data sources. One DAP may involve one or more raw data. The multiplicity is reflected by the cardinality [1..*] next to the feature "Raw Data" in Figure 6.6. Each raw data may have a set of properties, which are interpreted as its sub-features and constitute a sub-tree. These sub-features are: Pull, Shared, Sheddable, and Real-Time.

Pull "Pull" is a data acquisition scheme for collecting raw data. Using this scheme, the aggregator actively acquires data from the data source, as illustrated in Figure 6.7a. For instance, a traditional DBMS adopts the pull scheme, in which raw data are acquired from disks using SQL queries and aggregated in the main memory.

"Pull" is considered to be an optional feature of raw data, since not every DAP pulls data actively from data source. If raw data have the "pull" feature, pulling raw data actively from the data source is a necessary part of the aggregation process, including the selection of data as well as the shipment of data from the data source. If the raw data do not have the "pull" feature, they are pushed into the aggregator (Figure 6.7b). In this case, in our view the action of pushing data is the responsibility of another process outside of the DAP. From DAP's perspective, the raw data are already prepared for aggregation.

"**Persistently Stored**" is considered as an optional sub-feature of "Pull", since raw data to be pulled from data source may be stored persistently in a non-volatile storage, such as a disk-based relational DBMS. The retrieval of persistent raw data involves locating the data in the storage and the necessary I/O.

Shared Raw data of some DAP examples in Section 6.4 are read or updated by other processes at the same time when they are read for aggregation [3, 27, 26]. The same raw data may be aggregated by several DAP, or accessed by processes that do not perform aggregations. We use the optional "shared" feature to represent the characteristic that the raw data involved in the aggregation may be shared by other processes in the system.

Sheddable We classify the raw data as "sheddable", which is an optional feature, used in cases when data can be skipped for the aggregation. For instance, in TAG [5], the inputs from sensors will be ignored by the aggregation process if the data arrive too late. In a stream processing system, new arrivals may be discarded when the system is overloaded [25]. For raw data without the sheddable feature, every instance of the raw data is crucial and has to be computed for aggregation.

Real-Time (RD) The raw data involved in some of the surveyed DAP have real-time constraints. Each data instance is associated with an arrival time, and is only valid if the elapsed time from its arrival time is less than its **absolute validity interval**. "Real-time" is therefore considered an optional feature of raw data, and "absolute validity interval" is a mandatory sub-feature of the "real-time" feature. We name the real-time feature of raw data as "Real-Time (RD)" in our taxonomy, for differentiating from the real-time features of the aggregated data ("Real-Time (AD)" in Section 6.5.3) and the process ("Real-Time (P)" in Section 6.5.5).

Raw data with real-time constraints are classified as "**hard**", "**firm**" or "**soft**" real-time, depending on the strictness with respect to temporal consistency. They are represented as alternative sub-features of the real-time feature. As we have explained in Section 6.3, hard real-time data (such as sensor data from a field device [11]) and firm real-time data (such as surveillance data [29]) must be guaranteed up-to-date, while outdated soft real-time data are still of some value and thus can be used (e.g., the derived data from a neighboring node in VigilNet [29]).

MINT Raw data may arrive continuously with a Minimum INter-arrival Time (MINT), of which a fixed arrival time is a special case. For instance, in the surveillance system VigilNet [29], a magnetometer sensor monitors the environment and pushes the latest data to the aggregator at a frequency of 32HZ,

implying a MINT of 32.15 milliseconds. We consider "MINT" an optional feature of the raw data.

6.5.2 Aggregate Function

An aggregation process must have an aggregate function to compute the aggregated result from raw data. An aggregate function exhibits a set of characteristics that we interpret as features.

Duplicate Sensitive "Duplicate sensitivity" has been introduced as a dimension by Madden et al. [5] and Fasolo et al. [4]. An aggregate function is duplicate sensitive, if an incorrect aggregated result is produced due to a duplicated raw data. For example, COUNT, which counts the number of raw data instances, is duplicate sensitive, since a duplicated instance will lead to a result one bigger than it should be. MIN, which returns the minimum value of a set of instances, is not duplicate sensitive because its result is not affected by a duplicated instance. "Duplicate sensitive" is considered as an optional feature of the aggregate function.

Exemplary or Summary According to Madden et.al [5], an aggregate function is either "exemplary" or "summary", which are represented as alternative features in our taxonomy. An exemplary aggregate function returns one or several representative values of the selected raw data, for instance, MIN, which returns the minimum as a representative value of a set of values. A summary aggregate function computes a result based on all selected raw data, for instance, COUNT, which computes the cardinality of a set of values .

Lossy An aggregate function is "lossy", if the raw data cannot be reconstructed from the aggregated data alone [4]. For example, SUM, which computes the summation of a set of raw data instances, is a lossy function, as one cannot reproduce the raw data instances from the aggregated summation value without any additional information. On the contrary, a function that concatenates raw data instances with a known delimiter is not lossy, since the raw data can be reconstructed by splitting the concatenation. Therefore, we introduce "lossy" as an optional feature of aggregate functions.

Holistic or Progressive Depending on whether the computation of aggregation can be decomposed into sub-aggregations, an aggregate function can

be classified as either "progressive" or "holistic". The computation of a progressive aggregate function can be decomposed into the computation of sub-aggregates. In order to compute the AVERAGE of ten data instances, for example, one can compute the AVERAGE values of the first five instances and the second five instances respectively, and then compute the AVERAGE of the whole set using these two values. The computation of a holistic aggregate function cannot be decomposed into sub-aggregations. An example of holistic aggregate function is MEDIAN, which finds the middle value from a sequence of sorted values. The correct MEDIAN value cannot be composed by, for example, the MEDIAN of the first half of the sequence together with the MEDIAN of the second half.

6.5.3 Aggregated Data

An aggregation process must produce one aggregated result, denoted as mandatory feature "Aggregate Data" in the feature diagram. Aggregated data may have a set of features, which are explained as follows.

Push In some survey DAP examples, sending aggregated data to another unit of the system is an activity of the aggregator immediately after the computation of aggregation. This is considered as an active step of the aggregation process, and is represented by the feature "push". For example, in the group layer aggregation of VigilNet [29], each node sends the aggregated data to its leading node actively. An aggregation process without the "push" feature leaves the aggregate results in the main memory, and it is other processes' responsibility to fetch the results.

The aggregated data may be "pushed" into permanent storage, such as in [32] and [11]. The stored aggregated data may be required to be durable, which means that the aggregated data must survive potential system failures. Therefore, "**durable**" is considered as an optional sub-feature of the "push" feature.

Shared Similar to raw data, the aggregated data has an optional "shared" feature too, to represent the characteristic of some of the surveyed DAP that the aggregated data may be shared by other concurrent processes in the system. For instance, the aggregated results of one process may serve as the raw data inputs of another aggregation process, creating a hierarchy of aggregation [25, 29]. The results of aggregation may also be accessed by a non-aggregation process, such as a control process [9].

Time-to-live The "time-to-live" feature regulates how long the aggregated data should be preserved in the aggregator. For instance, Aurora system [25] can be configured to guarantee that the aggregated data are available for other processes, such as an archiving process or another aggregate process, for a certain period of time. After this period, these data can be discarded or overwritten. We use the optional feature "time-to-live" to represent this characteristic.

Real-Time (AD) The aggregated data may be real-time, as required in some of the surveyed DAP, if the validity of the data instance depends on whether its temporal consistency constraints are met. Therefore the "real-time" feature, which is named "Real-Time (AD)", is an optional feature of aggregated data in our taxonomy. The temporal consistency constraints on real-time aggregated data include two aspects, the absolute validity and relative validity, as explained in Section 6.3. "**Absolute validity interval**" and "**relative validity interval**" are two mandatory sub-features of the "Real-Time (AD)" feature.

Similar to raw data, the real-time feature of aggregated data has "**hard**", "**firm**" and "**soft**" as alternative sub-features. If the aggregated data are required to be hard real-time, they have to be ensured temporal consistent in order to avoid catastrophic consequences [32]. Compared with hard real-time data, firm real-time aggregated data are useless if they are not temporal consistent [29], while soft real-time aggregated data can still be used with less value (e.g., the aggregation in the remote server [11]).

6.5.4 Triggering Pattern

"Triggering pattern" refers to how the DAP is activated, which is a mandatory feature. We consider three types of triggering patterns for the activation of DAP, represented by the alternative sub-features "**periodic**", "**sporadic**" and "**aperiodic**".

A periodic DAP is invoked according to a time schedule with a specified "Period". A sporadic DAP could be triggered by an external "event", or according to a time schedule, possibly with a "MinT" (Minimum inter-arrival Time) and/or "MaxT" (Maximum inter-arrival Time). An aperiodic DAP is activated by an external "event" without a constant period, MinT or MaxT. The event can be an aggregate command (e.g. an explicit aggregation query [28]) or a state change in the system [32].

6.5.5 Real-time (P)

Real-time applications, such as automotive systems [9] and industrial monitoring systems [11], require the data aggregation process to complete its work by a specified deadline. The process timeliness, named "**Real-Time (P)**", is considered as an optional feature of the DAP, and "**deadline**" is its mandatory sub-feature.

Aggregation processes may have different types of timeliness constraints, depending on the consequences of missing their deadlines. For a **soft** real-time DAP, a deadline miss will lead to a less valuable aggregated result [30]. For a **firm** real-time DAP [11], the aggregated result becomes useless if the deadline is missed. If a **hard** real-time DAP misses its deadline, the aggregated result is not only useless, but hazardous [10, 9]. "Hard", "firm" and "soft" are alternative sub-features of the timeliness feature.

We must emphasize the difference between timeliness ("Real-Time (P)") and real-time features of data ("Real-Time (RD)" and "Real-Time (AD)"), although both of them appear to be classified into hard, firm and soft real-time. Timeliness is a feature of the aggregation process, with respect to meeting its deadline. It specifies when the process must produce the aggregated data and release the system resources for other processes. As for real-time features of data, the validity intervals specify when the data become outdated, while the level of strictness with respect to temporal consistency decides whether outdated data could be used. To meet the desired real-time strictness level of the data, the DAP may need to meet certain timeliness requirements, which will be discussed further in Section 6.6.

6.6 Design Rules and Heuristics

In the previous section we have introduced our taxonomy that encompasses the important features of a DAP. In this section, we formulate a set of design rules and heuristics, following the design implications imposed by the features. The design rules are the axioms that should be applied during the design. Violating the rules will result in infeasible feature combinations. Design heuristics, on the other hand, suggest that certain mechanisms may be needed, either to implement the selected features, or to mitigate the impact of the selected features.

6.6.1 Design Rules

The real-time features of data and process are commonly desired features of DAP among real-time applications. Among these features there exist dependencies, which should be respected when one is selecting and combining these features. In this subsection we analyze the dependencies among the real-time data features (the "Real-Time (RD)" and "Real-Time (AD)" features in the taxonomy) and the timeliness feature (the "Real-Time (P)" feature) of the aggregation process itself. Based on the analysis we formulate three design rules to eliminate the infeasible combinations.

As already introduced, real-time data can be classified as hard, firm or soft real-time according to the strictness w.r.t. the temporal consistency. The hard real-time feature imposes strongest constraints and represents highest level of strictness, while the soft real-time feature represents the lowest level of strictness. From the raw data to the aggregated data, the level of strictness can only decrease or remain the same. This is because the validity of aggregated data depends on the validity of raw data. Since the hard real-time aggregated data have to be both absolute valid and relative valid, which requires all involved raw data to be absolute valid, the raw data have to be hard real-time too. If the raw data is soft real-time, which indicates that outdated raw data may occur, the temporal consistency of the aggregated data cannot be guaranteed. Therefore, we get the following rule:

Rule 1 The real-time strictness level of the raw data must be higher than or equal to the real-time strictness level of the aggregated data.

The timeliness of the entire data aggregation process has an impact on meeting the strictness level of the aggregated data, since the validity of the aggregated data depends on the interval between the time when raw data are collected, and the time when the aggregated data are produced. If the aggregated data are required to be hard real-time, the DAP also has to be hard real-time. If the timeliness of the DAP is soft, the calculation may miss its deadline and produce an outdated aggregated result. If we consider the "hard", "firm" and "soft" features of the DAP as levels of strictness w.r.t. timeliness, this rule is formulated as follows:

Rule 2 The strictness level w.r.t. timeliness of the entire DAP must be higher than or equal to the real-time strictness level of the aggregated data.

The fact that both the raw and aggregated data may be shared by multiple processes imposes further consideration on the real-time strictness of

the shared data. If the raw data or the aggregated data are shared by several processes, and the requirements of these processes impose different real-time strictness, then the real-time constraint of this data is in accordance with the highest strictness required by these processes. For example, the raw data of an aggregate process happens to be the input of a control process that demands the input to be hard real-time. Even though the aggregation process can tolerate outdated raw data, the real-time strictness level of the data must be hard. Otherwise the data for the control process may be outdated and lead to catastrophic consequences. Hence we formulate the following rule:

Rule 3 The real-time strictness of the raw/aggregated data must meet the highest real-time strictness level imposed by all processes that share the data.

These rules should be applied when the application designer analyzes the features derived from the requirement specification. Consider a process aggregating data from three sensors and providing its aggregated data to a hard real-time control process. The specification of the aggregation process may allow outdated raw data, i.e., soft real-time raw data, and tolerate occasional deadline miss. However, since the control process requires its inputs (the aggregated data) to be hard real-time, both the raw data from the sensors and the DAP have to be hard real-time as well.

6.6.2 Design Heuristics

Accomplishing the design of a DAP involves the design of appropriate supporting run-time mechanisms. These mechanisms either achieve the selected features of the DAP, or mitigate the impact of the selected features in order to ensure other properties of the system. Such properties could be, for instance, the logical data consistency characterized by the ACID properties of the processes. In this subsection we introduce a set of design heuristics, which are suggestions of mechanisms that could be implemented in order to enforce certain features and system properties. The heuristics are organized as suggested mechanisms as follows.

Synchronization for "pull" and "push" features Pulling raw data from a data source may involve locating the data source, selecting the data and shipping data into the aggregator. Pushing aggregated data may involve locating the receiver and transmitting the data. These activities introduce higher risks of delayed and missing data that may breach the temporal and logical data

consistency. Overheads in time and computation resource are also introduced, which are impacting factors of the overall timeliness of the process. When designing for such systems, one may consider developing a synchronization protocol to mitigate such impacts and ensure the consistency of the data.

Load shedding for "sheddable" feature combined with real-time features Situations could occur when the DAP is not able to meet the real-time constraints, due to, for example, system overload. If the raw data are sheddable, one may consider implementing the load shedding mechanism [25], which allows raw data instances to be discarded systematically.

Approximation for "sheddable" feature combined with real-time features An alternative mechanism for sheddable raw data is to implement approximation techniques. For example, Deshpande et al. introduce an approximation technique into sensor network to improve the efficiency of aggregation [35]. Instead of reading data from all sensors, the DAP only collects raw data from some of the sensors that fulfill a probabilistic model.

Concurrency control for "shared" feature An implication of shared data is the concern of logical data consistency, which is a common consideration from concurrent data access. A certain form of concurrency control needs to be implemented to achieve a desired level of consistency. For example, the aggregate process may achieve full isolation from other processes, i.e., they can only see the aggregated result when the DAP completes, using serializable concurrency control [36]. To improve performance or timeliness, one may choose a less stringent concurrency control that allows other processes to access the sub-aggregate results of the DAP, which may lead to a less accurate final result. Without any concurrency control, the aggregation process may produce incorrect results using inconsistent data [37].

Logging and recovery for "durable" feature In order to ensure the "durable" aggregated data, logging and backward failure recovery techniques, which are commonly used to achieve durability in data management systems, may be applied to the DAP. For example, the operations on the aggregated data are logged immediately, and the actual changes are written into the storage periodically.

Filtering for "duplicate sensitive" aggregate functions Using a duplicate sensitive aggregate function indicates a higher risk of inconsistency caused by

duplicated values sent to aggregator. A filtering mechanism may be implemented to identify the duplicates and filter them away.

Caching for "lossy" aggregate functions Lossy aggregate functions disallow the reconstruction of raw data from the aggregated data. However, raw data may be needed to redo all changes when errors occur, in order to ensure the atomicity of a process. A caching mechanism may be implemented for the DAP as a solution, that raw data instances are cached in the aggregator until the process completes.

Decomposition of aggregation for "progressive" aggregate functions The implication of using a progressive aggregate function is that one may decompose the entire aggregation into sub-aggregates. Computing the sub-aggregates in parallel may benefit the performance of the entire DAP. Another useful application of the decomposition is error handling, especially when it is combined with a caching mechanism. Consider an aggregate process fetching data from several sensors. The process can perform aggregation upon the arrival of each sensor data and cache the sub-aggregate result so far. If an error occurs during the fetching of next sensor, the process can return the cached sub-aggregate result as an approximation [5], or only restart the fetching of the failed sensor, instead of restarting the whole process.

Buffers for raw data and aggregated data Raw data arrive in the aggregator with their "MINT", which could be different from the aggregation interval imposed by the"triggering pattern" of the process. Buffers may be necessary to keep the raw data available for aggregation. Buffers may also be necessary for the aggregated data, since the aggregated data are generated according to the "triggering pattern", and must be available for a specified period defined by the "time-to-live" feature. Buffer management is crucial for the accuracy of aggregation as well as the resource utilization. For instance, circular buffer is a common mechanism in embedded systems for keeping data in limited memory. When the buffer is full, the program will just overwrite the old content with new data from the beginning. With the features presented in our taxonomy, one may calculate buffer size based on worst-case scenarios for non sheddable data, or suffice buffer size for sheddable data, given the size of each data instance.

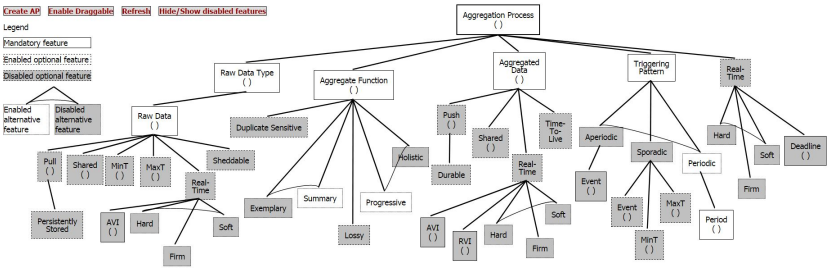


Figure 6.8: Interface of DAPComposer

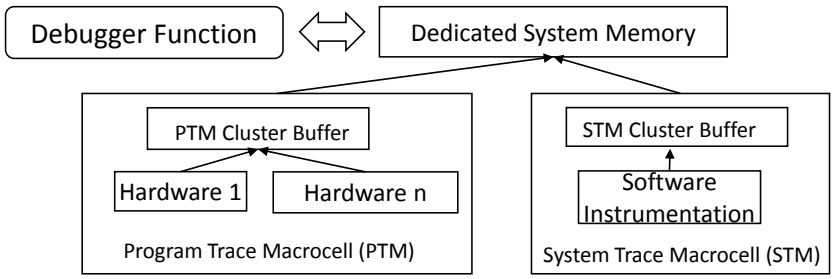


Figure 6.9: General architecture of the Hardware Assisted Trace system

6.7 Evaluation: an Industrial Case Study

In this section we evaluate the usefulness of our taxonomy in the design of a data aggregation application. Prior to the case study we have implemented a tool called DAPComposer (Data Aggregation Process Composer), shown in Figure 6.8. The tool provides a graphical user interface for designers to create DAP, by selecting and arranging the features in the diagram. Rules of mandatory, optional and alternative features are implemented. The mandatory features are always enabled, while optional and alternative features can be enabled/disabled by double-clicking the features. Annotations can be added to the selected features, such as the name of the data, or the actual value of the timing properties. It can also hide disabled features to provide a cleaner representation. Constraints can be typed as rules by users and saved in a rule base. The tool then validates the design against the specified rules. Although to the date only primitive constraints intrinsic to the feature model are checked by DAPComposer, we plan to mature the tool with more sophistic analysis capabilities, such as timing analysis, in the next version.

This evaluation is conducted on an industrial project, the Hardware Assisted Trace (HAT) [38] framework, together with its proposers from Ericsson. HAT, as shown in Figure 6.9, is a framework for debugging functional errors in an embedded system. In this framework, a debugger function runs in the same system as the debugged program, and collects both hardware and software runtime traces continuously. Together with the engineers we have analyzed the aggregation processes in their current design. At a lower level, a Program Trace Macrocell (PTM) aggregation process aggregates traces from hardware. These aggregated PTM traces, together with software instrumentation traces from the System Trace Macrocell (STM), are then aggregated by a higher level ApplicationTrace aggregation process, to create an informative trace for the debugged application.

We have analyzed the features of the PTM aggregation process and the ApplicationTrace aggregation process in HAT based on our taxonomy. The features of the PTM aggregation process are presented in Figure 6.10. Triggered by computing events, this process pulls raw data from the local buffer of the hardware, and aggregates them using an encoding function to form an aggregated trace into the PTM cluster buffer. The raw data are considered sheddable, since they are generated frequently, and each aggregation pulls only the data in the local buffer at the time of the triggering event. The aggregated PTM and STM traces then serve as part of the raw data of the ApplicationTrace aggregation process, which is shown in Figure 6.11. The ApplicationTrace process is triggered sporadically with a minimum inter-arrival time, and aggregates its raw data using an analytical function. The raw data of the ApplicationTrace should not be sheddable so that all aggregated traces are captured.

We have analyzed the features of the PTM aggregation process and the ApplicationTrace aggregation process in HAT based on our taxonomy. The features of the PTM aggregation process are presented in Figure 6.10. Triggered by computing events, this process pulls raw data from the local buffer of the hardware, and aggregates them using an encoding function to form an aggregated trace into the PTM cluster buffer. The raw data are considered sheddable, since they are generated frequently, and each aggregation pulls only the data in the local buffer at the time of the triggering event. The aggregated PTM and STM traces then serve as part of the raw data of the ApplicationTrace aggregation process, which is shown in Figure 6.11. The ApplicationTrace process is triggered sporadically with a minimum inter-arrival time, and aggregates its raw data using an analytical function. The raw data of the ApplicationTrace should not be sheddable so that all aggregated traces are captured.

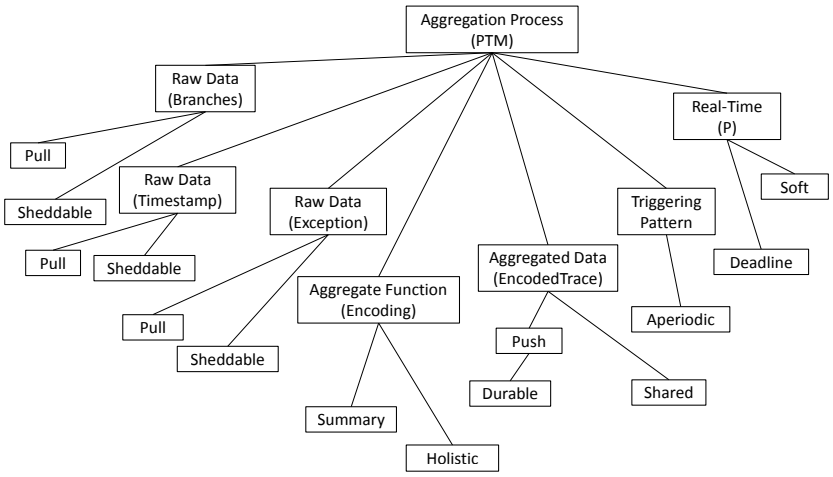


Figure 6.10: The aggregation process in the PTM

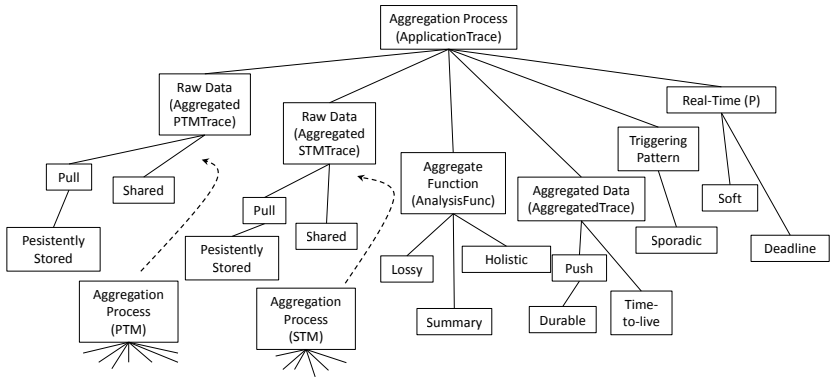


Figure 6.11: The aggregation processes in the investigated HAT system

6.7.1 Problem identified in the HAT design

With the diagrams showing the features of the aggregation processes, the engineers could immediately identify a problem in the PTM buffer management. In the current design, the buffer size is decided by both the hardware platform and the designer's experiences. The problem is that, the data in the buffer may

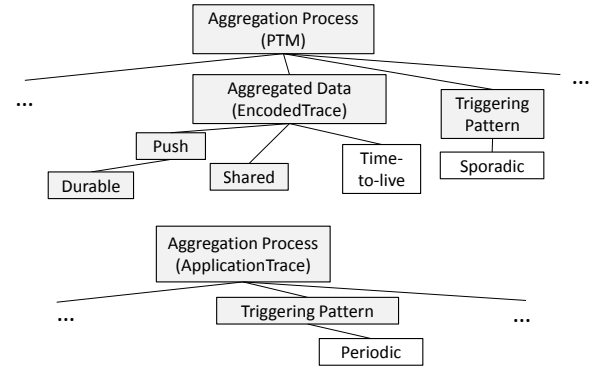
be overwritten before they are aggregated. This problem has been observed on Ericsson's implemented system, and awaits a solution. However, if the taxonomy would have been applied on the system design, this problem could have been identified before it was propagated to implementation.

This problem arises due to the lack of a holistic consideration on the PTM aggregation process and the ApplicationTrace aggregation process at design time. Triggered by aperiodic external events, the PTM process could produce a large number of traces within a short period and fill up the PTM buffer. The ApplicationTrace process, on the other hand, is triggered with a minimum inter-arrival time, and consumes the PTM traces as unsheddable raw data. When the inter-arrival time of the PTM triggering events is shorter than the MINT of the ApplicationTrace process, the PTM traces in the buffer may be overwritten before they could be aggregated by the ApplicationTrace process.

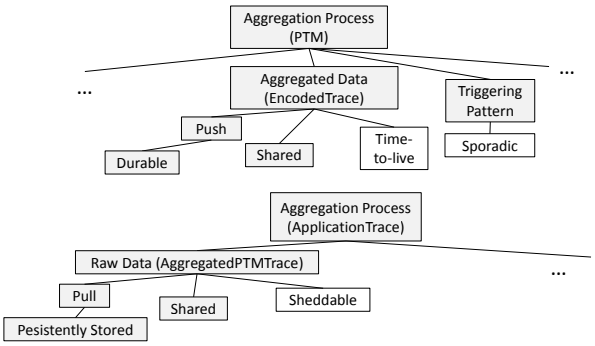
6.7.2 Solutions

Providing a larger buffer could be a choice to mitigate this problem. However, a larger provision might either still fail to meet the buffer consumption in some rare cases, or become a loss of resource due to pessimism. Considering the resource-constrained nature of the system, a better way is to derive the necessary buffer size at design time, given the size of each data entry. Based on our taxonomy, we and Ericsson engineers have come up with two alternative design solutions to fix this problem. Both solutions reuse most of the features in the current design.

Solution 1 To be able to derive the worst-case buffer size, one solution is to ensure more predictable behaviors of the aggregation processes, by adjusting the following features in the diagram (see Figure 6.12a): (i) Instead of selecting the "aperiodic" feature, the PTM process should select "sporadic", with a defined MINT; and, (ii) the "sporadic" feature of the ApplicationTrace process should be replaced by "periodic", so that the frequency of consuming the aggregated PTM traces can be determined. These changes of the features entail introducing extra real-time mechanisms into the current design, such as an admission control to ensure the MINT and a scheduler to schedule the processes. In addition, a "time-to-live" feature, whose value equals to the period of the ApplicationTrace process, should be added to the PTM process. To achieve this, a mechanism that prevents traces from being overwritten before the "time-to-live" value needs to be introduced. These new features allow the designer to analyze the worst-case production and consumption of the aggregated PTM



(a) Solution 1



(b) Solution 2

Figure 6.12: Illustration of Solution 1 and Solution 2. Unchanged features from the current design are marked in gray.

traces, and therefore derive the worst-case buffer size for the system using the actual values of the features. This solution also ensures that all PTM traces are aggregated by the ApplicationTrace process. Features reused from the current design are either marked in gray color, or omitted for readability.

Solution 2 An alternative to derive the exact buffer size is to allow overwriting in a controlled manner, as illustrated in Figure 6.12b. On one hand, as in Solution 1, we suggest to replace the "aperiodic" feature of the PTM process with "sporadic", so that the worst-case buffer size for PTM trace production

can be determined. On the other hand, the triggering pattern of the ApplicationTrace process remains unchanged ("sporadic"). However, a "sheddable" feature from the taxonomy is added to the raw data of the ApplicationTrace process, while a "time-to-live" is added to the PTM process. A shedding mechanism needs to be introduced, which in this case could be a logic in the buffer management that allows overwriting traces older than the "time-to-live" value. For instance, the designer may decide that the PTM traces produced 10 milliseconds ago are not valuable for the ApplicationTrace. When a PTM trace is older than 10 milliseconds, it might be overwritten even though it has not been aggregated. With the knowledge of the worst-case production of the PTM traces, and the "time-to-live" value of each trace, the designer is able to derive the needed buffer size.

Both solutions guarantee bounded buffers, while they require just a few features to be changed, and mechanisms introduced accordingly in the current design. Compared with Solution 2, which could lose traces, Solution 1 ensures all generated traces to be aggregated. However, to enforce a periodic triggering pattern as suggested in Solution 1, more efforts are required to provide real-time support, such as a real-time operating system.

6.7.3 Comparison with other taxonomies

Analysis based on other taxonomies could not easily identify the aforementioned bottleneck, since they characterize other aspects of data aggregation. The taxonomies proposed by Madden et al. [5] and Gray et al. [2] can only be applied to describe the aggregate functions of HAT, which are also captured by our taxonomy. The taxonomies of Solis et al. [7], Makhloufi et al. [6], and Rajagopalan [15], do not support modeling of data properties. Although Fasolo et al. [4] have considered data representations, aggregate functions and aggregation protocols, their taxonomy is defined at a much coarser level and does not allow for the analysis on such detailed data and process behaviors as the aforementioned bottleneck.

6.7.4 Summary

The engineers in the evaluation acknowledge that our taxonomy bridges the gap between the properties of data and the properties of the process, which has not been elaborated by other taxonomies. Our taxonomy enhances the understanding of the system by structuring the common and variable features of data aggregation processes. By applying analysis based on our taxonomy, design

flaws can be identified and fixed prior to implementation, which improves the quality of the system and saves money. Design solutions can be constructed by composing reusable features, and reasoned about based on the taxonomy, which contributes to a reduced design space. Due to these benefits, the engineers see great value in a potential design tool for data aggregation applications based on our taxonomy.

6.8 Discussion

The proposed taxonomy brings new lights on the understanding of the complexity of data aggregation in general. With a structured, feature-based organization, our taxonomy can be viewed as a common framework in which one can study the implications of the features, as well as dependencies between the features and the processes. Henceforth, one can reason about how the selection of one feature will influence other features, or even other processes. From an engineering perspective, as demonstrated by the HAT case study, applying our taxonomy to analysis can help designers identify flaws prior to implementation, and find new design solutions.

One direct usage of the dependencies between features is to eliminate infeasible feature combinations in the design. The rules regarding real-time strictness levels of data and process timeliness in Section 6.6 are one example. Although they appear straightforward and general, these rules regulate the qualitative relationships between real-time and timeliness features, and thus reduce the design space. For more accurate analysis, such as the derivation of buffer size in Section 6.7, values of quantitative features such as deadline and period should be involved into the calculation.

Conflicts may occur between real-time features and the other features. For instance, a DAP with durable aggregated data will ideally store each result into permanent storage, introducing frequent disk I/O which is time consuming and unpredictable. This may contradict the requirement of bounded computation time imposed by the timeliness feature. Such conflicts can be generalized as conflicts between logical data consistency, temporal data consistency and timeliness. Features such as “durable” and “shared” data are closely related to logical data consistency. The suggested mechanisms “logging and recovery” and “concurrency control” are common means to achieve durability and isolation respectively. They all have impacts on the temporal data consistency and timeliness. In such cases, a simple rule to detect potential conflicts is not possible. Neither is it possible to formulate a rule that resolves the conflicts that occur.

We believe the conflict detection, as well as the trade-offs among conflicting features, must come from careful analysis that relies on the selected features of a particular configuration with their values in the real case. Advanced analysis techniques, such as model-checking [39], can be applied on the configuration to verify whether the desired properties hold, and guide the trade-offs.

6.9 Conclusion

In this paper, we have investigated the characteristics of data aggregation processes in a variety of applications, and provided a taxonomy of the DAP, with a particular focus on the real-time properties. Our taxonomy is presented as a feature diagram, in which the common and variable characteristics are modeled as features. The taxonomy provides a comprehensive view of data aggregation processes for the designers, and allows the design of a DAP to be achieved via the selection of desired features and the combination of the selected features.

Based on the implications of the features in the taxonomy, we have introduced three rules that should be followed during the design of DAP. These rules eliminate some of the infeasible combinations of features during the design. We have also proposed a set of design heuristics, which help the designer to decide the necessary mechanisms for achieving the selected features and other system properties. The usefulness of the taxonomy has been demonstrated by an industrial case study. Flaws can be identified at design time, and solutions can be proposed at design level, by applying the taxonomy to the analysis.

Our taxonomy can be viewed as a framework for analyzing the dependencies between features and aggregation processes. For some potential conflicts among the desired features, we have highlighted that trade-offs must be decided based on careful analysis. More advanced analysis techniques are needed for reasoning about the conflicts among selected features, as well as the possible resolutions. In our future work, we plan to apply advanced analysis techniques, such as model-checking, to facilitate the trade-offs among features during the design of data aggregation processes.

Acknowledgment This work is funded by the Knowledge Foundation of Sweden (KK-stiftelsen) within the DAGGERS project.

Bibliography

[1] Imre J Rudas, Endre Pap, and János Fodor. Information aggregation in intelligent systems: An application oriented approach. *Knowledge-Based Systems*, 38:3–13, 2013.

[2] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[3] N. Iftikhar. Integration, aggregation and exchange of farming device data: A high level perspective. In *Proceedings of the 2nd International Conference on the Applications of Digital Information and Web Technologies*, pages 14–19, 2009.

[4] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *Wireless Communications, IEEE*, 14(2):70–87, 2007.

[5] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[6] Rafik Makhloufi, Guillaume Doyen, Grégory Bonnet, and Dominique Gaïti. A survey and performance evaluation of decentralized aggregation schemes for autonomic management. *International Journal of Network Management*, 24(6):469–498, 2014.

[7] Ignacio Solis and Katia Obraczka. In-network aggregation trade-offs for data collection in wireless sensor networks. *International Journal of Sensor Networks*, 1(3-4):200–212, 2006.

[8] Hani Alzaid, Ernest Foo, Juan Manuel Gonzlez Nieto, and DongGook Park. A taxonomy of secure data aggregation in wireless sensor networks. *International Journal of Communication Networks and Distributed Systems*, 8(1-2):101–148, 2012.

[9] G.R. Goud, N. Sharma, K. Ramamritham, and S. Malewar. Efficient real-time support for automotive applications: A case study. In *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 335–341, 2006.

[10] K. Bür, P. Omiyi, and Yang Yang. Wireless sensor and actuator networks: Enabling the nervous system of the active aircraft. *Communications Magazine, IEEE*, 48(7):118–125, 2010.

[11] Angélica Nieto Lee and José L Martínez Lastra. Data aggregation at field device level for industrial ambient monitoring using web services. In *Proceedings of the 9th IEEE International Conference on Industrial Informatics*, pages 491–496. IEEE, 2011.

[12] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[13] Radko Mesiar, Anna Kolesárová, Tomasa Calvo, and Magda Komorníková. A review of aggregation functions. In *Fuzzy Sets and Their Extensions: Representation, Aggregation and Models*, volume 220, pages 121–144. Springer Berlin Heidelberg, 2008.

[14] Jean-Luc Marichal. *Aggregation Functions for Decision Making*, pages 673–721. ISTE, 2010.

[15] R. Rajagopalan and P.K. Varshney. Data-aggregation techniques in sensor networks: A survey. *Communications Surveys Tutorials, IEEE*, 8(4):48–63, 2006.

[16] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[17] X. Song and J.W.S. Liu. How well can data temporal consistency be maintained? In *Proceedings of the 1992 IEEE Symposium on Computer-Aided Control System Design (CACSD)*, pages 275–284, 1992.

[18] Krzysztof Czarnecki and Eisenecker Ulrich. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[19] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, 2014.

[20] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[21] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992.

[22] Ines Fernando Vega Lopez, Richard T Snodgrass, and Bongki Moon. Spatiotemporal aggregate computation: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 17(2):271–286, 2005.

[23] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 318–329, 1996.

[24] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, 1997.

[25] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[26] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 623–634, 2006.

[27] Masafumi Oyamada, Hideyuki Kawashima, and Hiroyuki Kitagawa. Data stream processing with concurrency control. *SIGAPP Appl. Comput. Rev.*, 13(2):54–65, 2013.

[28] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[29] Tian He, Lin Gu, Liqian Luo, Ting Yan, J.A. Stankovic, and S.H. Son. An overview of data aggregation architecture for real-time tracking with sensor networks. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2006.

[30] Bruno Defude, Thierry Delot, Sergio Ilarri, Jose-Luis Zechinelli, and Nicolas Cenerario. Data aggregation in vanets: The vespa approach. In *Proceedings of the 5th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 13:1–13:6, 2008.

[31] B. Arai, G. Das, D. Gunopulos, and V. Kalogeraki. Approximating aggregation queries in peer-to-peer networks. In *Proceedings of the 22nd International Conference on*, pages 42–42, 2006.

[32] Jerry Baulier, Stephen Blott, Henry F. Korth, and Abraham Silberschatz. A database system for real-time event aggregation in telecommunication. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 680–684, 1998.

[33] A. Bar, P. Casas, L. Golab, and A. Finamore. Dbstream: An online aggregation, filtering and processing system for network traffic monitoring. In *Proceedings of the 2014 International Wireless Communications and Mobile Computing Conference*, pages 611–616, 2014.

[34] Lukasz Golab, Theodore Johnson, J. Spencer Seidel, and Vladislav Shkapenyuk. Stream warehousing with datadepot. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 847–854, 2009.

[35] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the 13th International Conference on Very Large Data Bases*, pages 588–599, 2004.

[36] Irina Botan, Peter M. Fischer, Donald Kossmann, and Nesime Tatbul. Transactional stream processing. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 204–215, 2012.

[37] Levent Gürgen, Claudia Roncancio, Cyril Labbé, and Vincent Olive. Transactional issues in sensor data management. In *Proceedings of the 3rd Workshop on Data Management for Sensor Networks*.

[38] Carlo Vitucci and Alf Larsson. Hat, hardware assisted trace: Performance oriented trace & debug system. In *Proceedings of 26th International Conference on Software & Systems Engineering and their Applications*, 2015.

[39] Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. Trading-off data consistency for timeliness in real-time database systems. In *Proceedings of 27th Euromicro Conference on Real-Time Systems*, pages 13–16, 2015.

Chapter 7

Paper B: A Formal Approach for Flexible Modeling and Analysis of Transaction Timeliness and Isolation

Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu.
In Proceedings of the 24th International Conference on Real-Time Networks and Systems, Brest, France, October 2016. ACM.

Abstract

Traditional Concurrency Control (CC) mechanisms ensure absence of undesired interference in transaction-based systems and enforce isolation. However, CC may introduce unpredictable delays that could lead to breached timeliness, which is unwanted for real-time transactions. To avoid deadline misses, some CC algorithms relax isolation in favor of timeliness, whereas others limit possible interleavings by leveraging real-time constraints and preserve isolation. Selecting an appropriate CC algorithm that can guarantee timeliness at an acceptable level of isolation thus becomes an essential concern for system designers. However, trading-off isolation for timeliness is not easy with existing analysis techniques in database and real-time communities. In this paper, we propose to use model checking of a timed automata model of the transaction system, in order to check the traded-off timeliness and isolation. Our solution provides modularization for the basic transactional constituents, which enables flexible modeling and composition of various candidate CC algorithms, and thus reduces the effort of selecting the appropriate CC algorithm.

7.1 Introduction

Many Real-time DataBase Management Systems (RTDBMSs) organize data-related computations as transactions in order to maintain logical data consistency, and guarantee the timeliness of the transactions, that is, meeting their deadlines [1]. As one of the cornerstones of transaction-based systems, Concurrency Control (CC) regulates the execution of concurrent transactions so that isolation, which means that transactions are not interfered by concurrent transactions [2], is achieved. While CC ensures the absence of undesired interference in a transactional computation, thus contributing to logical data consistency, it may cause unpredictable delays introduced by long blocking, arbitrary aborting and restarting, which could lead to deadline misses [3].

In an RTDBMS with hard real-time constraints, transaction timeliness is often the most crucial property and must be guaranteed. To avoid missing deadlines due to CC for full isolation (no interference from concurrent transactions at all), one general approach is to exploit CC algorithms enforcing lower levels of isolation that tolerate certain types of interferences [4, 5]. Consequently, unpredictable delays are reduced since transactions are less likely to be blocked or aborted, while a degraded (but acceptable) level of data consistency is provided due to the relaxed isolation [6]. An alternative is to leverage real-time constraints, such as priority and scheduling policy, in the CC [3, 7]. Such constraints enforce more restrictions on the possible interleavings, allow for more control over transaction executions and improve timeliness. Several solutions combine the relaxation of isolation with the enforcement of real-time constraints [5]. Given a priori knowledge of the worst-case execution times of individual data access operations, as well as the data access patterns, both timeliness and an acceptable level of isolation could be guaranteed, by selecting an appropriate CC algorithm for the particular transaction set at design time [8].

To achieve this, the system designer must be able to reason about both properties of the transaction set, under various CC algorithms, in order to decide the appropriate trade-offs. It is also highly beneficial to understand what data *inconsistency* could occur due to the relaxed isolation. Traditionally, the analyses of isolation and timeliness are separate tasks. In the database community, the analysis of isolation is based on analyzing dependencies exhibited in transaction execution history without incorporating timing [6]. In the real-time community, the common techniques are schedulability analysis [9] and experimental studies of different CC algorithms [10]. To our knowledge, less attention has been devoted to studying to which extent isolation can be achieved while timeliness is ensured.

In this paper, we propose an approach that facilitates the trade-off analysis of timeliness and isolation together with candidate CC algorithms via formal verification. Here, we choose one common type of CC, called Pessimistic Concurrency Control (PCC) [11], which utilizes locking techniques to prevent interferences. Nevertheless, the proposed approach can be applied to model other CCs, and integrated into the development of customized RTDBMSs.

Our approach relies on model checking transaction-based system models, against formalized timeliness and isolation together. The model of the transaction-based system is a network of timed automata representing the computational work to be executed, as well as automata representing the properties to be guaranteed. The resulting timed automata network is then verified with respect to both timeliness and isolation in UPPAAL [12]. To reduce the modeling efforts for various PCC algorithms, we propose a set of automata skeletons and parameterized patterns based on which the system models can be constructed modularly. Such skeletons model the basic structures of transactional constituents and common PCC algorithms, while the parameterized patterns model finer-grained recurring database operations, such as reads and writes. The modularization in our approach not only alleviates the complexity of the models, but also allows different PCC algorithms to be modeled and composed flexibly with the models of the computations.

The rest of the paper is organized as follows. In Section 7.2, we present the necessary background. In Section 7.3 we describe our approach for modeling transaction-based systems. Section 7.4 demonstrates the flexible adjustments for various PCC algorithms, followed by the verification results of the example models presented in Section 7.5. Section 7.6 compares our work with the related work, and in Section 7.7 we conclude the paper.

7.2 Preliminaries

In this section, we first briefly recall the concept of transaction, isolation and PCC algorithms, followed by UPPAAL timed automata.

7.2.1 The Concept of Transaction

A transaction is initially defined as a partially-ordered set of logically-related operations that, as a whole, ensure the so-called ACID properties [2], that is, Atomicity (a transaction either runs completely or rollbacks all changes), Consistency (a transaction executing by itself must not violate logical constraints),

Isolation (concurrent transactions do not interfere each other) and Durability (committed changes are made permanent). The partially-ordered set of operations without guarantees is called a *work unit* [13], which may include *read* operations that read data from the database, *write* operations that modify data in the database, and other *calculations* that do not interact with the database. We assume that interleaving among concurrently executing transaction can occur at any time except within read and write operations which we consider to be atomic. In hard real-time applications, in order to achieve predictability, the set of transactions is finite and the size of the database is bounded. In addition, the data access pattern of each transaction is known a priori.

Due to the semantic and performance limitations of the full ACID assurance, the original transaction model is sometimes adapted to relax the ACID properties [13], meaning that some of these properties are only partially ensured. For instance, isolation can be relaxed to allow certain types of interference [14]. In an RTDBMS for hard real-time applications timeliness must also be ensured, meaning that transactions must complete within their specified deadlines [4]. In this paper we only relax isolation, whereas atomicity, consistency, durability and timeliness are fully guaranteed.

A transaction is associated with transaction management primitives. The primitive *Begin* informs the transaction manager of the initiation of a transaction, whereas *Commit* and *Abort* indicate the transaction termination when all system resources possessed by the transaction are released. With full durability, when a transaction commits, the changes made by this transaction are saved permanently in the database, and become visible to other transactions. With full atomicity, when a transaction aborts, all changes it has made are undone. We consider two types of abort: *user abort* and *system abort*. User aborts are encoded in the logic of transactions by the designer, and are incurred by erroneous computational results at logical level. When a user abort occurs, the transaction is terminated. System aborts are triggered by the transaction manager when transactions are at such states that they cannot proceed. In this paper we only consider system aborts caused by concurrency control, for instance, due to conflicting data access.

Program 7.1 and 7.2 show two simple transactions T_0 and T_1 respectively. They consist of read and write operations on the same data (D_0 and D_1), which may lead to conflicts and expose the need for concurrency control. T_0 reads D_0, performs calculation, and writes D_1. In case any error occurs, T_0 will be aborted. T_1 simply updates D_0 and D_1. Both T_0 and T_1 have deadlines to meet, which are 11 and 4 time units respectively. In the following sections, we use r_i^j to denote that T_i reads D_j, w_i^j to denote that T_i writes D_j, c_i to denote

the commit of T_i, and a_i to denote the abort of T_i.

Program 7.1: Transaction T_0

```
Begin
read D0
calculate
if error, Abort
write D1
Commit
```

Program 7.2: Transaction T_1

```
Begin
write D0
if error, Abort
write D1
Commit
```

7.2.2 Isolation

Isolation refers to the property that the execution of one transaction is not interfered by other transactions executing concurrently [6]. Since full isolation may degrade performance and is not always necessary, the relaxation of isolation has been introduced by both industry and academia. Most commercial DBMSs support the *isolation levels* defined by ANSI/ISO SQL92 standard [14], which are SERIALIZABILITY (the most strict isolation), REPEATABLE READS, READ COMMITTED, and READ UNCOMMITTED (the most relaxed isolation).

Generalized by Adya et al. [6], the isolation levels are defined using the concept of *phenomenon*, which is a transaction execution that can lead to inconsistent data. For instance, the following execution involving T_0 and T_1 is considered as a phenomenon: $< r_0^0, w_1^0, w_1^1, w_0^1 >$, representing the execution "T_0 reads D_0, T_1 writes D_0, T_1 writes D_1, T_0 writes D_0". In this execution, T_0 writes D_1 based on the value it reads from D_0, without knowing that T_1 has updated D_0. The consequence is that the value of D_1 is not consistent with D_0.

An isolation level can then be defined as the property of avoiding a particular subset of the mentioned phenomena. For example, Adya et al. [6] classify phenomena into the following types: G0, G1 (including subtypes G1a, G1b, G1c), and G2. The SERIALIZABLE level precludes all the aforementioned phenomena, whereas READ COMMITTED only precludes G0 and G1. Since $< r_0^0, w_1^0, w_1^1, w_0^1 >$ is classified as a G2 phenomenon, the transaction system facilitating this execution violates SERIALIZABLE isolation. For more details about isolation, we refer to the literature [14, 6].

7.2.3 Pessimistic Concurrency Control (PCC)

A transaction-based system ensures isolation of concurrent transactions accessing the same data via concurrency control. We focus on one type of concurrency control, Pessimistic Concurrency Control (PCC), which is commonly applied in modern database systems [11]. PCC employs locking techniques to prevent interferences from concurrent transactions. A transaction needs to acquire a lock before it accesses the data, and release the lock after using the data. The CC manager receives locking and unlocking requests, and decides which transactions should be granted the lock, wait, or be aborted, according to the selected algorithm.

Among existing PCC algorithms, one popular algorithm to ensure isolation is *Two Phase Locking (2PL)* [11]. In 2PL, a transaction must acquire a write lock before writing to a data object, and must acquire a read lock or write lock before reading from a data object. If the data is already read locked, other transaction can still be granted read locks, but not write locks. If the data is write locked, no other transactions can be granted any lock. Transactions failing to acquire locks are added to a waiting queue. When a transaction unlocks data, both acquired read and write locks are released, and the next transaction in the waiting queue is granted the lock. Most importantly, a transaction is divided into two ordered phases, first a growing phase, and then a shrinking phase. In the growing phase the transaction can only require locks, whereas in the shrinking phase the transaction can only release locks. One widely applied variant of 2PL is rigorous 2PL, in which the shrinking phase occurs only when the transaction commits or aborts. As most 2PL algorithms, deadlocks may occur using rigorous 2PL.

Different PCC algorithms have been designed to rule out different types of interferences, thus achieving different levels of isolation, by for instance adjusting the locking time [15]. In the RTDBMS community, researchers have focused on leveraging real-time characteristics of transactions in CC, and have proposed a number of real-time PCC algorithms. One of the most widely applied real-time PCC is *Two Phase locking - High Priority (2PL-HP)* [7], which allows transactions with higher priority to lock a data that is already locked by another transaction with lower priority. Meanwhile, the transaction with lower priority gets aborted by the CC manager. Unlike rigorous 2PL, 2PL-HP is deadlock free.

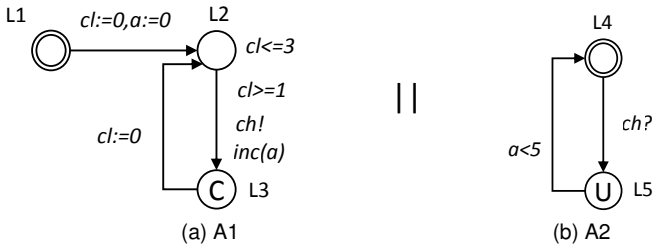


Figure 7.1: A network of timed automata

7.2.4 Timed Automata and UPPAAL

Timed Automata (TA), proposed by Alur and Dill [16], are finite-state automata extended with real-valued clock variables. UPPAAL is one of the most popular and mature verification tools based on timed automata, and extends the standard framework of TA with discrete variables as well as other modeling features [12]. We use UPPAAL TA in this paper, and introduce the syntax and semantics of UPPAAL TA via an example. To model a concurrent real-time system, several TA can be composed in an automata network. UPPAAL uses the Calculus of Communicating Systems (CCS) parallel composition operator ("||") [17] to build a network of TA. Using CCS, individual components are allowed to carry out internal actions (i.e., interleaving), while pairs of components can perform hand-shake synchronization.

Figure 7.1 shows a network of TA, composed of timed automata A1 and A2, which models a simple concurrent real-time system. In this system, automaton A1 sporadically increments the value of variable *a* and synchronizes with automaton A2. A timed automaton consists of a finite set of locations connected by edges. A1 consists of locations L1, L2 and L3, out of which L1 is the initial location. A clock variable *cl* is defined in A1 to measure the elapse of time, and progresses continuously. A discrete variable *a* is defined globally, and shared by A1 and A2. The semantics of a timed automaton is defined as a timed transition system where a state consists of the current location and the current values of clocks. There are two types of transitions: (i) delay (e.g. A1 may delay in L2 as long as cl $<=$ 3), and (ii) action/discrete transitions (e.g. L1 to L2 transition). At each location, if possible, A1 may non-deterministically delay at the location, or take a transition along an edge. A location may have an **invariant**, which is a conjunction of clock constraints. The TA must leave the location before the invariant is violated. In Figure 8.1a, A1 may stay at L2 until the value of *cl* reaches 3. Each edge may have a **guard**,

an **action** and an **assignment**. A guard is a finite conjunction of constraints on discrete variables or clock variables. A transition can be taken, only if the corresponding edge guard is satisfied. An action is the synchronization with other automata via a **channel**. Binary channels are used to synchronize one sender with a single receiver. An exclamation mark "!" following the channel name denotes the sending automaton, and a question mark "?" denotes the receiver. If the expected sender or receiver is not ready, the other automaton receiving or sending the message blocks. A broadcast channel is used between one sender and an arbitrary number of receivers. The sender does not block no matter how many receivers are ready. In Figure 7.1, A1 and A2 synchronize via a binary channel *ch*. An assignment resets the clock or sets a discrete variable when a transition is taken. Guards and assignments can be user-defined functions. In our example, when A1 moves from L2 to L3, the value of *a* is incremented by the function *inc(a)*.

A location can be **urgent** or **committed**. When an automaton reaches an urgent location, marked as "U", it must take the next transition without any delay in time. Another automaton may take transitions at the time, as long as the time does not progress. In our example, L5 is an urgent location. A committed location, marked as "C", indicates that no delay occurs on this location and the following transitions from this location will be taken immediately. When an automaton is at a committed location, another automaton may NOT take any transitions, unless it is also at a committed location. L3 is a committed location.

The UPPAAL model checker relies on a decidable subset of (Timed) Computation Tree Logic ((T)CTL) [18] as the specification language of properties, and supports verification of liveness and safety properties [12]. For example, one can specify the safety property "A1 never reaches location L3" as "$A[\,]\ not\ A1.L3$", in which "A" is a path operator and reads "for all paths", whereas "$[\,]$" is the "always" temporal operator. If a safety property is not satisfied, the model checker will provide a counterexample. Readers can refer to literature [12] for more information about UPPAAL.

7.3 Our Approach

In this section, we show how to model concurrent transactions under pessimistic concurrency control based on skeletons and patterns, aiming for the verification of timeliness and isolation.

Before explaining our modeling approach, we clarify the following as-

sumption. First, we assume that a hard RTDBMS runs on a single-processor platform, and comprises a finite set of transactions and data. Since the operations of each transaction are known a priori, the Worst-Case/Best-Case Execution Times (WCET/BCET) of the operations can be, either measured from legacy systems, or derived by analyzing the code, using established analysis methods [19]. Second, all read and write operations are atomic and cannot be preempted by other transactions. Third, since the time for a read/write operation and for waiting for a lock is significantly longer, the time spent on the lock resolution by the CCManager and the locking/unlocking actions is negligible. We also assume full durability and atomicity, which means that the changes of committed transactions are made permanent, while the changes of aborted transactions are undone by a rollback mechanism. The assumed rollback mechanism simply writes back the old values. One may however extend the models for more complex mechanisms.

We consider a real-time transaction-based system as a composition of a finite set of work units and the Concurrency Control Manager (CCManager). The system is modeled as a network of timed automata, in which each work unit and the CCManager are modeled as timed automata respectively. We introduce a set of IsolationOservers, which are automata that can be composed into the automata network and monitor the phenomena. Formally, we define a real-time transaction-based system N_S as follows:

$$N_S ::= A_0 \,||\, ... \,||\, A_{n-1} \,||\, A_{CCManager} \,||\, O_0 \,||\, ... \,||\, O_{k-1},$$

where A_0, ..., A_{n-1} are the TA of work units of transactions T_0, ..., T_{n-1}, respectively. Since in this paper we specifically target analysis of isolation and assume full atomicity and durability, we only model the interaction between A_i and the CC manager, while atomicity/durability management is not modeled explicitly. Instead, we augment A_i with locations representing the atomicity and durability semantics. $A_{CCManager}$ is the timed automaton of the CCManager. O_0, ..., O_{k-1} are the TA of IsolationOservers introduced to capture the phenomena precluded by the desired isolation, respectively. When a transaction performs an operation, the observers update their states accordingly. N_S must satisfy the timeliness property, that is, all transactions must meet their deadlines. N_S must also satisfy the desired isolation, which should be verified that no disallowed phenomena are observed by the IsolationObservers.

Our approach comprises a set of timed automata skeletons for modeling the basic structures of A_i, $A_{CCManager}$, and O_i, as well as parameterized patterns for modeling the operations within the automata. The skeletons are supposed to be adjusted and extended with respect to the particular system adopting a

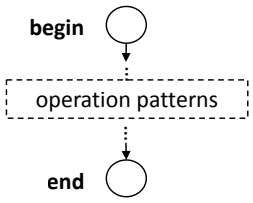


Figure 7.2: Timed automaton skeleton for a work unit

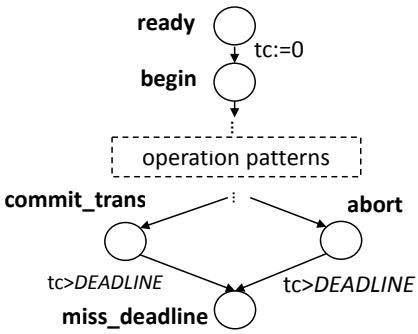


Figure 7.3: Work unit skeleton with atomicity and durability

selected PCC algorithm. The patterns, on the contrary, can be instantiated and reused as basic modeling units, to enrich the skeletons. The proposed skeletons and patterns are presented in the following subsections.

7.3.1 Work Unit Skeleton and Operation Patterns

Following its original definition, a work unit only consists of a set of operations. Figure 7.2 shows an automaton skeleton for a general work unit. The two locations *begin* and *end* represent the boundaries of the work unit. Within the boundaries, the operations, e.g., reads, writes and calculations, are modeled as patterns.

The work unit skeleton is augmented with the locations representing the committing and aborting mechanisms that achieve the assumed full atomicity and durability, as presented in Figure 7.3. In the future, if another variant of atomicity or durability is desired, the work unit should be augmented with the corresponding patterns. The model in Figure 7.3 comprises the location *begin*, while the *end* location in Figure 7.2 is replaced by *commit_trans* [1] and

[1] We do not use "commit" as the location name because "commit" is a reserved word in the

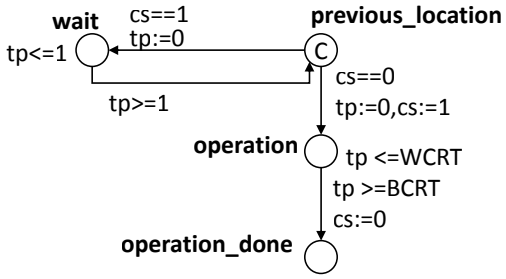


Figure 7.4: Read/write operation pattern

abort. The *ready* location models that the transaction is ready to be activated, before it actually begins. Since we are interested in verifying timeliness, a clock variable *tc* is defined, and reset once the transaction begins. When the transaction commits or aborts, the value of *tc* is compared with the transaction's *DEADLINE*, which is a positive integer value specified by the designer. If *tc* is greater than *DEADLINE*, the automaton will move to location *miss_deadline*. In the rest of this paper we use this work unit skeleton augmented with full atomicity and durability as the basic skeleton for work units.

Each operation is modeled as an instantiation of the "operation pattern". The atomic read/write operation pattern is defined in Figure 7.4. A clock variable *tp* is defined to measure the time spent on the operation. The variable *cs* stands for the critical section that models the CPU resource and ensures the atomic behavior. This pattern also contains locations *operation*, and *operation_done*, in which the word "operation" must be substituted by the actual operation, for instance "read_t1_d0" (representing r_1^0). When *cs* equals 1, meaning that the CPU is taken, the automaton moves to the *wait* location and waits for the CPU. For illustration, we simply let the automaton check whether the CPU is free every 1 time unit. This "polling" mechanism for CPU inquiry could be changed for more advanced scheduling, as we will present in Section 7.4. If *cs* equals 0, then the automaton obtains the CPU, sets *cs* to 1, and performs the operation. Before reaching location *operation_done*, which is the end of the operation, *cs* is set back to 0. *WCRT* and *BCRT* are parameters specified by the designer, representing the worst-case and best-case response time of the operation, respectively. The invariant $tp <= WCRT$ on location *operation* constrains that the execution of the operation takes at most WCRT time units. The guard $tp >= BCRT$ constrains that the execution takes at least *BCRT* time units. For atomic read and write operations, the WCRTs/BCRTs

UPPAAL tool.

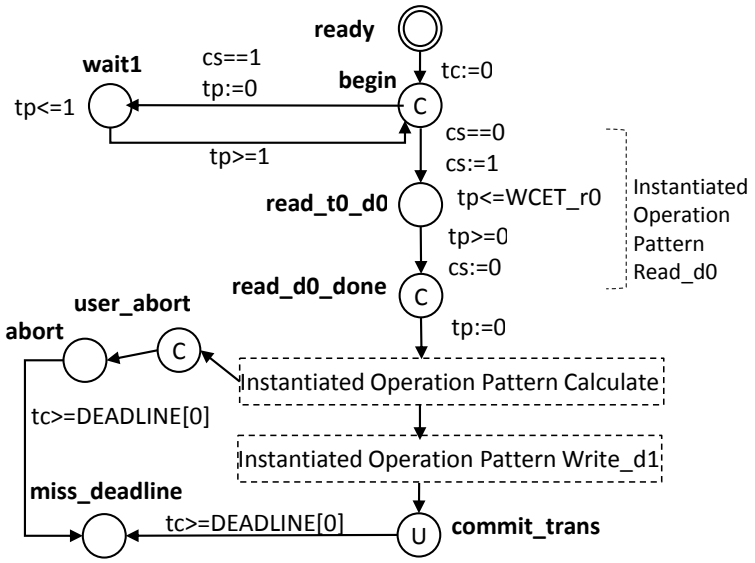


Figure 7.5: Work unit automaton of T_0

are equal to their WCETs/BCETs.

The pattern for non-atomic operations (calculations) is similar to the one for atomic operations, except that the value of *cs* is not updated during the operation. *WCRT*/*BCRT* could be derived by the designer based on the particular scheduling policy and the timing constraints, using for instance schedulability analysis techniques.

The instantiated operation patterns are composed into the skeleton to form the model of the work units. For example, the work unit of T_0 is modeled as shown in Figure 7.5. For readability, some *wait* locations are omitted. The instantiated operation patterns include **Read_d0**, **Calculate**, and **Write_d1**. These instantiated patterns, as we show in the remainder of the paper, can be reused to model a system of the same transactions but with different CC algorithms.

7.3.2 Concurrency Control Skeleton and Patterns

As explained in Section 7.2, PCC relies on locking/unlocking interactions between the CCManager and the transactions. We introduce the skeleton for the CCManager, as well as patterns to augment the work unit skeleton with CC

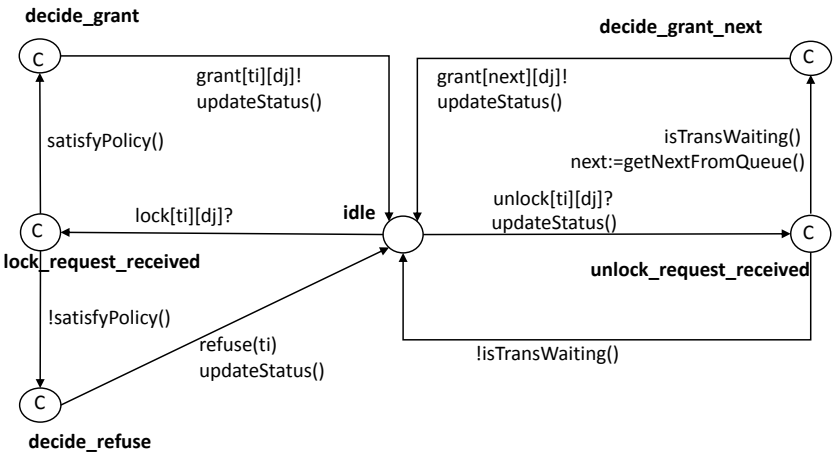


Figure 7.6: Automaton skeleton for a pessimistic concurrency control manager

related interactions.

The automaton skeleton for a PCC manager is shown in Figure 7.6. When the automaton receives a locking request via the channel *lock[ti][dj]?*, it takes the transition from the initial location *idle* to *lock_request_received*. A user-defined function that implements the resolution algorithm, *satisfyPolicy()* is defined, depending on the concurrency control algorithm, as a guard on the transitions from *lock_request_received*. Taking 2PL as an example, *satisfyPolicy()* evaluates to false if the data required by the transaction has already been locked by another transaction. If *satisfyPolicy()* returns true, the automaton moves to *decide_grant*. It then immediately sends the signal *grant[ti][dj]!* to transaction T_i, and updates the status of the transactions and the locks, using a user-defined function *updateStatus()*. If *satisfyPolicy()* returns false, the CCManager moves to *decide_deny*, and takes actions as implemented in function *deny()*, before it moves back to *idle*. Since the CCManager has the highest priority, and the time on lock resolution is negligible, all locations in this model are committed locations.

When receiving an unlocking request via *unlock[ti][dj]?*, CCManager updates the status, and moves to *unlock_request_received*. On the transitions from this location, the guards check if any transaction is waiting for locking the data, by a user-defined function *isTransWaiting()*. If this function returns true, the automaton sends a signal via *grant[next][dj]!*, to the next transaction obtained by the *getNextFromQueue()* function, and updates the status accordingly.

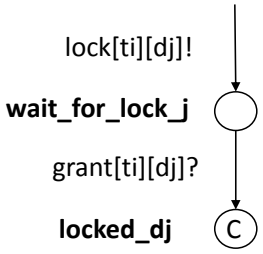


Figure 7.7: Locking pattern

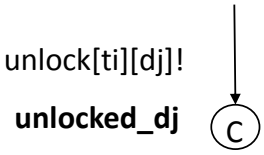


Figure 7.8: Unlocking pattern

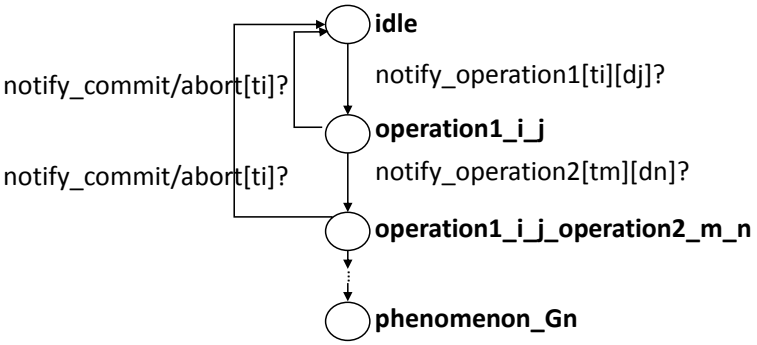


Figure 7.9: Automaton skeleton for an IsolationObserver

The work unit skeleton needs to be extended to model the interaction with the PCC manager. A locking pattern and an unlocking pattern are introduced in Figure 7.7 and Figure 7.8 respectively. After the transaction sends a message via *lock[ti][dj]!*, it waits at location *wait_for_lock_j*, until it receives the message *grant[ti][dj]?*. The patterns can be inserted into the work unit automata at particular positions depending on the selected PCC algorithm. For instance, using rigorous 2PL, write locks are released when the transaction commits or aborts. To model this PCC algorithm, the unlocking patterns for write locks must be inserted right before the *commit* or *abort* location in the transaction automaton. The model of rigorous 2PL is presented in the next subsection.

7.3.3 IsolationObserver Skeleton

To verify that a particular level of isolation is satisfied, we must verify the absence of the phenomena precluded by this level, in the presence of the underlying concurrency control algorithm. We introduce a set of IsolationObserver automata that capture the phenomena, by monitoring the operations performed

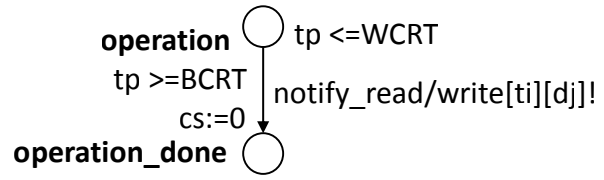


Figure 7.10: Operation pattern extended for IsolationObservers

by the work units.

The automaton skeleton for an IsolationObserver is described in Figure 7.9. The automaton starts from the *idle* location, and reaches the *phenomenon_Gn* location if the phenomenon is observed. Each location between *idle* and *phenomenon_Gn* is a subsequence of the operation sequence defining the phenomenon Gn. Without loss of generality, let us define Gn as the sequence $< op_i^j, op_m^n, ... >$. In Figure 7.9, via channel *notify_operation1[ti][dj]?*, the observer is notified when transaction T_i successfully completes op_i^j (read or write D_j), and transits to the location *operation1_i_j*. Subsequently, when T_m successfully completes op_m^n, the automaton moves from *operation1_i_j* to *operation1_i_j_operation2_m_n*. Since the observed phenomenon is started by an operation of transaction T_i, the end of T_i also means the end of the observation. Therefore, when T_i commits or aborts, the observer automaton gets a notification via channel *notify_commit/abort[ti]?*, and moves to location *idle*. Such iteration is repeated until the observer reaches *phenomenon_Gn*, indicating the existence of Gn, and thus the violation of the isolation level that precludes Gn.

Accordingly, the work unit skeleton and patterns introduced in the previous section must be extended to incorporate the notifications. In the work unit skeleton in Figure 7.3, *notify_commit[ti]!* and *notify_abort[ti]!* should be added to the edges leading to *commit_trans* and *abort* respectively. The operation pattern defined in Figure 7.4 is extended with *notify_read[ti][dj]!* or *notify_write[ti][dj]!* on the edge leading to *operation_done*, as shown in Figure 7.10.

7.3.4 Reference Algorithm: Rigorous 2PL

We illustrate our approach by modeling a concurrent transaction system implementing the rigorous 2PL algorithm presented in Section 7.2. Let us consider the two transactions in Programs 7.1 and 7.2. Assume the WCET of each read, write and calculation operation is 1 time unit. The BCETs are assumed to be 0 for simplicity.

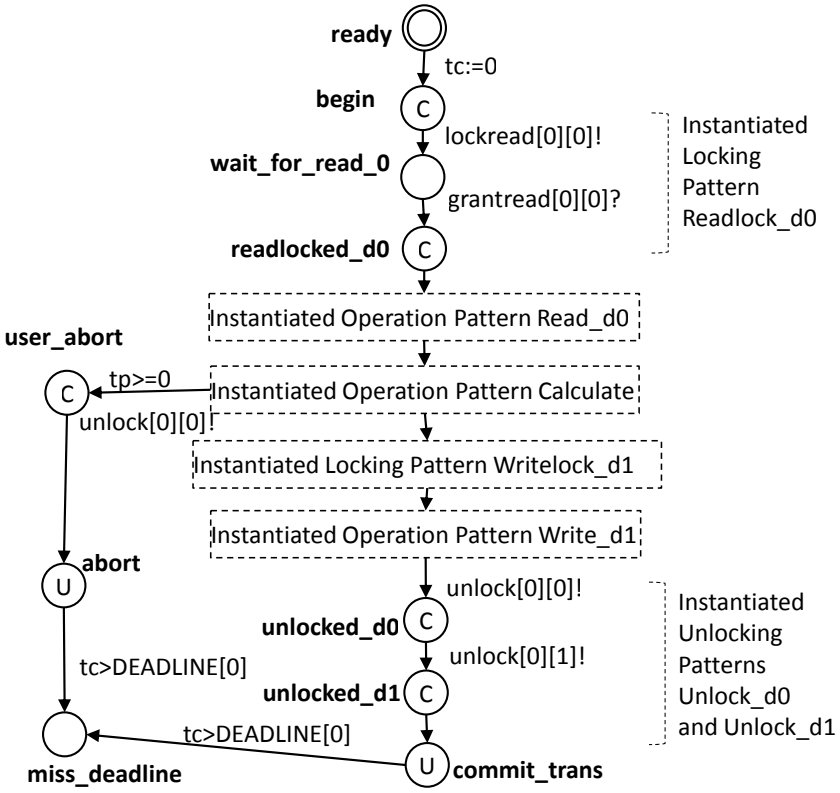


Figure 7.11: Rigorous 2PL: T_0 work unit automaton

Figure 7.11 shows the model of T_0 under rigorous 2PL. The work unit skeleton without CC contributes to the locations *ready*, *begin*, *commit_trans*, *abort*, *user_abort* and *miss_deadline*. The skeleton is enriched by the operations contained in the transaction, as well as the locking and unlocking patterns. As illustrated in Figure 7.11, the transaction first requires a read lock of D_0, which is modeled by an instantiated locking pattern **Readlock_d0**. The operation r_0^0 then follows, modeled by an instantiated operation pattern **Read_d0**. After r_0^0, the transaction performs calculation, acquires a write lock of D_0, and performs the operation w_0^1. At commit time, the transaction releases its locks using instantiated unlocking patterns. The instantiated operation patterns **Read_d0**, **Calculate** and **Write_d1** are reused from the model of T_0 without CC in Figure 7.5. The timed automaton of T_1 is modeled in a similar way.

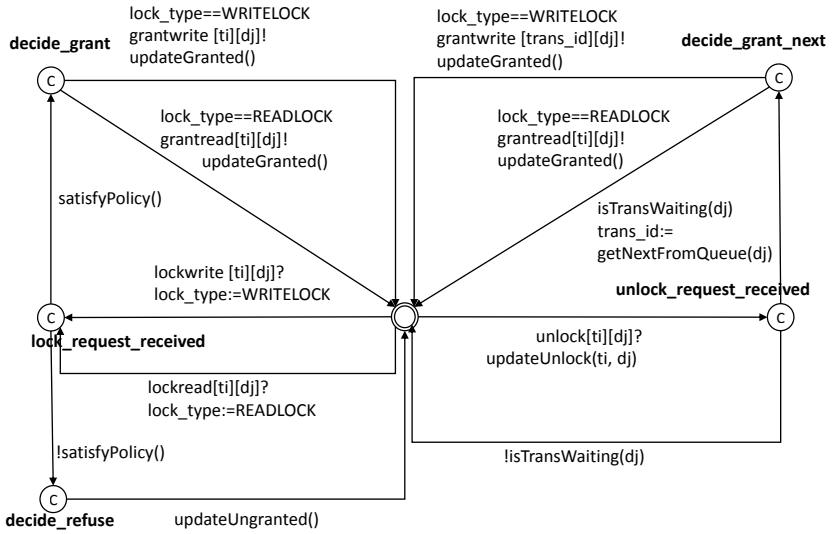


Figure 7.12: Rigorous 2PL: CCManager timed automaton

Figure 7.12 illustrates the timed automaton for the CCManager using rigorous 2PL algorithm. The PCC skeleton is extended to incorporate two types of locks, the read lock and the write lock. In this model, the user-defined functions implement the actual algorithm. The function *satisfyPolicy()* decides whether a transaction can be granted the required lock, based on the current status of the locked data. The functions *updateGranted()*, *updateNotgranted()* and *updateUnlock()* update the status of transactions and data after a locking, denying or unlocking action is taken, respectively. The function *isTranswaiting()* checks if any transaction is waiting due to unfulfilled lock requests. The function *getNextFromQueue()* fetches the next transaction in the waiting queue.

According to the definitions of phenomena in [6], phenomenon G1a can be described as: $< w_1^0, r_0^0, a_1 >$, representing the sequence that T_1 writes D_0, T_0 reads D_0, and then T_1 gets aborted. The IsolationObserver automaton for G1a is shown in Figure 7.13, which captures the sequence of operations of this phenomenon. The complete programs and UPPAAL models of T_0 and T_1, the CCManager and other IsolationObservers are included in our report [20].

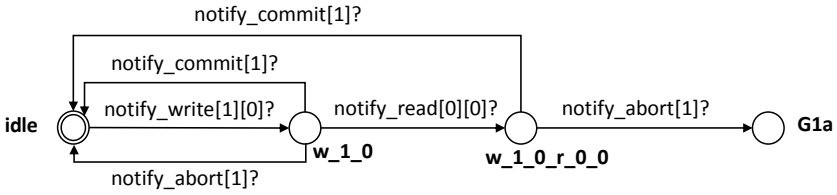


Figure 7.13: IsolationObserver for G1a

7.4 Adjustments for Various PCC

In this section, we demonstrate the flexibility of our approach by modeling two different PCC algorithms: one algorithm relaxing isolation, the other leveraging real-time constraints. The new models are constructed by adjusting and composing the skeletons and patterns from the models of the reference Rigorous 2PL.

7.4.1 Concurrency Control for Relaxed Isolation

One solution to guarantee timeliness under concurrency control is to relax the isolation level by adjusting the CC algorithm. By adjusting the instantiated patterns and composing them, our approach allows designers to easily build new models for other PCC algorithms from the existing rigorous 2PL models. In this subsection we show how to model PCC algorithms for relaxed isolation levels, through minor adjustments in the models of rigorous 2PL.

One type of adjustments for relaxing isolation is to adjust the points of locking and unlocking within the transaction control flow. By applying such adjustments one can develop different pessimistic concurrency control algorithms that achieve different isolation levels [15, 6]. For example, SERIALIZABLE can be achieved by exploiting long read locks and long write locks, as in rigorous 2PL. These locks are released when the transaction is committed. If the read locks are changed to have short duration, which means the read locks are released immediately after the read operation, a lower level of isolation such as READ COMMITTED could be achieved. The READ COMMITTED level can be further relaxed to READ UNCOMMITTED by, for instance, removing read locks entirely. This type of adjustment is easy to implement in our model. Since locking and unlocking are modeled as parameterized patterns composed with the work unit skeleton, one can move them to the desired locations to achieve different durations. The adjustments of locking types and durations are illustrated in Figure 7.14. The dashed rectangles represent the

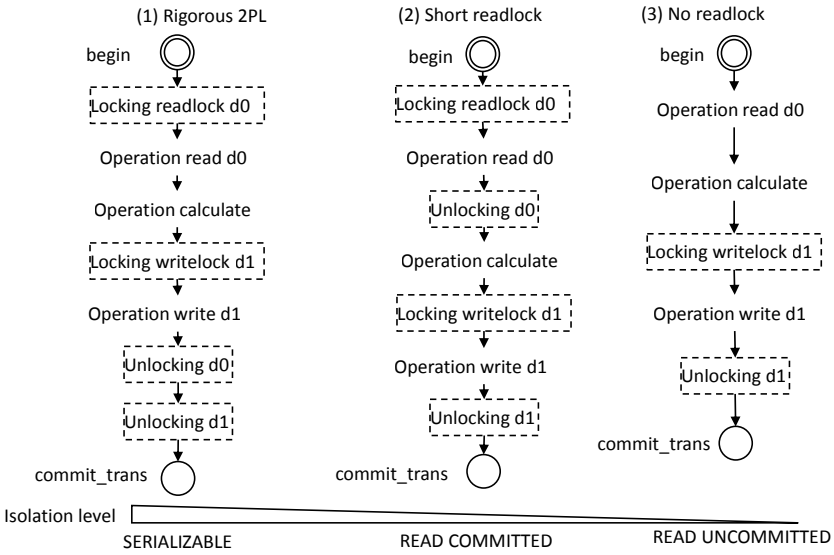


Figure 7.14: Adjusting rigorous 2PL for different isolation levels

instantiated locking and unlocking patterns in the automaton of T_0. The adjustments for different PCC algorithms can easily be accomplished by adding, removing, or moving around the locking and unlocking patterns.

The system of T_0 and T_1 exploiting short readlock CC algorithm can be easily adjusted from the Rigorous 2PL models. The automata for T_0 and T_1 are adjusted according to Figure 7.14. The CCManager automaton and the IsolationObservers are exactly the same as the ones in Section 7.3 for Rigorous 2PL.

7.4.2 Real-time Concurrency Control

As an alternative to relaxing isolation, researchers have worked for solutions to provide hard real-time guarantees while preserving a high level of isolation. A common approach is to identify the real-time characteristics of the system, such as the scheduling policy and priorities of the transactions, and leverage them in the CC algorithms. The reason behind such approaches is that the real-time constraints in the CC algorithms can restrict possible interleavings, which improves the timeliness of the transactions and thus the predictability of the system. In this section, we show that our approach is flexible for modeling real-time PCC algorithms. We first present how the automata skeletons and

patterns in Section 7.3 are extended for common real-time PCC algorithms. We then compose these instantiated skeletons and patterns to model a set of transactions under a popular real-time PCC algorithm called 2PL-HP [7], and verify the timeliness and isolation.

We assume that transactions are running on a real-time operating system and are scheduled according to a certain scheduling policy. For illustration we assume that the scheduling policy is the Fixed Priority Scheduling (FPS). However, the modeling approach can be easily adapted to other real-time scheduling policies. Under FPS, priorities of transactions are assigned at design time, and remain unchanged during execution. Transactions can be preempted by the ones with higher priorities, unless they are executing in "non-preemptive" regions, which are atomic operations in our case. We assume that the CCManager executes at the highest priority.

The real-time assumptions impose changes in the operation pattern (Figure 7.10) in Section 7.3, which assumes a simple "polling" scheme for inquiring CPU resource and no preemptions from other transactions. The adjusted atomic operation pattern is shown in Figure 7.15. We use the variable *cs* to model the critical section for atomic operations, and encode the scheduling policy (e.g. FPS) in the *sch()* function that decides which transaction should get the CPU. The flexibility of using the user-defined *sch()* function is that, if another scheduling policy is selected, the changes will be limited to the code of the *sch()* function without affecting the models. At the **previous_location**, which is right before the atomic operation, the transaction may be preempted, which is actually reflected by the result of *sch()*. The automaton can move to perform an operation, only if *sch()* returns its id *ti*. If not, this means that another transaction obtaining the CPU is either in its atomic operation, or has a higher priority than T_i. Then the automaton moves to the *wait* location, and waits for a *cpu_free* signal. The *cpu_free* signal is sent when a transaction finishes an atomic operation, gets blocked on locking requests, or terminates (commits or aborts).

The pattern for non-atomic operations (calculations) is similar to the one for atomic operations, except that the value of *cs* is not updated during the operation. *WCRT*/*BCRT* could be derived by the designer based on the particular scheduling policy and the timing constraints, using for instance schedulability analysis techniques.

The locking pattern (Figure 7.7) in Section 7.3 also needs to be modified in order to model real-time behaviors. The adjusted pattern is shown in Figure 7.16. When transaction T_i requires to lock D_j, it may get from the CCManager either a *grant[ti][dj]?* or a *wait[ti][dj]?* signal. The *wait[ti][dj]?* signal lets

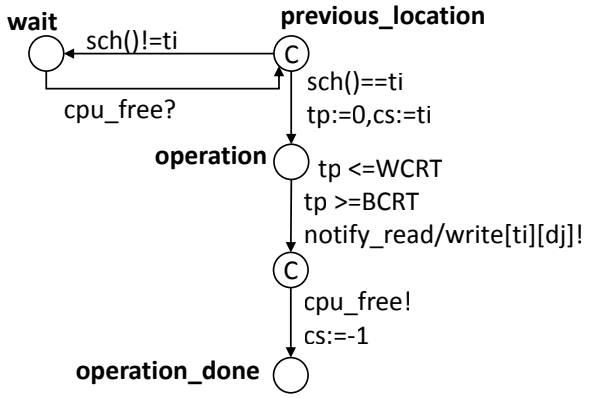


Figure 7.15: Operation pattern extended for real-time CC

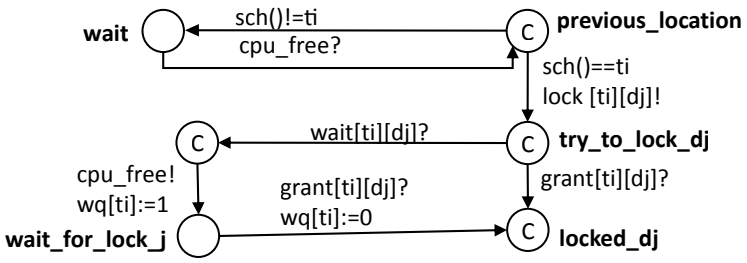


Figure 7.16: Locking pattern extended for real-time CC

the automaton to move to *wait_for_lock_j* and release the CPU via the channel *cpu_free!*. The variable *wq[ti]* keeps track whether T_i is blocked due to an unfulfilled lock request. When the automaton receives a *grant[ti][dj]?* signal from CCManager, it moves to *locked_dj*.

The work unit skeleton and the CCManager skeleton are the same as the ones in Section 7.3. These skeletons need to be adjusted according to the actual system, as shown by the 2PL-HP example in the next subsection. The isolation observers are constructed in the same way as described in Section 7.3.

We apply our approach to modeling a set of transactions under 2PL-HP [7], a widely applied CC algorithm in real-time database systems, by composing the aforementioned instantiated patterns. 2PL-HP allows transactions with higher priorities to lock the data that are already locked by transactions with lower priorities. The lockers with lower priorities are aborted by the CCManager. The aborted transactions are scheduled to be restarted, according to a

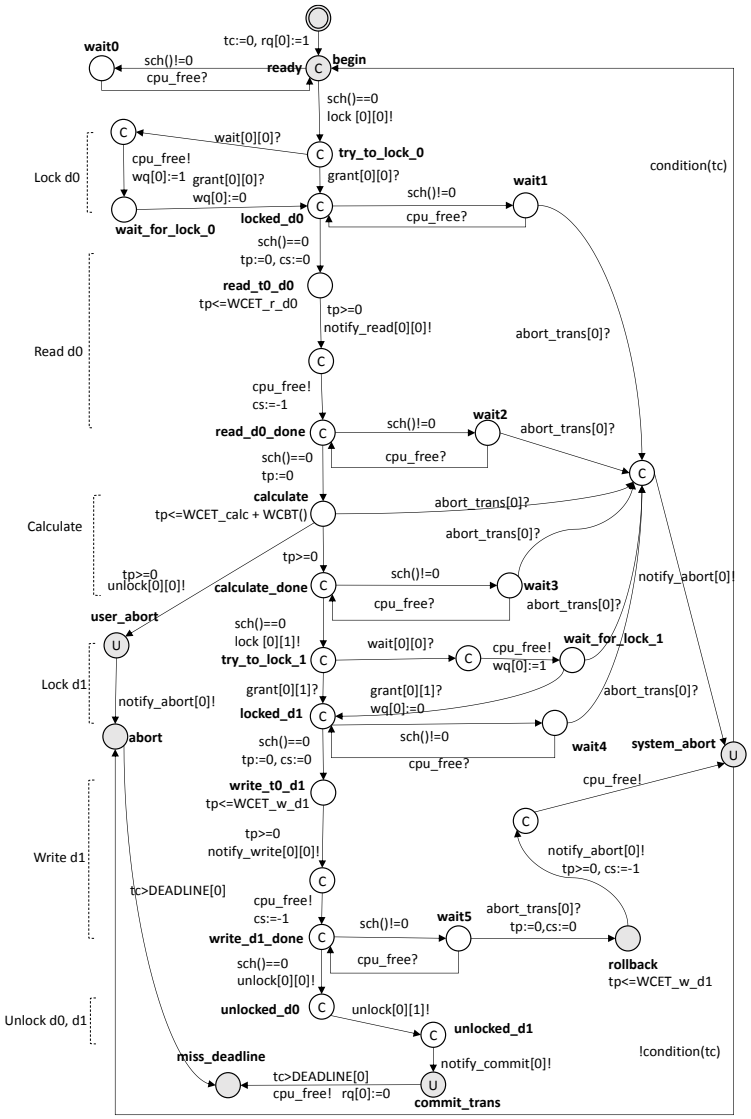


Figure 7.17: 2PL-HP: Work unit of T_0 with full atomicity and durability

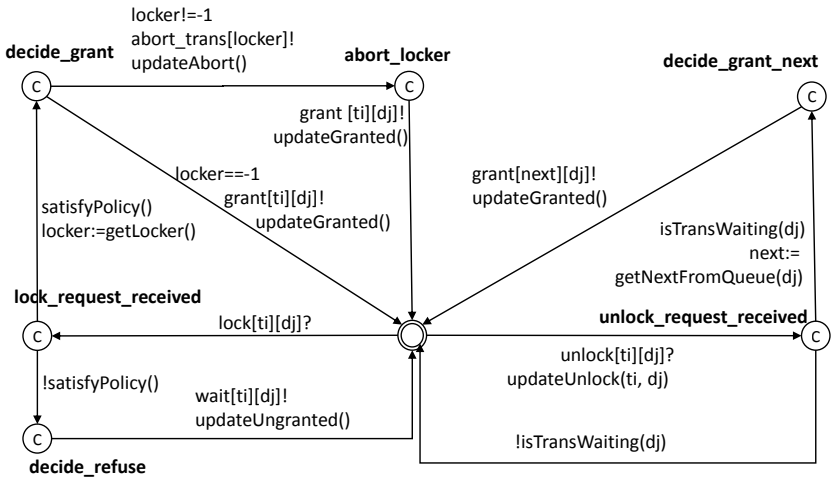


Figure 7.18: Automaton for the 2PL-HP CCManager

predefined criterion. In our case we assume that transactions are restarted if they have not missed their deadlines.

The work unit automaton for T_0 is shown in Figure 7.17. The variable *rq[0]* indicates whether T_0 is started. The locations in gray color are the ones from the work unit skeleton. Several differences exist in this model compared to the model under rigorous 2PL in Figure 7.11, addressing the specific aborting and restarting mechanism in 2PL-HP. This model comprises the *system_abort* location since transactions may be aborted by the CCManager. A *rollback* location models undoing the change in D_1 when T_0 gets aborted by CCManager after its write operation. The function *condition(tc)*, which could be adjusted for other criteria, compares the current time with the deadline. When the automaton arrives at the *system_abort* location, it continues to either *begin*, if *condition(tc)* returns true, or *abort* otherwise. Locking and the operations are instantiated based on the patterns in Figure 7.15 and Figure 7.16.

The CCManager of 2PL-HP is modeled as shown in Figure 7.18. In this model, the *satisfyPolicy()* function is implemented according to 2PL-HP, i.e., returning true if the data is either free, or locked by a transaction with lower priority. If *satisfyPolicy()* returns true, the locker, which is obtained by the *getLocker()* function, is aborted, while the lock is granted to the requester instead. Otherwise a *wait[ti][dj]!* signal is sent to the requester. When an unlocking signal is received, the CCManager picks the next transaction from the waiting queue using the *getNextFromQueue()* function, which also takes the priorities

into consideration.

7.5 Verification

We use UPPAAL model checker to verify timeliness and isolation. The timeliness property can be specified as a safety property in (T)CTL, requiring that the *miss_deadline* locations are not reachable. The timeliness of T_i can be specified as:

$$A[\,]\, not\, T0.miss\_deadline.$$

With the IsolationObservers composed into the models, we can verify that a transaction set achieve the desired isolation level under a certain CC algorithm, by proving the absence of phenomena precluded by the isolation level. For instance, to verify that T_0 and T_1 achieve SERIALIZABLE isolation, one must prove that none of the phenomena G0, G1 (G1a, G1b and G1c) and G2 could occur. This is equivalent to proving that the *phenomenon_Gn* location of IsolationObserver O_n is not reachable, which can be specified as:

$$A[\,]\, not\, On.phenomenon\_Gn.$$

Finding a complete set of the undesired phenomena is not the main concern of this paper. However, we believe that this process can be automated since the data access patterns of all transactions are known a priori. A naive approach is to enumerate all possible operation sequences for each pair of transactions that share data, and match the sequences with the phenomena in [6]. Although more efficient algorithms probably exist, we use this approach for our evaluation in the following sections.

Verification of rigorous 2PL We model check timeliness and SERIALIZABLE isolation of T_0 and T_1 under the reference algorithm Rigorous 2PL, which are modeled in Section 7.3. The verification shows that T_1 may miss its deadline under rigorous 2PL. From the trace we realize that T_1 is trying to lock D_1 before w_1^1, while D_1 is already locked by T_0 before r_1^1 until T_0 commits. Such long blocking time introduced by the concurrency control causes breaching the timeliness of T_1. In order to reduce the blocking times and meet all deadlines, we can either choose a less restrictive CC that achieves a lesser degree of isolation, or a real-time CC that leverages temporal constraints to enforce timeliness.

By definition [6], G0 is exhibited only if there exists a write-dependency loop between T_0 and T_1, which is not possible considering the operations of

Table 7.1: Verification results using the short readlock CC

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\, not\, T0.miss\_deadline$	0.001s	1179	Satisfied
S2	$A[]\, not\, T1.miss\_deadline$	0.001s	1179	Satisfied
S3	$A[]\, not$ $IsolationObserverG1a.G1a$	0.001s	1179	Satisfied
S4	$A[]\, not$ $IsolationObserverG2.G2$	0.001s	966	Not satisfied

these two transactions. Similarly, G1b and G1c do not occur. Therefore, in order to verify SERIALIZABLE isolation, we only need to prove the absence of G1a and G2, which can be described as $< w_1^0,\, r_0^0,\, a_1 >$, and $< r_0^0,\, w_1^0,\, w_1^1,\, w_0^1 >$, respectively. The verification shows that neither location *G1a* nor location *G2* are reachable, which means that the verified system achieves SERIALIZABLE isolation level.

Verification of short readlock algorithm As one solution to ensure timeliness, we apply the short readlock algorithm to relax isolation, and adjusted the models according to Section 7.4.1. The UPPAAL verification results are listed in Table 7.1. The results show that both transactions meet their deadlines. S3 is satisfied, indicating that phenomenon G1a does not occur, meaning that the modeled system reaches READ COMMITED isolation level. As expected, S4, which checks the absence of G2, is not satisfied, meaning that the system does not meet the SERIALIZABLE isolation level. The consequence is that the execution $< r_0^0,\, w_1^0,\, w_1^1,\, w_0^1 >$ degrades data consistency, because T_0 writes to D_1 based on the old value it reads from D_0, which has been changed by T_1.

Verification of 2PL-HP Our last example applies 2PL-HP to control the concurrency of a transaction set, consisting of T_0 and T_1 listed in Programs 7.1 and 7.2 in Section 7.2, as well as T_2 and T_3 in Programs 7.3 and 7.4. D_0 and D_1 are shared by T_0, T_1 and T_2, while D_2 is shared by T_2 and T_3. The deadlines for T_0, T_1, T_2 and T_3 are 11, 4, 22 and 13 time units respectively. The priorities are assigned, from highest to lowest, as follows: T_1, T_0, T_3, T_2. The phenomena possible to occur during the execution of the example transactions are listed in Table 7.2.

Table 7.2: Possible phenomena during execution

Phenomenon	Operation sequences
G1a	$< w_1^0, r_0^0, a_1 > < w_0^1, r_2^0, a_0 > < w_1^0, r_2^0, a_1 > < w_1^1, r_2^1, a_1 > < w_2^2, r_3^2, a_2 >$
G2	$< r_0^0, w_1^0, w_1^1, w_0^1 > < w_1^0, r_2^0, r_2^1, w_1^1 > < r_2^0, w_1^0, w_1^1, r_2^1 > < r_3^2, w_2^2, w_3^2 >$

Program 7.3: Transaction T_2

```
Begin
read D0
read D1
write D2
Commit
```

Program 7.4: Transaction T_3

```
Begin
read D2
write D2
Commit
```

The models are adjusted according to Section 7.4.2. The complete models are included in our report [20].

To verify SERIALIZABLE isolation, one needs to verify that none of the listed phenomena could actually occur. The verification results are listed in Table 7.3. Specification S1, S2, S3 and S4 encode timeliness of T_0, T_1, T_2, and T_3, respectively. S5 specifies that none of the locations indicating a phenomenon is reachable. All listed specifications are satisfied, which means that both timeliness and SERIALIZABLE isolation are guaranteed by 2PL-HP.

Discussion The authors of 2PL-HP have proved that the algorithm guarantees serializability [7], which validates the verification results regarding isolation.

We compare the model checking results of timeliness with the results of schedulability analysis. We argue that existing schedulability analysis techniques cannot be directly applied to analyze the schedulability of the transaction set. For instance, analysis of tasks in the Abort-and-Restart (AR) model assumes that higher priority tasks immediately abort lower priority tasks that are later restarted [21]. The transaction model in 2PL-HP is more complex. A transaction may be blocked by a lower priority transaction because of the atomic operations and rollback. A transaction may be aborted (and then restarted) by a higher priority transaction if they share the same data, or be preempted if they do not share data.

Since the considered transaction set consists of only four transactions, for the purpose of validating that the model-checked transactions are indeed schedulable, we analyze the worst case for each transaction manually, assuming each

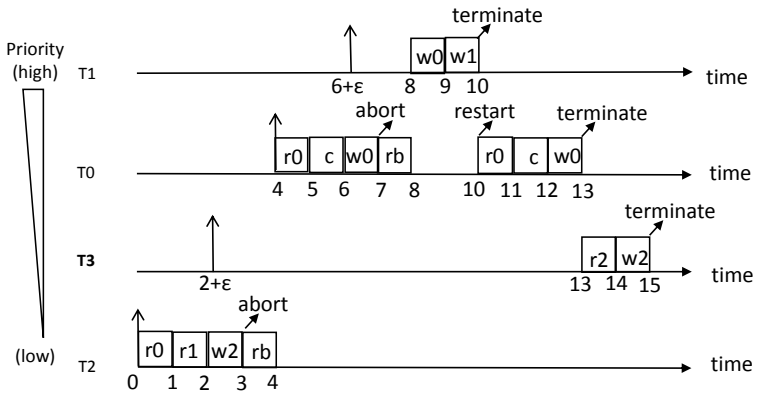


Figure 7.19: Worst case for T_3

transaction is modeled as a real-time task. As an example, we show the worst case for transaction T_3 in Figure 7.19. In this case, a lower priority transaction T_2 has read D_0 and D_1 (denoted as "r0" and "r1"), and started an atomic write operation on D_2 ("w2") at time 2. T_3 is activated at time $2 + \epsilon$, and tries to read D_2. T_2 is aborted due to conflicts, but before T_3 starts, T_2 must finish the atomic operation, and perform the rollback ("rb"). When the rollback is completed at time 4, T_0 is activated, which has a higher priority than T_3, and thus preempts T_3. However, before T_0 could complete its work, it gets aborted by T_1 at time 7, gets restarted at time 10, and terminates at time 13. T_3 is then allowed to execute, and terminates at time 15. The worst case response time of T_3 is therefore $13 - \epsilon$, smaller than its deadline 13. T_3 is indeed schedulable. Similar analysis shows that T_0, T_1 and T_2 can all meet their deadlines, whose worst case response times are 11, 4 and 22, respectively. Therefore, the model checking results with respect to timeliness are validated. For a larger transaction set under 2PL-HP, the validation via schedulability analysis should be automated, which is not trivial and out of our current scope.

Compared with schedulability analysis, our approach can perform more exact analysis for more complex transaction models. For instance, a variant of 2PL-HP conditionally aborts transactions based on their current time [7]. While existing schedulability analysis techniques can be applied but with large pessimism, our approach can easily model the conditional aborting behavior by extending the current models, and perform more exact analysis.

Table 7.3: Verification results using 2PL-HP

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\ not\ T0.miss\_deadline$	1.592s	161126	Satisfied
S2	$A[]\ not\ T1.miss\_deadline$	1.606s	161126	Satisfied
S3	$A[]\ not\ T2.miss\_deadline$	1.623s	161126	Satisfied
S4	$A[]\ not\ T3.miss\_deadline$	1.638s	161126	Satisfied
S5	$A[]\ not$ $(IsolationObserverT0T1G1a.G1a$ $or\ IsolationObserverT0T1G2.G2$ $or\ IsolationObserverT0T2G1a.G1a$ $or\ IsolationObserverT1T2G1a\_0.G1a$ $or\ IsolationObserverT1T2G1a\_1.G1a$ $or\ IsolationObserverT1T2G2\_0.G2$ $or\ IsolationObserverT1T2G2\_1.G2$ $or\ IsolationObserverT2T3G1a.G1a$ $or\ IsolationObserverT2T3G2.G2)$	1.669s	161126	Satisfied

7.6 Related Work

Several approaches for modeling and verifying real-time transactions exist already. Xiong et al. [22] propose the Real-Time ACTA framework to specify transactional properties, including isolation and timeliness, and verify the consistency of the specification. However, the models in Real-Time ACTA are conceptual and based on axioms, making it difficult to model different concurrency control algorithms, and reason about isolation relaxation. Our approach supports modeling of various CC algorithms, and reasoning about the low level behaviors of transactions under the assumed CC algorithm. Gallina et al. [23] propose a modular specification of advanced transaction models, and verification of isolation variants using the Alloy verification tool. The authors model neither the timing behaviors of transactions, nor the concrete CC algorithms. In the real-time community, schedulability analysis has been applied to analyze timeliness of transactions, such as Han et al. [9] and Wong et al.[21]. Such work, however, does not consider isolation and various concurrency control algorithms, and the analysis could be pessimistic for more complex transaction models.

Timed automata have been used to model real-time transactions. Lanotte et al. [24] propose a framework using timed automata for modeling long running transactions with timing constraints. The authors have also proposed automata patterns for different commit protocols. Their work, however, focuses on commit protocols and atomicity, instead of CC and isolation. Another related work is done by Kot [25], which models several real-time CC algorithms in UP-

PAAL. Despite the similarity in the modeled target, our work is different in several aspects. First, the authors show the feasibility of modeling transactions and CC in UPPAAL, while we strive to contribute to a general, reusable and flexible modeling approach for common CC mechanisms by using skeletons and patterns. Second, our models are designed for flexible verification of various levels of isolation relaxation, while their work checks more general properties, like absence of deadlocks, starvation, etc.

7.7 Conclusion

In this paper we have proposed an approach for modeling transaction based systems and verifying transaction timeliness and isolation in a unified framework based on UPPAAL timed automata. A set of automata skeletons and patterns have been proposed for modeling the work units of the transactions, the CC manager, and the observers for isolation. Timeliness and isolation are verified by model checking the automata network.

Such modularization not only reduces the complexity of the models, but also enhances the flexibility of our approach, in that different PCC algorithms can be easily modeled with a few adjustments, and composed into the automata network flexibly. Different PCC algorithms can be modeled by changing the points of locking and unlocking within transactions, adding new locking/unlocking patterns for other types of locks, or adjusting the user-defined functions such as *satisfyPolicy()*. Designers can then decide the appropriate PCC algorithm by analyzing the achieved isolation and timeliness, and trade off the two properties according to the application semantics. The decided PCC algorithm is guaranteed to meet transaction deadlines as well as a desired level of isolation.

Since our approach applies exhaustive model checking, verification of large transaction sets may not terminate due to state explosion. However, our approach can be applied to the down-scaled system focusing on the core transaction set. In our future work, we plan to mitigate state explosion by partitioning the transactions according to the data dependencies, and model check only transactions competing for the same data. For a large transaction set, we plan to apply bounded model checking, such as statistical model checking implemented in UPPAAL-SMC [26], which does not explore the entire state space. Although the verification result using statistical model checking does not offer the same degree of assurance as exact symbolic model checking, it provides a probabilistic guarantee with a certain degree of accuracy, and should suffice

for many soft real-time applications.

Besides addressing the state explosion problem, we are going to extend our models to support other types of CC, such as optimistic CC, and possibly automate the method by developing tool support. Our ultimate goal is to develop a formal framework capable of modeling a large spectrum of advanced real-time transaction models, in which trade-off analysis of all ACID properties could be carried out [8]. Such an endeavor would eventually facilitate the automation of customizing an RTDBMS.

Acknowledgment

This work is funded by the Knowledge Foundation of Sweden (KK-stiftelsen) within the DAGGERS project.

Bibliography

[1] Krithi Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1(2):199–226, 1993.

[2] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[3] A. Datta and S.H. Son. A study of concurrency control in real-time, active database systems. *IEEE Trans. Knowl. Data Eng.*, 14(3):465–484, 2002.

[4] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *Computer*, 32(6):29–36, 1999.

[5] L. B. C. DiPippo and V. F. Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of RTSS*, pages 87–96, 1993.

[6] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th ICDE*, pages 67–78, 2000.

[7] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, September 1992.

[8] S. Cai, B. Gallina, D. Nyström, and C. Seceleanu. Trading-off data consistency for timeliness in real-time database systems. In *27th ECRTS*, pages 13–16, 2015.

[9] S. Han, K. Lam, J. Wang, K. Ramamritham, and A.K. Mok. On co-scheduling of update and control transactions in real-time sensing and control systems: Algorithms, analysis, and performance. *IEEE Trans. Knowl. Data Eng.*, 25(10):2325–2342, 2013.

[10] X. Song and J.W.S. Liu. Maintaining temporal consistency: pessimistic vs. optimistic concurrency control. *IEEE Trans. Knowl. Data Eng.*, 7(5):786–796, 1995.

[11] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2004.

[12] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152, 1997.

[13] B. Gallina. *PRISMA: a software product line-oriented process for the requirements engineering of flexible transaction models*. PhD thesis, University of Luxembourg, 2010.

[14] ISO/IEC 9075:1992 Database Language SQL. Standard, International Organization for Standardization.

[15] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Readings in database systems. chapter Granularity of Locks and Degrees of Consistency in a Shared Data Base, pages 175–193. Morgan Kaufmann Publishers Inc., 1998.

[16] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[17] R. Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

[18] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[19] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, 2008.

[20] S. Cai, B. Gallina, D. Nyström, and C. Seceleanu. Flexible verification of transaction timeliness and isolation. `http://www.es.mdh.se/publications/4276-`, 2016.

[21] H. C. Wong and A. Burns. Schedulability analysis for the abort-and-restart (ar) model. In *Proceedings of 22nd RTNS*, pages 119:119–119:128, 2014.

[22] M. Xiong and K. Ramamritham. Specification and analysis of transactions in real-time active databases. In *Real-Time Database and Information Systems: Research Advances*, volume 420, pages 327–351. Springer, 1997.

[23] B. Gallina, N. Guelfi, and P. Kelsen. Towards an alloy formal model for flexible advanced transactional model development. In *Proceedings of the 33rd Annual IEEE Software Engineering Workshop*, pages 94–103, 2009.

[24] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina. Modeling long-running transactions with communicating hierarchical timed automata. In *Formal Methods for Open Object-Based Distributed Systems*, pages 108–122. Springer, 2006.

[25] M. Kot. Modeling selected real-time database concurrency control protocols in uppaal. *Innovations in Systems and Software Engineering*, 5(2):129–138, 2009.

[26] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, pages 80–96. Springer, 2011.

Chapter 8

Paper C: Towards the Verification of Temporal Data Consistency in Real-Time Data Management

Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu.
In Proceedings of the 2nd International Workshop on modeling, analysis and control of complex Cyber-Physical Systems, Vienna, Austria, April 2016. IEEE.

Abstract

Many Cyber-Physical Systems (CPSs) require both timeliness of computation and temporal consistency of their data. Therefore, when using real-time databases in a real-time CPS application, the Real-Time Database Management Systems (RTDBMSs) must ensure both transaction timeliness and temporal data consistency. RTDBMSs prevent unwanted interferences of concurrent transactions via concurrency control, which in turn has a significant impact on the timeliness and temporal consistency of data. Therefore it is important to verify, already at early design stages that these properties are not breached by the concurrency control. However, most often such early on guarantees of properties under concurrency control are missing. In this paper we show how to verify transaction timeliness and temporal data consistency using model checking. We model the transaction work units, the data and the concurrency control mechanism as a network of timed automata, and specify the properties in TCTL. The properties are then checked exhaustively and automatically using the UPPAAL model checker.

8.1 Introduction

In a Cyber-Physical System (CPS), the control of physical working units is decided by the computational operations based on timely monitored environmental data [1]. Many CPS applications are real-time systems, which means that the results of the computation must be not only logically correct, but also temporally correct [1]. The temporal correctness of a result depends both on the time when the result is produced, and on the temporal consistency of the data used for the computation. For instance, consider a robot arm picking up objects from the conveyor of an assembling line. In order to pick up the object correctly, the robot arm must adjust its rotation angle according to the position of the approaching object. A computer in the arm calculates the rotation angle, based on current angle of the arm, and the position of the target. The computational result is useless, if either the calculation misses its specified deadline, or the position data are outdated.

One common way of managing the temporal environmental data and computational results is to store them in a Real-Time Database (RTDB) [2]. The temporal consistency of the data requires that the states of the RTDB must be consistent with the corresponding environmental states timely [3]. Since computations on data are implemented as transactions in the database, the Real-Time Database Management System (RTDBMS) must therefore ensure both the transaction timeliness and the temporal data consistency [4]. However, it is not trivial to verify these properties, partly due to the concurrency control mechanisms used by RTDBMSs to eliminate unwanted interferences from concurrent transactions. Transactions may be blocked or aborted by the concurrency control manager, which on the one hand may lead to breached timeliness and temporal consistency, and on the other hand increases the complexity of the analysis. Some of existing work towards analysis of temporal consistency (e.g., Song et al. [3]) are based on simulation, and thus lack formal guarantees. Other work either provide analysis of temporal consistency without considering concurrency control [5], or focus on other properties of concurrent transaction systems such as isolation [6] or absence of deadlock [7].

In our recent work [6], we have proposed a formal approach based on timed automata [8], Temporal Computational Tree Logic (TCTL) [9] and UPPAAL [10] for verifying timeliness and isolation of transactions in a unified manner. Here, we develop our approach further, focusing on the tradeoff between timeliness and temporal data consistency instead. We consider the targeted system as a composition of the following constituents: the data accessed by either the

sensors or the computational units of the CPS, the transaction work units [11], which are the logical operations in the transactions, and the concurrency control manager that coordinates concurrent transactions. We first transform these constituent parts into a formal model, which is a network of timed automata. Then we specify the timeliness and temporal consistency in a logic formalism called TCTL, using a set of specification patterns. Finally, we use the UPPAAL model checker [10] to check whether these formalized properties are satisfied by the model. The approach is exemplified on a concrete example in detail.

The remaining part of the paper is organized as follows. Section 8.2 introduces the background of the paper, consisting of the concepts of temporal data consistency in RTDBMS, and the needed knowledge on timed automata and UPPAAL. In Section 8.3 we present the assumed CPS system with exemplary transactions and relevant requirements. In Section 8.4 we describe our modeling approach for transactions, data and the lock manager of the assumed system. The formal specification of the requirements, as well as the verification results, are presented in Section 8.5. We compare our work to the related work in Section 8.6, after which we conclude the paper in Section 8.7.

8.2 Background

In this section, we first recall the concepts of temporal data consistency in real-time databases, followed by a brief introduction of timed automata and the UPPAAL model checker.

8.2.1 Temporal Data Consistency

Data in an RTDBMS can be classified into base data and derived data. In real-time applications, which often monitor the environment states and react accordingly, base data are the representations of the environment states in the database. A typical example of base data is the readings from sensors that monitor the speed of the conveyor in our example CPS. Derived data are the results of computations based on a set of base data objects. For instance, a transaction takes the conveyor speed and the position of the robot arm as inputs to compute the rotation angle. The rotation angle is a derived data. Each data object is associated with a timestamp. For a base data object, the timestamp indicates the time when it is collected, whereas for a derived data object, it refers to the time when it is derived.

As mentioned in Section 8.1, RTDBMSs must guarantee the temporal data

consistency, which includes two aspects: the **absolute validity** and the **relative validity** [3] of data. Absolute validity refers to the property that the data must always reflect the environment timely. If we define the age of a data object as the difference between the current time and its timestamp, a base data object is absolute valid if the age of the data is smaller than a specified interval, called **absolute validity interval**. A derived data object is absolute valid if all participating data are absolute valid.

In order to compute a valid derived data, the set of base data may have to be collected close enough to each other in time. For instance, the conveyor speed and the position of the robot arm must be collected within 50 milliseconds. A set of data objects are relative valid, if the difference between the ages of every object is within a specified **relative validity interval**.

The original absolute validity requires data to be absolute valid all the time. This however imposes restrictions on the database performance and the timeliness of other transactions, since the data may need to be updated frequently, even though it is not accessed by any other transaction. Therefore, Kao et al. [12] propose the **weak absolute validity** as a relaxation, which requires that, the age of the data accessed by a transaction should be smaller than its absolute validity interval only when the transaction accesses it. Similarly, one can define the **weak relative validity**, which requires that the age differences of the base data should be within the relative validity interval when they are accessed by a transaction.

8.2.2 Timed Automata and UPPAAL

UPPAAL [10] is the state-of-art model checker for real-time systems, based on timed automata [8]. Basically, a system is modeled as a network (a parallel composition) of timed automata in UPPAAL. A timed automaton is a finite-state automaton extended with real-valued clock variables and discrete variables. In UPPAAL, clock variables progress synchronously. The locations of all automata, together with the values of clock variables, define the state of a system.

The action to be taken at one location can either be a delay at the same location, or a transition to another location following an edge. An invariant, which is a predicate (boolean set of states) over clock variables, may be associated with a location setting an upper-bound on the delay. A guard, which is a predicate of clock or discrete variables, may be associated with an edge as the required condition to take the underlying transition. During the transition, discrete variables can be updated, while clock variables can be reset. An au-

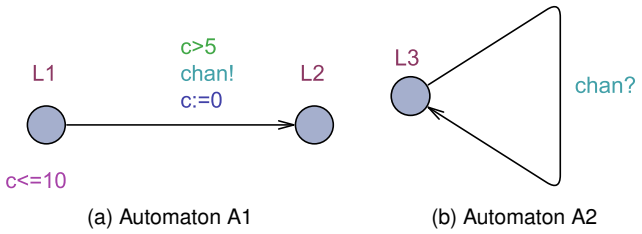


Figure 8.1: Example of automata in UPPAAL

tomaton can synchronize with another automaton via channels. Data can be shared by all automata by shared variables. A location marked as "U" is an "urgent" location, indicating that the next transition (not necessarily from this same location) should be taken without delay. A location marked as "C" is a committed location, indicating that the transitions from this location should be taken immediately.

Figure 8.1 shows two automata, A1 and A2, in UPPAAL notation. A1 has two locations, *L1* and *L2*, and has defined a clock variable c. *L1* has an invariant $c <= 10$, indicating that A1 may delay at *L1* at most until c equals 10 time units. The guard $c > 5$ requires that the value of c must be bigger than 5 in order to take the transition to *L2*. A1 synchronizes with A2 via channel *chan*. The "!" denotes sending the signal, and the "?" denotes receiving the signal. When A1 transits from *L1* to *L2*, it sends a signal via *chan* and resets c. When receiving the signal, A2 takes the transition from location *L3* back to *L3*, in turn.

UPPAAL uses a decidable subset of TCTL (Timed Computational Tree Logic) to formalize requirements that need to be proven as properties of the system by model-checking. These formalized specifications, called queries, can be verified exhaustively on the network of timed automata (e.g., A1 || A2 in Figure 8.1). In this paper we will use the following queries:

- $A[]\,p$: **Invariant property** (For all possible execution paths p always holds).
- $p \rightarrow q$: **Leads-to property** (Whenever p holds, q will eventually hold).

Property p is a logic expression that may contain logical operators such as "and", "or", "not" or "imply". In case an invariant property fails the verification, the model checker provides a counter-example, and step-by-step simulation of the counter-example. Readers can refer to the literature [10] for more

information about UPPAAL.

8.3 Assumed System

The assumed system is a CPS including sensors that monitor the environment, and control processes that control the actuators based on the sensor readings. The real-time data are stored in an RTDB. The access and manipulation of the data are managed by an RTDBMS as transactions. We identify the following transaction types in the RTDBMS.

Update transaction An update transaction is a write-only transaction that updates a real-time data object with the sampled value in the database. It is triggered with a period that is decided by the sampling rate.

Control transaction A control transaction reads data from the database, performs application-dependent computation based on the data, and may write data into the database. In a real-time application, control transactions are often periodic, or are triggered with a minimum inter-arrival time. They often have specified deadlines to meet.

Such update and control transactions may be executed concurrently. The RTDBMS applies a certain concurrency control mechanism to prevent unwanted interferences.

We consider the following computations in the system. Two update transactions, T0 and T1, update data D0 and D1, respectively. T0 has a period of 7ms, and T1 has a period of 8ms. A control transaction, T2, reads D0 and D1, and does some computation based on the values. T2 has a period of 15ms, and a deadline of 15ms. The worst-case time to read a data is 1ms, while the worst-case time to write is 2ms. In this paper, we assume that a lock-based concurrency control is applied. Before a transaction is able to read from or write to a data object, it needs to acquire the lock of that data. When a transaction commits, it releases the locks it holds so that the locked data become accessible to other transactions. The locking and unlocking are assumed to be instantaneous.

The following temporal consistency and timeliness requirements are specified. Among them, Requirement 1 and 2 refer to absolute validity, Requirement 3 refers to relative validity, and Requirement 4 refers to the transaction timeliness.

- **Requirement 1.1** D0 should never be older than its absolute validity interval, which is 15ms.
- **Requirement 2.1** D1 should never be older than its absolute validity interval, which is 16ms.
- **Requirement 3.1** The difference between the ages of D0 and D1 should never be larger than the relative validity interval, which is 18ms.
- **Requirement 4** T2 should not miss its deadline, which is 15ms.

The weak absolute validity and weak relative validity are specified as follows.

- **Requirement 1.2** D0 should not be older than its absolute validity interval, which is 15ms, when it is accessed by T2.
- **Requirement 2.2** D1 should not be older than its absolute validity interval, which is 16ms, when it is accessed by T2.
- **Requirement 3.2** The difference between the ages of D0 and D1 should not be larger than the relative validity interval, which is 18ms, when accessed by T2.

8.4 Modeling Transaction Work Units and Data

In this section, we describe our approach for modeling the work units [11] of the transactions, the data and the transaction manager. The high-level description of the modeling approach is presented in Figure 8.2. The work units, data and the transaction manager are modeled as timed automata respectively. Similar to our previous work [6], a work unit automaton models the operations within a transaction with respect to timing, as well as the interactions with the data and the transaction manager. A transaction manager automaton models the concurrency control mechanism in this paper. In order to verify temporal data consistency, in this paper we extend the approach with a data automaton that models the updated time and the age of the data. A work unit automaton sends signals to the data automata, when the data is updated by the transaction. The data automaton will then update the timestamps of the data.

The assumed RTDBMS applies a lock-based concurrency control mechanism to manage concurrent transactions. Therefore, the transaction manager is called the lock manager in the rest of the paper. The work unit automata may

send signals to require for locks from the lock manager. The lock manager either sends signals to the transactions and grant the locks, or lets the transactions wait if the data are already locked. In the remaining part of this section we will discuss how to model the work units, the data and the lock manager. To simplify the illustration, we omit error handling of the transactions, and assume that the lock manager makes decisions instantaneously.

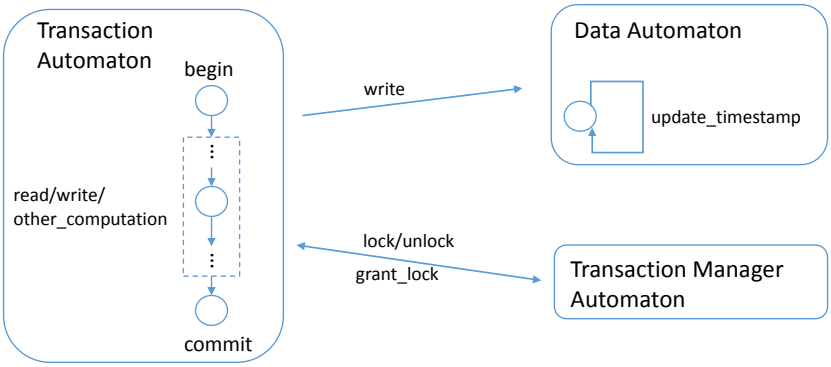


Figure 8.2: High-level description of the approach

8.4.1 Modeling Transaction Work Units

Similar to our previously proposed approach [6], a transaction is composed by its work-unit operations that include the operations on data and other computation, and transaction management operations, including begin, commit, and synchronizing with the lock manager. We model the work unit, as well as the interactions with the lock manager imposed by the concurrency control, as an automaton in UPPAAL. In such a model, the operations are modeled as a set of locations. A transition from one location to another models the execution order of the operations. Since we especially target temporal data consistency, we explicitly model the interaction between transactions and data. When a transaction updates a data, it sends a corresponding signal to the data automaton via the designated channel.

Figure 8.3 shows the timed automaton for the update transaction T0 described in Section 8.3. In this automaton, the transaction starts from the initial location *begin*. A variable *cs* represents the CPU resource. If *cs* is 1, meaning

the CPU is taken by another transaction, T0 must wait until CPU is free, which is modeled by the *cs_free* signal. When *cs* is 0, T0 tries to lock the data D0 via channel *lock_data_0[id]*, where *id* is the identifier of this transaction. It then waits until the lock is granted via channel *grant_lock_0[id]*, and proceeds to write the new value of D0 at location *write_d0*. Due to the timing constraints, we use a clock *temp* together with the invariant *WRITE_TIME* to model that it takes in worst case *WRITE_TIME* time units (in this case 2ms) to write the data. After this, the transaction immediately unlocks the data, which is modeled by an urgent location, and a consecutive *unlock_data_0[id]* channel. Then it notifies the observer that D0 is updated, via channel *update[data_id]*. The commit of the transaction is modeled as the *commit_work* location, with an invariant bounded by the commit time. To model the execution time of T0 we create a clock variable *t_u0*, which is reset when T0 is started. The periodic behavior of T0 is modeled such that T0 will be restarted if *t_u0* is equal to T0's period.

The model of T1 is very similar to the automaton of T0. The channels are defined as locking, unlocking and updating operations on D1 instead of D0, and the values of deadline and period are T1 specific.

The modeling of the control transaction T2 follows the same principles. We show the automaton of T2 in Figure 8.4. The reading D0 and D1 operations are modeled as locations *read_d0* and *read_d1* respectively, and between the locations channels are used to model the locking mechanism. Computational operations other than read and write are abstracted as a location *other_work*. The *temp* clock, the invariant *temp*$<=$*MAX_WORK_TIME* and the guard *temp* $>=$ *MIN_WORK_TIME* together enforce the best and worst case execution time of the computation. A clock variable *t_tran* keeps track of the time of T2. If *t_tran* is bigger than the deadline, T2 will reach the *deadline_miss* location, indicating a deadline miss.

8.4.2 Modeling the Age of Data

In order to model-check temporal data consistency, we need to model the age of the data. Our solution is to use observer automata that observe the write operations on the data, and update the age of the data accordingly. The observer automaton of D0 is shown in Figure 8.5. In this figure, the age of data D0 is modeled by a clock variable *age*. When data D0 is updated, the transaction sends a signal to the data automaton via channel *update[0]*, and the observer automaton then resets *age*. Therefore, at any given time point, *age* represents the age of D0.

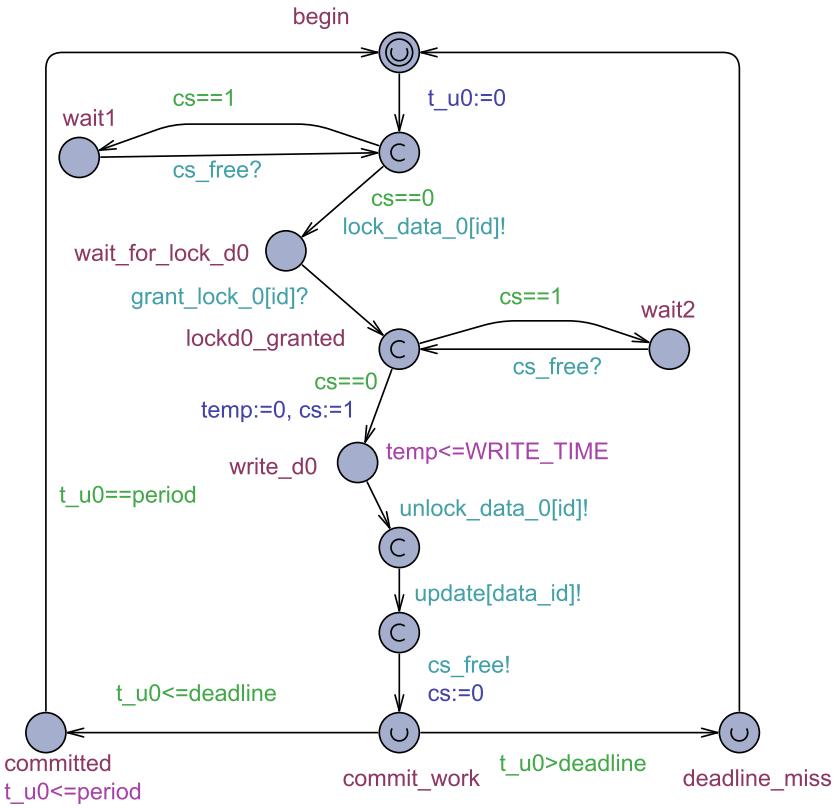


Figure 8.3: The work unit automaton of the updater transaction T0

8.4.3 Modeling the Lock Manager

The lock manager handles lock requests and releases from transactions, and grants available locks to the transactions. The modeling of the lock manager is heavily application dependent. In the assumed system, the lock manager grants locks to the transactions in a First-In-First-Serve manner. The lock manager implements a queue that holds the transactions requiring for a lock. New requiring transactions are appended to the end of the queue, while the head of the queue is the one that will be granted the lock.

The model of the lock manager is shown in Figure 8.6. The queue of wait-

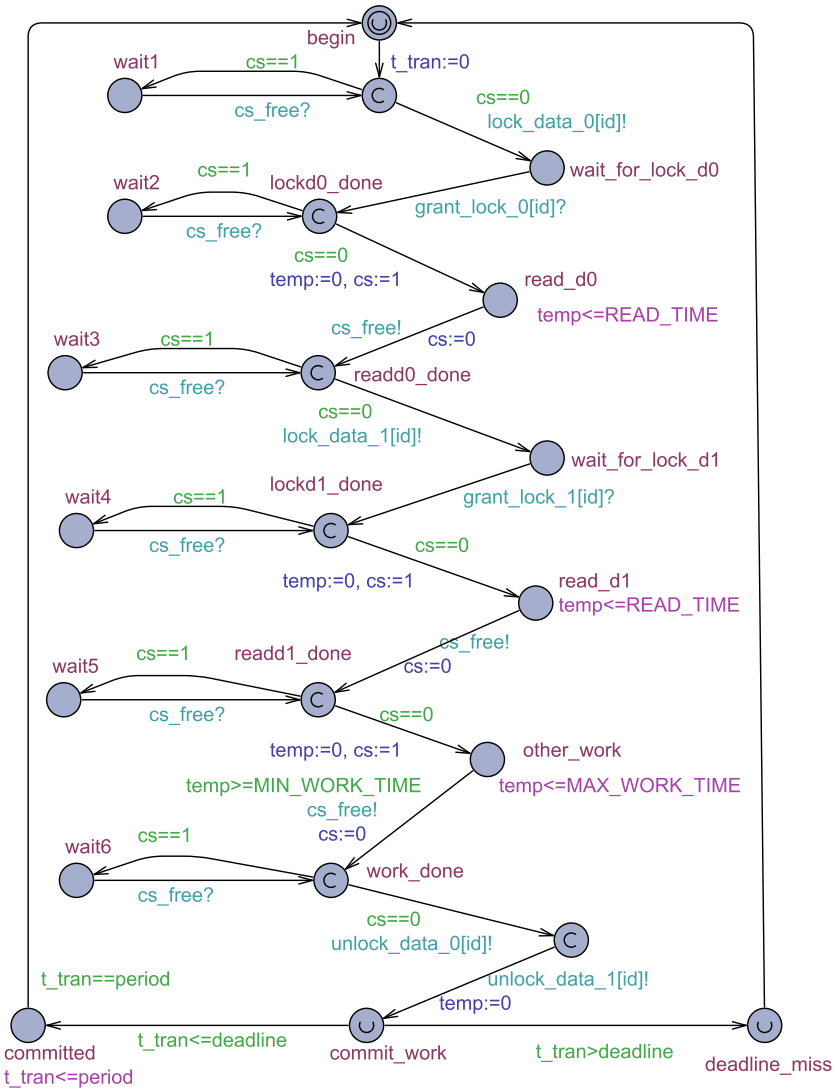


Figure 8.4: The work unit automaton of the control transaction T2

ing transaction ids is modeled by an array, whose first transactions' id is as-

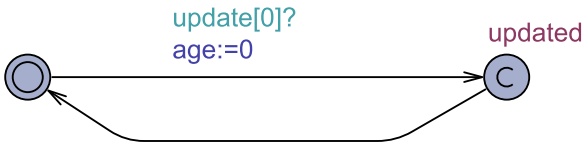


Figure 8.5: UPPAAL model of data D0

Program 8.1: The functions in the lock manager automaton

```
void enqueue(int tran_id) {
queue[len]=tran_id;
len++;
head=queue[0];}

void dequeue() {
if(len==1) {
queue[len]=0;
len--;
head=0;
} else {
int i;
for(i=0;i<len;i++)
queue[i]=queue[i+1];
queue[len]=0;
len--;
head=queue[0]; }}
```

signed to a variable *head*. An *enqueue()* function appends a new transaction id to the end of the array, while a *dequeue()* function removes the first transaction id from the array, and updates the index and the head of the queue. The *enqueue()* and *dequeue()* functions are shown in Listing 8.1. When the lock manager gets requests from a transaction with *tran_id* for locking D0 via channel *lock_data_0[tran_id]*, it moves to the *lock_data* location. Meanwhile, during the transition, the transaction is inserted into the queue of D0 by *enqueue()*. The lock manager then checks if the data is currently being locked. If the data is not locked, which means the requiring transaction is the head of the queue, the lock is granted via the channel *grant_lock_0[head]*. If the data is locked, the lock manager just returns to the initial location, and the requiring

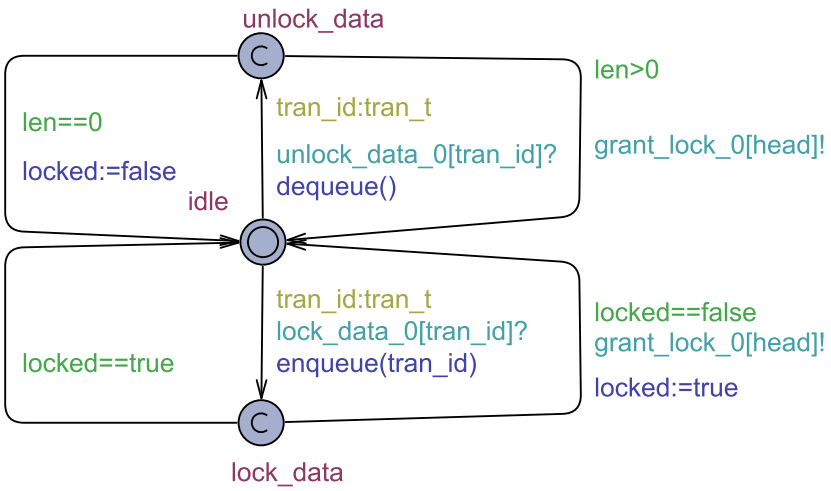


Figure 8.6: UPPAAL model of the lock manager

transaction has to wait. The location *lock_data* is a committed location, indicating the transition sequence from location *idle* to *lock_data* and then back to *idle* is instantaneous and atomic.

When a transaction unlocks data D0 via channel *unlock_data_0[tran_id]*, it is removed from the queue by function *dequeue()*. If the array is not empty, the lock will be granted to the head of the queue via channel *grant_lock_0[head]*.

8.5 Verification of Temporal Data Consistency and Timeliness

In this section we formulate the requirements of temporal consistency and timeliness as UPPAAL verification queries, and verify these properties for the assumed system.

8.5.1 Formalizing the Requirements

To model-check the requirements in Section 8.3 using UPPAAL, we have to first specify these requirements in TCTL. We propose a set of specification

patterns to help the formulation of the temporal consistency and the timeliness of transactions.

The absolute validity requirements, i.e., Requirement 1.1 and 2.1 from Section 8.3, can be specified as: the clock *age* of the automaton of Di must always be smaller than or equal to its absolute validity interval AVI(i). This is a property that must hold invariantly, and can be specified using the $A[]$ operators in TCTL as:

$$A[\,]\, Di.age <= AVI(i).$$

The weak absolute validity requirement can be specified as: whenever Tj reads Di, the age of Di must be smaller than or equal to its absolute validity interval AVI(i). This property is specified as:

$$A[\,]\,(Tj.read\_di\, imply\, Di.age <= AVI(i)).$$

The relative validity, as described in Requirement 3, requires the age differences of Di and Dj to be smaller than or equal to the relative validity interval RVI(i,j). Intuitively, this requirement can be specified as: $A[\,](Di.age - Dj.age <= RVI(i,j)\, and\, Dj.age - Di.age <= RVI(i,j))$. However, the verification of this query might not terminate, due to a large state-space during verification .

In Figure 8.7 we illustrate the update of the age variables with respect to time, using D0 and D1 as examples. During the period shown in the figure, D0 is updated in t1 and t4, while D1 is updated in t2 and t6. Without loss of generality, we consider the relative validity at t3 and t5. At t3, the values of D0.age and D1.age are t3-t1 and t3-t2 respectively. The difference between the ages is hence t2-t1. This is actually the age of D0 when D1 is updated. Similarly, the age difference at t5 is equal to t4-t2, which is the age of D1 when D0 is updated. Therefore, we formulate relative validity of Di and Dj as the following query, which explores fewer states compared with the original one we mentioned in the previous paragraph:

$$\begin{array}{c} A[\,]\,((Di.updated\, imply\, Dj.age <= RVI(i,j)) \\ and\,(Dj.updated\, imply\, Di.age <= RVI(i,j))). \end{array}$$

Similarly, the weak relative validity requirement requires that whenever Tk reads Di or Dj, the age differences of Di and Dj to be smaller than or equal to the relative validity interval RVI(i,j). This can be formulated as:

Figure 8.7: Illustration of updates of data with respect to time

$$A[]\,((Tk.read\_di\ or\ Tk.read\_dj)\ imply\\ ((Di.updated\ imply\ Dj.age <= RVI(i,j))\ and\\ (Dj.updated\ imply\ Di.age <= RVI(i,j))))$$

The verification of timeliness equals to proving that location *Ti.deadline_miss* is not reachable. This requirement is equivalent to the following invariant property:

$$A[]\ not\ Ti.deadline\_miss.$$

The proposed specification patterns are summarized in Table 8.1. In each row, the table shows the informal description of the property, as well as the corresponding query patterns in UPPAAL TCTL.

8.5.2 Verification Results

We model the assumed system as described in Section 8.4 and verify properties using UPPAAL 4.1.19. The properties are specified using the specification patterns from the previous subsection. The results are listed in Table 8.2. All requirements have passed the verification. The table also lists the time it takes to verify each query, as well as the memory consumption. Since the system we have modeled is not complex, the time and memory costs look promising.

8.6 Related Work

Kung [13] applies pushed automata techniques to model and verify temporal constraints in a database. However, the temporal constraints he has verified do not include temporal data consistency. Song et al. [3] introduced the concept

Table 8.1: Specification Patterns

Property	Informal Specification	Query Pattern
Absolute validity	The age of Di must always be smaller than or equal to its absolute validity interval AVI(i)	$A[]\, Di.age <= AVI(i)$
Weak absolute validity	Whenever Tk reads Di, the age of Di must always be smaller than or equal to its absolute validity interval AVI(i)	$A[]\,(Tk.read\_di\, imply$ $Di.age <= AVI(i))$
Relative validity	The age differences of Di and Dj to be smaller than or equal to the relative validity interval RVI(i,j)	$A[]\,((Di.updated\, imply$ $Dj.age <= RVI(i,j))$ $and\,(Dj.updated\, imply$ $Di.age <= RVI(i,j)))$
Weak relative validity	Whenever Tk reads Di or Dj, the age differences of Di and Dj to be smaller than or equal to the relative validity interval RVI(i,j)	$A[]\,((Tk.read\_di\, or\, Tk.read\_dj)$ $imply$ $((Di.updated\, imply$ $Dj.age <= RVI(i,j))\, and$ $(Dj.updated\, imply$ $Di.age <= RVI(i,j))))$
Transaction timeliness	Transaction Tk will not miss its deadline	$A[]\, not\, Tk.deadline\_miss$

Table 8.2: Verification Results

Req	Query	Verification Time	Memory Consumption	Explored States	Status
1.1	$A[]\, D0.age <= 15$	0.577s	9716KB	54996	Satisfied
1.2	$A[]\, D1.age <= 16$	0.609s	9844KB	54643	Satisfied
2.1	$A[]\,(T2.read\_d0\, imply\, D0.age <= 15)$	0.608s	9860KB	54996	Satisfied
2.2	$A[]\,(T2.read\_d1\, imply\, D1.age <= 16)$	0.608s	9868KB	54643	Satisfied
3.1	$A[]\,((D0.updated\, imply\, D1.age <= 18)$ $and\,(D1.updated\, imply\, D0.age <= 18))$	0.765s	10008KB	57893	Satisfied
3.2	$A[]\,((T2.read\_d0\, or\, T2.read\_d1)\, imply$ $((D1.updated\, imply\, D0.age <= 18)\, and$ $(D0.updated\, imply\, D1.age <= 18)))$	0.780s	10044KB	57893	Satisfied
4	$A[]\, not\, T2.deadline\_miss$	0.468s	9680KB	58729	Satisfied

of temporal consistency. In their work, temporal consistency is evaluated via simulation, instead of formal verification. Han et al. [5] apply schedulability analysis on transactions to maintain temporal consistency. However, they do not consider concurrency control in the analysis. Both Lauer et al. [14] and Le Berre et al. [15] use formal methods to verify temporal data consistency in networked systems. Compared with our work, their work do not deal with database assumptions and do not model transaction behaviors.

Several researchers have used UPPAAL TA to model various aspects of database transactions and transaction management mechanisms. For example, Kot [7] models several selected transaction concurrency control mechanisms in UPPAAL and verify properties such as free of deadlock. Al-Bataineh et al. [16] uses UPPAAL to model a two-phase commit protocol for an RTDBMS. In our previous work, we have proposed a flexible approach to verify transaction timeliness and isolation using UPPAAL [6]. Although these works also use model checking and UPPAAL, they focus on other aspects of the database than temporal data consistency.

8.7 Conclusion

In this paper we have described a model-checking approach for verification of transaction timeliness and temporal data consistency in a real-time database within a cyber-physical system. We have modeled the transaction work units, the data and the concurrency control mechanisms as a network of automata. This work continues our previous work [6] in an attempt to create a framework for verification of RTDBMS, with respect to verifying transaction timeliness v.s. temporal data consistency. The properties are specified in TCTL using our proposed specification patterns. The formalized properties are model checked using UPPAAL. An example RTDBMS with update and control transactions has been used to demonstrate the model checking approach. All properties have been proved satisfied within short time and with low memory consumption.

One possible problem of this approach is the potential state explosion when the modeled system incorporates a large number of transactions and data. This can be mitigated by partitioning the transactions and data according to their dependencies. For example, many CPSs apply distributed data management. Instead of modeling the entire system directly, it may be possible to verify the properties in the local databases, and the model check, or deductively prove, the properties of the entire system.

Acknowledgment This work is funded by the Knowledge Foundation of Sweden (KK-stiftelsen) within the DAGGERS project.

Bibliography

[1] Lui Sha, Sathish Gopalakrishnan, Xue Liu, and Qixin Wang. Cyber-physical systems: A new frontier. In *Machine Learning in Cyber Trust*, pages 3–13. Springer US, 2009.

[2] Kyoung-Don Kang and S.H. Son. Real-time data services for cyber physical systems. In *Distributed Computing Systems Workshops, 2008. ICDCS '08. 28th International Conference on*, pages 483–488, June 2008.

[3] X. Song and J.W.S. Liu. Performance of multiversion concurrency control algorithms in maintaining temporal consistency. In *Proceedings of the Fourteenth Annual International Computer Software and Applications Conference*, pages 132–139, 1990.

[4] Krithi Ramamritham, Sang H. Son, and Lisa Cingiser Dipippo. Real-time databases and data services. *Real-Time Syst.*, 28(2-3):179–215, 2004.

[5] Song Han, Kam yiu Lam, Jiantao Wang, Sang H. Son, and Aloysius K. Mok. Adaptive co-scheduling for periodic application and update transactions in real-time database systems. *Journal of Systems and Software*, 85(8):1729 – 1743, 2012.

[6] S. Cai, B. Gallina, D. Nyström, and C. Seceleanu. Flexible verification of transaction timeliness and isolation. Technical report, 2016.

[7] Martin Kot. Modeling selected real-time database concurrency control protocols in uppaal. *Innovations in Systems and Software Engineering*, 5(2):129–138, 2009.

[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[9] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, 1993.

[10] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[11] B. Gallina. *PRISMA: a software product line-oriented process for the requirements engineering of flexible transaction models*. PhD thesis, University of Luxembourg, 2010.

[12] Ben Kao, K. Y. Lam, Brad Adelberg, Reynold Cheng, and Tony Lee. Updates and view maintenance in soft real-time database systems. In *Proceedings of the Eighth International Conference on Information and Knowledge Management*, CIKM '99, pages 300–307, 1999.

[13] C. H. Kung. On verification of database temporal constraints. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, pages 169–179, 1985.

[14] Michal Lauer, Frdric Boniol, Claire Pagetti, and Jrme Ermont. End-to-end latency and temporal consistency analysis in networked real-time systems. *International Journal of Critical Computer-Based Systems*, 5(3-4):172–196, 2014.

[15] Tanguy Le Berre, Philippe Mauran, Gérard Padiou, and Philippe Quéinnec. A Data Oriented Approach for Real-Time Systems. In *17th International Conference on Real-Time and Network Systems*, pages 147–158, Paris, France, October 2009.

[16] O. Al-Bataineh, T. French, and T. Woodings. Formal modeling and analysis of a distributed transaction protocol in uppaal. In *Temporal Representation and Reasoning (TIME), 2012 19th International Symposium on*, pages 65–72, Sept 2012.

Chapter 9

Paper D: Design of Cloud Monitoring Systems via DAGGTAX: a Case Study

Simin Cai, Barbara Gallina, Dag Nyström, Cristina Seceleanu, and Alf Larsson.

In Proceedings of the 8th International Conference on Ambient Systems, Networks and Technologies (ANT 2017), Madeira, Portugal, May 2017. Elsevier.

Abstract

Efficient auto-scaling of cloud resources relies on the monitoring of the cloud, which involves multiple aggregation processes and large amounts of data with various and interdependent requirements. A systematic way of describing the data together with the possible aggregations is beneficial for designers to reason about the properties of these aspects as well as their implications on the design, thus improving quality and lowering development costs. In this paper, we propose to apply DAGGTAX, a feature-oriented taxonomy for organizing common and variable data and aggregation process properties, to the design of cloud monitoring systems. We demonstrate the effectiveness of DAGGTAX via a case study provided by industry, which aims to design a cloud monitoring system that serves auto-scaling for a video streaming system. We design the cloud monitoring system by selecting and composing DAGGTAX features, and reason about the feasibility of the selected features. The case study shows that the application of DAGGTAX can help designers to identify reusable features, analyze trade-offs between selected features, and derive crucial system parameters.

9.1 Introduction

In the prominent evolution to the Fifth Generation (5G) of mobile technology, both network applications and infrastructural services are increasingly deployed as virtualized software instances running in the cloud [1]. A main advantage of this shift is the automatic scaling of resource provision for dynamic and heterogeneous applications, a promising technique that ensures the Quality of Service (QoS) of applications with efficient use of resources. To achieve this, the run-time states from various layers of the cloud such as hardware layer and virtual machines need to be continuously monitored, aggregated and analyzed, in order to be able to efficiently allocate resources on demand.

Instead of adopting off-the-shelf monitoring tools directly, many companies choose to design their own monitoring functionality for scaling, either from scratch or by extending existing frameworks, in order to meet their particular needs [2]. This requires the system designers to decide what data and how they should be collected, aggregated, and propagated. Since each layer of the cloud may impose different requirements for the data aggregation, the designed solution must ensure the various properties of both the data and the processes. For designers choosing to extend existing frameworks, analyzing dependencies between the new extensions and existing system components also adds complexity to the design. Due to these complexity and heterogeneity issues, designing such systems is prone to faults that might compromise the efficiency and effectiveness of the monitoring tool, preventing it from realizing its full potential [2]. The systematic analysis of the design decisions at early design stages, based on a thorough understanding and systematic analysis of the data as well as the aggregation process, holds the promise of alleviating the issue.

In this paper, we present a case study provided by Ericsson that applies our previously proposed taxonomy called DAGGTAX (Data AGGregation TAXonomy) [3] to support the design of a cloud monitoring system. DAGGTAX provides a high-level characterization of Data Aggregation Processes (DAPs), focusing on a systematic representation of common and variable features of the data as well as the aggregation process itself.

The problem that we tackle is to systematically design a cloud monitoring system for an enhanced auto-scaling functionality in a video streaming system, such that potential unfeasible design decisions are prevented. The designed system extends the existing open-source OpenStack framework[1], whose major components for achieving auto-scaling are shown in Figure 9.1. Among them, *Ceilometer* monitors the run-time states of the cloud by collecting var-

[1] `https://www.openstack.org/software/`

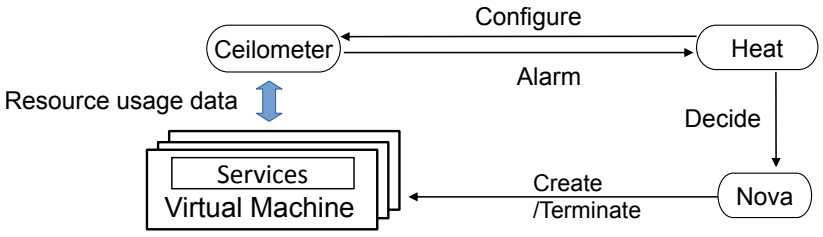


Figure 9.1: Auto-scaling related components in the OpenStack framework.

ious resource data, such as CPU usage of each virtual machine. Once a measurement meets a predefined condition, *Heat* is alerted to decide if a scaling action should be taken, and notifies *Nova* to start or terminate a virtual machine. According to this solution, however, only hardware resource usage of virtual machines are taken into account, regardless of the status of the actual applications. To achieve finer-tuned auto-scaling, Ericsson requires the cloud controller to monitor not only the hardware usage by each virtual machine, but also the application performances within the virtual machines.

We apply DAGGTAX to both the analysis of the current OpenStack framework, and the design of the new solution. The DAPs in the current framework are represented as a feature diagram instantiated from DAGGTAX. The new solution is designed by selecting and composing the DAGGTAX features, among which, some are identified as reusable features from the current framework. Based on these feature diagrams, we further analyze the feasibility and correctness of the design. Our experience from the analysis demonstrates that the taxonomy helps to gain a better understanding of the data and data aggregation processes in the cloud monitoring system, which enables the designers to identify reusable features, analyze trade-offs between the desired features, and derive crucial system parameters. From an industrial perspective, the ability to have a common nomenclature has also been found very useful, since it bridges the various descriptions and specifications of a data aggregation design used in the company today.

The remainder of the paper is organized as follows. We recall DAGGTAX in Section 9.2, and present our case study and the application of DAGGTAX in Section 9.3. Section 9.4 presents the lessons we have learned from the case study. In Section 9.5 we discuss the related work, after which we conclude the paper in Section 9.6.

9.2 Background

A Data Aggregation Process (DAP) is defined as the process of producing a synthesized form from multiple data items [4]. DAGGTAX [3] provides a global, high-level characterization of DAPs, in the form of a *feature diagram* [5], presented in Figure 9.2. In this diagram, nodes represent features of a DAP, and edges represent decomposition of features. A node with a solid dot represents a common feature mandatory for every DAP, whereas a node with an empty circle represents an optional feature. A group of alternative features is represented by a group of nodes associated with a spanning curve, from which one feature must be selected by a particular DAP. The cardinality [m..n] annotated with a node denotes how many instances of the feature, including the entire sub-tree, can be considered as children of the feature's parent in a DAP.

The top level features in Figure 9.2 include the main constituents of an aggregation process (**Raw Data**, **Aggregate Function** and **Aggregated Data**), as well as features characterizing the entire DAP, including the **Triggering Pattern** of the process, and **Real-Time (P)**, which refers to the optional timeliness property of the entire process. In the following we briefly explain the concepts underlying each feature. For more details, we refer to our previous work [3].

Raw Data A data aggregation process must involve at least one *Raw Data Type*. Each type of raw data consists of at least one instance of *Raw Data*. The sub-features are:

Pull - Raw data are actively pulled from the data source by the aggregation process.

Persistently Stored - Raw data are stored persistently.

Shared - Raw data involved in the aggregation are shared by other processes in the system.

Sheddable - Raw data can be skipped for the aggregation, due to trade-offs between different system properties.

MinT - Minimum inter-arrival Time of raw data.

MaxT - Maximum inter-arrival Time of raw data.

Real-Time (RD) - Each raw data instance is associated with an arrival time, and is only valid if the elapsed time from its arrival time is less than its *Absolute Validity Interval*. Outdated *Hard* real-time data will result in loss of life or money. On the contrary, outdated *Firm* real-time data bare no value, while outdated *Soft* real-time data produce less value.

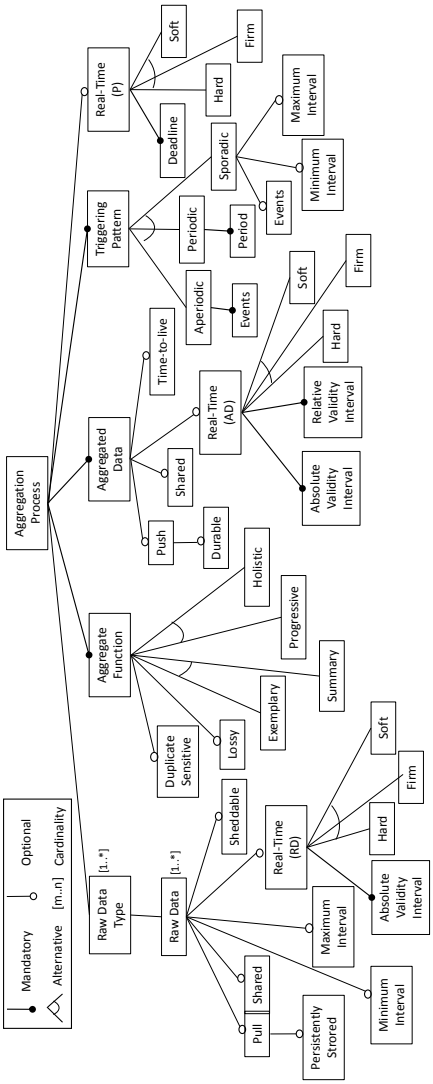


Figure 9.2: DAGGTAX depicted as a feature diagram.

Aggregate Function An aggregate function performs the aggregation computation. Its sub-features include:

Duplicate sensitivity - The aggregated result is incorrect if a raw data is duplicated.

Lossy - Raw data cannot be reconstructed from the aggregated data alone.

Exemplary/Summary - An exemplary aggregate function returns one or several representative values of the selected raw data. A summary aggregate function computes a result based on all selected raw data.

Progressive/Holistic - The computation of a progressive aggregate function can be decomposed into the computation of sub-aggregates, whereas a holistic aggregate function must be computed on the entire data set at once.

Aggregated Data An aggregation process must produce one *aggregated data*. Its sub-features include:

Push - Sending aggregated data to another unit of the system is apart of the DAP.

Durable - The aggregated results should survive potential system failures.

Shared - The aggregated data are shared by other processes in the system.

Time-to-live - The aggregated data should remain available for a specified period of time in the aggregator.

Real-Time (AD) - An aggregated data is absolute valid if all participating raw data are absolute valid. The *absolute validity interval* of the aggregated data depends on the intervals and ages of the raw data that are used to derive the aggregated data. In addition, all raw data involved in the aggregation should be sampled within a specified interval, called *relative validity interval*. Similar to raw data, the strictness of real-time aggregated data can be classified as *hard*, *firm* and *soft*.

Triggering Pattern A DAP is activated with a triggering pattern, specified as *Periodic*, *Sporadic* or *Aperiodic*. A periodic DAP is invoked according to a time schedule with a *Period*. A sporadic DAP could be triggered by an external event, or according to a time schedule, possibly with a *MinT* (Minimum inter-arrival Time) and/or *MaxT* (Maximum inter-arrival Time). An aperiodic DAP is activated by an external event without a constant period, MinT or MaxT.

Real-time (P) A DAP may need to satisfy timeliness requirements, named as "*Real-Time (P)*". The real-time DAP need to complete its work by a specified *Deadline*. It can be classified as *Hard* real-time, meaning missing the deadline will cause intolerable loss of life or profit and thus must be avoided. A *Firm* real-time process will bring no value, while a *Soft* real-time process will provide less value, if the deadline is missed.

9.3 Case Study and Results

In this section, we describe the industrial case study, in which we apply DAG-GTAX to design a cloud monitoring system by extending the open-source OpenStack framework.

9.3.1 Case Study Description

The target cloud system consists of a collection of hardware resources (physical servers and network capacities), virtualized by complex management software. Software services, including video streaming services, are deployed on a number of Virtual Network Functions (VNFs), which are virtual machines spawned and terminated by a VNF manager, and running on top of the virtualized hardware. Each VNF consists of a set of Virtual Network Function Components (VNFCs), each representing a collection of applications running in the cloud. For instance, one such VNFC may contain all video streaming services responsible for the users requests, while another is dedicated to handle security issues.

In order to maintain the desired QoS while maximizing the resource utilization, VNFs should be started or terminated according to the status of the applications. Such auto-scaling decisions are taken by the cloud controller based on run-time measurements. OpenStack supports resource-usage measurements such as CPU usage of each VNF for auto-scaling. However, the coarse-grained, VNF-based measurements may not be adequate for efficient auto-scaling decisions. For instance, resources may appear to be exhausted

soon, either (i) by video streaming services that are critical to end users, or (ii) by a routinely security check while the streaming requests are low. While the former case may indicate insufficient provision that may degrade the QoS, the latter is a temporary maintenance phenomenon that will pass soon, for which the scaling-up will cause unnecessary system overhead and become a waste of resource.

Therefore, we consider to take into account the measurements from within the VNFs. The data to be collected include: (i) CPU usage of each VNF; (ii) CPU usage of each VNFC within each VNF; (iii) Throughput and the number of dropped packets of each application within each VNFC.

A new VNF should be spawned (scaling up), if either of the following rules is satisfied: (i) the average CPU usage of any VNF is higher than 90% for 60 seconds; (ii) the average CPU usage of any VNF is higher than 80% for 60 seconds, and the packet loss of the video streaming services is higher than zero.

An existing VNF should be terminated (scaling down), if the following rule is satisfied: the average CPU usage of the VNF is lower than 5% for 60 seconds, and the packet loss of the video streaming services is zero.

9.3.2 Application of DAGGTAX

We apply DAGGTAX to organize the data aggregation processes in the existing auto-scaling functionality of OpenStack, as well as to select and compose features for the desired enhanced auto-scaling functionality.

In the OpenStack framework, two levels of aggregation take place: one generating alarms from aggregating CPU usages of the VNFs, the other making the scaling decisions from aggregating the alarms. These aggregation processes are identified using DAGGTAX and presented in Figure 9.3, in which each box is an instantiated feature from Figure 9.2. In the bottom level, a DAP called CPUAlarmStatusEvaluation aggregates periodically a set of VNFCPUs raw data pulled from the Ceilometer database. These raw data are sampled by the hypervisor prior to the DAP by another process with a predefined frequency (MinT and MaxT have the same value). All CPU statistics within the interval between two aggregation periods are aggregated by an aggregate function, which computes the average value of the CPU data, compares the value with a threshold value, and produces the alarm status as a result. The aggregate function is duplicate sensitive, lossy, progressive and computes a summary. The aggregated alarm status is then pushed to Heat for the auto-scaling decision. In the top level, ScalingDecisionMaking process is triggered by the alarm

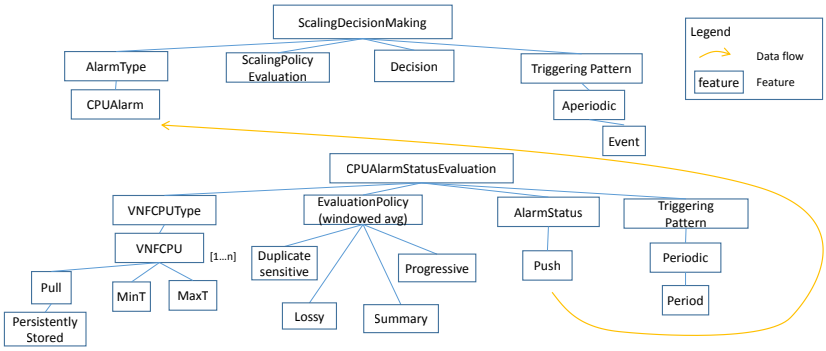


Figure 9.3: Data aggregation processes in OpenStack auto-scaling functionality.

event. A ScalingPolicyEvaluation function is applied to the alarm status. If the status shows that the CPU usage is higher than the threshold, and the time from last scaling action is longer than one minute, an auto-scaling decision will be taken, either to add or to terminate a VNF.

The DAPs for the new auto-scaling functionality, together with the design decisions on the data flow management, are presented in Figure 9.4. The features in gray color are already implemented in OpenStack. For better readability, we only show the features of raw data, aggregated data, triggering patterns and the real-time properties of new DAPs, and omit the features of the aggregate functions and existing features. In this design, the top-level ScalingDecisionMaking process takes two types of raw data: the CPU alarm status as in the existing solution, as well as a set of VNFProfiles, which are status profiles of currently active VNFs. Each VFNProfile is generated by VNFProfileGeneration process, aggregating a set of VNFCProfiles, representing the status of the VNFCs in this VNF. Each VNFCProfile is an aggregation of the CPU usage of this VNFC (VNFCCPU), and a set of AppProfiles, which are the status profiles of the applications in this VNFC. In the lowest level, an AppProfile is an aggregation of the throughput and the number of dropped packets that are sampled for each application. The VNFCProfileGeneration and AppProfileGeneration processes are desired to meet their deadlines in order not to interfere with the video streaming services.

ScalingDecisionMaking and CPUAlarmStatusEvaluation are deployed in the controller node, while the other DAPs for VNF, VNFC and application pro-

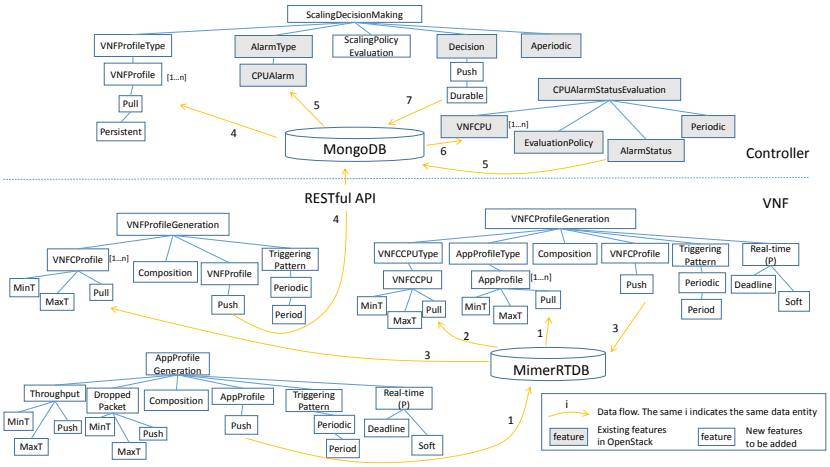


Figure 9.4: Data aggregation processes in the designed system.

files are deployed in the VNFs. Data between DAPs, within the controller and the VNF respectively, are communicated via databases. In the controller we use MongoDB [2] as it is by default used in the current framework, whereas in the VNF we use Mimer SQL Real-Time Edition (MimerRTDB) [3], which provides predictable real-time data access as required by the VNFCProfileGeneration and AppProfileGeneration processes.

9.3.3 System Implementation

We have developed a prototype that implements the designed DAPs by extending OpenStack (version: Newton). The architecture is shown in Figure 9.5. The prototype is deployed on a PC with a 2.7 GHz quad-core process and 16 GB memory. Each VNF is a Linux virtual machine that hosts two VNFCs and a MimerRTDB. VNFC1 holds two video streaming applications, whereas VNFC2 holds two applications that are less critical. Each application is simulated by a process written in C that updates the throughput and dropped packets in the database. For each application, an AppProfileGeneration process is executed to aggregate the application data and generate its AppProfile. Similarly,

[2] https://www.mongodb.com/
[3] http://www.mimer.se/Products/MimerSQLRealtime.aspx

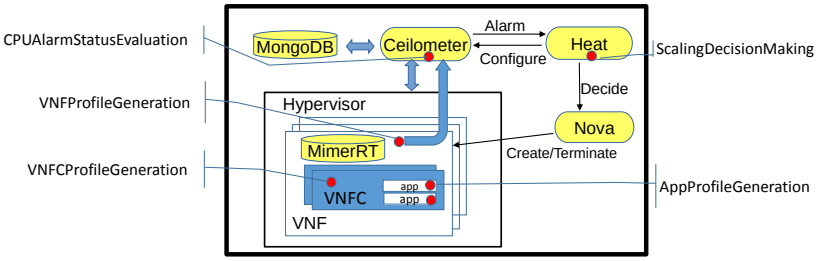


Figure 9.5: Architecture of the enhanced auto-scaling for the cloud video streaming system.

in each VNFC, an aforementioned VNFCProfileGeneration process is executed to create its VNFCProfile, while in each VNF a VNFProfileGeneration process is executed. The VNFProfiles are then sent to Ceilometer, via a new service entry point, and saved in MongoDB. The new scaling rules, as specified in Section 9.3.1, are defined in Heat.

We analyze the prototype with simulated data provided by Ericsson. The timing parameters of raw data and the processes are listed in Table 9.1 and 9.2, respectively. The simulation workloads, as well as results, are presented in Table 9.3, which shows that the prototype system achieves more accurate auto-scaling compared to the current OpenStack framework according to the specified rules. In particular, as expected, our prototype remains unchanged when the CPU usage of the VNF exceeds 80% but no packets have been dropped for the streaming service (Mode 2), and successfully scales up when the packets start to be dropped due to overload (Mode 3). As a contrast, the current OpenStack framework cannot distinguish these two modes and scale up for both cases.

Table 9.1: Timing parameters of raw data

Data	MinT	MaxT
Throughput	1s	1s
Dropped Packet	1s	1s

Table 9.2: Timing parameters of DAPs

DAP	Period
AppProfileGeneration	1s
VNFCProfileGeneration	60s
VNFProfileGeneration	60s
CPUAlarmStatusEvaluation	60s

Table 9.3: Simulation workloads and results

Mode	VNF CPU usage	Dropped packet in VNFC1	Expected Result*	DAGGTAX Prototype Result*	Current OpenStack Result*
1	50%	0	NC	NC	NC
2	81%	0	NC	NC	SU
3	81%	1	SU	SU	SU
4	91%	0	SU	SU	SU
5	4%	0	SD	SD	SD

* NC: No Change. SU: Scale Up. SD: Scale Down.

9.4 Benefits of DAGGTAX

During this case study, we recognize several benefits from applying DAGGTAX in the early design stage. In general, similarly to what we have already experienced [3], the taxonomy provides a structured representation to organize the features of data aggregation processes, which enhances the understanding about the data, the aggregation processes, as well as their interplays in the designed cloud system. As a result, a DAP can be designed by composing these features, and crucial design decisions could be made based on systematical reasoning about the desired features.

Identify reusable and new components As shown in Figure 9.4, we identify that the CPUAlarmStatusEvaluation process, as well as some features of the ScalingDecisionMaking process, can be reused from the existing framework since they incorporate the features that are desired by the new system. Other DAPs and features need to be implemented, and integrated with the existing solution.

Data flow management design The intrinsic characteristics of data and DAP revealed by the taxonomy, as well as the dependencies between different DAPs, serve as the basis to design the data flow management of the system. The design decisions include the deployment of data and DAPs. For instance, one may decide to deploy the VNFC profiles, the VNFProfileGeneration process and the VNF profiles in the cloud controller. However, since the delay between VNF and the controller is significantly long, the system cannot collect VNFC

profiles too frequently because of communication overheads. Since the feature diagram explicitly incorporates the periods of the processes and the MinTs/-MaxTs of the data, we are able to analyze the amounts of data and calculate the overhead of different deployment decisions. In the designed system, we decide to deploy only the VNF profiles in the controller, while the VNFC profiles and the VNFProfileGeneration process in the VNF nodes. By only sending the aggregated VNFProfile to the controller node, we can reduce the communication overhead, which allows the system to collect VNFC measurements more frequently and form more accurate VNFC profiles.

We can also reason about the database solutions for the system with the feature graph. We decide to use MongoDB in the controller node hosting VNFProfiles, alarms and decisions, because it is already used by the current OpenStack framework for auto-scaling, and it provides durability for the auto-scaling decisions required by the ScalingDecisionMaking process. In the VNF nodes, we choose to use MimerRTDB, which provides in-memory data management with predictable execution times. This is because the VNFCProfileGeneration and AppProfileGeneration processes are desired to meet their deadlines, and do not require persistence of the data.

Eliminating infeasible design Based on the feature graph, we can eliminate infeasible feature combinations in an early stage, and thus reduce the design space. In the VNFCProfileGeneration process, we cannot ask for persistent raw data (VNFCCPU and AppProfile) nor durable aggregated data (VNFCProfile) if we require the process to meet its deadline, since the overhead of disk I/O is usually not predictable, which contradicts the real-time property of the process. Since feature diagrams have a well-defined semantics in Boolean logic, full formal reasoning on the feasibility of the design based on DAGGTAX is possible, using existing analysis tools for feature models.

To improve the correctness of the scaling decision, one desired property is to guarantee that the VNFProfiles are collected within a certain period, for instance 5 seconds, when an alarm triggers the ScalingDecisionMaking process. In other words, the “Decision” feature should have a “Real-time AD” subfeature, which has a “Relative Validity Interval” of 5 seconds. To enforce this, the VNFProfile should be pushed by VNFProfileGeneration with a “MaxT” of 5 seconds This means that the sum of the execution time of the VNFProfileGeneration process, the communication time between the VNF and the controller, and the time spent in MongoDB, should not be longer than 5 seconds. Since this cannot be satisfied under our current hardware and software architecture, we give up this infeasible requirement in our design.

Deciding time-related parameters Some of the crucial timing properties are explicitly included in the taxonomy, such as MinT, period, deadline, etc. Thanks to these timing properties, as well as the dependencies revealed by the taxonomy, we are able to informally reason about other time-related parameters. In our designed system, MimerRTDB embodies a circular buffer for each data entity whose size should be specified. With the feature diagram, we can easily decide the minimum size required for the design. For instance, the periods of AppProfileGeneration and VNFCProfileGeneration are 1 second and 60 seconds respectively. This indicates that the circular buffer for each AppProfile should host at least 60 records.

9.5 Related Work

The design of monitoring systems for cloud management has attracted much research attention in recent years. For instance, Bruneo et al. [6] have proposed a framework for designing a system collecting measurements from multiple layers of a cloud. Montes et al. [7] have created a taxonomy of cloud monitoring systems, based on which they propose an approach for designing cloud monitoring. Although systematic analysis of architectural characteristics are applied in the design approaches of both works, they do not consider the detailed characteristics of the constituents of data aggregation, as we have done in this paper. Ward et al. [2] have proposed a taxonomy for cloud monitoring, and have discussed some key aspects of designing monitoring strategies. On the contrary, we emphasize to analyze the details of DAPs using DAGGTAX when designing such a monitoring strategy, which is not the authors' focus.

A number of works have been conducted by researchers in order to understand various aspects of data aggregation, and thus to aid the design of systems applying data aggregation. Gray et al. [8] and Madden et al. [9] have proposed taxonomies, of which the main purposes are for helping the understanding and design of aggregate functions. Fasolo et al. [10] propose a taxonomy to reason about aggregate functions and routing protocols for systems applying in-network aggregation. Our work, compared to theirs, aim to design a system based on analysis of the characteristics of the data, the aggregate functions and the processes, which are all covered by DAGGTAX.

Existing modeling notations such as UML activities can be used for structural representation of data flows and processes [11]. However, UML has many flavors of semantics requiring non-trivial transformations in order to perform formal analysis. Our taxonomy is presented as a feature diagram, which has

formal semantics in Boolean logic, and thus enables easy feasibility checks with a SAT solver.

9.6 Conclusion and Future Work

In this paper we present an industrial case study, in which we apply DAGGTAX to design a cloud monitoring system for an enhanced auto-scaling functionality in a cloud video streaming system. We analyze the current framework based on the features of the DAPs, and design new DAPs by selecting and composing features from DAGGTAX. The application of DAGGTAX promotes a deeper understanding of the systems behavior, and raises awareness about characteristics that need to be considered as well as issues that need to be solved during the design. It helps designers to perform better analysis than otherwise, such as to identify reusable design solutions, make data management decisions, eliminate infeasible feature combinations, and calculate time-related parameters. It also holds potential for full formal reasoning on the feasibility of the design and consistency of the decisions. Although we have only demonstrated the benefits of DAGGTAX on a cloud monitoring system in this paper, we believe that these benefits also apply to the design of other data-intensive systems with multi-levels of data aggregation.

In the future we plan to integrate DAGGTAX with state-of-art architectural and process modeling languages, so that designers can analyze other crucial properties, such as logical data consistency, in the context of data aggregation. Another future direction is a tool based on DAGGTAX that supports automatic feature selection and combination to create new data aggregation processes, as well as the evaluation with respect to the efficiency of the approach.

Acknowledgements This work is funded by the Knowledge Foundation of Sweden (KK-stiftelsen) within the DAGGERS project.

Bibliography

[1] Ericsson. 5g systems enabling the transformation of industry and society. Technical report, January 2017.

[2] Jonathan Stuart Ward and Adam Barker. Observing the clouds: a survey and taxonomy of cloud monitoring. *Journal of Cloud Computing*, 3(1):1, 2014.

[3] Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu. Daggtax: A taxonomy of data aggregation processes. Technical report, 2016.

[4] Imre J Rudas, Endre Pap, and János Fodor. Information aggregation in intelligent systems: An application oriented approach. *Knowledge-Based Systems*, 38:3–13, 2013.

[5] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[6] D. Bruneo, F. Longo, and C. C. Marquezan. A framework for the 3-d cloud monitoring based on data stream generation and analysis. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 62–70, May 2015.

[7] Jess Montes, Alberto Snchez, Bunjamin Memishi, Mara S. Prez, and Gabriel Antoniu. Gmone: A complete approach to cloud monitoring. *Future Generation Computer Systems*, 29(8):2026 – 2040, 2013.

[8] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube:

A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[9] Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[10] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *Wireless Communications, IEEE*, 14(2):70–87, 2007.

[11] Harald Störrle. Semantics and verification of data flow in uml 2.0 activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35 – 52, 2005.