Integration framework for artifact-centric processes in the internet of things

Maroun Abi Assaf

N°d'ordre NNT : 2018LYSEi059

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
l'INSA Lyon

Ecole Doctorale ED512
INFOMATHS

Spécialité de doctorat : Informatique

Soutenue publiquement le 09/07/2018, par :
Maroun ABI ASSAF

Integration Framework for Artifact-centric Processes in the Internet of Things

---

Devant le jury composé de :

LAFOREST, Frédérique	Professeure (Université Jean Monet)	Présidente
VERDIER, Christine	Professeure (Université de Grenoble Alpes)	Rapporteure
LAURENT, Anne	Professeure (Université de Montpellier)	Rapporteure
CAUVET, Corine	Professeure (Université Aix-Marseille)	Examinatrice
BADR, Youakim	Maître de Conférences - HDR (INSA-Lyon)	Directeur de thèse
AMGHAR, Youssef	Professeur (INSA-Lyon)	Co-directeur de thèse
BARBAR, Kablan	Professeur (Université Libanaise)	Co-directeur de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales – Quinquennal 2016-2020

SIGLE	ECOLEDOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
CHIMIE	CHIMIE DE LYON http://www.edchimie-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage secretariat@edchimie-lyon.fr INSA : R. GOURDON	M. Stéphane DANIELE Institut de recherches sur la catalyse et l'environnement de Lyon IRCELYON-UMR 5256 Équipe CDFA 2 Avenue Albert EINSTEIN 69 626 Villeurbanne CEDEX directeur@edchimie-lyon.fr
E.E.A.	ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE http://edeea.ec-lyon.fr Sec. : M.C. HAVGOUDOUKIAN ecole-doctorale.eea@ec-lyon.fr	M. Gérard SCORLETTI École Centrale de Lyon 36 Avenue Guy DE COLLONGUE 69 134 Écully Tél : 04.72.18.60.97 Fax 04.78.43.37.17 gerard.scorletti@ec-lyon.fr
E2M2	ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION http://e2m2.universite-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : H. CHARLES secretariat.e2m2@univ-lyon1.fr	M. Philippe NORMAND UMR 5557 Lab. d'Ecologie Microbienne Université Claude Bernard Lyon 1 Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69 622 Villeurbanne CEDEX philippe.normand@univ-lyon1.fr
EDISS	INTERDISCIPLINAIRE SCIENCES-SANTÉ http://www.ediss-lyon.fr Sec. : Sylvie ROBERJOT Bât. Atrium, UCB Lyon 1 Tél : 04.72.44.83.62 INSA : M. LAGARDE secretariat.ediss@univ-lyon1.fr	Mme Emmanuelle CANET-SOULAS INSERM U1060, CarMeN lab, Univ. Lyon 1 Bâtiment IMBL 11 Avenue Jean CAPELLE INSA de Lyon 69 621 Villeurbanne Tél : 04.72.68.49.09 Fax : 04.72.68.49.16 emmanuelle.canet@univ-lyon1.fr
INFOMATHS	INFORMATIQUE ET MATHÉMATIQUES http://edinfomaths.universite-lyon.fr Sec. : Renée EL MELHEM Bât. Blaise PASCAL, 3e étage Tél : 04.72.43.80.46 Fax : 04.72.43.16.87 infomaths@univ-lyon1.fr	M. Luca ZAMBONI Bât. Braconnier 43 Boulevard du 11 novembre 1918 69 622 Villeurbanne CEDEX Tél : 04.26.23.45.52 zamboni@maths.univ-lyon1.fr
Matériaux	MATÉRIAUX DE LYON http://ed34.universite-lyon.fr Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction ed.materiaux@insa-lyon.fr	M. Jean-Yves BUFFIÈRE INSA de Lyon MATEIS - Bât. Saint-Exupéry 7 Avenue Jean CAPELLE 69 621 Villeurbanne CEDEX Tél : 04.72.43.71.70 Fax : 04.72.43.85.28 jean-yves.buffiere@insa-lyon.fr
MEGA	MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE http://edmega.universite-lyon.fr Sec. : Marion COMBE Tél : 04.72.43.71.70 Fax : 04.72.43.87.12 Bât. Direction mega@insa-lyon.fr	M. Jocelyn BONJOUR INSA de Lyon Laboratoire CETHIL Bâtiment Sadi-Carnot 9, rue de la Physique 69 621 Villeurbanne CEDEX jocelyn.bonjour@insa-lyon.fr
ScSo	ScSo* http://ed483.univ-lyon2.fr Sec. : Viviane POLSINELLI Brigitte DUBOIS INSA : J.Y. TOUSSAINT Tél : 04.78.69.72.76 viviane.polsinelli@univ-lyon2.fr	M. Christian MONTES Université Lyon 2 86 Rue Pasteur 69 365 Lyon CEDEX 07 christian.montes@univ-lyon2.fr

*ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

A mes parents
A tous ceux qui me sont Chers

Acknowledgements

The four years and five months that lasted my Ph.D. were extremely intense and rich. I had never worked so hard before this period, but I have gained a lot of experience and maturity. I would like to use this page to thank all those who have helped and supported me during this period.

First of all, I would like to thank my main supervisor, Prof. Youakim Badr for his unreserved support, for the endless hours he devoted to turn me into a researcher, for all the amazing discussions we had on scientific and non-scientific topics during these years.

I would also like to express my deep gratitude towards my other supervisors, Prof. Kablan Barbar and Prof. Youssef Amghar for giving me this once in a lifetime opportunity and for sharing their deep knowledge and insights with me.

Last but not least, I thank my family and friends for their unconditional moral and physical support during my Ph.D.

Abstract

The emergence of fixed or mobile communicating objects poses many challenges regarding their integration into business processes in order to develop smart services. In the context of the *Internet of Things*, connected devices are heterogeneous and dynamic entities that encompass cyber-physical features and properties and interact through different communication protocols. To overcome the challenges related to interoperability and integration, it is essential to build a unified and logical view of different connected devices in order to define a set of languages, tools and architectures allowing their integrations and manipulations at a large scale. Business artifact has recently emerged as an autonomous (business) object model that encapsulates attribute-value pairs, a set of services manipulating its attribute data, and a state-based lifecycle. The lifecycle represents the behavior of the object and its evolution through its different states in order to achieve its business objective. Modeling connected devices and smart objects as an extended business artifact allows us to build an intuitive paradigm to easily express integration data-driven processes of connected objects. In order to handle contextual changes and reusability of connected devices in different applications, data-driven processes (or artifact processes in the broad sense) remain relatively invariant as their data structures do not change. However, service-centric or activity-based processes often require changes in their execution flows.

This thesis proposes a framework for integrating artifact-centric processes and their application to connected devices. To this end, we introduce a logical and unified view of a "global" artifact allowing the specification, definition and interrogation of a very large number of distributed artifacts, with similar functionalities (smart homes or connected cars, ...). The framework includes a conceptual modeling method for artifact-centric processes, inter-artifact mapping algorithms, and artifact definition and manipulation algebra. A declarative language, called *AQL (Artifact Query Language)* aims in particular to query continuous streams of artifacts. The *AQL* relies on a syntax similar to the *SQL* in relational databases in order to reduce its learning curve. We have also developed a prototype to validate our contributions and conducted experimentations in the context of the *Internet of Things*.

Keywords: Business Process Modeling and Merging – Query Languages – Data Integration – Smart Processes – Internet of Things

Résumé

La démocratisation des objets communicants fixes ou mobiles pose de nombreux défis concernant leur intégration dans des processus métiers afin de développer des services intelligents. Dans le contexte de l'*Internet* des objets, les objets connectés sont des entités hétérogènes et dynamiques qui englobent des fonctionnalités et propriétés cyber-physiques et interagissent via différents protocoles de communication. Pour pallier aux défis d'interopérabilité et d'intégration, il est primordial d'avoir une vue unifiée et logique des différents objets connectés afin de définir un ensemble de langages, outils et architectures permettant leur intégration et manipulation à grande échelle.

L'artéfact métier a récemment émergé comme un modèle d'objet (métier) autonome qui encapsule ses données, un ensemble de services, et manipulant ses données ainsi qu'un cycle de vie à base d'états. Le cycle de vie désigne le comportement de l'objet et son évolution à travers ses différents états pour atteindre son objectif métier. La modélisation des objets connectés sous forme d'artéfact métier étendu nous permet de construire un paradigme intuitif pour exprimer facilement des processus d'intégration d'objets connectés dirigés par leurs données. Face aux changements contextuels et à la réutilisation des objets connectés dans différentes applications, les processus dirigés par les données, (appelés aussi « artifacts » au sens large) restent relativement invariants vu que leurs structures de données ne changent pas. Or, les processus centrés sur les services requièrent souvent des changements dans leurs flux d'exécution.

Cette thèse propose un cadre d'intégration de processus centré sur les artifacts et leur application aux objets connectés. Pour cela, nous avons construit une vue logique unifiée et globale d'artéfact permettant de spécifier, définir et interroger un très grand nombre d'artifacts distribués, ayant des fonctionnalités similaires (maisons intelligentes ou voitures connectées, …). Le cadre d'intégration comprend une méthode de modélisation conceptuelle des processus centrés artifacts, des des algorithmes d'appariement inter-artifacts et une algèbre de définition et de manipulation d'artifacts. Le langage déclaratif, appelé *AQL (Artifact Query Language)* permet en particulier d'interroger des flux continus d'artifacts. Il s'appuie sur une syntaxe de type *SQL* pour réduire les efforts d'apprentissage. Nous avons également développé un prototype

pour valider nos contributions et mener des expériences dans le contexte de l'*Internet* des objets.

Mots-Clés: Modélisation et fusion de processus métier - Langages de requête - Intégration de données - Processus intelligents - Internet des objets

Contents

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

List of Tables

List of Figures

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

List of Definitions

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

List of Examples

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

1

Introduction

Chapter Outline

1.1. Context

Artifact-centric process modeling is a *Business Process Modeling* approach that seeks to explicitly unify data and process, and consequently eliminates the dichotomy that has separated the *Database* and the *Business Process Management* communities.

Artifact-centric processes were first introduced by *IBM* research labs in 2003 [NiCa03]. The artifact-centric approach, rather than *Relation Schema* modeling in *Databases* [AbHV95], or *Workflow* modeling in *Business Process Management* [DTKB03], combines both data and process into self-contained entities, known as *Artifacts* that serve as the basic building blocks from which models of (business) processes are constructed.

In general, an artifact-centric process referred to as an *Artifact System* [BGHL07] is formed from three main components:

1) *Artifact Classes* including *Information Models* for data related to the artifacts and state-based *Lifecycles* describing possible stages,
2) *Services* the basic units of work that operate on Artifacts, and
3) (Business) *Rules* describing the possible ways that *Services* can be invoked on *Artifacts* by following transitions between states of their *Lifecycles*.

An *Artifact System* is thus a blend of data and process about dynamic entities that capture their end-to-end journeys and evolve according to specified lifecycles in order to achieve particular goals.

Figure 1.1 illustrates an example of an *Artifact System* about an *Order* artifact. The *Information Model* of the *Order Artifact* class has *data attributes* for registering information about the order id, product, quantity, client, shipment address, whether the product is available in stock, date retrieved, and delivery date. The *Lifecycle* includes *states* for representing the different stages of an *Order Artifact* including; *Created*, *NotAvailable*, *Available*, *Retrieved*, and *Delivered*. The list of *Services* acting on *Order Artifact* includes:

1) *Create Order*: creates a new *Order Artifact* instance and registers necessary information.
2) *Check Availability*: checks if the requested product and quantity are available in stock.

3) *Retrieve Product*: retrieves the product from the stock. And,

4) *Deliver Product*: delivers the product to the shipment address.

Rules are declarative *ECA (Event-Condition-Action)* rules that are represented as arrows in Figure 1.1. They are responsible for invoking *Services* and changing the state of *Artifact* instances.

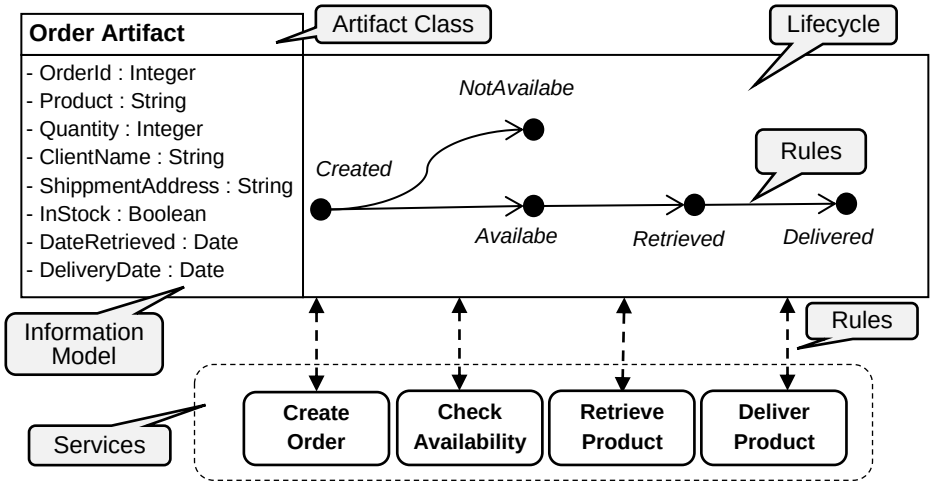


Figure 1.1 Artifact System example

By leveraging process models into a semantic level, artifact-centric processes provide an intuitive and flexible framework for executing and managing data-driven processes. As reported in [CoHu09, Hull08], the artifact-centric approach has successfully been applied to process management and case handling, and has demonstrated many advantages such as enabling a natural modularity and componentization of processes, supporting process transformations and changes, providing a framework of varying levels of abstraction, and understanding the interplay between data and process in ways not supported by previous *Computer Science* abstractions. As a result, end-users can manage, control, and transform artifact-centric processes from day to day with minimal to no intervention from *IT* specialists and experts.

Over the last few years, artifact-centric processes have proliferated at a phenomenal pace with the wide range of promising applications including finance, monitoring, and virtual organization. Yet another promising application of artifacts is the *Internet of Things (IoT)* in which smart objects link networks of sensors and actuators. In this context, smart objects can be modeled as self-evolving artifacts gathering data streams from various sensors, detecting complex events, and performing actions on actuators. The *Internet of Things*, where numerous connected devices are integrated with

Internet-based protocols in order to build high-level business services, can be architecturally divided into three layers as illustrated in Figure 1.2:

1) *Devices layer*: This is the lowest-level layer, which consists of a set of sensors and actuators interacting with their physical environment.
2) *Gateway layer*: This is the intermediate layer, which is able to provide a unified access point to the variety of connected objects.
3) *Business services layer*: The *Internet of Things* presents considerable business opportunities, not only from a device manufacturing perspective, but also from a business perspective through business services and high-level applications.

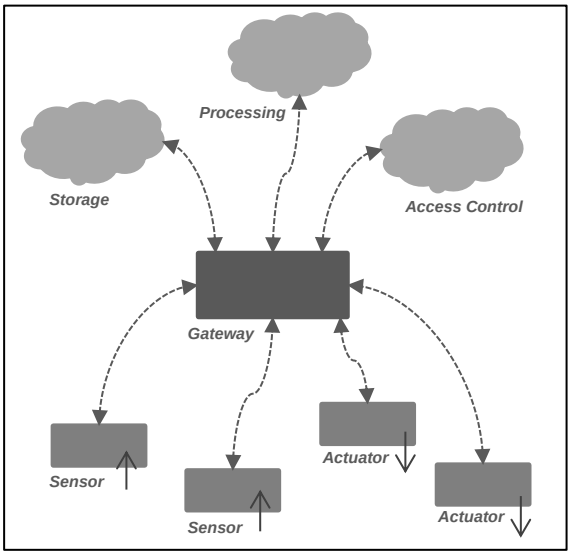


Figure 1.2 Internet of Things architecture

The artifact-centric approach can provide an abstraction over the three layer architecture of *Internet of Things* and its various components. Through the use of *Artifact Classes*, *Services*, and *Rules*, an *Artifact System* can represent all the low-level components of *Internet of Things* including *sensors*, *actuators*, *storage*, *processing*, *access control*, and *gateways*.

As a result, the artifact-centric approach demonstrates many advantages and benefits including; a natural modularity and componentization of self-contained entities and a framework of varying levels of abstraction in order to develop goal-oriented components instead of function-oriented components in the case of *Web Services*.

1.2. Problem Description

Many of today's businesses are formed by mergers and acquisitions with other businesses and as a result, business people have to deal with a number of heterogeneous *Business Processes* and *Databases* performing similar or different functionalities (i.e. manufacturing processes, sale processes) [PaSp00]. The same situation is applicable to the *Internet of Things* in which a large number of connected devices require to merge their data or provide a *Unified View* in order to be easily managed in a simple way rather than dealing with a large number of rows in distributed *Database* tables.

As described in [Lenz02], a convenient approach to manage heterogeneous processes and *Databases* consists of using a *Unified View* that centralizes the access to information and tasks available in these processes. A *Unified View* is a virtual process or data model that can be uniformly used to supervise, execute, and interrogate heterogeneous distributed processes and data entities without dealing with their differences and complexities. As a result, a uniform query based on a *Unified View* is transformed using mapping rules into the corresponding heterogeneous queries of the distributed data entities (i.e., *Databases*) or processes. Result sets of the heterogeneous queries are then transformed and merged using the mapping rules into a uniform result set compatible with the *Unified View*. The benefits of using *Unified Views* are managing huge number of entities at a large scale, facilitating evaluation and analysis of their data and behavior, and providing a centralized access point for administrators and casual users. Figure 1.3 illustrates the integration mechanisms.

Since artifact-centric processes have emerged as a new modeling paradigm and provided interesting applications in the context of *Internet of Things* to model artifact-based connected devices, this thesis' work focuses on the problem of integrating heterogeneous artifact-centric processes. Integrating artifact-centric processes raises an acute problem because of the complexity of matching and mapping two or more artifacts at the level of their components (i.e. *Information Models*, *Lifecycles*, *Services*, and *Rules*). And as a result, traditional data integration and process merging solutions and techniques like [ChTr12, KuYY14, PaSp00] fail to address the complexity of artifact-centric process integration.

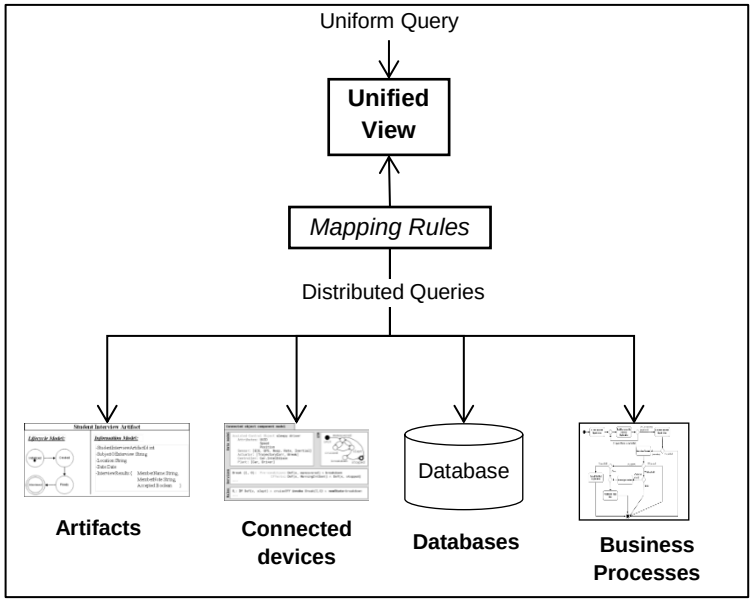


Figure 1.3. Integration mechanism

Moreover, given an artifact-based connected device, different variants can exist to handle different usages in different application domains. Variations may occur in the *Information Model* (semi-structured data), *States* (i.e. new intermediary states), *Services* (i.e. new services or same services with different signatures ...), and *Rules* (specifically tailored to an application domain).

Variants of artifacts consequently lead to heterogeneous artifact-centric processes. As a result, the integration of artifact-centric processes from different sources becomes a major challenge when we need to provide *Unified Views* for managing, querying and executing very large number of artifacts distributed across the *Internet of Things*.

Since artifact-centric processes combine three main components; *Artifact Classes*, *Services* and *Rules*, the integration problem of two or more artifacts requires simultaneously the integration of their *Information Models*, *Lifecycles*, *Services* and *Rules*. However, integration problems have been extensively investigated in disciplines such as *Data Integration*, *Databases*, *Business Process Merging* but the complexity and richness of artifact structures requires specialized integration semantics and approaches.

The challenges facing artifact-centric process integration can be classified into four different levels:

1) **Integration Semantics Level:** *Artifact Systems* combine both process and data aspects into three components; *Artifact Classes* including *Information Models* and *Lifecycles*, *Services*, and *Rules*. As a result, specialized

integration semantics that address these three components should be defined. Moreover, these integration semantics should support different kinds of semantic relationships (i.e. unique, equivalent, and composition) between elements of the different *Artifact Systems*' components.

2) **Conceptual Artifact Model Level:** In order to define effective integration semantics, *Artifact Systems* should be represented using conceptual models that capture their three components. These conceptual models should not only be simple, intuitive and holistic but can also be used to generate working *Artifact Systems*.
3) **Artifact-specific Language Level:** In order to effectively create, execute, manipulate, and interrogate *Artifact Systems*, artifact-specific languages should exist to specifically target artifacts and take full advantage of their semantic nature. Additionally, Artifact-specific languages should be used in order to interrogate generated *Unified Views*, and,
4) **Extended Artifact Level:** Since artifacts are mainly applied to traditional business processes, artifacts require to be extended with data stream capabilities in order to support modern *IoT-based* processes.

1.3. Contributions

In this thesis, we focus on the problem of artifact-centric process integration in the context of the *Internet of Things* through the representation of *Artifact Systems,* using conceptual models and merging of conceptual models according to correspondence relationships between different artifact component elements.

To this end, we propose a global artifact-centric process integration framework as illustrated in Figure 1.4. The integration framework is based on four main phases: *Modeling*, *Specification*, *Integration*, and *Execution*.

In the *Modeling Phase*, we model *Artifact Systems* using conceptual models that we refer to as *Conceptual Artifact Models (CAMs)*. We propose the *Conceptual Artifact Modeling Notations (CAMN)* as minimalistic graphical notations that we use in order to model *CAMs*. *CAMs* are not only characterized by containing all required information for generating *Artifact Systems*, but they also form the basis of the integration and generation of *Unified Views of heterogeneous artifacts*.

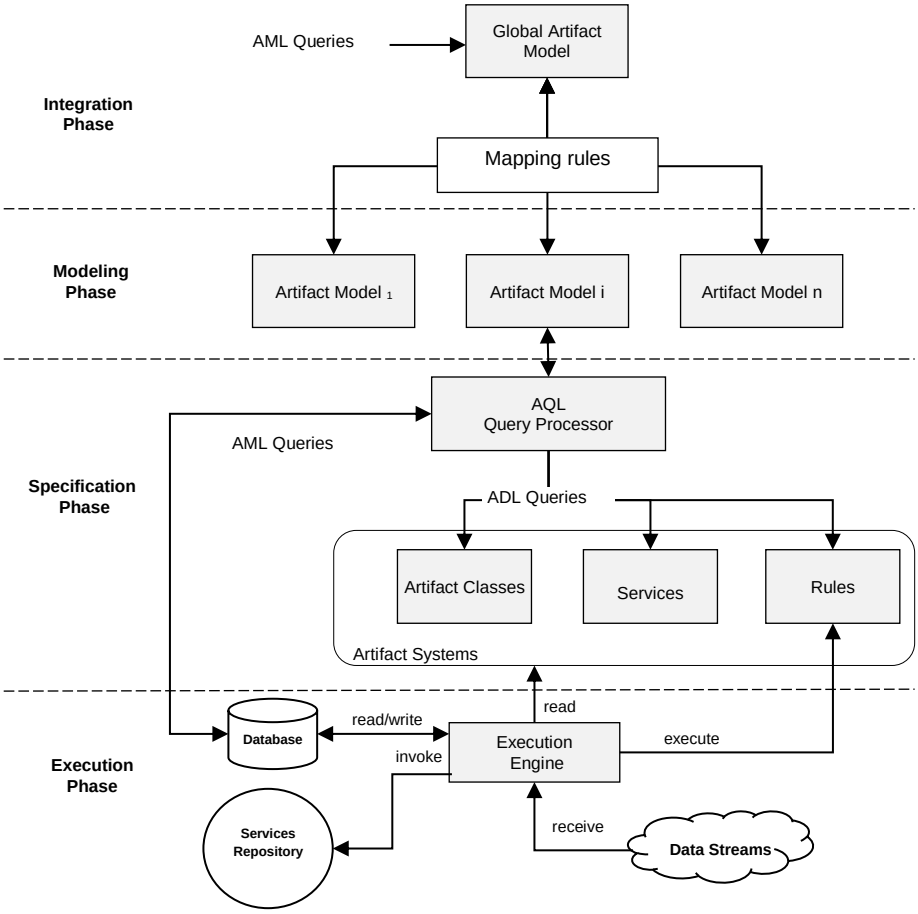


Figure 1.4. Artifact-centric process integration framework

In the *Specification Phase*, we generate *Artifact Systems* from *CAMs* that are modeled in the modeling phase. We also propose the *Artifact Query language (AQL),* an artifact-specific language, that we use in order to express and implement *Artifact Systems*. The *AQL* is a high-level declarative language similar to the *SQL* and specifically targets artifacts taking full advantage of their semantic and data structutres. The *AQL* is made of two parts: the *Artifact Definition Language (ADL)*, and the *Artifact Manipulation Language (AML)*. The *ADL* contains statements to define *Artifact Classes*, *Services*, and *Rules*. The *AML* contains statements to instantiate, manipulate, and interrogate artifact instances. Moreover, the *AQL* supports *Data Streams* and *Continuous Query* capabilities and allows *Complex Event Processing (CEP)* over data streams through the use of *Artifact Rules*.

In the *Integration Phase*, we integrate several local *CAMs* in order to generate one global *CAM* that is used as a *Unified View of heterogeneous artifacts*. We propose semantic-based integration based on:

a) The identification of three types of correspondence relationships (*unique*, *equivalent*, and *composition*) between different elements of *CAMs*,

b) The generation of a global *CAM* by merging local *CAMs* based on the identified correspondences, and

c) The specification of mapping rules in order to translate *AML* queries and data between the global and local *CAMs* with regards to the type of *CAM* elements.

Finally, in the *Execution Phase*, we execute *Artifact Systems* using an execution engine based on translating *AQL* queries into *semantic* queries. *Semantic* queries are then executed on a *Database Management System* in order to perform relational and stream operations. The execution engine is also responsible for invoking *Services*.

We validate our artifact-centric process integration framework by developing a prototype, consisting of several modules and graphical user interfaces.

1.4. Document Organization

The remainder of the document is organized as follow. Chapter two is a survey of related works. Chapter three defines the *Artifact System* and presents the *Artifact Query Language (AQL)*. Chapter four presents the *Conceptual Artifact Modeling Notations (CAMN)* and defines *Conceptual Artifact Models (CAMs)*. It also describes the semantics of generating *Artifact Systems* from *CAMs*. Chapter five defines integration semantics of *CAMs*. Chapter six describes the prototype implementation. Finally, chapter seven concludes our contributions with new research perspectives and open research issues.

2

Related Works

Chapter Outline

In this thesis, we treat problems related to *Business Process Modeling*. In particular, we focus on modeling, execution, and integration of artifact-centric *Business Processes* and their applications to modern smart processes and services.

In this chapter, we present the state of the art of existing works and compare them with regard to our research problem. The reviewed research fields and domains include: *Business Process Modeling, Data Integration*, *Business Process Merging*, and *Query Languages*.

In Section 2.1, we start with a brief introduction of *Business Process Models* that puts into perspective *activity-centric* and *artifact-centric* approaches.

In Section 2.2, we make an overview of activity-centric *Business Process Modeling*. Moreover, we illustrate advantages and disadvantages of the *activity-centric* approach in contrast to the *artifact-centric* approach.

In Section 2.3, we present *artifact-centric Business Process Modeling*. We describe the *artifact-centric* approach and its existing formal models. We also make a comparison between existing formal models and the proposed formal model illustrating the advantages of our proposition.

In Section 2.4, we describe existing artifact modeling notations and framework and illustrate their advantages and disadvantages. We then make a comparison between existing artifact modeling notations and frameworks and the proposed artifact modeling notations and framework.

In Section 2.5, we investigate *data integration* and show the need for an alternative approach to integrate artifact-centric processes.

In Section 2.6, we describe *Business Process Merging and Views* approaches. Similarly to *Data Integration*, we demonstrate the need for an alternative approach to integrate artifact-centric processes.

In Section 2.7, we describe existing *Query Languages* including one-time and continuous query languages. We also compare existing query languages to the proposed query language while illustrating its advantages.

2.1. Business Process Models

Business Process is the way an organization conducts its business in order to achieve its business goals. As stated in [Ritt04], a *Business Process* is one of the first things to consider when organizations need to improve their effectiveness and efficiency. Not having a well-defined and standardized

Business Process, may lead an organization to unfavorable consequences since the choice of how to conduct the business is left for each employee to make.

The *Business Process Model (BPM)* is an abstract representation of the *Business Process* of an organization. This abstract representation is most often a graphical representation that consists of a set of interconnected modeling primitives. As described in [LiBW07], a *Business Process Model* allows the analysis of the *Business Process* and the reasoning about how to conduct it in the most efficient and effective manner.

Two major approaches to modeling *Business Processes* exist:

1. The *Activity-centric* approach in which *Business Processes* are modeled as sequences of *Tasks* or *Activities* [DuTe01]. And,
2. The *Artifact-centric* approach in which *Business Processes* are modeled based on semantic entities referred to as *Business Artifacts* [NiCa03].

[EsWi03, TrAS08, VTKB03] describe three major aspects that are involved in any *Business Process Model*:

1. The *process aspect (Control-Flow)* describes *Tasks* or *Activities,* and their logical order of execution.
2. The *data aspect (Data-Flow)* describes the evolution of data in the *Business Process*.
3. The *resource aspect* concerns the organizational structure and describes roles that are responsible for executing tasks.

In the activity-centric *Business Process Modeling*, the *process aspect* is fundamental and vital to the *Business Process Model* and represents its core [VTKB03]. On the other hand, *data* and *resource aspects* are secondary and are layered on top of the *process aspect* and provide it with necessary support. In this approach, models describes actions that need to be performed by business actors (human or system) using resources of an organization and their logical order of execution in order to achieve business goals [LiBW07].

On the other hand, in artifact-centric *Business Process Modeling* both *process* and *data aspects* are involved from the beginning in the *Business Process Model*. The main building blocks of this approach are *Business Artifacts* [NiCa03], combining both process and data aspects of a *Business Process*. In this approach, *Business Process Models* are built from *Lifecycles* of *Business Artifacts*.

As described in [CoHu09], *activity-centric Business Process Models* have been found to have many disadvantages especially when applied in situations that require dynamic modelling and transformations like in knowledge-driven *Business Processes*. In this kind of situations where the *Control-Flow* is not known before executing the process, an artifact-centric approach is more flexible and suitable than an activity-centric approach.

2.2. Activity-Centric Business Process Modeling

In activity-centric *Business Process Modeling*, the core components of the process model comprise *Activities,* and *Control-Flows*. *Activities* are the units of work that represent single logical steps within a *Business Process.* An *Activity* may be performed by a human actor or an automated system. *Control-Flows* are used to connect *Activities* together to form logical steps of a *Business Process*.

Activity-centric Business Process Models can be supported by a *Workflow Management System (WFMS)* in order to automate suitable parts of the *Business Process* as described in [Ritt04]. In this case, the *Business Process Model* is referred to as a *Workflow* model. An overview of *Workflow Management* is described in [GeHS95] where a *Workflow* model can be read, executed, and controlled by a *Workflow Management System (WFMS)*.

Different modeling languages and notations related to *Business Process* and *Workflow* modeling exist. For examples, *UML Activity Diagrams* [DuTe01], *Business Process Model and Notation (BPMN)* [Fort09], and *Yet Another Workflow Language (YAWL)* [VaTe05] are the most common and used modeling languages and notations. Moreover, these modeling languages and notations are interchangeably used by *business analysts* and *IT specialists*.

From an analysis perspective, [VTKB03] performs a study on *Workflow* patterns. In this study, twenty six *Workflow* patterns are identified and described. Moreover, [VTKB03] concludes that current *Workflow Management Systems (WFMS)* can only support a subset of the twenty six workflow patterns. related work such as [MSMP05, VaVa04] describe four major *Workflow* patterns that are involved in any *Workflow* including *Sequence*, *Parallel Path*, *Alternative Path*, and *Iteration* where:

1. *Sequence* pattern represents several *Activities* that should be executed in order, one after the other.

2. *Parallel Path* pattern represents several *Activities* that can be executed at the same time or in no particular order.
3. *Alternative Path* pattern path represents the decision of executing one or more *Activities* from several *Activities*.
4. *Iteration* pattern is when several *Activities* must be repeatedly executed until a condition is met.

From the Unified Markup Language (*UML)* perspective, [DuTe01, Ritt04] describe the use of *UML Activity Diagrams* in modeling *Workflows*. The *UML Activity Diagram* [StHa04] provides basic constructs that can be combined in order to model the major patterns of *Workflows*. Figure 2.1 illustrates the *UML Activity Diagram* constructs that includes: *Start*, *End*, *Fork/Join*, *Decision/Merge*, *Activity* and *Activity Edge.* [StHa04, StHa05] define formal semantics for *UML Activity Diagrams*. While [EsWi03] makes a comparison between *Petri Nets* and *UML Activity Diagrams*. On the other hand, [MSMP05] presents an approach to the modeling and analyzing of *Business Processes* based on *UML Activity Diagrams* and *Petri Nets* where;

1. *Sequence* patterns are modeled using *Activities* and *Activity Edges*.
2. *Parallel Path* patterns are modeled using *Fork/Join* constructs.
3. *Alternative Path* patterns are modeled using *Decision/Merge* constructs.
4. *Iteration* patterns are modeled using a combination of *Decision/Merge* and *Activity Edges* constructs. Finally,
5. *Start* and *End* constructs are respectively used to denote the start and end of the *Business Process*.

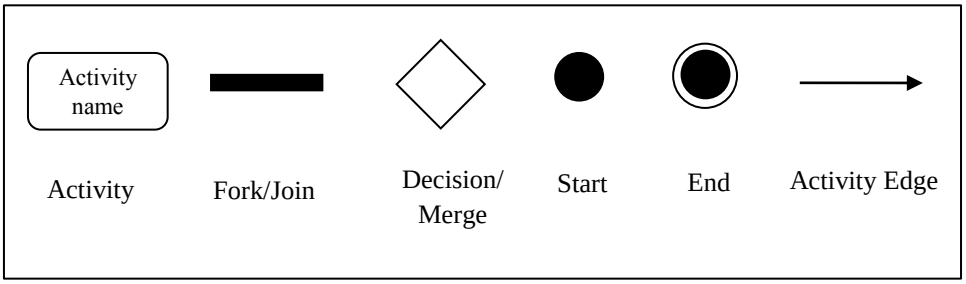


Figure 2.1 UML Activity Diagram modeling constructs

Figure 2.2 illustrates an example of a *Workflow* model about an order process modeled as *UML Activity Diagram*. After creating an order, it is checked for availability. If it is available, the product is retrieved and delivered. Otherwise, the process terminates.

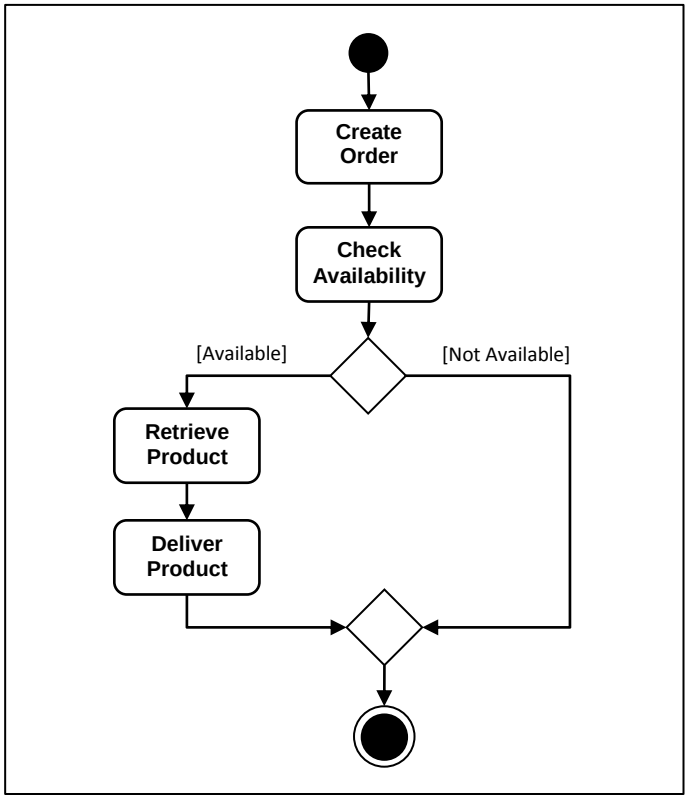


Figure 2.2 UML Activity Diagram example

From a different perspective, [Fort09] describes the *Business Process Modeling Notation (BPMN)*. While [VaTe05] describes the *Yet Another Workflow Language (YAWL)*. Works such as [Whit04] and [FoFo09] illustrate and compare how to model *Workflow* patterns described in [VTKB03] using *UML Activity Diagrams*, *BPMN*, or *YAWL*. They conclude that the three modeling notations can be adequately used to model most *Workflow* patterns. Similarities between the three modeling notations exist and are due to the fact that they are designed to solve the same basic problem; the diagramming of procedural business processes. Moreover, differences between the three modeling notations also exist and are due to the target users and goals of the modeling notations; *BPMN* was created as a graphical notation for business people to use. *UML* was created in order to standardize modeling for software development. *YAWL* was created in order to provide comprehensive support for *Workflow* patterns and to design executable *Workflow* models.

In summary, activity-centric *Business Process Modeling* focuses on modeling *Business Processes* based on *Activities*. An activity-centric *Business Process Model* tends to be easier to be developed than an artifact-centric *Business Process Model* since key *Activities* in a *Business Process*

can be easily identified. Moreover, business logics are defined explicitly using *Control-Flows* which result in an explicit process model which contributes towards *Business Process* awareness in the organization. However, data are incorporated at a limited level as inputs and outputs of *Activities* which limits the understanding of possible effects of processing steps on key business entities [NgOt13]. Moreover, the explicit definition of *Control-Flows* is not suitable in situations that require dynamic modelling and transformations like in knowledge-driven *Business Processes* that are characterized by not having predefined *Control-Flows.* Instead, skilled and knowledgeable workers decide the best course of action according to each case like in the healthcare domain. In this kind of processes, an artifact-centric approach is more flexible and suitable than the activity-centric approach as described in [MaHV12, Whit09].

2.3. Artifact-Centric Business Process Modeling

In artifact-centric *Business Process Models*, the *data aspect* of *Business Processes* is involved from the beginning as opposed to the activity-centric approach that leaves it to later stages as input and output of tasks.

In this approach, *data* and *process* aspects are combined into semantic entities referred to as *Business Artifacts* [NiCa03]. A *Business Artifact* is composed from an *Information Model* and a *Lifecycle*. The *Information Model* is a set of attribute/value pairs representing business related data and objects. The *Lifecycle* describes the possible stages that a *Business Artifact* can pass through during the *Business Process*. *Business Artifacts* are used by business people in order to record and track progress toward completing business goals.

The concept of using *Business Artifacts* as building blocks for *Business Processes Models* was first introduced in [NiCa03] and was further described in [BhHS09, CoHu09, Hull08]. In this approach, *Lifecycles* of *Business Artifacts* are represented using finite state machines. *Lifecycle* of a *Business Artifact* can interact with the *Lifecycles* of other *Business Artifacts* in the *Business Process*. A *Business Process Model* is formed from all *Lifecycles* of *Business Artifacts* involved in the *Business Process*.

In [Hull08], a framework for analyzing and working with artifact-centric *Business Process Models* is presented. The framework, referred to as the *BALSA* framework, describes four dimensions that are involved in any artifact-centric *Business Process Model,* including: *Business Artifact*

(Information Model), *Lifecycle*, *Services*, and *Associations*. Figure 2.3 illustrates an example of the *BALSA* framework representation of an *Order* artifact.

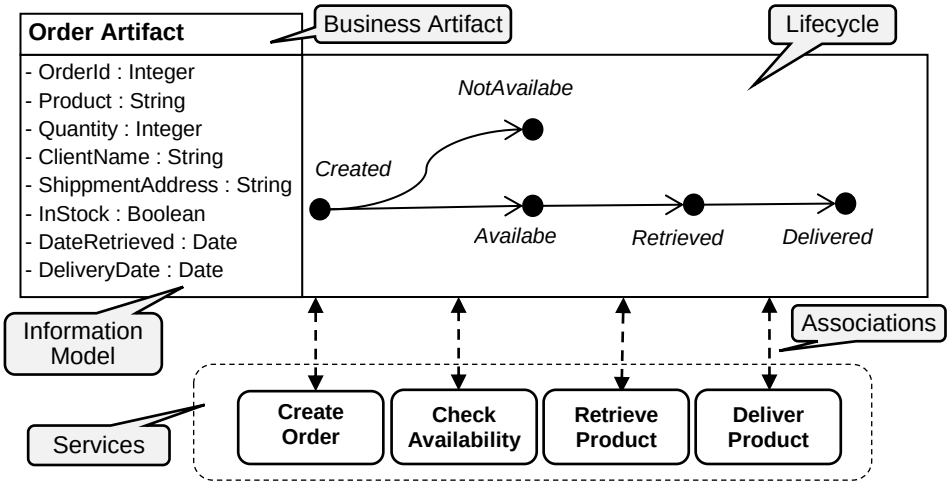


Figure 2.3 BALSA Framework example

By varying the implementations of the four dimensions of the *BALSA* framework, different kinds of artifact-centric *Business Process Models* ranging from procedural to declarative can be constructed. A procedural model describes *"how"* processing should be done in a step-by-step manner using a finite-state machine. While a declarative model describes *"what"* should be done using statements in a particular language.

In [GeBS07, GeSu07], a formal model for procedural artifact-centric *Business Process Models* is defined. In this formal model, *Associations* and *Lifecycles* are represented using finite-state machines composed from *Tasks*, *Repositories*, and *Flow Connectors* where:

1. *Tasks* represent units of work that operate and update *Information Models* of *Business Artifacts*.
2. *Repositories* represent storage locations for *Business Artifacts* awaiting future processing if any. And,
3. *Flow Connectors* transport *Business Artifacts* between *Tasks* and *Repositories*.

Figure 2.3 illustrates the *Lifecycle* of the *Order* artifact represented as a finite-state machine composed from *Tasks*, *Repositories*, and *Flow Connectors*. Moreover, the problem of verification of procedural artifact-centric *Business Process Models* is studied in [GeSu07] where a language for specifying and verifying artifact lifecycle behaviors is presented.

Transformation between activity-centric and artifact-centric *Business Process Models* is performed in [KuLW08, MeWe13].

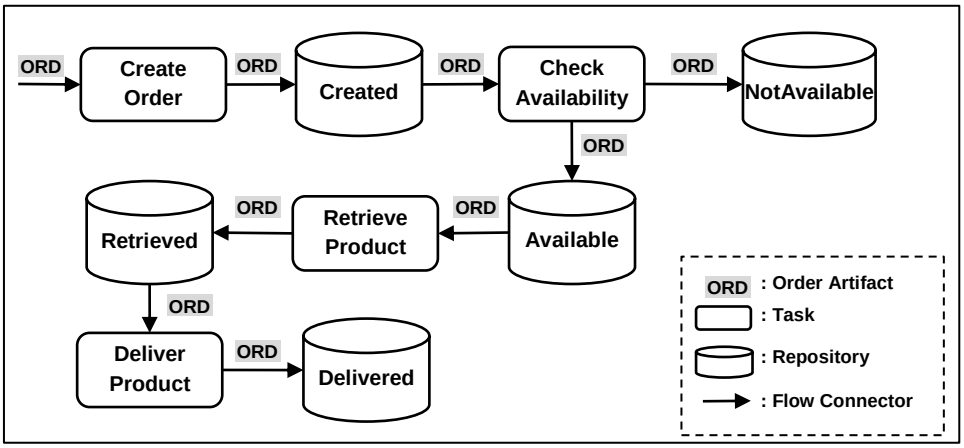


Figure 2.4 Lifecycle of Order artifact as Tasks, Repositories and Flow Connector

On the other hand, [BGHL07] defines a formal model for declarative artifact-centric *Business Process Models*. In this formal model, *Associations* are represented as declarative *Business Rules*. *Services* are also declaratively represented as *Semantic Web Services* specifications. *Business Rules*, as variants of *Event-Condition-Action (ECA) Rules* describe conditions under which actions should be performed. Conditions involve *Information Models* and *Lifecycles'* states. Actions invoke *Services*, or change *Lifecycles'* states. The problem of verification of correctness properties for declarative artifact-centric *Business Process Models* is studied in [DHPV09, HCDD11].

A formal model for artifact-centric *Business Process Models* based on declarative *Guard-Stage-Milestones (GSM) Lifecycles* was introduced and defined [DaHV13, HDDF11a, HDDF11b]. In this formal model, *Lifecycles* are represented using *Guards, Stages*, and *Milestones*. *Milestones* correspond to business objectives that a *Business Artifact* might achieve. *Stages* correspond to collections of *Tasks* that are intended to achieve *Milestones*. And, *Guards* control when *Stages* can be opened for execution. The *GSM* provides a declarative model that supports parallelism and hierarchies in *Business Artifact*'s lifecycles. The problem of verification of *GSM* models based on symbolic model checking was studied in [GoGL12]. In [PoDu12, PoFD15], *Petri Net Lifecycles* generated from event logs are transformed into *GSM* models.

In summary, three different formal models for artifact-centric *Business Process Models* exist. The first formal model is based on procedural finite state machines. The second formal model is based on declarative *Business Rules*. And, the third formal model is based on declarative *GSM Lifecycles*.

Moreover, a declarative model is better suited than a procedural model in the context of *Business Process* transformation and customization. In this case, adding declarative rules or editing existing declarative rules is simpler than editing, recompiling, and deploying of a finite-state machine. As a result, the formal model we propose in this thesis is based on declarative *Business Rules* we refer to as *Artifact Rules*.

On the other hand, existing artifact formal models lack the support for data streams generated by sensors, and actuators. As a result, existing artifact formal models are not suitable for modeling of modern day smart processes and thus smart services that require the integration of data streams, sensors, and actuators into same processes.

In comparison, the proposed artifact formal model offers support for data streams in the form of *Stream Attributes* included in artifacts' *Information Models*. Additionally, the proposed artifact formal model defines two types of *Services* that can operate on artifacts;

- *Ad-hoc Services* that perform one time actions on actuators. And,
- *Stream Services* that continuously read data from stream sources like sensors.

Moreover, existing artifact formal models define *Information Models* as sets of simple attribute/value pairs or as sets of database relations (i.e., tuples). When using relational databases to model and manage artifacts, manipulating artifacts must be performed by manipulating different database relations using *SQL* queries. This approach of dealing with low-level database relations or attributes does not take full advantage of the semantic nature of artifacts and the different relationships that can exist between them.

In comparison, in our proposed artifact formal model, which defines *Information Models* as data structures, introduce four types of data attributes; *Simple*, *Complex*, *Reference*, and *Stream*. As a result, these types of data attributes allow the manipulation of artifacts at a high-level and hide the low-level database relations and *SQL* queries.

Similarly, the proposed artifact formal model represents artifact relationships using high-level *Reference* data attributes that hide the low-level database relations. Table 2.1 summarizes the differences between existing and proposed artifact formal models.

Table 2.1 Comparison between existing and proposed artifact formal models

	Existing Formal Models	**Proposed Formal Model**
Data Streams	do not support data streams	support data streams using *Stream Attributes*
Sensors	do not support sensors	support sensors using *Stream Services*
Actuators	do not support actuators	support actuators using *Ad-hoc Service*
Manipulation	low-level dealing with attribute/value pairs and database relations	high-level dealing with simple, complex, reference, and stream data attributes
Artifact Relationships Representation	low-level using database relations	high-level using reference attributes

2.4. Artifact Modeling Notations and Frameworks

From modeling notations and frameworks perspective, many works have focused on defining formal and graphical notations to model and execute artifact-centric *Business Processes*. We analyze and distinguish between these notations according to three criteria:

1. *Conceptual vs Executable Notations*: The first criterion distinguishes between whether notations are used to construct *Conceptual Models* or *Execution Models*. A *Conceptual Model* is a graphical model that is designed by a business person and is used to represent a *Business Process* at a conceptual level. On the other hand, an *Execution Model* is an IT related technical and textual model that is defined by an IT specialist or a computer system and is used to execute a *Business Process*. An *Execution Model* can also be automatically generated from a *Conceptual Model*.

2. *Procedural vs Declarative Notations*: The second criterion distinguishes between procedural or declarative notations. Procedural notations are used to construct finite-state based artifact models whereas declarative notations are used to construct rule-based artifact models.

3. *Graphical vs Textual Notations*: The third criterion distinguishes between graphical or textual notations. Graphical notations are used to

model declarative or procedural *Conceptual Models*. Textual notations are used to model declarative or procedural *Execution Models*.

With the exception of the *GSM* notation [DaHV13], existing graphical modeling notations are based on the modeling constructs and patterns described in [LiBW07, NiCa03]. [NiCa03] describes three modeling constructs; *Task*, *Repository*, and *Flow Connector* that can be used to model artifact lifecycles. While [LiBW07] describes nine modeling patterns including; *Pipeline*, *Repository*, *Branch*, *Convergence*, *Project*, *Creation*, *Synchronization*, *Rework*, and *Disposal* that can be employed in *Business Artifact* modeling.

Cohn et al. [CDHP08] introduce *Siena* to graphically model *Business Artifact Lifecycles* as procedural finite-state machines based on *Tasks*, *Repositories*, and *Flow Connectors*. *Siena* provides the capability to generate *XML*-based *Business Artifacts* that are then deployed and executed in the *Siena Runtime Container*. The procedural specification of finite-state machines in *Siena* makes it unsuitable for *Business Process* transformation and customization where a declarative approach is more favorable.

In [LoNy11], *Business Process Modeling Notation (BPMN)* extensions have been introduced to model artifact-centric *Business Process Models*. These *BPMN* extensions include; *Artifacts*, *Object Lifecycles*, *Location Information*, *Access Control*, *Goal States*, and *Policies*. *Artifacts* represent the process model's basic building blocks. *Object Lifecycles* specify artifacts' states. *Location Information* specifies how artifacts change their location. *Access Control* specifies the remote accessibility of artifacts. *Goal States* specify desired final states. And, *Policies* are used to remove undesired behavior. *BPMN* extensions provide graphical notations and environment for modeling procedural artifact-centric processes but do not support a declarative modeling approach.

In [LLQS10], *Artifact Conceptual Flow* or *ArtiFlow* (re-named *EZ-Flow* in [XSYY11]) is introduced. The *ArtiFlow* model relies on four types of constructs: *Business Artifacts*, *Services*, *Repositories*, and *Events*. *ArtiFlow* models are executed by translating them into *BEPL*-based *Workflows*. In order to make the translation into the *BPEL* feasible, *ArtiFlow* manages execution control through the use of *Events*. A modeling tool for *Artiflow* models is designed and developed in [ZYLL11]. Figure 2.5 illustrates an *Artiflow* example. Similarly to *BPMN* extensions, *ArtiFlow* provides graphical notations and environment for modeling procedural artifact-centric processes but do not support a declarative modeling approach.

Moreover, the use of a generated event when a *Task* is executed in order to invoke the next *Task* makes *Artiflow* more process oriented.

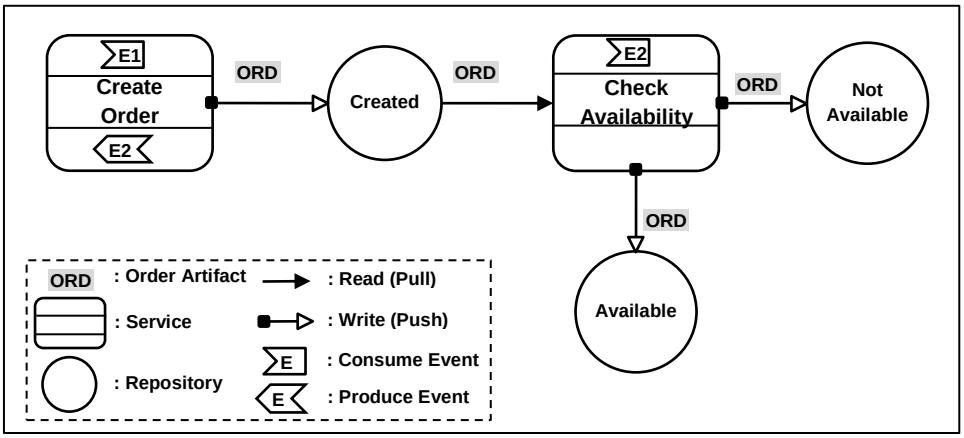


Figure 2.5 Artiflow example

The ArtiNets model, a variant of artifact-centric process models is introduced in [KuSu10]. It supports the specification of constraints on artifact lifecycles. Similarly to the *Declarative Service Flow Language* (*DecSerFlow*) [VaPe06], *ArtiNets* also allow declarative style in specifying constraints on artifact lifecycles. The key components of *ArtiNets* model are: *Artifacts*, *Services*, *Places*, and *Transitions*. *ArtiNets* model is closely related to *Petri Nets* [Mura89], but only they differ in two aspects: *Artifacts* in *ArtiNets* replaces *Tokens* in *Petri Nets*, and *ArtiNets* have different transition firing rule than *Petri Nets*.

[EQST12] models artifact-centric business processes with the *Unified Modeling Language (UML)* [RuJB04]. In this work, *Business Artifacts (Information Models)* are represented as *Class Diagrams*. *Lifecycles* are represented as *State Machine Diagrams*. *Services* are declaratively specified as preconditions and post-conditions using the *Object Constraint Language (OCL)* [CaGo12]. Finally, *Associations* are procedurally specified using *Activity Diagrams*.

On the other hand, modeling notations for declarative *Business Artifact* models based on the *GSM (Guard-Stage-Milestones)* formal model are introduced in [DaHV13, HDDF11a, HDDF11b]. The *GSM* paradigm seeks to graphically model *Business Artifact* lifecycles using *Guards*, *Stages*, and *Milestones*. Declarative rules referred to as *Sentries* open or close *Stages*, and consequently validate or invalidate *Milestones*. By using *Guards*, *Stages* and *Milestone* as modeling primitives, the *GSM* notation allows parallelism and hierarchies in *Business Artifact* lifecycles. Moreover, *GSM* notation served as the foundation for the *Case Management Modeling and Notation (CMMN)*

[MaHV12] core model. The *Barcelona* prototype provides design editor and runtime environment for *Business Artifact* models based on the *GSM* paradigm [HBGV13]. Figure 2.5 illustrates *GSM* representation of the Lifecycle of the *Order* artifact. The *GSM* approach not only supports a declarative specification of artifact-centric business processes but it also provides graphical modeling notations and environment which simplifies the modeling phase. However, *GSM* does not support data streams, sensors, and actuators and is developed to be used by *Business People* which makes it unsuitable for modeling modern day smart processes.

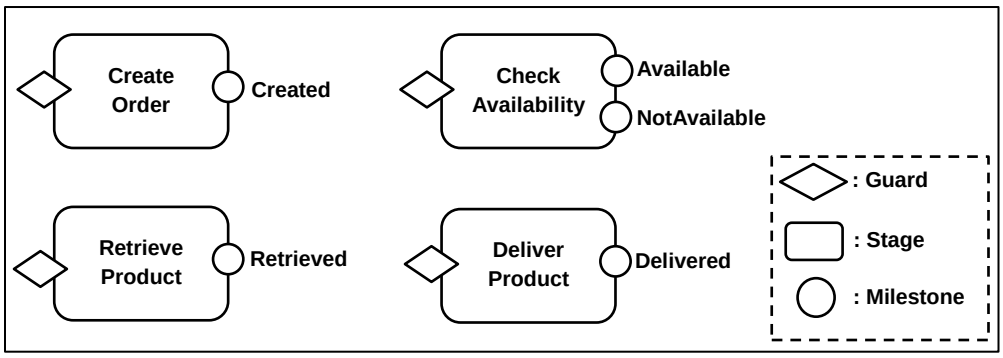


Figure 2.6 GSM representation of the Lifecycle of the Order Artifact

From the textual notations perspective, the *Business Entities and Business Entity Definition Language (BEDL)* are introduced in [NKMH10, NPKM11]. The *BEDL* is an *XML*-based language that specifies artifact-centric *Business Process Models* based on; *Business Entities (or Artifacts)*, *Lifecycles*, *Access Policies*, and *Notifications.* The *BPEL4DATA* is then used to consume *Business Entities* and execute processes. Even though *BEDL* is a textual language, *BEDL* does not support a declarative specification of artifact-centric business process models. Instead, the *BEDL* specifies artifact-centric business process models as technical XML-based finite state machines. As a result, the *BEDL* complicates the design and modeling phase of artifact-centric business process models.

Abiteboul et al. [ABGM09] formalize business artifact processes using the *Active XML (AXML)* approach where a *Business Artifact* instance is written as an *XML* document with embedded function calls. The *Business Artifact* process is thus executed by invoking embedded functions when associated condition holds and assigning their results to *Business Artifact* attributes. The *Active XML* introduces a declarative specification of artifact-centric business process models which makes it suitable for *Business Process* transformation and customization. Nonetheless, the *Active XML* does not provide a graphical modeling notation that simplifies the modeling phase and contribute towards process awareness in the organization.

In [YoLi10], the *Artifact-Centric Process Model* (*ACP model*) is introduced in order to support inter-organizational business process modeling. In [YoLZ11], an extended version of the *ACP model* is presented and is referred to as *Artifact-Centric Collaboration Model* (*ACC model*). The core modeling constructs of *ACC model* include: *Role*, *Artifact*, *Task* and *Business Rule*. *Roles* are organization roles participating in the collaboration. Moreover, *ACC model* distinguishes between two types of artifacts; *local* and *shared* artifacts. *Local* artifacts are the artifacts used internally within an organization. *Shared* artifacts serves as a contract between involved organizations, and it is used to indicate the agreed business stages towards the completion of the collaborative process. In [NgYL12], a framework for realizing artifact-centric process models in *Service-Oriented Architecture (SOA)* has been proposed and implemented based on the *ACC model* proposed in [YoLZ11]. One of the main advantages of *ACC model* is its support for declarative specification of *Business Rules* which are specified as constraints expressed with *Event-Condition-Action Rules*.

In comparison to existing works, we propose a graphical modeling notation referred as the *Conceptual Artifact Modeling Notations (CAMN)*. The *CAMN* includes six modeling constructs; *Task*, *Repository*, *Flow Connector*, *Data Attribute List, Condition*, and *Event*. These six modeling constructs allows the design of procedural *Conceptual Models* that we refer to as *Conceptual Artifact Models (CAM)*. The *CAM* is characterized by capturing both *Information Models* and *Lifecycles* of artifacts in the same *Conceptual Model* resulting in reduced design time and improves process awareness. Additionally, we describe eleven modeling patterns that cover the nine modeling patterns described in [LiBW07], in addition to modeling patterns that are required when data streams are involved. Moreover, the *CAM* includes required information for automatically generating declarative *Execution Models* that comply with our proposed formal model. A textual notation, referred to as *Artifact Query Language (AQL)*, is also proposed in order to directly construct declarative *Execution Models* and interrogate them.

Table 2.2 and 2.3 summarize and compare different artifact modeling notations and frameworks covered in this section. Table 2.2 compares the *Conceptual Models* of the modeling approaches, if any, whereas Table 2.3 compares the *Execution Models* of the modeling approaches, if any. Similarly to [KuYY14], we perform our comparison based on the four dimensions of the *BALSA* framework; *Information Model*, *Lifecycle*, *Services*, and *Associations*. The *'-'* symbol signifies that the *BALSA*

component does not have a representation at the level of comparison in question.

Table 2.2 Comparing conceptual models of different artifact modeling approaches

	Information Model	**Lifecycle**	**Services**	**Associations**
CAM	*Embedded in conceptual model as Data Attribute Lists attached to Tasks*	*Procedural (Tasks, Repositories, Flow Connectors)*	*Procedural (Tasks)*	*Procedural (Flow Connectors with attached Events and Conditions)*
Siena	*Separate from conceptual model*	*Procedural (Graphical Finite-State Machine)*	*Procedural (Tasks)*	*Procedural (Graphical Finite-State Machine)*
BPMN Extensions	*Separate from conceptual model*	*Procedural (Activities, Events, Gateways, Sequence Flow)*	*Procedural (Activities)*	*Procedural (Sequence Flow)*
ArtiFlow	*Separate from conceptual model*	*Procedural (Graph of Services, Repositories and Edges)*	*Procedural (Services)*	*Procedural (Graph Edges)*
ArtiNets	*Separate from conceptual model*	*Procedural (Artifacts, Places, Services and Transitions)*	*Procedural (Tasks)*	*Declarative (ECA-rules)*
UML	*Class Diagram*	*Procedural (State Machine Diagrams)*	*Declarative (Pre and Post Conditions in OCL)*	*Procedural (Activity Diagrams)*
GSM	*Separate from conceptual model*	*Declarative (Guard, Stage Milestone)*	*Declarative (Tasks)*	*Declarative (Sentries)*
BEDL	-	-	-	-
AXML	-	-	-	-
ACC model	-	-	-	-

Table 2.3 Comparing execution models of different artifact modeling approaches

	Information Model	**Lifecycle**	**Services**	**Association**
CAM	*Programming Data Types*	*Declarative (States)*	*Declarative (Semantic Web Services Specification)*	*Declarative (Artifact Rules)*
Siena	*Database*	*Procedural (XML finite-state machine)*	*Procedural (Tasks)*	*Procedural (XML finite-state machine)*
BPMN Extensions	-	-	-	-
ArtiFlow	*XML Schema*	*Procedural (BPEL)*	*Procedural (BPEL)*	*Procedural (BPEL)*
ArtiNets	-	-	-	-
UML	-	-	-	-
GSM	-	-	-	-
BEDL	*XML Schema*	*Procedural (XML finite-state machine)*	*Procedural*	*Procedural (XML finite-state machine)*
AXML	*XML Schema*	*Declarative (AXML Embedded Function Calls with Conditions)*	*Declarative (Function Calls)*	*Declarative (AXML Embedded Function Calls with Conditions)*
ACC model	*XML Schema*	*Declarative (States in XML)*	*Declarative (Web Service Details in XML)*	*Declarative* (*ECA Rules in XML*)

Analysis of Tables 2.2 and 2.3 lead to several conclusions:

Firstly, the *CAM* is the only approach that embeds the *Information Model* in the *Conceptual Model*, which leads to a holistic representation of *Business Processes* and simplification of the modeling phase. The *UML* approach represents the *Information Model* as a separate *Conceptual Model* using a *Class Diagram*. The remaining approaches define *Information Models* separately from *Conceptual Models*.

Secondly, several modeling approaches, including *BEDL*, *AXML*, and *ACC model*, do not make use of a *Conceptual Model*. Instead, they directly define *Executable Models* using a technical language like *XML* which complicates the modeling phase and does not support process awareness.

Thirdly, the *CAM* is the only approach that incorporates both procedural and declarative modeling approaches according to the modeling level. At the conceptual level, *CAM* is graphically presented as a procedural *Conceptual Model* based on minimalistic and intuitive constructs which simplifies the design phase. Moreover, it provides a graphical model that is easy to understand and analyze. At the execution level, the *CAM* is automatically translated into a declarative *Execution Model* based on *ECA Rules* which provide high level of control and flexibility over the execution of the *Business Process.*

Finally, the *CAM* is the only approach that offers both graphical and textual notations. The graphical notation *CAMN* is used to design procedural *Conceptual Models* which are then automatically translated into *Execution Models.* The textual notation *AQL* is used to directly define and query declarative *Execution Models*.

In summary, existing artifact modeling notations and frameworks fall into two categories: Notations that model procedural finite-state machines which fail to provide a declarative and flexible framework. And, notations that provide declarative notations based on rules which fail to provide a simple and representative *Conceptual Models*. Our work combines both conceptual and declarative approaches in order to provide holistic models that are used as the basis of the *Artifact Integration Framework*.

2.5. Data Integration

Dealing with solutions for artifact integration leads to analyze integration problems in various disciplines such as *Databases* and the *Business Process Management*.

From the *Database* perspective, *Data Integration* is the problem of combining data residing at heterogeneous data sources, and providing users with a *Unified View* of these data [Hale01, Hull97, Ullm00].

Figure 2.5 illustrates the architecture of a typical *Data Integration System* as described in [KaPa09]. *Sources* store data in a variety of formats including relational databases, text files, spreadsheets, XML…etc. *Wrappers* handle the heterogeneity in the data formats by transforming each *Source*'s data model to a common data model used by the integration system. The wrapped data sources are usually referred to as *Local Schemas* or source databases. A *Unified View,* also called *Global Schema* is then exported by the *Mediator*. *Mappings* expressed in a certain mapping language specify the relationship between the *Local Schemas* and the *Unified View* or *Global Schema*. Finally, users can retrieve data from the *Sources* indirectly by querying the *Global Schema*.

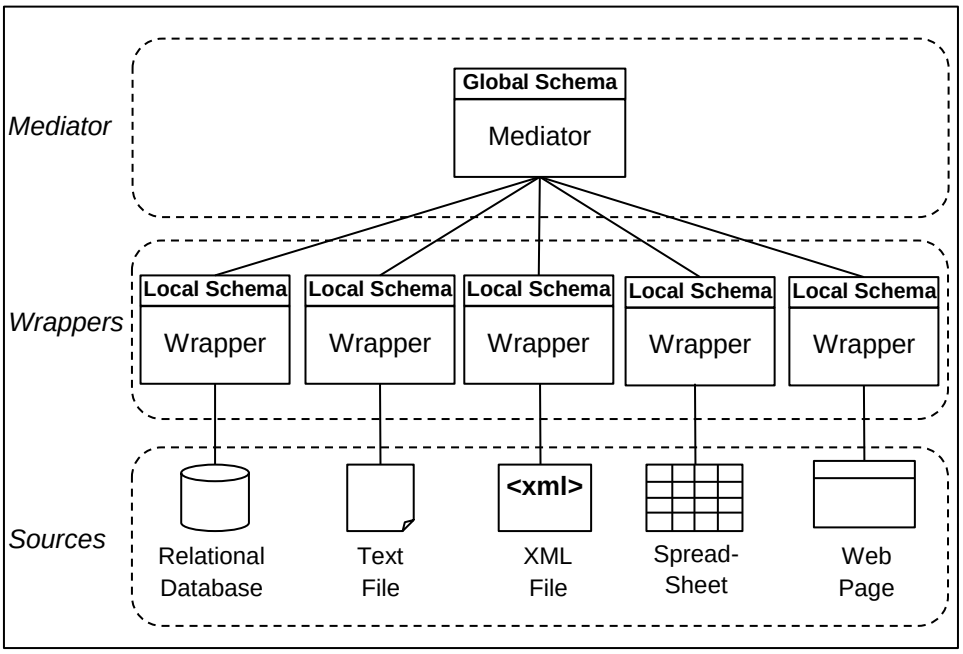


Figure 2.7 Artifact Integration System

On the other hand, [HuZh96a, HuZh96b, YaKL97, ZHKF95] describes two major approaches to manage *Data Integrations*:

1) *The Materialized Approach* by which data at the sources are copied into *Data Warehouses* where it is accessed from. And,
2) *The Virtual Approach* by which data are accessed from sources through a *Unified View*.

Consequently, a query against the *Global Schema* in the *Materialized Approach* is answered directly by accessing *Data Warehouses*. In contrast, a query against the *Global Schema* in the *Virtual Approach* cannot be answered directly but has to be translated into sub-queries against the *Local Schemas*.

Moreover, two types of *Data Integration* systems can be implemented in the *Virtual Approach*:

1) *The Central Data Integration System* in which a global schema provides the user with a uniform interface to access information stored in the data sources as described in [CrXi05, Lenz02, Xiao06]. And,
2) *The Peer-to-Peer Data Integration System* in which there are no global points of control on the data sources (or peers). Instead, any peer can accept user queries to manipulate distributed data in the whole system as described in [AKKK03, CDLR04, HIST03].

On the other hand, in [Lenz02, PaSp00], *Local Schemas* are integrated into a *Global Schema* and mapping rules that translate data between local and global schemas are generated. Moreover, *Global Schemas* are generated based on correspondence relationships between elements of *Local Schemas*.

In [ADMR05, HaOt07], correspondences are acquired as a result of *Match Operations*. [DoMR03, RaBe01] describe two types of *Match Operations*: *manual operations*, where users specify the corresponding elements using a graphical interface, and *semi-automatic operations*, using the help of algorithms and/or ontologies.

As described in [BoVe11, MaPl06], *Schema Mapping* deals with transforming data structured respecting a source schema into data structured conforming a target schema. In the context of *Data Integration*, *Schema Mapping* transforms data structures between global and local schemas [HaOt07, HaOt07]. As described in [Lenz02], two main approaches to *Schema Mapping* exist; *Global-As-View (GAV)* and *Local-As-View (LAV)*. In the *GAV* approach, elements in the *Global Schema* are mapped to views over *Local Schemas*. Therefore, querying strategies are simple, but the evolution of *Local Schemas* is not easily supported. In the *LAV* approach, elements in *Local Schemas* are mapped to views over the *Global Schema*. The *LAV* allows thus changes to *Source Schemas* without affecting the *Global Schema* but complicates query processing. Since the mapping associates to each source a view over the global schema, it is not immediate to infer how to use the sources in order to answer queries expressed over the global schema [Lenz02]. In order to overcome these limitations of both *GAV* and *LAV*, related works such as [KaPa09] propose a hybrid approach known as *Global-*

Local-As-View (GLAV) in which views over *Local Schemas* are mapped to views over the *Global Schema*.

From a different perspective, recent works such as [DLLP18, ChPe17], employ *Ontologies* in order to perform *Semantic Data Integration*. An *Ontology* is a formal, explicit specification of a shared conceptualization for domain knowledge [CrXi05]. In *Semantic Data Integration* systems, *Ontologies* are used to represent *Global Schemas* [WVVS01]. Moreover, works such as [LaWB08, Pasq08], uses *Semantic Web* [DMVF00] technologies such as *Resource Description Framework (RDF)* [KlCa06] and *Web Ontology Language (OWL)* [AnVa04] in order to define *Ontologies* and process queries.

In comparison to the *Artifact Integration System*, the data sources in the *Artifact Integration System* correspond to distributed and heterogeneous artifacts. *Local Schemas* corresponds to *Conceptual Artifact Models*. Moreover, several local *Conceptual Artifact Models* are integrated in order to build a global *Conceptual Artifact Model* that acts as a *Global Schema*. Since the local models in our case represent distributed *processes* that executes separately, the *Artifact Integration System* that we propose is based on a *Virtual Approach* in which a *Unified View* is generated and no data is replicated in a *Warehouse*.

Moreover, since the purpose of the *Artifact Integration System* is to provide a centralized access point for managing local artifact-centric processes, we propose in our contribution to implement a central *Artifact Integration System* in which a global *Conceptual Artifact Model* is used to access local *Conceptual Artifact Models.*

Furthermore, we generate the global *Conceptual Artifact Model* based on identified correspondences as a result of *Match Operations*. Finally, we generate specialized mapping specifications between local and global models that are used for data translation and transformation. Since we integrate local *Conceptual Artifact Models* representing different existing processes, we employ a *GAV* mapping approach in order to simplify query processing. However, *Data Integration* only deals with data and ignores processes. As a result, artifact integration requires specialized integration semantics that take into consideration process elements and their *Control-Flows* such as in the fields of *Business Process Merging and Views*.

2.6. Business Process Merging and Views

From the *Business Process* perspective, process integration is extensively studied in order to build *Business Processes* by merging seerveral *Business Processes*.

As described in [SuKY06], *Merging Business Processes* is primarily based on *Control-Flows* including operators such as sequential, parallel, conditional, and iterative to reconnecting and re-branching activities.

In [LDUD10], three requirements to merging *Business Process Models* are presented:

1) The behavior of the merged process model should subsume the source process models.
2) Given an element in the merged process model, analysts should be able to trace back from which source process model(s) the element in question originates.
3) One should be able to derive source process models from the merged processes.

[KGFE08] proposes a prototype for consolidating multiple versions of one shared *Business Process Model* that is manipulated in the context of business-driven development. The prototype visualizes differences between versions of process models and enables the resolution of differences, by applying change operations in an iterative way to automatically build the *Control-Flow*.

[LDUD13] presents an algorithm for generating the union of multiple *Business Process Models* referred to as *Merged Models*. Additionally, an algorithm that extracts intersections of multiple *Business Process Models* referred to as *Digests* from a *Merged Model* is also presented.

From the *Business Process Views* perspective, works in the field of artifact-centric inter-organizational business process modeling and choreography such as [LoWo10, YoLZ11, YYZO15] seek to provide collaborative organizations with the ability to modify and/or hide specific parts of their internal processes while exposing necessary parts by using *Business Artifacts*. In this case, *Business Artifacts* ensure the correctness of collaboration processes. For example, the framework proposed in [YoLi10, YoLZ11] exposes private and public views to integrate artifact based processes.

Similarly to *Business Process Merging and Views*, we merge in our framework several source *Conceptual Artifact Models* into one *Conceptual Artifact Model*. We handle the merged model as the union of source models. Moreover, we employ merging algorithms that re-branch and reconnect *Flow Connectors* between *Tasks* and *Repositories* in the merged model. in our work, the *Control-Flow* is not directly modeled in *Conceptual Artifact Models* and is substituted by *Repositories*, representing different data states, which hence implies the need for specialized merging approaches that include *Data Integration* mechanisms.

2.7. Query Languages

To the best of our knowledge, current works on artifacts still lack effective languages and tools that take full advantage of the artifact semantic nature in order to define, manipulate and interrogate their instances.

With the exception of [JoBa14], most existing languages are graphical or textual notations, which mainly focus on modeling and executing artifact processes. They lack of declarative query languages similar to *SQL* in relational databases for managing artifact instances and handling their data streams.

The SQL for Business Artifacts (BASQL), introduced in [JoBa14], was a first attempt to describe *SQL*-like statements in order to define and manipulate artifact instances. However *BASQL* fails to treat artifacts as semantic entities formed from an underlying model of relations and streams, instead of treating them as simple database relations.

From *Data Streaming*, *Continuous Query*, and *Complex Event Processing* perspectives have received abundant contributions in the literature. The roots of continuous query go back to *Materialized Views* [GuMO95] where data views are continuously updated to reflect changes to databases.

The concept of *Data Streaming* was first introduced in [JaMS95] under the name of Chronicles as an extension of materialized views. The *Tapestry* [TGNO92] was a pioneer by introducing the notion of continuous queries using a declarative language called *TQL* by which queries are executed once every time instant and their results are merged using set union statements.

Since then, many sophisticated continuous query languages and data stream management systems have been proposed including *Aurora*

[ACÇC03], *TelegraphCQ* [CCDF03], *STREAM* [ABBC16], and *Odysseus* [00a] to just name a few.

However, *Complex Event Processing* goes back to *Active Databases* [SPAM91, WiCe96] where *ECA* rules, referred to as *Triggers*, fire when events of interest occur and if their conditions are satisfied to perform relevant actions. Since events in *Triggers* are simple update, insert, or delete operations performed on relations, works in [CKAK94] focuses on the specification of complex or composite events which are constructed from primitive events.

As sensors and data stream processing have become main-streams in the Internet of Things, recent works on *Complex Event Processing* techniques such as [DGPR07, JiAC07], allow expressing specialized continuous queries that detect complex events from input event streams.

In summary, existing query languages treat low level streams and/or relations and are not suitable for treating high-level *Artifact* instances and their data. For this reason, our proposed *Artifact Query Language (AQL) is* specifically designed to target *Artifact* instances and take full advantage of their semantic nature in order to define, manipulate, and interrogate *Artifact* instances. Moreover, our *AQL* support data streams and services invocation capabilities in order to model modern day processes that require reading sensor data and performing actions on actuators.

2.8. Conclusion

In this chapter, we present and analyzed existing works related to the problem of this thesis. First, we cover with a brief introduction to *Business Process Models* that puts into perspective its *activity-centric* and *artifact-centric* approaches. We then overview activity-centric *Business Process Modeling* and illustrated its advantages and disadvantages in comparison to the artifact-centric approach. We then investigate existing artifact modeling notations and framework and illustrate their advantages, disadvantages and compared them to our proposed artifact modeling notation and framework. We then study *Data Integration* and *Business Process Merging* and show the need for specialized *Artifact Integration* semantics that combines mechanisms from both approaches and adds specific integration mechanisms related to artifact integration. Finally, we describe existing *Query Languages* including one-time and continuous query languages and compared them to the proposed query language while illustrating its advantages.

In the next chapter, we present the *Specification Phase* of the *Artifact Integration Framework*. Moreover, we define a formal model for an *Artifact System* that implements and executes artifact-centric processes. We also propose a declarative query language that takes full advantage of the semantic nature of artifacts in order to define, manipulate and interrogate artifact instances.

3

Specifying Artifact Systems

Chapter Outline

Several formal models for executing artifact-centric processes exist in the literature and can be grouped into two types; procedural and declarative models. Procedural formal models such as [GeBS07] are based on finite-state machines and are difficult to customize and transform. On the other hand, declarative formal models like [BGHL07, DaHV13] are based on *Event-Condition-Action (ECA) Rules* and are flexible for customization and transformation processes.

Nevertheless, existing formal models fail to define practical approaches that can be semantically used in order to define, manipulate and query artifact-centric processes in a simple and intuitive manner. Instead, existing formal models directly exposes the relational database model and require technical expertise in order to be defined, manipulated, and queried. As a result, artifact-centric processes that are based on existing formal models cannot be effectively used by casual users and business people.

Most of existing formal models are defined in order to specify traditional artifact-centric *Business Process Models*. As a result, they lakes the support of data streaming, sensors, and actuators descriptions that are needed in smart processes and smart services in *IoT*.

In this chapter, we introduce a formal model for the *Artifact System* that implements and executes artifact-centric processes. The proposed formal model is based on declarative *ECA Rules* allowing customization and transformation of processes. Moreover, our proposed *Artifact System* hides low-level relational database model and exposes high-level data attributes that can be used to semantically define, manipulate and query *Artifact Systems* by casual users and business people. Additionally, the proposed formal model supports data streams, sensors and actuators that are relevant for *IoT* applications.

The proposed *Artifact System* is based on three main components:

1. *Artifact Classes* which include four categories of high-level data attribute types including; *simple*, *complex*, *reference*, and *stream*.
2. *Services*, which are the basic units of work that operate on artifacts, perform actions on actuators, and data stream generated by sensors. And,
3. *Artifact Rules*, which are a variant of *Event-Condition-Action (ECA)* rules, execute artifact processes by detecting situations based on states of artifacts and taking appropriate actions in a timely manner.

We also define the *Artifact Query Language* (*AQL*) a simple and declarative query language inspired by the *SQL*. The *AQL* is used to define

Artifact Systems and manipulate and query artifact instances. The *AQL* is characterized by hiding the underlying relational model and semantically operating on artifact instances.

The remainder of this chapter is organized as follow:

In Section 3.1, we define and formalize *Artifact Systems*.

In Section 3.2, we present the syntax of the *Artifact Query Language* (*AQL*) and its two sub-parts; *Artifact Definition Language* (*ADL*) and *Artifact Manipulation Language* (*AML*).

In Section 3.3, we describe the semantics and execution strategies of *AQL* based on fundamental *Mathematical Logic* and *Relational Algebra*.

3.1. Artifact Systems

In artifact-centric processes, semantic entities, known as artifacts, are the basic building blocks from which the process is constructed [CoHu09]. The purpose of every artifact is to achieve a particular goal. Artifacts evolve according to their lifecycles and interact with each others in order to reach their business goals.

An Artifact is composed from both an *Information Model* and a *Lifecycle* [NiCa03]. The information model is used to register data about the artifact-centric process and involved objects. The state-based lifecycle dictates the possible states and possible state transitions of artifacts and when services, performing basic units of work can be invoked on artifacts.

Based on these definitions and concepts, we define an *Artifact System* that implements artifact-centric processes based on a declarative specification of artifact *Lifecycles* using *Artifact Rules* as described in [BGHL07]. A declarative specification describes *"what"* should be done using *Event-Condition-Action (ECA) Rules* whereas a procedural specification describes *"how"* processing should be done in a step-by-step manner using a finite-state machine. The declarative specification benefits the incremental construction of artifact processes and introduces higher level of flexibility when performing process transformation in contrast to the procedural specification [KuYY14].

Our proposed *Artifact System* is composed of three main components: *Artifact Classes*, *Services*, and *Artifact Rules*.

Definition 3.1 (Artifact System) *An Artifact System W is a triplet (C, S, R) where C is the set of Artifact Classes, S is the set of Services, and R is the set of Artifact Rules.*

Example 3.1 (Artifact System) *A simplified Artifact System implementing a smart process about house fire detection and control in the context of a smart city could be composed of the following:*

- *Two Artifact Classes; FireControlArtifact that deals with fire detection and performs reactive procedures, and FireStationAlertArtifact that deals with locating and alerts near fire stations.*
- *Several Services such as TurnOnAlarm that turns on an alarm actuator, StreamIndoorTemperature that streams indoor temperature from a temperature sensor, LocateFireStation that locates a close fire station, etc.*
- *Several Artifact Rules such as a rule that detects a fire incident when indoor temperature becomes greater than 57°C and changes the state of a FireControlArtifact to the FireDetected state as a result, and another rule that invokes the TurnOnAlarm when a FireControlArtifact state becomes FireDetected, etc.*

3.1.1. Artifact Classes

An *Artifact Class* represents the schema for a set of artifact instances of the same type. It thus specifies both the *Information Model* and the *Lifecycle of a given artifact*.

The *Information Model* defines a set of data attributes that are used to register data about an artifact and involved objects. Data attributes are expressed as *name-type* pairs. Data attributes fall into four categories:

1. *Simple Attributes*: hold one value at a time and are used to record information about the artifact itself. For example the artifact *identifier* and its creation date are simple attributes. The simple attribute types are: *Boolean*, *Integer*, *Real*, *String*, *Date*, and *TimeStamp*.
2. *Complex Attributes*: represent relations and are specified as lists of simple attributes. In mathematics, relations represent *Database* tables. Similarly to relations, complex attributes can hold many tuples or table rows at a time. They are used to record information about various business objects that are related to the artifact. For example *House* is a complex attribute consists of two simple attributes; *Address*, and *Surface*.

3. *Reference attributes*: refer to other artifacts (i.e., children artifacts) that are directly related to the parent artifact in a *Parent-Child* relationship. The *Parent-Child* relationship can be one-to-one or one-to-many. Thus, a parent artifact can have one or more children artifacts. For example a fire control artifact is the parent of several fire station alert artifacts.
4. *Stream attributes*: represent data streams that are generated by data stream sources and are specified as a list of simple attributes including a timestamp attribute representing the insertion time of tuples. Data streams are considered a non-persisted, append-only, and unbounded bag of elements that include a timestamp. For example the house's indoor temperature is a stream attribute consisting of two simple attributes; *Temperature* and *TimeStamp*.

The *Lifecycle* of an *Artifact Class* is a set of states that artifact instances can pass through towards their goals. States represent stages or milestones in the business context that are achieved by artifact instances. The set of states can have one initial state in addition to any number of intermediate and final states. For example, in the fire control artifact, *Normal* is the initial state. *FireDetected* is an intermediate state. And, *FireExtinguished* is a final state. Additionaly, an *Initialized* state is used by default to represent an artifact instance that is created but has not yet moved to its initial state.

The *Artifact Class* has a minimal structure that must contain at least an artifact identifier and state attributes in its *Information Model*, and one state in its *Lifecycle*.

Assuming the existence of the following pairwise disjoint countably infinite sets: $\mathcal{C}$ of artifact names, $\mathcal{A}$ of attribute names, $\mathcal{Q}$ of artifact states, $\mathcal{V}$ of simple data types, including: *Boolean*, *Integer*, *Real*, *String*, *Date*, *Null,* and *TimeStamp*, and $\mathcal{S}$ of services names.

Definition 3.2 (Artifact Class) *An Artifact Class* c *is a tuple* $(c, A, \gamma_{sim}, \gamma_{com}, \gamma_{ref}, \gamma_{str}, Q, s, F)$ *where:*

- $c \in \mathcal{C}$ *is the artifact class name,*
- $A \subseteq \mathcal{A}$ *is a finite set of artifact attributes that include four subsets; a simple attribute partition* A_s*, a complex attribute partition* A_c*, a reference attribute partition* A_r*, and a stream attribute partition* A_t*,*
- $\gamma_{com} : A_c \rightarrow \mathcal{A}^n$*, the complex type function is a partial map that maps the complex attributes in A to a list of simple attributes in* $\mathcal{A}$*.*

- $\gamma_{ref} : A_r \rightarrow \mathcal{C}$, *the reference type function is a partial map that maps the reference attributes in A to their corresponding Artifact Class in* $\mathcal{C}$.
- $\gamma_{str} : A_t \rightarrow \mathcal{A}^n$, *the stream type function is a partial map that maps the stream type attributes in A to a list of simple attributes in* $\mathcal{A}$ *such that one of the simple attributes is of the* TimeStamp *type.*
- $\gamma_{sim} : A_s \cup \gamma_{com} \cup \gamma_{str} \rightarrow \mathcal{Y}$, *the simple type function is a partial map that maps the simple attributes in A in addition to the simple attributes constituting complex and stream attributes to their simple data types in* $\mathcal{Y}$.
- $Q \subseteq \mathcal{Q}$ *a finite set of states, where* $s \in Q$ *and* $F \subseteq Q$ *are respectively initial and final states.*

Example 3.2 (Artifact Class) *An Artifact Class* c *specifying a fire control artifact is defined as follow:*

- $c = FireControlArtifact$
- $A = \{FireControlArtifactId, FireDate, House, IsAlarmTurnedOn, FireStationAlert, IndoorTemperature, Address, Surface\}$ *where:*
  - $A_s = \{ FireControlArtifactId, FireDate, IsAlarmTurnedOn\}$
  - $A_c = \{ House \}$
  - $A_r = \{ FireStationAlert \}$
  - $A_t = \{ IndoorTemperature \}$
- $\gamma_{com}(House) = \{Address, Surface\}$
- $\gamma_{ref}(FireStationAlert) = FireStationAlertArtifact$
- $\gamma_{str}(IndoorTemperature) = \{Time, Tmp\}$
- $\gamma_{sim}(FireControlArtifactId)=Integer$, $\gamma_{sim}(FireDate)=Date$, $\gamma_{sim}(IsAlarmTurnedOn)=Integer$, $\gamma_{sim}(Address)=String$, $\gamma_{sim}(Surface)=Real$, $\gamma_{sim}(Time)=TimeStamp$, $\gamma_{sim}(Tmp)=Integer$
- $Q = \{Normal, FireDetected, AlarmTurnedOn, FireStationAlerted, FireExtinguished\}$ *where* $s = Normal$ *and* $F = \{ FireExtinguished \}$

3.1.2. Services

Services are basic units of work that operate on artifacts and update their attributes, and trigger state transitions according to artifact lifecycles. We define two types of *Services*:

1. *Ad hoc Services*: are stateless units of work that are executed when certain situations occur as specified by *Artifact Rules*. When executed,

Ad hoc Services perform certain actions, update artifact instances, and finish execution as soon as possible. *Ad hoc Services* can perform calculations, read data objects, and/or perform actions, for instance, on sensors, actuators or software modules. They also can be automatic services, requiring no human intervention, manual services, requiring human intervention, or semi-automatic services. *Ad hoc Services* affect simple, complex and reference attributes. For example, *TurnOnAlarm* is an *Ad hoc Service* that is invoked when fire is detected in order to activate an alarm actuator.

2. *Stream Services*: are units of work that are invoked when artifacts are instantiated to connect to sensors and pull out data streams. From the moment they are invoked, stream services are continuously executed until the artifact instance is disposed. They produce data streams that are read from various sources (i.e., temperature sensors, RSS feeds, CSV files). For example, *StreamIndoorTemperature* is a stream service that transmits the house's indoor temperature from a physical temperature sensor into an *IndoorTemperature* stream attribute.

In order to provide a declarative specification of *Services*, we define (artifact) *Services* using *Input*, *Output*, *Precondition*, and *Effect (IOPE)* in a similar way to *Semantic Web Services* where:

- *Input* is a list of artifacts that are read by the *Service*.
- *Output* is a list of artifacts that are manipulated or created by the *Service*. *Output* may include artifacts from the *Input* list.
- *Precondition* is a condition on the *Input* artifacts that holds before executing the *Service*.
- *Effect* is a condition on the *Output* artifacts that holds after executing the *Service*.

Definition 3.3 (Service) *A Service s is a tuple (s, C_I, C_O, P, E) where $s \in S$ is a service name, C_I is a finite set of input artifact classes that are read by the service, C_O is a set of output artifact classes that are modified or created by the service, P and E are respectively the Precondition and Effect of the service defined below.*

Definition 3.4 (Service Precondition) *A Service Precondition P is an expression that is formed from the conjunction of the following predicates and their negations over data attributes: $\mathsf{opened}(c.A_t)$, $\mathsf{defined}(c.A)$, and scalar comparison predicates $(>, <, \leq, \geq, =, \neq)$ where $c \in C_I \cup C_O$. The conjunction is expressed using the logical conjunction symbol: $\wedge$, while the negation is expressed using the negation symbol: $\neg$.*

Definition 3.5 (Service Effect) *A Service Effect E of a service is an expression that is formed from the conjunction of the following predicates and their negations over data attributes or Artifact Classes: new(c), opened(c.A_t), defined(c.A), and scalar comparison predicates (>, <, ≤, ≥, =, ≠) where $c \in C_I \cup C_O$. The conjunction is expressed using the logical conjunction symbol: ∧, while the negation is expressed using the negation symbol: ¬.*

The semantics of the predicates involved in *Services*' *Precondition* and *Effect* are defined as follow:

- *opened(c.A_t)*: The *opened* predicate implies that the stream attribute A_t of the *Artifact Class c* has started receiving data tuples from its stream source.

- *defined(c.A)*: The *defined* predicate implies that the attribute *A* of the *Artifact Class c* is not null and has a defined value.

- *new(c)*: The *new* predicate implies that a new instance of the *Artifact Class c* is created.

Consequently, a *Stream Service* is a service that can only have the *opened* predicate and its negation in the *Precondition* and *Effect* expressions. While an *Ad hoc Service* can only have the *defined*, *new* and scalar composition predicates and their negations in *Precondition* and *Effect* expressions.

Example 3.3 (Ad hoc Service 1) *The TurnOnAlarm Ad hoc Service activates alarm actuator of the corresponding FireControlArtifact (FCA) and is defined as follow:*

- $s = TurnOnAlarm$
- $C_I = \{FCA\}$
- $C_o = \{FCA\}$
- $P = defined(FCA.FireControlArtifactId) \wedge \neg defined(FCA.IsAlarmTurnedOn)$
- $E = defined(FCA.IsAlarmTurnedOn)$

Example 3.4 (Ad hoc Service 2) *The IssueFireStationAlert Ad hoc Service creates a new instance of the FireStationAlertingArtifact (FSAA) as a child artifact of the corresponding FireControlArtifact (FCA) and is defined as follow:*

- $s = IssueFireStationAlert$
- $C_I = \{FCA\}$
- $C_o = \{FCA, FSAA\}$
- $P = \neg defined(FCA.FireStationAlert)$
- $E = new(FCA.FireStationAlert) \wedge defined(FSSA.FireStationAlertArtifactId) \wedge defined(FSSA.House)$

Example 3.5 (Stream Service) *The StreamIndoorTemperature Stream Service streams temperature readings from the temperature sensor of the corresponding FireControlArtifact (FCA) into its IndoorTemperature attribute and is defined as follow:*

- $s = StreamIndoorTemperature$
- $C_I = \{FCA\}$
- $C_o = \{FCA\}$
- $P = \neg opened(FCA.IndoorTemperature)$
- $E = opened(FCA.IndoorTemperature)$

3.1.3. Artifact Rules

Artifact Rules are variants of *declarative Event-Condition-Action (ECA)* rules that execute artifact processes in conformance with the artifact state-based lifecycle. Artifact rules fall into two types:

1. *Artifact Rules* that invoke *Services* on artifacts, and
2. *Artifact Rules* that perform *Lifecycle*'s state transitions

Artifact Rules specify predicate conditions over attribute, in particular stream attributes. Moreover, they can only invoke *Ad hoc Services* whereas *Stream Services* are automatically invoked upon artifact's instantiation.

Definition 3.6 (Artifact Rule) *An Artifact Rule r is a logical implication $P \rightarrow Q$ in which its antecedent P is formed from the conjunction of the following predicates; $event(e)$, $state(c,q)$, $defined(c.A)$ and their negations, and scalar comparison predicates ($>, <, \leq, \geq, =, \neq$). And its consequent Q is one of the following predicates: $invoke(S)$, or $state(c,q)$.*

The semantics of the newly introduced predicates used in *Artifact Rules* are as follow:

- $event(e)$: The $event$ predicate implies that a timely, or user-generated event e is received. Timely events are used to trigger actions that should be performed on a timely basis, i.e., every day at 20 P.M. a backup should be performed. User-generated events are events that occur in the environment, i.e., a student submits an application form, an employee creates an order, etc.
- $state(c, q)$: The $state$ predicate implies that an instance of *Artifact Class* c is in the state q.
- $invoke(S)$: The $invoke$ predicate implies that the *Ad hoc Services* S are invoked.

Example 3.6 (Artifact Rule 1) *The Artifact Rule that detects a fire incident and changes the state of the corresponding FireControlArtifact (FCA) from the Normal state to the FireDetected state is defined as:*

$$\boldsymbol{state}(\boldsymbol{FCA}, \boldsymbol{Normal}) \wedge \boldsymbol{FCA.IndoorTemperature.Tmp} > \mathbf{57} \rightarrow \boldsymbol{state}(\boldsymbol{FCA}, \boldsymbol{FireDetected})$$

Example 3.7 (Artifact Rule 2) *The Artifact Rule that invokes the TurnOnAlarm service when a fire incident is detected for the corresponding FireControlArtifact (FCA) is defined as:*

$$\boldsymbol{state}(\boldsymbol{FCA}, \boldsymbol{FireDetected}) \wedge \neg \boldsymbol{defined}(\boldsymbol{FCA.IsAlarmTurnedOn}) \rightarrow \boldsymbol{invoke}(\boldsymbol{TurnOnAlarm})$$

Example 3.8 (Artifact Rule 3) *The Artifact Rule that invokes the CreateFireControlArtifact (CreateFCA) service when a CreateFireControlArtifactEvent (CFCAE) event is received is defined as:*

$$\boldsymbol{event}(\boldsymbol{CFCAE}) \rightarrow \boldsymbol{invoke}(\boldsymbol{CreateFCA})$$

Artifact Rules are considered to be sound and thus can be executed if they do not contain any conflict. Conflicting *Artifact Rules* are rules that have overlapping conditions and distinct actions. For example, an *Artifact Rule* states that if the state of an *FCA* artifact is *Normal* and the indoor temperature is greater than 57°C then the state should be changed to *FireDetected*. Another *Artifact Rule* states that if an *FCA* artifact is *Normal* and the indoor temperature is greater than 50°C then the state should be changed to *PossibleFire*. These two rules are conflicting since an *FCA* artifact which is in the *Normal* state and has an indoor temperature of 60°C can trigger both rules. In this case, one *Artifact Rule* should be refined in order to resolve the conflict. For example, the second *Artifact Rule* should be refined to if the state is *Normal* and the indoor temperature is between 50°C and 56°C, then the state should be changed to *PossibleFire*. The conflict resolution is performed by a reasoner that detects and flags any conflicting *Artifact Rules* that must then be updated by the user.

After resolving conflicts, *Artifact Rules* can then be executed by the *Rule Execution Engine* which is implemented in the prototype (see chapter 6). Whenever an artifact instance is modified, the *Rule Execution Engine* is notified. After that, all the *Artifact Rules* that involve the corresponding *Artifact Class* are evaluated on the artifact instance. Finally, the matching *Artifact Rules* are incrementally executed on the artifact instance.

3.2. Artifact Query Language

Based on the *Artifact System* we proposed in Section 3.1, we define the *Artifact Query Language (AQL)* in order to declaratively specify, manipulate and query *Artifact Systems* and artifact instances. The *AQL* differs from other query languages like *SQL* [Slaz04], *BASQL* [JoBa14], and *CQL* [ArBW06] by targeting the high-level artifact model instead of targeting low-level database relations and streams. The *AQL* benefits include simpler and intuitive queries that hide the technical details of the underlying relational model. It consists of nine statements divided into two parts: the *Artifact Definition Language (ADL)* and the *Artifact Manipulation Language (AML)*. Table 1 lists the *AQL* statements and their syntaxes.

The *ADL* has three statements to define *Artifact Classes*, *Services* and *Artifact Rules*: The *Create Artifact* statement defines *Artifact Classes* as a list of data attributes and states. The *Create Service* statement defines *Services* by specifying their *Input, Output, Pre-condition, and Effect (IOPE)* in a similarly way to *Semantic Web Services*. The *Create Rule* statement defines *Event-Condition-Action (ECA)* rules that invoke *Services* or change artifact states.

The *AML* instantiates, manipulates, and interrogates artifact instances using the following statements: *New, Update, Insert Into, Remove From, Delete,* and *Retrieve.* The *New* statement instantiate a new artifact instance and invoke its *Stream Services*. *Update*, *Update In*, *Insert Into*, *Remove From*, and *Delete* statements are used to manipulate simple, complex, and reference attributes and artifact instance states. On the other hand, since stream attributes are append-only bags of elements that are neither persisted nor modified, stream attributes cannot be manipulated. Finally, the *Retrieve* statement retrieves artifact instances and allows the specification of conditions and a *Sliding Window* on stream attributes.

In the following sub-sections, we describe, define, and provide examples for *AQL* statements. We also specify their context-free grammars where production rules are separated by semi-colon symbols (;). Moreover, the name of a rule is written at the left-side of a colon symbol (:), while its expansion is written at the right-side. Finally, variables are written in uppercase and bold styles, while terminal symbols are written in normal style and are enclosed by double quotes.

Table 3.1 AQL Statements

Statement	Syntax
Create Artifact	**Create Artifact** <name> **Attributes** <list of attributes> **States** <list of states>
Create Service	**Create Service** <name> **Input** <list of artifacts> **Output** <list of artifacts> **Precondition** <expression> **Effect** <expression>
Create Rule	**Create Rule** <name> (**On** <event> | **On** <event> **If** <condition> | **If** <condition>) (**Change State Of** <artifact> **To** <state> | **Invoke** <list of services>)
New	**New** <artifact> <list of simple attributes> **Values** <list values> {<complex attribute> **Include** <list of tuples>} {<reference attribute> **Having** <condition>} {<stream attribute> **Using** <stream service>} [**Set state to** <state>]
Retrieve	**Retrieve** <list of attributes> **From** <list of artifacts> [**Where** <condition>] [**Within** <range>]
Update	**Update** <artifact> **Set** <list of assignments> [**Where** <condition>]
Insert Into	**Insert** <attribute> **Into** <artifact> <list of tuples> [**Where** <condition>]
Remove From	**Remove** <attribute> **From** <artifact> [**Where** <condition>]
Delete	**Delete** <artifact> [**Where** <condition>]

3.2.1. Artifact Definition Language

In this section, we present the statements of the *Artifact Definition Language (ADL)* which are used to declaratively specify *Artifact Systems*.

3.2.1.1. Create Artifact Statement

The *Create Artifact* statement is used to define an *Artifact Class* with respect to its formal model. It consists of a list of data attributes and a list of states.

The list of data attributes supports the four categories or datatypes; *simple, complex, reference, and stream*. Simple attributes are specified using the "*name:type*" syntax. Complex attributes are specified using the "*name₁:{name₁:type₁, name₂:type₂,…}*" syntax. The cardinality of the complex attribute is set to one or to many by appending respectively "*As One*" or "*As Many*" to the complex attribute. Reference attributes are specified using the "*name:artifact*" syntax. Stream attributes are specified using the "*name₁:{name₂:TimeStamp, name₃:type₁,…} As Stream*" syntax in which one of the attributes must be of the *TimeStamp* type.

The list of states supports the specification of *initial*, *final*, or *intermediate* states. The *initial state* is specified using the "*As Initial State*" keyword whereas *final states* are specified using the "*As Final State*" keyword. The other states are by default intermediate states. Figure 3.1 illustrates the context-free grammar of the *Create Artifact* statement.

```
CREATEARTIFACT: "Create Artifact" BANAME
                ATTRIBUTECLAUSE
                STATECLAUSE;
ATTRIBUTECLAUSE: "Attributes (" ATTRIBUTELIST ")";
ATTRIBUTELIST: ATTRIBUTE | ATTRIBUTE "," ATTRIBUTELIST;
ATTRIBUTE: ATTRIBUTENAME ":" ATTRIBUTETYPE;
ATTRIBUTETYPE: SIMPLETYPE | COMPLEXTYPE | REFERENCETYPE | STREAMTYPE;
SIMPLETYPE: "Boolean" | "Integer" | "Real" | "String" | "Date";
COMPLEXTYPE: "{" ATTRIBUTELIST "}" ("As One" | "As Many");
STREAMTYPE: "{" ATTRIBUTELIST ","
                ATTRIBUTENAME ": TimeStamp } As Stream";
REFERENCETYPE: BANAME;
STATESCLAUSE: "States (" STATELIST ")";
STATELIST: STATE | STATE "," STATELIST;
STATE: STATENAME
       | STATENAME "As Initial State"
       | STATENAME "As Final State";
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
STATENAME: IDENTIFIER;
IDENTIFIER: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Figure 3.1 *Create Artifact* Statement Grammar

Example 3.9 (Create Artifact Statement) *The Create Artifact query defining the FireControlArtifact (FCA) is defined as follow:*

```
Create Artifact FCA
Attributes (
  FireControlArtifactId : Integer,
  FireDate : Date,
  FireDuration : Integer,
  AreWaterEjectorsActivated : Boolean,
  IsAlarmTurnedOn : Boolean,
  House : { Address : String, Surface : Real } As One,
  Habitats : { Name : String, PhoneNum : Integer } AS Many,
  FireStationAlert : FireStationAlertArtifact,
  IndoorTemperature : { Time : TimeStamp,
                        Tmp : Integer } As Stream,
  SmokeLevel : { Time : TimeStamp,
                 Lvl :  Integer } As Stream
)
States (
  Normal As Initial State,
  FireDetected,
  PrimaryProcedurePerformed,
  FireExtinguished As Final State
)
```

3.2.1.2. Create Service Statement

The *Create Service* statement defines a *Service* by specifying its *Input, Output, Pre-condition,* and *Effect (IOPE)*. *Input* is a list of comma separated *Artifact Classes* that are read by the *Service*. *Output* is a list of comma separated *Artifact Classes* that are modified or instantiated by the service. *Precondition* is the condition on the *Input* artifacts that holds before the invocation of the service. *Effect* is the condition on the *Output* artifacts that holds after the invocation of the service. Condition expressions are formed from the conjunctions of the following predicates using the *And* keyword: *opened(Attribute)*, *closed(Attribute)*, *defined(Attribute)*, *notDefined(Attribute)*, *new(Artifact)*, and scalar comparison predicates *(>, <, ≤, ≥, =, ≠)* where *closed* and *notDefined* predicates denotes respectively the negation of *opened* and *defined* predicates. Figure 3.2 illustrates the context-free grammar of the *Create Service* statement.

```
CREATESERVICE: "Create Service" SNAME
       INPUTCLAUSE
       OUTPUTCLAUSE
       PRECONCLAUSE
       EFFECTCLAUSE;
INPUTCLAUSE: "Input" BANAMELIST;
OUTPUTCLAUSE: "Output" BANAMELIST;
BANAMELIST: BANAME | BANAME "," BANAMELIST;
PRECONCLAUSE: "Precondition" PREDICATELIST;
EFFECTCLAUSE: "Effect" PREDICATELIST;
PREDICATELIST: PREDICATE | PREDICATE "and" PREDICATELIST;
PREDICATE: NEWPRED | DEFPRED | NDEFPRED | OPEPRED | CLOPRED | SCALPRED;
NEWPRED: "new(" BANAME ")";
DEFPRED: "defined(" ATTRIBUTEIDENTIFICATION ")";
NDEFPRED:"notDefined(" ATTRIBUTEIDENTIFICATION ")";
OPEPRED: "opened(" ATTRIBUTEIDENTIFICATION ")";
CLOPRED: "closed(" ATTRIBUTEIDENTIFICATION ")";
SCALPRED: ATTRIBUTEIDENTIFICATION SCALAROP CONSTANTVALUE;
SCALAROP: "=" | "<" | ">" | "<=" | ">=";
ATTRIBUTEIDENTIFICATION: ATTRIBUTENAME
       | BANAME "." ATTRIBUTEIDENTIFICATION
       | ATTRIBUTENAME "." ATTRIBUTEIDENTIFICATION;
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
IDENTIFIER: CONSTANTVALUE;
CONSTANTVALUE: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Figure 3.2 *Create Service* Statement Grammar

Example 3.10 (Create Service Statement 1) *The CreateFCA service creates a new instance of the FireControlArtifact (FCA) and defines its FireControlArtifactId, House, and Habitats. The corresponding Create Service query is defined as follow:*

```
Create Service CreateFCA
Input  -
Output FCA
Precondition -
Effect new(FCA)
  And defined(FCA.FireControlArtifactId)
  And defined(FCA.House)
  And defined(FCA.Habitats)
```

Example 3.11(Create Service Statement 2) *The IssueFireStationAlert service creates a new instance of the FireStationAlertArtifact (FSAA) as a child artifact of the corresponding FireControlArtifact (FCA). It defines its FireStationAlertArtifactId and House attributes. It is defined using the following Create Service query:*

```
Create Service IssueFireStationAlert
Input FCA
Output FCA, FSAA
Precondition notdefined(FCA.FireStationAlert)
Effect new(FCA. FireStationAlert)
```

```
And defined(FSAA.FireStationAlertingArtifactId)
And defined(FSAA.House)
```

Example 3.12 (Create Service Statement 3) *The StreamIndoorTemperature service streams data tuple from temperature sensors into the IndoorTemperature attribute of the corresponding FireControlArtifact. Its Create Service query is defined as follow:*

```
Create Service StreamIndoorTemperature
Input FCA
Output FCA
Precondition closed(FCA.IndoorTemperature)
Effect opened(FCA.IndoorTemperature)
```

3.2.1.3. Create Rule Statement

The *Create Rule* statement is used to define *Artifact Rules*. There are two types of rules:

- "***If*** *condition* ***Invoke*** *services*", and
- "***If*** *condition* ***Change State of*** *artifact* ***To*** *state*".

An optional "***On*** *event*" clause can be appended to rules. The event in this case represents an external, timely, or user generated event, i.e., creation of a new application, submission of required documents, etc.

In the case that the "***On*** *event*" clause is specified, the "***If*** *condition*" clause can be omitted. The condition expression is thus formed from the conjunction of the following predicates: *state(Artifact, State)*, *defined(Selector)*, *notDefined(Selector)*, and scalar comparison predicates ($>$, $<$, $\leq$, $\geq$, $=$, $\neq$) where *Selector* is a cascading reference to attributes inside artifacts, such as *Selector = {Artifact.Attribute1, Artifact. Attribute1.Attribute2}*. *services* are a list of comma separated services to be invoked using the syntax "*servicename(Artifact$_1$, Artifact$_2$, ...)*" where the list of input artifacts is specified between parenthesis. Figure 3.3 illustrates the context-free grammar of the *Create Rule* statement.

```
CREATERULE: "Create Rule" RNAME
        (ONCLAUSE ACTCLAUSE
        | ONCLAUSE IFCLAUSE ACTCLAUSE
        | IFCLAUSE ACTCLAUSE);
ONCLAUSE: "On" EVNAME;
IFCLAUSE: PREDICATELIST;
PREDICATELIST: PREDICATE | PREDICATE "and" PREDICATELIST
    | PREDICATE "or" PREDICATELIST | "(" PREDICATELIST ")";
PREDICATE: STATEPRE | DEFPRED | NDEFPRED | SCALPRED;
STATEPRE: "state(" BANAME "," STATENAME ")";
DEFPRED: "defined(" ATTRIBUTEIDENTIFICATION ")";
NDEFPRED: "notDefined(" ATTRIBUTEIDENTIFICATION ")";
SCALPRED: ATTRIBUTEIDENTIFICATION SCALAROP CONSTANTVALUE;
SCALAROP: "=" | "<" | ">" | "<=" | ">=";
ATTRIBUTEIDENTIFICATION: ATTRIBUTENAME
        | BANAME "." ATTRIBUTEIDENTIFICATION
        | ATTRIBUTENAME "." ATTRIBUTEIDENTIFICATION;
ACTCLAUSE: INVOKESERVICESCLAUSE | CHANGESTATECLAUSE;
INVOKESERVICESCLAUSE: "Invoke" SERVICEINVLIST;
SERVICEINVLIST: SERVICEINVOKATION
        | SERVICEINVOKATION "," SERVICEINVLIST;
SERVICEINVOKATION: SERVNAME   "(" CONSTANTVALUELIST ")";
CONSTANTVALUELIST: CONSTANTVALUE | CONSTANTVALUE "," CONSTANTVALUELIST;
CHANGESTATECLAUSE: "Change State of" BANAME "To" STATENAME;
RNAME: IDENTIFIER;
EVNAME: IDENTIFIER;
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
STATENAME: IDENTIFIER;
IDENTIFIER: CONSTANTVALUE;
CONSTANTVALUE: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Figure 3.3 *Create Rule* Statement Grammar

Example 3.13 (Create Rule Statement 1) *Rule 1 invokes the CreateFCA service when a Create Fire Control Artifact Event is received. It is defined using the following Create Rule query:*

```
Create Rule r1
On CreateFireControlArtifactEvent
Invoke CreateFCA()
```

Example 3.14 (Create Rule Statement 2) *Rule 2 changes the state of FireControlArtifact to the Normal state if it is initialized and its FireControlArtifactId, House, and Habitats attributes are defined. Its Create Rule query is defined as follow:*

```
Create Rule r2
If state(FCA, Initialized)
  And defined(FireControlArtifactId)
  And defined(House)
  And defined(Habitats)
Change State of FCA To Normal
```

Example 3.15 (Create Rule Statement 3) *Rule 3 changes the state of FireControlArtifact to the FireDetected state if it is in the Normal state, house temperature sensor values are higher than 57°C, and smoke sensor levels exceed a threshold of 3. It is defined using the following Create Rule query:*

```
Create Rule r3
If state(FCA, Normal)
  And FCA.IndoorTemperature.Tmp >= 57
  And FCA.SmokeLevel.Lvl >= 3
Change State of FCA To FireDetected
```

Example 3.16 (Create Rule Statement 4) *Rule 4 invokes the TurnOnAlarm and ActivateWaterEjectors services if a FireControlArtifact is in the FireDetected state and its IsAlarmTurnedOn and AreWaterEjectorsActivated attributes are not defined. Its corresponding Create Rule query is defined as follow:*

```
Create Rule r4
If state(FCA, FireDetected)
  And notdefined(FCA.IsAlarmTurnedOn)
  And notdefined(FCA.AreWaterEjectorsActivated)
Invoke TurnOnAlarm(FCA), ActivateWaterEjectors(FCA)
```

3.2.2. Artifact Manipulation Language

In this section, we present the statements of the *Artifact Manipulation Language (AML),* which are used to declaratively manipulate and query artifact instances.

3.2.2.1. New Statement

The *New* statement instantiates a new artifact instance from an artifact class, initializes its attribute values and states, and invokes its stream services. The *New* statement has several modes of usages that can be combined in order to initialize:

1. Some or all of the simple attributes using: "*(attribute₁, attribute₂, …)* ***Values*** *(value₁, value₂, …)*".
2. Complex attributes by using the syntax: "*attribute* ***Include*** { *(value₁, value₂, …), (value₃, value₄, …), …*}" where a list of tuples is inserted into the complex attribute relation.
3. Reference attributes using: "*attribute* ***Having*** *(condition)*" where the child artifact referenced by attribute should satisfy condition.

4. State of the artifact using: "***Set State To*** *state*".

In addition, the *New* statement is used to invoke stream services on stream attributes using: "*attribute* ***Using*** *service*". The context-free grammar of the *New* statement is illustrated in Figure 3.4.

```
NEW: "New" BANAME
     SIMPLEATTCLAUSE COMPLEXATTCLAUSE? REFERENCEATTCLAUSE?
     STREAMATTCLAUSE? STATECLAUSE?;
SIMPLEATTCLAUSE:"(" ATTRIBUTENAMELIST ")Values(" CONSTANTVALUELIST ")";
ATTRIBUTENAMELIST: ATTRIBUTENAME | ATTRIBUTENAME "," ATTRIBUTENAMELIST
CONSTANTVALUELIST: CONSTANTVALUE | CONSTANTVALUE "," CONSTANTVALUELIST;
COMPLEXATTCLAUSE: COMPLEXATTRIBUTELIST;
COMPLEXATTRIBUTELIST: COMPLEXATTRIBUTE
             | COMPLEXATTRIBUTE " " COMPLEXATTRIBUTELIST;
COMPLEXATTRIBUTE: ATTRIBUTENAME "Include" "{" TUPLELIST "}";
TUPLELIST: TUPLE | TUPLE "," TUPLELIST;
TUPLE: "(" CONSTANTVALUELIST ")";
REFERENCEATTCLAUSE: REFERENCEATTRIBUTELIST;
REFERENCEATTRIBUTELIST: REFERENCEATTRIBUTE |
     REFERENCEATTRIBUTE  " " REFERENCEATTRIBUTELIST;
REFERENCEATTRIBUTE: ATTRIBUTENAME "Having (" CONDITION ")";
CONDITION: CONDITIONPREDICATELIST;
CONDITIONPREDICATELIST: CONDITIONPREDICATE |
       CONDITIONPREDICATE "And" CONDITIONPREDICATELIST;
CONDITIONPREDICATE: ATTRIBUTENAME PREDICATEOP CONSTANTVALUE;
PREDICATEOP: "=" | "<" | ">" | "<=" | ">=";
STREAMATTCLAUSE:  STREAMATTRIBUTELIST;
STREAMATTRIBUTELIST: STREAMATTRIBUTE
             | STREAMATTRIBUTE  " " STREAMATTRIBUTELIST;
STREAMATTRIBUTE: ATTRIBUTENAME "Using" SERVICEINVOKATION;
SERVICEINVOKATION: SERVNAME;
STATECLAUSE: "Set State To" STATENAME;
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
SERVNAME: IDENTIFIER;
IDENTIFIER: CONSTANTVALUE;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER
       | CONSTANTVALUE DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Figure 3.4 *New* statement Grammar

Example 3.17 (New Statement) *In the following, the New query creates an instance of the FireControlArtifact (FCA). The simple attribute FireControlArtifactId is initialized to the value of 100235, the tuple ("20 Av. Albert Einstein", 64) is inserted into its House complex attribute, and two tuples ("John", 00330675839457) and ("Sam", 00330625374883) are inserted into its Habitats complex attribute. Moreover, its FireStationAlert reference attribute is initialized to refer to the FireStationAlertArtifact instance having100200 as the value of its FireStationAlertArtifactId. The StreamIndoorTemperature service is passed the current artifact instance and is invoked on the IndoorTemperature stream attribute. Similarly, StreamSmokeLevel is invoked on SmokeLevel stream attribute. Finally, the state of the artifact instance is set to Normal.*

```
New FCA
(FireControlArtifactId) Values (100235)
House Include {("20 Av. Albert Einstein", 64)}
Habitats Include { ("John", 00330675839457),
           ("Sam", 00330625374883) }
FireStationAlert Having FireStationAlertArtifactId = 100200
IndoorTemperature Using StreamIndoorTemperature
SmokeLevel Using StreamSmokeLevel
Set State To Normal
```

3.2.2.2. Retrieve Statement

The *Retrieve* statement selects tuples that meet certain conditions from artifact relations, in addition to related tuples from complex attributes, stream attributes and child artifact relations. The condition of the *Retrieve* statement is specified using the "***Where*** *condition*" clause. Three types of filtering condition are introduced:

1. Conditions on simple, complex, stream attributes and states are possible using scalar comparison operators and *state* predicates. Simple attributes inside complex and stream attributes can be accessed using cascading references, i.e., *Artifact.Attribute1.Attribute2*.
2. Conditions on complex attributes using the *Include* keyword where the complex attribute is tested for containing certain tuples;
   "*complex attribute* ***Include {****tuple list****}***".
3. Conditions on reference attributes using the *Having* keyword; "*attribute* ***Having*** *(condition)*" where the reference *attribute* should match *condition*.

Additionally, the *Retrieve* statement supports the specification of *Sliding Windows* when stream attributes are involved. The *Sliding Window* is specified using the optional *"**Within** range"* clause where range is a time interval. If no *Sliding Window* is specified using the *"**Within** range"* clause, then the current instance (a.k.a. ***now***) *Sliding Window* is applied by default. The context-free grammar of the *Retrieve* statement is illustrated in Figure 3.5.

```
RETRIEVE: "Retrieve" ATTRIBUTENAMELIST
          "From" BANAME
          WHERECLAUSE?
          WITHINCLAUSE?;
WHERECLAUSE: "Where" WHEREPREDICATELIST;
WHEREPREDICATELIST: WHEREPREDICATE
       | WHEREPREDICATE "And" WHEREPREDICATELIST
       | "(" WHEREPREDICATELIST  ")";
WHEREPREDICATE: STATEPREDICATE | DEFPREDICATE | NOTDEFPREDICATE
       |COMPARISIONPREDICATE | INCLUDEPREDICATE | HAVINGPREDICATE;
STATEPREDICATE: "state (" STATENAME ")";
DEFFPREDICATE: "defined (" ATTRIBUTEIDENTIFICATION ")";
NOTDEFPREDICATE: "notDefined (" ATTRIBUTEIDENTIFICATION ")";
COMPARISIONPREDICATE: ATTRIBUTEIDENTIFICATION PREDICATEOP CONSTANTVALUE;
PREDICATEOP: "=" | "!=" | "<" | ">" | "<=" | ">=";
INCLUDEPREDICATE: ATTRIBUTENAME "Include {" TUPLELIST "}";
TUPLELIST: TUPLE | TUPLE "," TUPLELIST;
TUPLE: "(" CONSTANTVALUELIST ")";
CONSTANTVALUELIST: CONSTANTVALUE | CONSTANTVALUE "," CONSTANTVALUELIST;
HAVEPREDICATE: ATTRIBUTENAME "Having (" CONDITION ")";
CONDITION: CONDITIONPREDICATELIST;
CONDITIONPREDICATELIST: CONDITIONPREDICATE
       | CONDITIONPREDICATE "And" CONDITIONPREDICATE;
CONDITIONPREDICATE: WHEREPREDICATE;
ATTRIBUTEIDENTIFICATION: ATTRIBUTENAME
       | BANAME "." ATTRIBUTEIDENTIFICATION
       | ATTRIBUTENAME "." ATTRIBUTEIDENTIFICATION;
WITHINCLAUSE: "Within" RANGE TIMEUNIT;
TIMEUNIT: "Seconds" | "Minutes" | "Hours" | "Days";
BANAME: IDENTIFIER;
STATENAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
IDENTIFIER: CONSTANTVALUE;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER
       | CONSTANTVALUE DIGIT;
RANGE: DIGIT | RANGE DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Figure 3.5 *Retrieve* statement Grammar

Example 3.18 (Retrieve Statement 1) *The following Retrieve query selects all the attributes of the FireControlArtifact instances that are in the FireDetected state and their indoor temperature is higher than 100°C over a window of 10 seconds.*

```
Retrieve *
From FCA
Where state(FCA, FireDetected)
      And FCA.IndoorTemperature.Tmp > 100
Within 10 Seconds
```

Example 3.19 (Retrieve Statement 2) *The following Retrieve query retrieves the IndoorTemperature stream attribute of the FireControlArtifact instances that are in the Normal state over a window of 3 seconds.*

```
Retrieve IndoorTemperature
From FCA
Where state(FCA, Normal)
Within 3 Seconds
```

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

3.2.2.3. Remaining Artifact Manipulation Statements

The remaining statements of the *AML* manipulate artifact instances and include statements such as *Update*, *Insert Into*, *Remove From*, and *Delete*. The *Update* statement is used to update simple and complex attributes and states of artifacts. The *Insert Into* statement is used to insert tuples into complex attributes. It is also used to insert child artifact references into reference attributes. On the other hand, the *Remove From* statement is used to remove tuples from complex attributes and child references from reference attributes. Finally, the *Delete* statement is used to entirely delete artifact instances including values of their complex, reference, and stream attributes. Since streams are append-only bags that are neither persisted nor modified, stream attributes cannot be manipulated using the described statements. Figure 3.6 illustrates the context-free grammar of the remaining manipulation statements where the **WHERECLAUSE** production rule is the same as in Figure 3.5.

```
UPDATE: "Update"  BANAME
         SETCLAUSE
         WHERECLAUSE;
SETCLAUSE: SETSTATE | SETATTRIBUTES;
SETSTATE: "Set State To" STATENAME;
SETATTRIBUTES: "Set" ATTRIBUTEASSIGNMENTLIST;
ATTRIBUTEASSIGNMENTLIST: ATTRIBUTEASSIGNMENT
                | ATTRIBUTEASSIGNMENT "," ATTRIBUTEASSIGNMENTLIST;
ATTRIBUTEASSIGNMENT: ATTRIBUTENAME "="  CONSTANTVALUE;
INSERT: "Insert" ATTRIBUTENAME
        "Into" BANAME
        COMPLEXATTCLAUSE?
        WHERECLAUSE;
COMPLEXATTCLAUSE: "{" TUPLELIST "}";
REMOVE: "Remove" ATTRIBUTENAME
        "From" BANAME
        WHERECLAUSE;
DELETE: "Delete" BANAME
        WHERECLAUSE;
TUPLELIST: TUPLE | TUPLE "," TUPLELIST;
TUPLE: "(" CONSTANTVALUELIST ")";
BANAME: IDENTIFIER;
STATENAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
IDENTIFIER: CONSTANTVALUE;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER
               | CONSTANTVALUE DIGIT;
```

Figure 3.6 Manipulation statements Grammar

Example 3.20 (Update Statement) *The following Update query updates the phone number of the habitat with the name "John" of the FireControlArtifact instance having 100325 as id.*

```
Update FCA
Set Habitats.PhoneNum = 0033763423758
```

```
Where Habitats.Name = "John"
  And FCA.FireControlArtifactId = 100325
```

Example 3.21 (Insert Into Statement) *The following Insert Into query inserts two tuples into the Habitats complex attribute of the FireControlArtifact instance having 100325 as id.*

```
Insert Habitats Into FCA
{ ("Sebastien", 0033823459876),
  ("Nicole", 003357643214) }
Where FCA.FireControlArtifactId = 100325
```

Example 3.22 (Remove From) *The following Remove From query delete a tuple having "John" as value of the Name attribute from the Habitats complex attribute of the FireControlArtifact instance having 100325 as id.*

```
Remove Habitats From FCA
Where FCA.FireControlArtifactId = 100325
  And Habitats.Name = "John"
```

Example 3.23 (Delete Statement) *The following Delete query deletes the FireControlArtifact instance having 100325 as id including all its complex reference and stream attributes values.*

```
Delete FCA
Where FCA.FireControlArtifactId = 100325
```

3.3. Artifact Query Language Semantics

3.3.1. Preliminaries

This section describes the semantics of the *AQL* which formally define the functioning of the *AQL* statements and how they should be executed by an *AQL processor*. Since *Artifact Systems* are based on *Database* relations, we specify the semantics of *AQL* using fundamental *Mathematical Logic* [Mend09] and *Relational Algebra* [Maie83] concepts.

We start by assuming the existence of the following pairwise disjoint countably infinite sets: $\mathcal{D}$ for constants; i.e. data values. $\mathcal{C}$ of artifact names, $\mathcal{A}$ of attribute names, $\mathcal{Q}$ of artifact states, $\mathcal{Y}$ of simple data types, including: *Boolean*, *Integer*, *Real*, *String*, *Date*, and *TimeStamp*, and $\mathcal{S}$ of services names. We also make uses of the definitions made in Section 3.1 related to *Artifact System W*, *Artifact Class c*, *Service s*, and *Artifact Rule r*.

We also give some simple notations for relations and relation schemas. For a given relation schema *R*, we denote by **schema**(R) $\subseteq \mathcal{A}$ the set of attributes in *R*. The primary key of *R* is denoted by ***key(R)*** $\subseteq$ ***schema(R)***. A

tuple t over R is an element of $\mathcal{D}^{|schema(R)|}$, and a relation r over R is a finite set of tuples over R such that $r \subseteq \mathcal{D}^{|schema(R)|}$. We also assume the existence of a relation *states* over a relation schema *States* used to store information about states of lifecycles with ***schema****(States)={Artifact, State, Type}* and ***key****(States)={Artifact, State}*.

Since the *AQL* is used to manipulate and query an underlying model of relations and streams, we also make use of the relational and stream algebra operators: *Selection*, *Projection*, *Cartesian Product*, *Window,* and *Assignment*. As described in [AbHV95, ArBW06], these operators provide the necessary functionalities in order to manipulate and query relations and streams. The *Selection* operator is denoted by $\sigma_{c(r)}$ where a subset of tuples that meet condition c is selected from the relation r. The *Projection* operator is denoted by $\pi_{a1,\ldots,an}(r)$ where the result is a relation of n attributes obtained by erasing from the relation r the attributes that are not listed in $a_1,\ldots,a_n$. The *Cartesian Product* operator is denoted by $r_1 \times r_2$ where the result is a relation that combines r_1 and r_2. *Window* is denoted by $\mathcal{W}_{range}(s)$ where the result is a subset relation obtained by returning the stream tuples with a timestamp matching *range* from the stream s. Relational algebra expressions can be constructed using *Selection*, *Projection*, *Cartesian Product* and *Window* operators in addition to mathematical union and set difference operators. The *Assignment* operator is denoted by $r \leftarrow E$ where the result of the relational algebra expression E is assigned to the relation r. Using the assignment operator, we can define insert, delete and update operations on relations. Inserting a tuple t into a relation r is defined as $r \leftarrow r \cup t$. Deleting a tuple t from a relation r is defined as $r \leftarrow r - t$. Updating a tuple t in a relation r is defined as $r \leftarrow r - t \cup t'$ where t' is the updated tuple.

3.3.2. The Artifact Definition Language

In this section, we formally define how the underlying model of relations and streams, representing an *Artifact System*, is created using the statements of the *Artifact Definition Language (ADL)*.

3.3.2.1. Create Artifact Statement

The *Create Artifact* statement of the *ADL* defines *Artifact Classes* according to Definition 3.2. The semantics of executing a *Create Artifact* query is defined as follow:

1. A relation schema R_c representing an *Artifact Class* c is created. The schema of R_c contains the simple attributes of c such that $\textbf{\textit{schema}}(R_c) = \{a \mid a \in A_s\}$. Additionally, R_c contains two more attributes: $a_{pk}=\textbf{\textit{concat}}(c, \text{"\_PK"})$ is the primary key of R_c such that $\textbf{\textit{key}}(R_c)=a_{pk}$, and $a_{st}=State$ is the current state of the artifact. Taking the *Create Artifact* query of Example 3.9 as an example, we obtain the following relation schema:

   FCA(FCA_PK, FireControlArtifactId, FireDate, FireDuration, AreWaterEjectorsActivated, IsAlarmTurnedOn, State)

2. For every complex attribute a_c such that $a_c \in A_c$, we create an associated relation schema R_{ac} containing the simple attributes constituting a_c such that $\textbf{\textit{schema}}(R_{ac})=\{a \mid a \in \gamma_{com}(a_c)\}$. Additionally, $\textbf{\textit{schema}}(R_{ac})$ contains a primary key attribute a_{cpk} such that $\textbf{\textit{key}}(R_{ac})=a_{cpk}$ and $a_{cpk}=\textbf{\textit{concat}}(a_c, \text{"\_PK"})$. Moreover, $\textbf{\textit{schema}}(R_{ac})$ also contains a reference to the artifact relation in the form of a foreign key a_{cfk} of R_c such that $a_{cfk}=\textbf{\textit{concat}}(c, \text{"\_FK"})$. In reference to the *Create Artifact* query of Example 3.9, we obtain the following relation schemas:

   House(House_PK, FCA_FK, Address, Surface)
   Habitats (Habitats_PK, FCA_FK, Name, PhoneNum)

3. For every reference attribute a_r of c such that $a_r \in A_r$, we create an associated relation schema R_{ar} that contains foreign keys of parent and child artifacts such that $\textbf{\textit{schema}}(R_{ar})=\{a_{parent}, a_{child} \mid a_{parent}=\textbf{\textit{concat}}(c, \text{"\_PFK"})$ *and* $a_{child}=\textbf{\textit{concat}}(\gamma_{ref}(a_r), \text{"\_CFK"})\}$. Additionally, both foreign keys form the primary key of R_{ar} such that $\textbf{\textit{key}}(R_{ar})=\{a_{parent}, a_{child}\}$. Taking the *Create Artifact* query of Example 3.9, we obtain the following relation schema:

   FireStationAlert(FCA_PFK, FSAA_CFK)

4. For every stream attribute a_t of c such that $a_t \in A_t$, we create an associated relation schema R_{at} that contains the simple attributes constituting a_t such that $\textbf{\textit{schema}}(R_{at})=\{a \mid a \in \gamma_{str}(a_t)\}$. Moreover, $\textbf{\textit{schema}}(R_{at})$ also contains a reference to the artifact relation in the form of a foreign key a_{tfk} of R_c such that $a_{tfk}=\textbf{\textit{concat}}(c, \text{"\_FK"})$. On the other hand, since stream data may not be unique, the relational schema R_{at} does not have a primary key. Taking the *Create Artifact* query of Example 3.9 as an example, we obtain the following relation schema:

   IndoorTemperature (FCA_FK, Time, Tmp)
   SmokeLevel (FCA_FK, Time, Lvl)

5. For every state q of c, we insert a tuple t into the *states* relation such that:
   - $states \leftarrow states \cup \{(c, q, \text{“”})\}$ *if* $q \in Q$ *and* $q \neq s$ *and* $q \notin F$.
   - $states \leftarrow states \cup \{(c, q, \text{“initial”})\}$ *if* $q \in Q$ *and* $q = s$.
   - $states \leftarrow states \cup \{(c, q, \text{“final”})\}$ *if* $q \in Q$ *and* $q \in F$.

3.3.2.2. Create Service Statement

The *Create Service* statement of the *ADL* defines *Services* according to Definition 3.3. The semantics of invoking and executing a *Service* s is defined as follow:

1. When a *Service* s is invoked, primary keys of the artifacts included in the input list C_I are passed as parameters.
2. The *Service Precondition* P is evaluated on the input artifacts specified in the input list C_I according to the following predicate semantics:
   - The predicate *defined(c.A)* checks if *the* attribute A of c has a value and is defined when *Service* s is executed.
   - The predicate *notDefined(c.A)* checks if *the* attribute A of c does not have a value and is undefined when *Service* s is executed.
   - The *opened* predicate, *opened(c.A),* checks if *the* stream attribute A of c is receiving stream readings when *Service* s is executed.
   - The *closed* predicate, *closed(c.A),* checks if *the* stream attribute A of c is not receiving stream readings when *Service* s is executed.
   - The scalar comparison predicates $(>, <, \leq, \geq, =, \neq)$ checks if the left and right operands match with the specified operator.

   As a result, if the input artifacts C_I do not match with the *Service Precondition* P then the service execution is aborted.
3. The primary keys of overlapping artifacts between the input artifact list C_I and the output artifact list C_O are copied from the input artifact list C_I into the corresponding output artifacts in the output artifact list C_O.
4. The *Service Effect* E is executed according to the following predicate semantics:

- The *new* predicate *new(c)* creates a new instance of *Artifact Class* c and assigns its primary key to the corresponding output artifact $o \in C_0$.
- if the *new* predicate is of the form *new(c.A)* where A is a reference attribute of the artifact instance referenced by an output artifact $o_1 \in C_0$ of the *Artifact Class* c, then a new instance of *Artifact Class* $\gamma_{ref}(A)$ is created as a child artifact of o_1 and its primary key is assigned to the corresponding output artifact $o_2 \in C_0$ referencing the *Artifact Class* $\gamma_{ref}(A)$.
- The *defined* predicate, *defined(c.A),* assigns a simple value or tuple(s) to the simple attribute A (respectively the complex attribute A) of the artifact instance of the *Artifact Class* c referenced by the output artifact $o \in C_0$. The assigned value can be retrieved from one of three sources which are implemented in the prototype (see chapter 6) : 1) A GUI Form, 2) A Web Service Call, or 3) A User Defined Function.
- The scalar comparison predicates $(>, <, \leq, \geq, =, \neq)$ checks if values assigned by the *defined* predicates match certain conditions and if not, the service effect is rolled back and the invocation is aborted.
- The *opened* predicate, *opened(c.A)*, describing the effect of a *Stream Service* continuously reads into the stream attribute A of the artifact instance of the *Artifact Class* c identified by an output artifact $o \in C_0$. The stream source can be specified using one of two choices implemented in the prototype (see chapter 6) : 1) A Web Service Call, or 2) A text file.

3.3.2.3. Create Rule Statement

The *Create Rule* statement of the *ADL* defines *Artifact Rules* according to Definition 3.6. The semantics of executing *Artifact Rules* is defined as follow:

1. The set of *Artifact Rules* R of an *Artifact System* W is re-evaluated every time a manipulation operation is performed on the relational database model of an artifact instance i, or when a user-generated or timely event e related to an artifact instance i is created using the *Graphical User Interface* of the prototype (see chapter 6).

2. All *Artifact Rules* that do not involve the *Artifact Class* c of the artifact instance i are discarded.
3. If a user-generated or timely event e_1 is created then all the *Artifact Rules* that do not have an *event* predicate $event(e_2)$ in their conditions such that $e_1 = e_2$ are discarded.
4. The *state* predicates $state(c, q_1)$ of the remaining *Artifact Rules* is matched with the state q_2 of the artifact instance i of *Artifact Class* c. If $q_1 \neq q_2$ then the *Artifact Rule* is discarded.
5. Predicates involving simple attributes (respectively complex attributes) of an *Artifact Class* c in the condition of the remaining *Artifact Rules* are matched with the simple attribute values of the artifact instance i of *Artifact Class* c. *Artifact Rules,* which do not meet the simple attributes (respectively complex attributes) predicate conditions, are thus discarded.
6. Predicates involving stream attributes of *Artifact Class* c in the condition of the remaining *Artifact Rules* are matched with the last (or current) tuple of the corresponding stream attribute relations of artifact instance i of *Artifact Class* c. *Artifact Rules* that do not meet the stream attributes predicate conditions are discarded.
7. Predicates involving reference attributes of *Artifact Class* c in the condition of the remaining *Artifact Rules* are matched with the corresponding child artifact instances of the artifact instance i of *Artifact Class* c. *Artifact Rules* that do not meet the reference attributes predicate conditions are discarded.
8. Action parts of the remaining *Artifact Rules* are executed as follow:
   - If the action changes the state of an artifact using a *state* predicate $state(c, q)$, then an update operation that changes the *State* attribute of the corresponding artifact instance relation to the q state is performed.
   - If the action invokes some services using the *invoke* predicate $invoke(S)$, then the set of services S are invoked and the primary keys of the input artifacts are passed as parameters.

3.3.3. Artifact Manipulation Language

In this section, we formally define how the underlying model of relations and streams are manipulated and queried using the statements of the *Artifact Manipulation Language (AML)*.

3.3.3.1. New Statement

The *New* statement of *AML* instantiates an artifact instance of an *Artifact Class* c by inserting necessary tuples into the different relations constituting the artifact relational model. The semantics of executing a *New* query is defined as follow:

1. Values of simple attributes specified in the *"(attribute$_1$, attribute$_2$, ...)* ***Values*** *(value$_1$, value$_2$...)"* clause are inserted in a new tuple into the artifact main relation r_c and its automatically generated primary key is retained such that: $r_c \leftarrow r_c \cup \{(k_{parent}, v_1, \dots, v_n, state)\}$ where k_{parent} is the automatically generated primary key of the artifact instance. If the state is not yet specified in the query, the state of the artifact is set to ***Initialized***. Similarly, if the state is specified in the query, it is validated using the expression: $\sigma_{Artifact=c \wedge State=state}(states)$. The retained primary key k_{parent} is used as the foreign key for the insert operation.
2. For every complex attribute tuple clause specified in *"attribute* ***Include*** *(tuple$_1$, tuple$_2$...)"*, an insert operation is performed on the corresponding complex attribute relation r_{ac} such as: $r_{ac} \leftarrow r_{ac} \cup \{(k_{ac1}, k_{parent}, tuple_1), (k_{ac2}, k_{parent}, tuple_2), \dots\}$ where k_{ac1} and k_{ac2} are automatically generated primary keys of the inserted tuples and k_{parent} is the retained foreign key of the parent artifact.
3. Similarly, for every reference attribute clause *"attribute* ***Having*** *condition"*, an insert operation is performed on the corresponding reference attribute relation r_{ar} such as: $r_{ar} \leftarrow r_{ar} \cup \{(k_{parent}, k_{child})\}$. In this case, the k_{parent} is the retained foreign key of the parent artifact and k_{child} is the retrieved foreign key of the child artifact. The k_{child} is retrieved according to the specified condition using the expression: $\pi_{Artifact\_PK}(\sigma_{condition}(\gamma_{ref}(attribute)))$.
4. Finally, for every stream attribute invocation clause *"attribute* ***Using*** *service(k_{parent})"*, the *service* is invoked and passed the primary key k_{parent} of the parent artifact instance in order to continuously stream data readings into the specified stream attribute relation.

3.3.3.2. Update Statements

The *Update* statement of *AML* updates simple and complex artifacts' attributes, in addition to the artifact states. The semantics of executing an *Update* query is defined as follow:

1. The required artifact instance tuple is retrieved from the artifact relation r_c using a selection operation: $t \leftarrow \sigma_{condition}(r_c)$ where condition is the condition specified in the query.
2. An update operation involving the simple attributes and state of the artifact instance is performed on the artifact relation such as: $r_c \leftarrow r_c - t \cup t'$ where t′ is the tuple t updated with the new values of the simple attributes and state of the artifact instance.
3. Updating a complex attribute relation r_{ac} is done by first retrieving the complex attribute tuple from the complex attribute relation using a *Cartesian Product* operation such as:

$$t \leftarrow \pi_{schema(r_{ac})}(\sigma_{condition \wedge Artifact\_PK=Artifact\_FK}(r_c \times r_{ac}))$$

4. An update operation can be performed on the complex attribute relation r_{ac} such as: $r_{ac} \leftarrow r_{ac} - t \cup t'$ where t′ is the tuple t updated with the new values of the simple attributes of the complex attribute.

3.3.3.3. Insert Into Statements

The *Insert Into* statement of the *AML* inserts tuples into complex or reference attributes relations. The semantics of executing the *Insert Into* query is defined as follow:

1. In order to insert a tuple $(value_1,...,value_n)$ into a complex attribute, the primary key of the corresponding artifact is retrieved using projection and selection operations such as: $k_{parent} \leftarrow \pi_{Artifact\_PK}(\sigma_{condition}(r_c))$. An insert operation is then performed on the complex attribute relation r_{ac} such as: $r_{ac} \leftarrow r_{ac} \cup \{(k_{ac}, k_{parent}, value_1,...,value_n)\}$.
2. In order to insert a reference into a reference attribute, the primary keys of the parent and child artifacts are retrieved using projection and selection operations: $k_{parent} \leftarrow \pi_{Artifact\_PK}(\sigma_{cparent}(r_{parent}))$ where *cparent* is the condition related to the parent artifact. And $k_{child} \leftarrow \pi_{Artifact\_PK}(\sigma_{cchild}(r_{child}))$ where *cchild* is the condition related to the child artifact. An insert operation is then performed on the reference attribute relation r_{ar} as follow: $r_{ar} \leftarrow r_{ar} \cup \{(k_{parent}, k_{child})\}$.

3.3.3.4. Remove Statements

The *Remove From* statement of the *AML* deletes tuples from complex or reference attribute relations. The semantic of executing a *Remove From* query is defined as follow:

1. Removing a tuple t from a complex attribute relation r_{ac} is performed similarly to the *Update* statement for complex attributes. But, a delete operation is used instead of an update operation such as: $r_{ac} \leftarrow r_{ac} - t$.
2. On the other hand, removing a tuple from a reference attribute relation r_{ar} is performed similarly to the insert statement for reference attributes. But, a delete operation is used instead of an insert operation: $r_{ar} \leftarrow r_{ar} - \{(k_{parent}, k_{child})\}$.

3.3.3.5. Delete Statement

The *Delete* statement deletes instances of a given artifact class or schema, in addition to all related children artifacts (cascade deletion.

3.3.3.6. Retrieve Statement

The *Retrieve* statement selects tuples that meet conditions from the artifact System. The semantics of executing a *Retrieve* query is defined as follow:

1. Tuples from the artifact relation that meet condition on simple attributes and state of the artifact are selected as follow: $r_1 \leftarrow \sigma_{cparent}(r_c)$ where *cparent* is the condition related to the simple artifact attributes and states.
2. As for conditions on artifact complex attributes, further selections are performed on the *Cartesian Product* of r_1 and related complex attribute relation r_{ac} such as: $\sigma_{ccomplex \wedge Artifact\_PK=Artifact\_FK}(r_1 \times r_{ac})$ where *ccomplex* is the condition related to the complex attribute.
3. Similarly, for conditions on artifact reference attributes, a selection is performed on the Cartesian Product of r_1, the reference attribute relation r_{ar}, and the artifact relation r_c such as: $\sigma_{cchild \wedge r1.Artifact\_PK=Artifact\_PFK \wedge Artifact\_CFK=artifact.Artifact\_PK}(r_1 \times r_{ar} \times r_c)$.
4. Finally, for conditions on artifact stream attributes, first a window operator is applied to the stream attribute relation r_{at} such that $\mathcal{W}_{range}(r_{at})$ where *range* is specified by the *"Within range"* clause of

the *Retrieve* query. If the *"Within range"* clause is not specified then the default current window (a.k.a *now*) is specified by default such as $\mathcal{W}_{now}(r_{at})$. Further selections are then performed on the *Cartesian Product* of r_1 and the windowed stream attribute relation $\mathcal{W}_{range}(r_{at})$ such as: $\sigma_{cstream \wedge Artifact\_PK=Artifact\_FK}(r_1 \times \mathcal{W}_{range}(r_{at}))$ where *cstream* is the condition related to the stream attribute.

3.4. Summary of Specifying Artifact Systems

In this chapter, we propose a formal model for *Artifact Systems* that is specifically adapted to extended artifacts adequate to the *IoT* in a simple and intuitive manner. The formal model supports data streams generated by sensors and actuators which are parts of *IoT* based processes. Moreover, the formal model allows definitions, manipulations, and querying of *Artifact Systems* at a high-level without dealing with underlying details of relations and streams. The formal model makes use of four high-level attribute categories; *simple*, *complex*, *reference*, and *stream*. By such, stream attributes are used to represent data streams. Complex attributes are used to represent complex relationships between artifacts, and reference attributes which are used to represent *Parent-Child* relationships between artifacts. Moreover, the formal model presents two types of *Services* to respectively perform actions on actuators and stream data from various sensors. The proposed *Artifact System* serves thus as the basis of the *Artifact Integration Framework* that is further described in the remaining chapters.

We also present the *Artifact Query Language (AQL)* to declaratively define *Artifact Systems* and manipulate artifact instances without dealing with low level relations and streams. The *AQL* provides simple and declarative statements that hide underlying database relations and streams. Statements are grouped into the *Artifact Definition Language (ADL)*, and *the Artifact Manipulation Language (AML)*. The *ADL* provides statements to define *Artifact Systems*. The *AML* provides statements to manipulate and query artifact instances. Moreover, the *AML* will be used in order to uniformly manipulate and query distributed *Artifact Systems* using the *Unified Views* in the *Artifact Integration Framework*.

In the next chapter, we will model *Artifact systems* to promote process awareness through simple graphical notations and enable the *Artifact Integration Framework* with *Unified Views* that serve as centralized access platforms to supervise and manage distributed *Artifact Systems*.

4

Modeling Artifact Systems

Chapter Outline

As described in Chapter 3, *Artifact Systems* can be specified by writing appropriate *AQL* queries. We extend *AQL* queries with conceptual models. In fact, conceptual models of representing artifact-based processes are visual-oriented approach to practical and user-friendly design that can be automatically translated into *Artifact Systems*. They can be also effectively used for integrating heterogeneous *Artifact Systems* and generating *Unified Views* for supervising, managing, and querying heterogeneous *Artifact Systems* in a uniform manner.

Existing artifact modeling notations and frameworks such as [DaHV13, LLQS10, LoNy11] have several limitations. Firstly, their conceptual models do not present a holistic representation of *Artifact System's* components: *Artifacts (Information Models)*, *Lifecycles*, *Services*, and *Associations*. Instead, *Lifecycles*, *Services* and *Associations* are often included in the conceptual models while *Information Models* are defined separately from the conceptual model. As a result, these models cannot be effectively used for generating representative *Unified Views* that serve as centralized supervision and management platforms for artifact-centered processes.

Moreover, existing artifact modeling notations fall under two categories; procedural and declarative notations. Procedural modeling notations like [CDHP08, EQST12, NKMH10] are based on simple finite-state machines and they fail to provide customizable and flexible frameworks. Declarative modeling notations like [ABGM09, DaHV13, YoLZ11] are based on customizable *Event-Condition-Action (ECA) Rules* but still limited and cannot easily provide simple and representative models that support process awareness.

In addition, they are used to model *Business Process Models* with workflow patterns as described in [LiBW07]. These patterns lack data stream capabilities and thus are not suitable for modeling smart services and smart processes, integrating data streams.

In order to overcome the disadvantages of existing artifact modeling notations and frameworks, we propose the *Conceptual Artifact Modeling Notation (CAMN)* to construct *Conceptual Artifact Models (CAM)*. The *CAMs* include all components of *Artifact Systems* into the same model and thus provides a holistic process representation. Moreover, *CAMs* are procedural models based on simple finite-state machines that can be automatically translated into declarative *Artifact Systems* based on customizable *Event-Condition-Action (ECA) Rules*. As a result, our proposed modeling framework combines the advantages of both procedural and declarative modeling approaches and eliminates their disadvantages.

Furthermore, we propose a set of modeling patterns to include data stream specific modeling patterns and ensure the generation of valid *Artifact Systems*. The remaining sections of Chapter 4 are organized as follow:

In Section 4.1, we introduce the *Conceptual Artifact Modeling Notation (CAMN)* which is based on six modeling primitives.

In Section 4.2, we describe the modeling patterns that ensure the generation of valid *Artifact Systems*.

In Section 4.3, we describe the semantics of generating *Artifact Systems* from the *Conceptual Artifact Models (CAM)* based on the set of modeling patterns.

4.1. Conceptual Artifact Modeling Notation

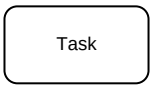
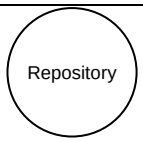
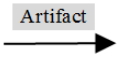
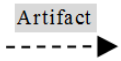
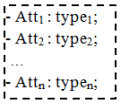
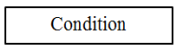
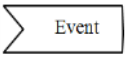
In Table 4.1, we summarize the *Conceptual Artifact Modeling Notation (CAMN)* constructs and their graphical representations to design *Conceptual Artifact Models (CAMs)*. The notation's main focus is to capture artifact *Lifecycles* by describing how artifact instances flow between *CAMN* constructs.

The graphical notation is based on six modeling constructs: *Task*, *Repository*, *Flow connector* (read-only and read/write), *Data Attribute List*, *Condition*, and *Event*. Using these constructs, an artifact-centric process can be represented at a conceptual level by modeling interacting artifact *Lifecycles*.

1. *Tasks* correspond to *Services* in *Artifact Systems* and represent units of work to be performed in order to manipulate artifact instances and evolve them in their lifecycle thus achieving goals.
2. *Repositories* denote state-based storage locations into which artifacts can be stored, awaiting for future processing. For every artifact state in the lifecycle, an associated *Repository* is used to store all artifact instances that are in that state. Artifact instances can then be pushed into or pulled from particular *Repositories* as needed using *Flow Connectors*.
3. *Flow Connectors* connect *Tasks* and *Repositories* and allow artifact instances to be transferred between them. Read/write *Flow Connectors* indicate that artifact instances are transferable between tasks and repositories where they are manipulated and evolved with respect to their lifecycles. Read-only *Flow Connectors* indicate that artifact

content is required in read-only mode and no modification is performed, thus the artifact instance remains in the same *Repository*. Figure 4.1 (a) illustrates a *Repository* connected to a *Task* through a read/write *Flow Connector*.

Table 4.1 Conceptual Artifact Modeling Notation (CAMN)

Modeling Construct	Graphical Notation	Description
Task		Units of work operating on artifacts
Repository		State-based storage locations for artifacts
Read/write Flow Connector		Transit artifacts between tasks and repositories
Read-only Flow Connector		Read artifact content from a repository
Data Attribute List		List of attribute-type pairs that are manipulated by a Task
Condition		Conditions associated to flow connectors
Event		Event associated to flow connectors

4. *Data Attribute Lists* are associated to *Tasks* and describe the set of data attributes of artifact's *Information Models* that are manipulated by the *Task*. Simultaneous definitions of artifact's *Information Model* and *Lifecycle* in the same conceptual model leads us to building artifact processes incrementally without dealing with fine-grained details related to artifact models. Additionally, the aggregation of *Data Attribute Lists* also allows the generation of *Information Models* of interrelated artifacts. Data attributes are written as *"artifact.attribute:type"* triplets. Figure 4.1 (b) depicts a *Data Attribute List* attached to a *Task*.
5. *Events* are attached to *Flow Connectors* and specify received external events that trigger activation of *Flow Connectors*. For example in the

fire control process, a *Create Fire Control Artifact* event causes the invocation of the *CreateFCA* task.

6. *Conditions* are also attached to *Flow Connectors* and specify constraints that should be satisfied in order to activate a *Flow Connector*. The condition expresses constraints over artifact attributes by using the *defined*, *notdefined* and scalar comparison predicates ($>$, $<$, $\leq$, $\geq$, $=$, $\neq$). Figure 4.1 (c) illustrates an *Event* and a *Condition* constructs attached to a *Flow Connector*.

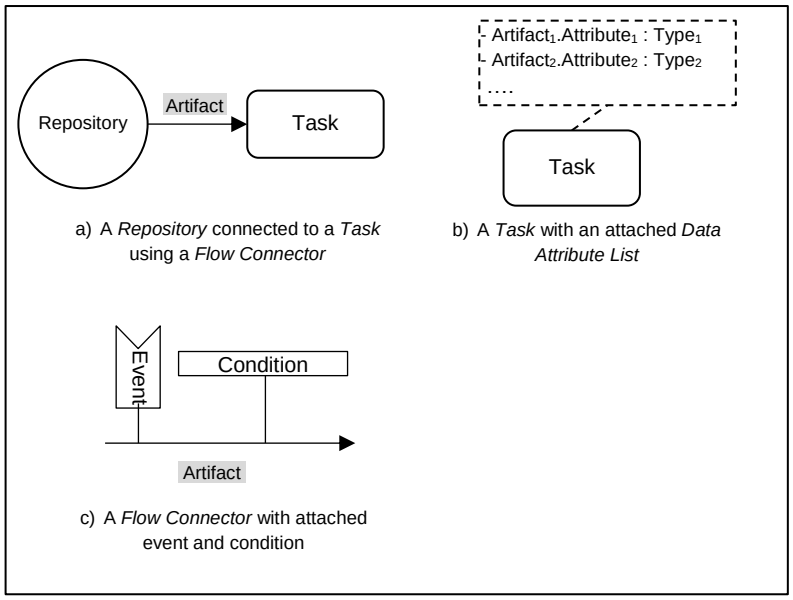


Figure 4.1 Examples of CAMN Combinations

Using the six *CAMN* constructs, a *Conceptual Artifact Model (CAM)* can be constructed as illustrated in Figure 4.2 and depicts part of the fire control process. First, when the *CreateFireControlArtifact (CFCA)* event is received, the *CreateFCA Task* is invoked. An instance of the *FireControlArtifact (FCA)* is created, its *FireControlArtifactId*, *House*, and *Habitats* attributes are also defined. The instance is then passed into the *Normal Repository*. If the indoor temperature becomes greater than 57 and smoke level becomes greater than 3, the *FCA* instance is passed from the *Normal Repository* into the *FireDetected Repository*. Consequently, the *TurnOnAlarm* and *ActivateWaterEjectors Tasks* are invoked. Finally, if the alarm is successfully turned on and water ejectors are successfully activated, the *FCA* instance is passed into the *PrimaryProcedurePerformed Repository*. Otherwise, the *FCA* instance is passed into the *Failure repository*. Moreover, a *StreamIndoorTemperature Task* continuously streams readings from a

temperature sensor into the *IndoorTemperature* stream data attribute starting at artifact instantiation.

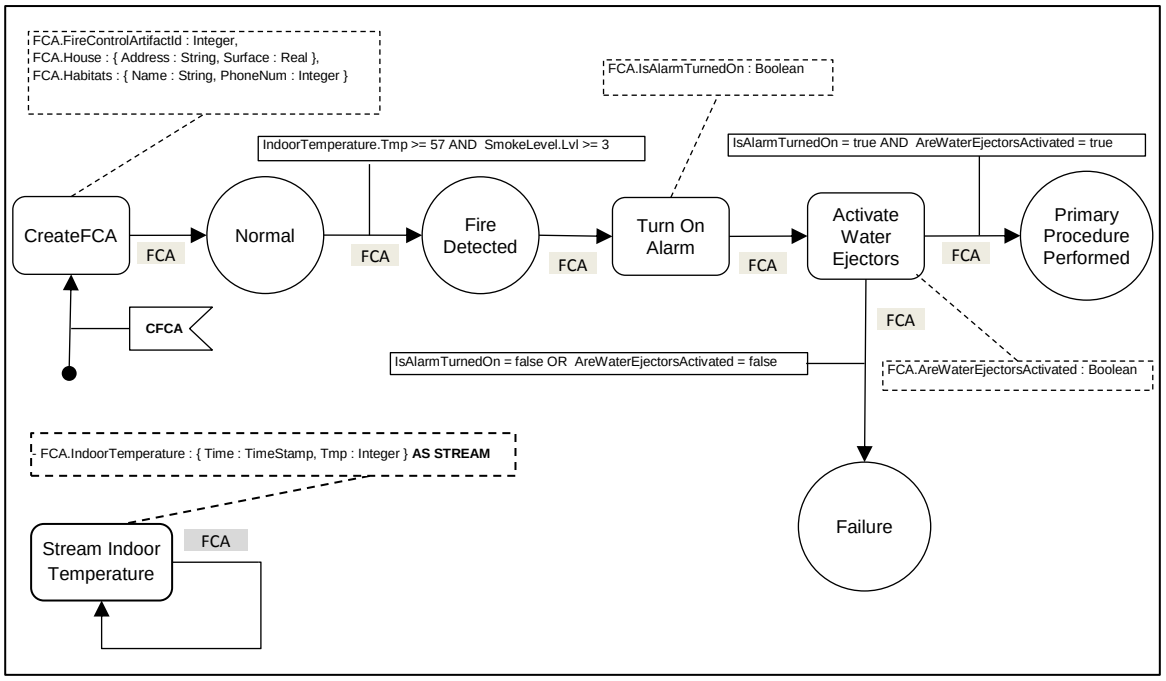


Figure 4.2 Part of Fire Control Conceptual Artifact Model

4.2. Modeling Patterns

We have defined a set of modeling patterns that ensure the generation of valid *Artifact Systems* and provide best practices and modeling guidelines. The set of modeling patterns deal with various operational semantics like transitions, creation, branching, and iteration based on the modeling patterns described in [LiBW07].

4.2.1. Transition Patterns

Transition patterns deal with passing artifact instances between *Tasks* and *Repositories*. We define four types of transition patterns: *Repository-to-Task Transition Pattern*, *Task-to-Repository Transition Pattern*, *Task-to-Task Transition Pattern*, and *Repository-to-Repository Transition Pattern*.

4.2.1.1. Repository-to-Task Transition Pattern

In the *Repository-to-Task Transition Pattern*, a *Task* is invoked when an input artifact instance is available in its input *Repository*. In this case, the artifact instance is passed from the *Repository* into the invoked *Task* through a read/write *Flow Connector*. Figure 4.3 illustrates the *Repository-to-Task Transition Pattern.*

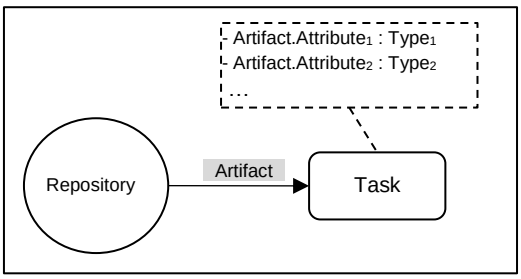


Figure 4.3 Repository-to-Task Transition Pattern

Figure 4.4 illustrates an example of the *Repository-to-Task Transition Pattern* from the fire control scenario. When a *FireControlArtifact* instance is in the *FireDetected Repository*, it is passed into the *TurnOnAlarm Task* which defines its *IsAlarmTurnedOn Boolean* attribute.

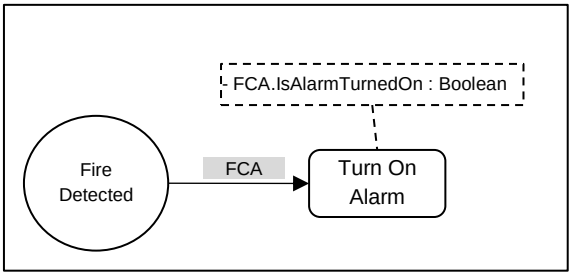


Figure 4.4 Repository-to-Task Transition Pattern Example

4.2.1.2. Task-to-Repository Transition Pattern

In the *Task-to-Repository Transition Pattern*, an artifact instance is sent to a *Repository* after it has been manipulated by a *Task* using read/write *Flow Connector*. Figure 4.5 illustrates the *Task-to-Repository Transition Pattern.*

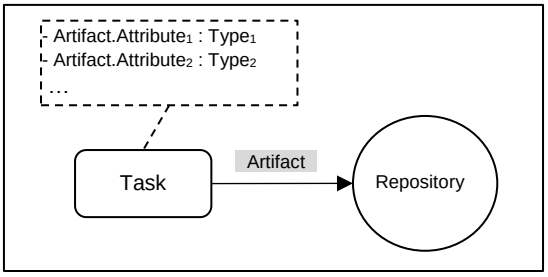


Figure 4.5 Task-to-Repository Transition Pattern

Figure 4.6 illustrates an example of the *Task-to-Repository Transition Pattern* from the fire control scenario. After a *FireControlArtifact (FCA)* instance is manipulated by the *ActivateWaterPumps Task* which defines its *AreWaterPumpsActivated Boolean* attribute, it is then passed into the *EjectorsRefilled Repository*.

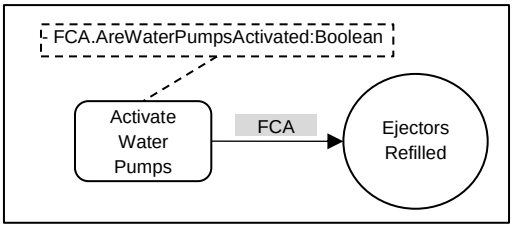


Figure 4.6 Task-to-Repository Transition Pattern Example

4.2.1.3. Task-to-Task Transition Pattern

In the *Task-to-Task Transition Pattern*, an artifact instance is manipulated by two *Tasks* in sequence. First, the artifact instance is manipulated by a *Task*, and then it is directly sent using a read/write *Flow Connector* into a second *Task* without passing by a *Repository*. Figure 4.7 illustrates the *Task-to-Task Transition Pattern.*

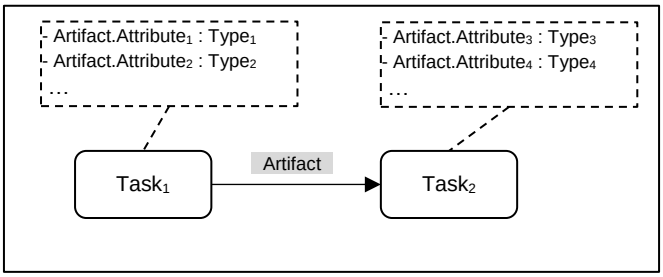


Figure 4.7 Task-to-Task Transition Pattern

Figure 4.8 illustrates an example of the *Task-to-Task Transition Pattern* from the fire control scenario. After a *FireControlArtifact (FCA)* instance is

manipulated by the *TurnOnAlarm Task* which defines its *IsAlarmTurnedOn Boolean* attribute, it is directly passed into the *ActivateWaterEjectors Task* which defines its *AreWaterEjectorsActivated Boolean* attribute.

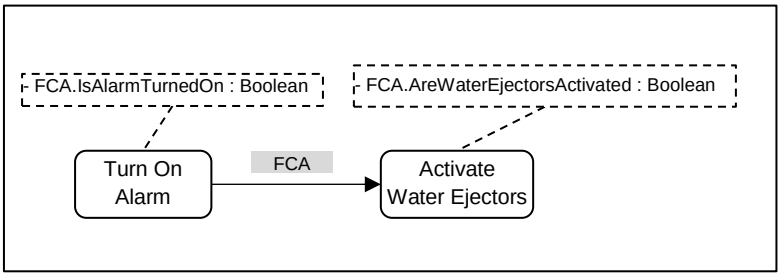


Figure 4.8 Task-to-Task Transition Pattern Example

4.2.1.4. Repository-to-Repository Transition Pattern

In the *Repository-to-Repository Transition Pattern*, an artifact instance in a first *Repository* is directly sent to a second *Repository* when a certain condition is met. The artifact instance passes from the first *Repository* into the second *Repository* when the condition attached to the connecting read/write *Flow Connector* is met. Figure 4.9 illustrates the *Repository-to-Repository Transition Pattern.*

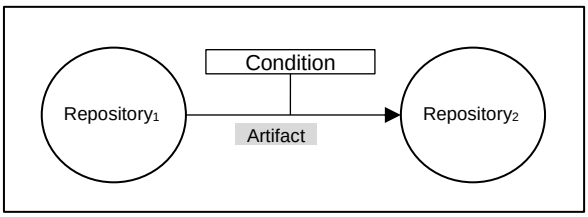


Figure 4.9 Repository-to-Repository Transition Pattern

Figure 4.10 illustrates an example of the *Repository-to-Repository Transition Pattern* from the fire control scenario. When a *FireControlArtifact (FCA)* instance is in the *Normal Repository* and its indoor temperature becomes greater than or equal to 57 and its smoke level becomes greater than or equal to 3, it is passed into the *FireDetected Repository*.

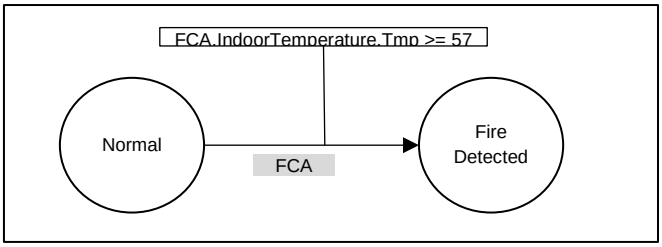


Figure 4.10 Repository-to-Repository Transition Pattern Example

4.2.2. Creation Patterns

Creation patterns deal with creating artifact instances based on the functioning of tasks. We distinguish between the *Parent Artifact Creation Pattern* and the *Child Artifact Creation Pattern.*

4.2.2.1. Parent Artifact Creation Pattern

In the *Parent Artifact Creation Pattern,* a task creates a new artifact instance that is then sent to a *Repository*. In this case, the task has no input artifacts. Additionally, an *Event* can be created to trigger the invocation of the *Task*. Figure 4.11 illustrates the *Parent Artifact Creation Pattern.*

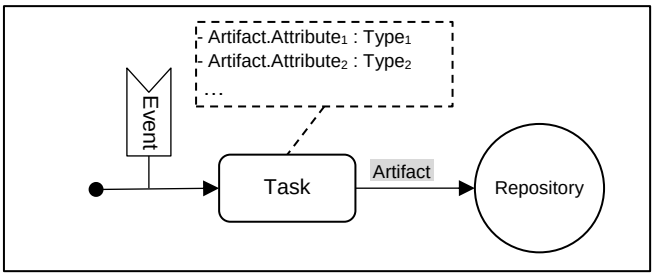


Figure 4.11 Parent Artifact Creation Pattern

Figure 4.12 illustrates an example of the *Parent Artifact Creation Pattern* from the fire control scenario. When a *Create Fire Control Artifact (CFCA)* event is received, the *CreateFCA Task* is invoked in order to create a new instance of the *FireControlArtifact (FCA)* and define its *FireControlArtifactId*, *House*, and *Habitats* attributes.The *FireControlArtifact (FCA)* instance is then passed into the *Normal Repository.*

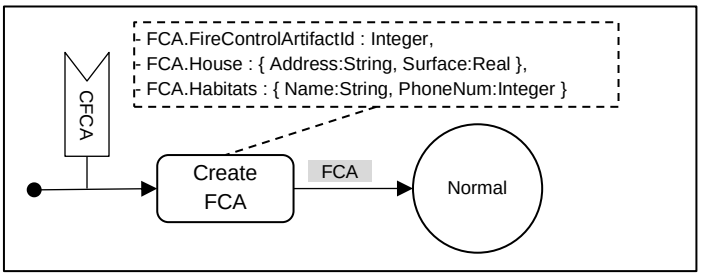


Figure 4.12 Parent Artifact Creation Pattern Example

4.2.2.2. Child Artifact Creation Pattern

In the *Child Artifact Creation Pattern*, a *Task* takes an artifact instance from a *Repository*, manipulates the artifact instance, creates a child artifact instance, and passes both parent and child artifact instances into two different *Repositories*. Figure 4.13 illustrates the *Child Artifact Creation Pattern.*

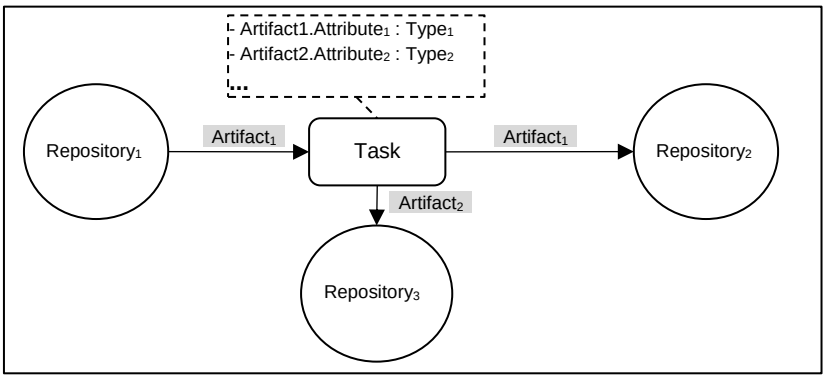


Figure 4.13 Child Artifact Creation Pattern

Figure 4.14 illustrates an example of the *Child Artifact Creation Pattern* from the fire control scenario. When a *FireControlArtifact (FCA)* instance is in the *PrimaryProcedurePerformed Repository*, it is passed into the *IssueFireStationAlert Task* which will create a new artifact instance of the *FireStationAlertArtifact (FSSA)* as a child artifact of the parent *FireControlArtifact (FCA)* instance. The reference to the child artifact instance is stored in the *FireStationAlert* attribute. The *FireStationAlertArtifactId* and *House* attributes of the child artifact instance are also defined. Finally, the parent artifact instance *(FCA)* is passed into the *ClosestFireStationAlerted Repository*, while the child artifact instance *(FSAA)* is passed into the *Issued Repository.*

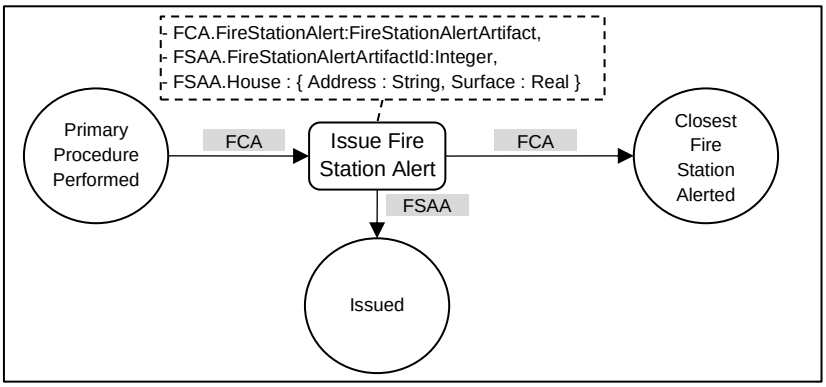


Figure 4.14 Child Artifact Creation Pattern Example

4.2.3. Branch Pattern

In the *Branch Pattern*, an artifact instance can pass through one of two or more divergent *Flow Connectors*. The choice is made according to the conditions attached to the divergent *Flow Connectors*. In this case, the conditions should be disjoint in order to generate valid *Artifact Systems.* In other words, only one of the conditions of the divergent *Flow Connectors* can be true at a time and the others are false. The *Branch Pattern* can be *Task*-centered; the divergent *Flow Connectors* depart from a *Task*. Or, the *Branch Pattern* can be *Repository*-centered; the divergent *Flow Connectors* depart from a *Repository*. Figure 4.15 illustrates the *Task*-centered *Branch Pattern.*

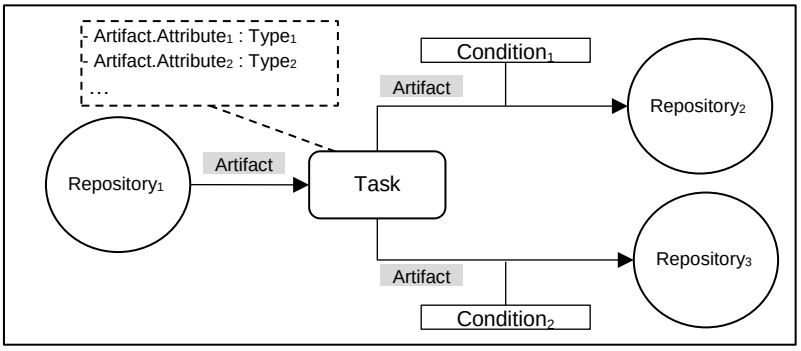


Figure 4.15 Task-centered Branch Pattern

Figure 4.16 illustrates the *Repository*-centered *Branch Pattern.*

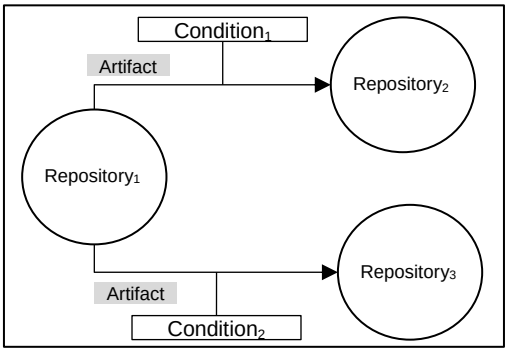


Figure 4.16 Repository-centered Branch Pattern

Figure 4.17 illustrates an example of the *Branch Pattern* from the fire control scenario. When a *FireStationAlertArtifact (FSAA)* instance is in the *Located Repository*, it is passed into the *AlertFireStation Task* which defines its *IsSuccessfullyAlerted Boolean* attribute. If the value of the *IsSuccessfullyAlerted* is true, the *FireStationAlertArtifact (FSAA)* instance is

passed into the *Success Repository*. If it is false, the the *FireStationAlertArtifact (FSAA)* instance is passed into the *Failed Repository.*

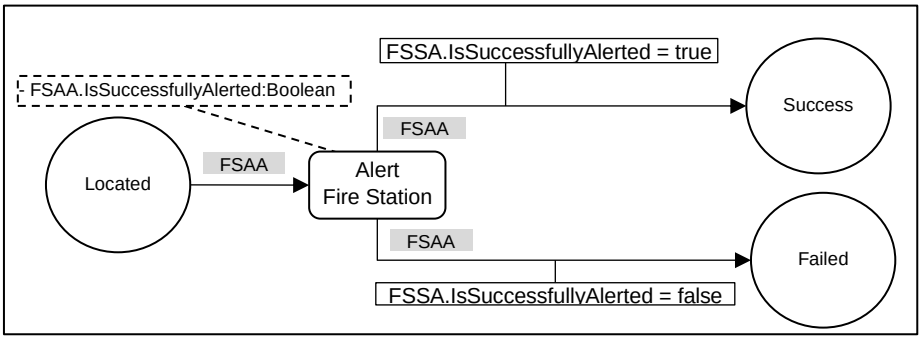


Figure 4.17 Branch Pattern Example

4.2.4. Convergence Pattern

In the *Convergence Pattern*, an artifact instance can arrive into a *Task* or *Repository* from one of two or more convergent *Flow Connectors*. The converging *Flow Connectors* are the result of an earlier *Branch Pattern*. The *Convergence Pattern* can be *Task*-centered; the convergent *Flow Connectors* arrive into a *Task*. Or, the *Convergence Pattern* can be *Repository*-centered; the convergent *Flow Connectors* arrive into a *Repository*. Figure 4.18 illustrates the *Task*-centered *Convergence Pattern.*

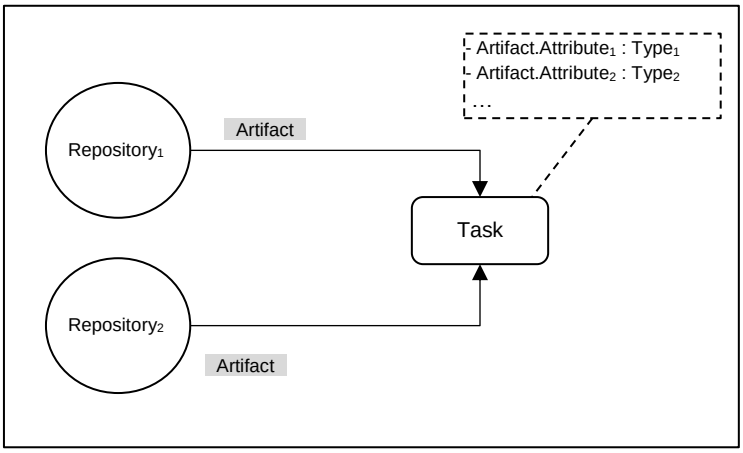


Figure 4.18 Task-centered Convergence Pattern

Figure 4.19 illustrates an example of a *Repository*-centered *Convergence Pattern* from the fire control process. In this example, a *FireControlArtifact (FCA)* instance can be passed into the *FireExtinguished Repository* from the *HabitatsInformed* or the *EjectorsRefilled Repositories* whenever the indoor

temperature becomes less than 50 and the smoke level becomes less than or equal to 1. Since the *Convergence Pattern*, in this example, is made of two *Repository-to-Repository Transition Patterns*, *Conditions* on *Flow Connectors* are mandatory.

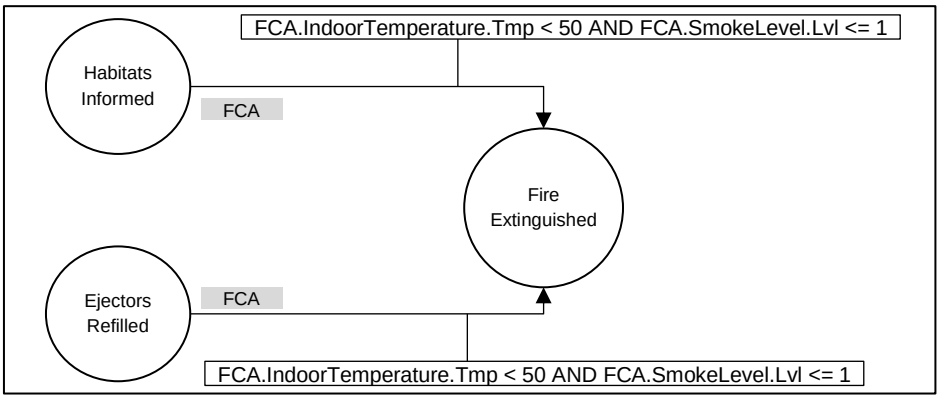


Figure 4.19 Repository-centered Convergence Pattern

4.2.5. Rework Pattern

In the *Rework Pattern*, an artifact instance must repass through one or more previously invoked *Tasks* until certain condition is met. Figure 4.20 illustrates the *Rework Pattern.*

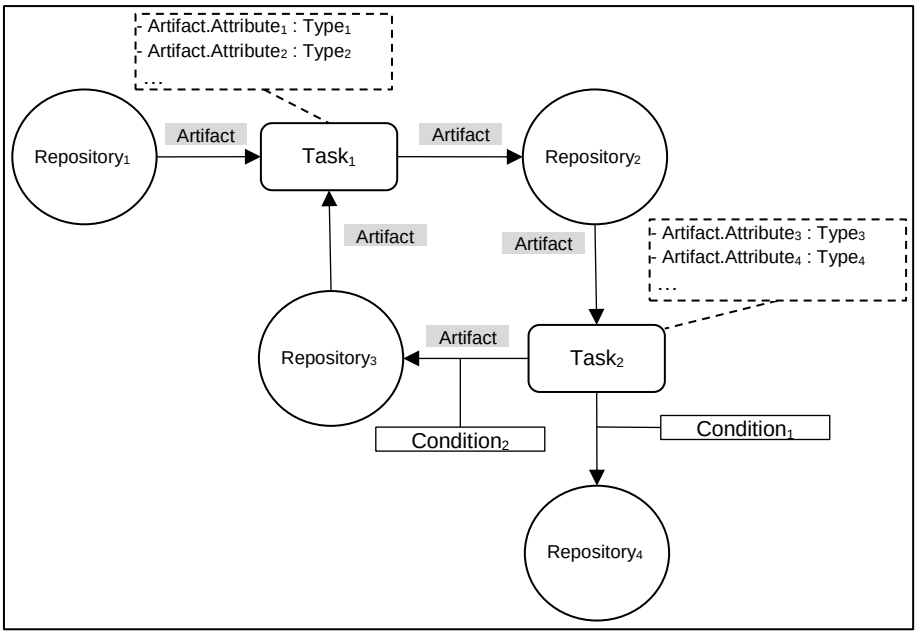


Figure 4.20 Rework Pattern

Figure 4.21 illustrates an example of the *Rework Pattern* from the fire control scenario. A *FireStationAlertArtifact (FSAA)* instance repasses through the *LocateFireStation* and *AlertFireStation Tasks* until a located fire station is successfully alerted.

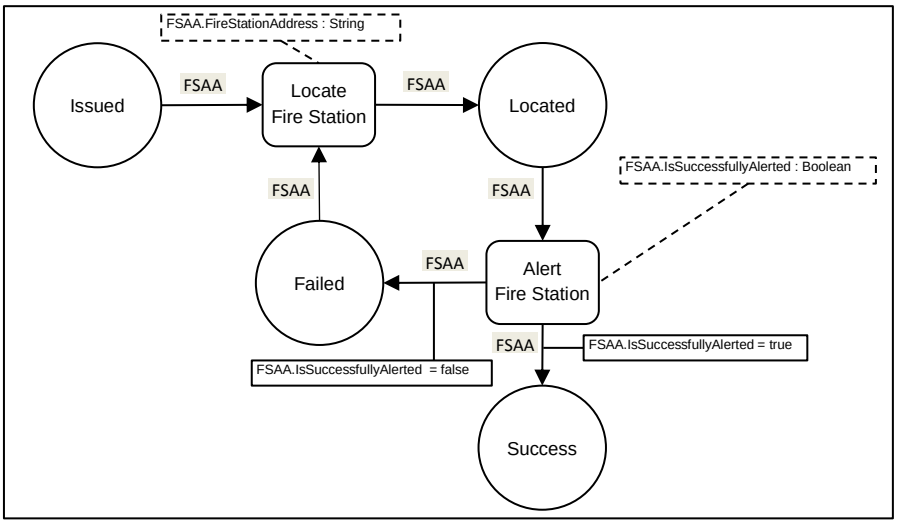


Figure 4.21 Rework Pattern Example

4.2.6. Synchronization Pattern

In the *Synchronization Pattern*, an artifact instance must read data from another artifact instance in order to advance in its *Lifecycle*. In this case, two artifact instances are passed to one *Task*. One of the artifact instances is passed in read/write mode using a read/write *Flow Connector*, while the second artifact instance is passed in a read-only mode using the read-only *Flow Connector*. Then, the *Task* manipulates the read/write artifact instance by reading the content of the read-only artifact instance. Finally, the manipulated artifact instance is passed to a different *Repository*, while the consulted artifact instance remains in the same *Repository*. Figure 4.22 illustrates the *Synchronization Pattern.*

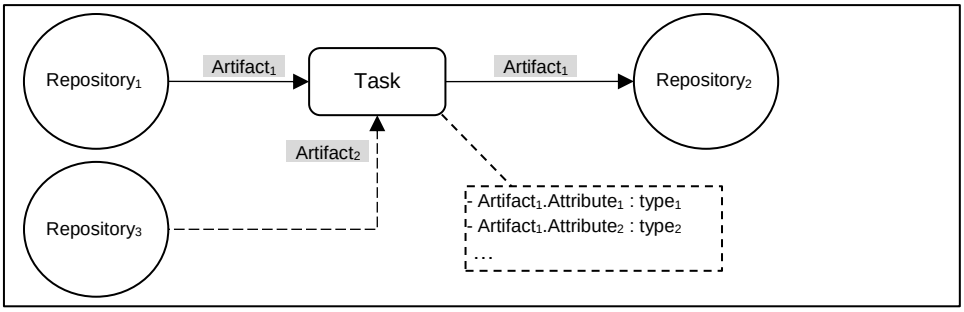


Figure 4.22 Synchronization Pattern

Figure 4.23 illustrates an example of the *Synchronization Pattern* from the fire control scenario. In this example, the *LocateFireStation Task* takes two artifact instances as input; a *FireStationAlertArtifact (FSAA)* instance from the *Issued Repository*, and a *MapArtifact (MPA)* instance from the *Active Repository*. The *FireStationAddress* attribute of the *FireStationAlertArtifact (FSAA)* is updated with the address of the closest fire station by consulting the *MapArtifact (MPA)* and is then passed into the *Located Repository.*

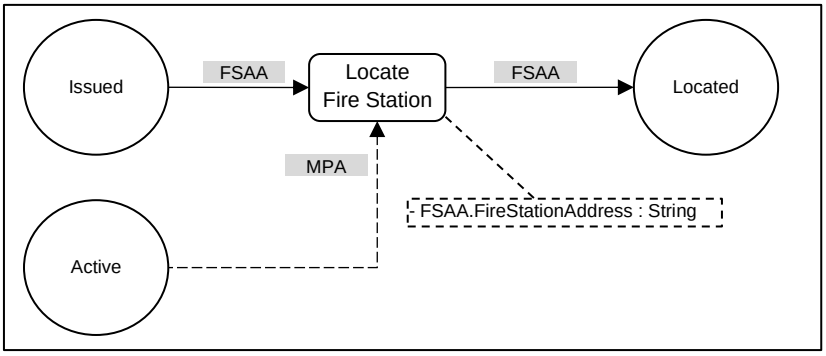


Figure 4.23 Synchronization Pattern Example

4.2.7. Streaming Pattern

In the *Streaming Pattern*, an artifact instance is continuously updated with readings data stream sources. In this case, an artifact instance repeatedly passes in a closed loop between a *Task* and a read-write *Flow Connector*. In this case, the *Task* corresponds with a *Stream Service* in *Artifact Systems* and its *Data Attribute List* has one stream attribute. Figure 4.24 illustrates the *Streaming Pattern*.

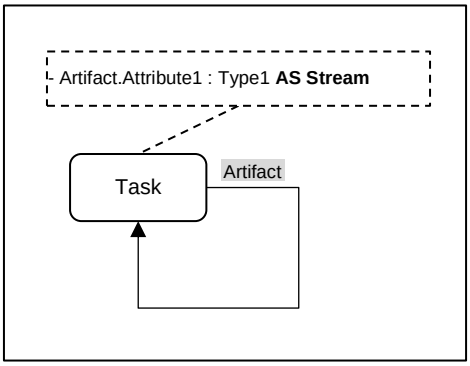


Figure 4.24 Streaming Pattern

Figure 4.25 illustrates an example of the *Streaming Pattern* from the fire control scenario. A *FireControlArtifact (FCA)* instance is continuously updated with readings from a temperature sensor using the

StreamIndoorTemperature Task. In this case, the *"AS STREAM"* keywords specify that the *IndoorTemperature* attribute manipulated by the *StreamIndoorTemperature Task* is a stream attribute.

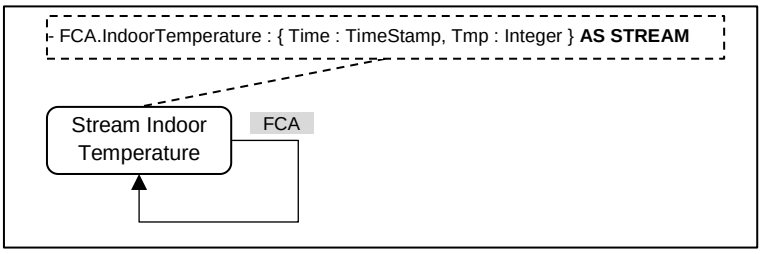


Figure 4.25 Streaming Pattern Example

4.3. Conceptual Artifact Model Semantics

In this section, we define the semantics of generating *Artifact Systems* from *Conceptual Artifact Models (CAMs)* and propose a formalism for *Conceptual Artifact Models (CAMs*). Based on this formalism, we define semantics of generating different elements of an *Artifact System.*

4.3.1. Conceptual Artifact Model

Based on the *Conceptual Artifact Modeling Notation (CAMN)*, we present a formal presentation for *Conceptual Artifact Models (CAMs)*. First, we assume the existence of the following pairwise disjoint countably infinite sets:

- $\mathcal{C}$ of *Artifact Class* names.
- $\mathcal{A}$ of attribute names.
- $\mathcal{K}$ of *Task* names.
- $\mathcal{P}$ of *Repository* names.
- $\mathcal{N}$ of *Condition* expressions.
- $\mathcal{E}$ of *Event* names.
- $\mathcal{Y}$ of simple data types, including: *Boolean*, *Integer*, *Real*, *String*, *Date*, and *TimeStamp;*

Definition 4.1 (*Conceptual Artifact Model*) *A Conceptual Artifact Model (CAM) is an augmented graph G in which Tasks and Repositories are vertices, and Flow Connectors are edges such as:*

$$G = (P, K, W, E, N, \delta, \alpha, \beta)$$

Where:

- $P \subseteq \mathcal{P}$ *is the set of Repositories defined below,*
- $K \subseteq \mathcal{K}$ *is the set of Tasks defined below,*
- W *is the set of Flow Connectors defined below,*
- $E \subseteq \mathcal{E}$ *is the set of Events,*
- $N \subseteq \mathcal{N}$ *is the set of Conditions,*
- δ *is the partial function that maps W to the set of Events E, thus δ attaches Events to some Flow Connectors,*
- α *is the partial function that maps W to the set of Conditions N, thus α attaches conditions to some Flow Connectors,*
- β *is the total function that maps the set of edges (Flow Connectors W) to the set of endpoints (Tasks K and Repositories P) such as:*

  $$\beta : W \rightarrow P \times K \cup K \times P \cup K \times K \cup P \times P \cup \perp \times K$$

  Where $\perp$ is the null symbol. In other words, β specifies the source and destination of Flow Connectors where:

  - $P \times K$ *represents Repository-to-Task Transition Pattern,*
  - $K \times P$ *represents Task-to-Repository Transition Pattern,*
  - $K \times K$ *represents Task-to-Task Transition Pattern,*
  - $P \times P$ *represents Repository-to-Repository Transition Pattern, and*
  - $\perp \times T$ *represents Parent Artifact Creation Pattern in which the Flow Connector have no source and a Task as destination.*

Definition 4.2 (Repository) *A Repository is a tuple (p, c_p), where $p \in P$ is the Repository name, $c_p \in C$ is the Artifact Class associated to the repository. When clear from the context, a Repository (p, c_p) is simply referred to as p.*

Definition 4.3 (Task) *A Task is a tuple $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim})$ where:*

- $k \in \mathcal{K}$ *is the Task name.*

- $A_k \subseteq \mathcal{A}$ *is a finite set of artifact attributes associated to the Task* k. A_k *includes three partitions; a simple attribute partition* A_s, *a complex attribute partition* A_c, *a reference attribute partition* A_r, *and a stream attribute partition* A_t.
- $\gamma_{art} : A_k \rightarrow \mathcal{C}$, *the artifact type function is a total map that maps the attributes in* A_k *to the Artifact Class they belong to in* $\mathcal{C}$.
- $\gamma_{com} : A_c \rightarrow \mathcal{A}^n$, *the complex type function is a partial map that maps the complex attributes in* A_k *to a list of simple attributes in* $\mathcal{A}$.
- $\gamma_{str} : A_t \rightarrow \mathcal{A}^n$, *the stream type function is a partial map that maps the stream type attributes in* A_k *to a list of simple attributes in* $\mathcal{A}$ *such that one of the simple attributes is of the* $\mathit{TimeStamp}$ *type.*
- $\gamma_{ref} : A_r \rightarrow \mathcal{C}$, *the reference type function is a partial map that maps the reference type attributes in* A_r *to an Artifact Class in* $\mathcal{C}$.
- $\gamma_{sim} : A_s \cup \gamma_{com} \cup \gamma_{str} \rightarrow \mathcal{Y}$, *the simple type function is a partial map that maps the simple attributes in* A *in addition to the simple attributes constituting complex and stream attributes to their simple data types in* $\mathcal{Y}$.

When clear from the context, a Task $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{sim})$ *is simply referred to as* k.

Definition 4.4 (Flow Connector) *A Flow Connector* w *is the tuple* $w = (c, ro)$, *where* $c \in \mathcal{C}$ *is the Artifact Class associated to* w, $ro \in \mathit{Boolean}$ *indicates if* w *is read only:* true $(w.ro)$, *or not:* false $(\neg w.ro)$.

Example 4.1(Repository) *The FireDetected Repository from the fire control process is a tuple* (p, c_p) *where* p=$\mathit{FireDetected}$, *and* c_p=*FCA.*

Example 4.2(Task) *The CreateFCA Task from the fire control process is a tuple* $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim})$ *where:*

- *The Task name:* k= $\mathit{CreateFCA}$
- *The Task attributes:* A_k={ $\mathit{FireControlArtifactId}$, House, $\mathit{Habitats}$ }
- *The Artifact Class mappings:* $\gamma_{art}(\mathit{FireControlArtifactId})$=FCA, $\gamma_{art}(\mathit{House})$=FCA, $\gamma_{art}(\mathit{Habitats})$=FCA
- *The complex attributes mappings:* $\gamma_{com}(\mathit{House})$={ $\mathit{Address}$, $\mathit{Surface}$ }, $\gamma_{com}(\mathit{Habitats})$={ Name, $\mathit{PhoneNum}$ }
- *The stream attributes mappings:* $\emptyset$
- *The reference attributes mappings:* $\emptyset$
- *The simple attributes mappings:*

$\gamma_{sim}(FireControlArtifactId)=Integer$, $\gamma_{sim}(Address)=String$,
$\gamma_{sim}(Surface)=Real$, $\gamma_{sim}(Name)=String$, $\gamma_{sim}(PhoneNum)=Integer$

Example 4.3 (Flow Connector) *A Flow Connector w connecting the Normal Repository to the FireDetected Repository in the fire control process, is a tuple $w = (c, ro)$ where $w.c=FCA$ and $w.ro=false$.*

Example 4.4 (Conceptual Artifact Model) Considering the *Conceptual Artifact Model G* made of the *Branch Pattern* depicted in Figure 4.17, $G = (P, K, W, E, N, \delta, \alpha, \beta)$ is specified as follow:

- *The set of Repositories $P=\{p_1, p_2, p_3\}$ where*
  $p_1=Located$, $p_2=Success$, $p_3=Failed$
- *The set of Tasks $K=\{k_1\}$ where $k_1=AlertFireStation$*
- *The set of Flow Connectors $W=\{w_1, w_2, w_3\}$*
- *The set of Events $E=\emptyset$*
- *The set of Condition expressions $N=\{n_1, n_2\}$ where*
  $n_1=$"FSSA.IsSuccessfullyAlerted=true" *and*
  $n_2=$"FSSA.IsSuccessfullyAlerted=false"
- *The Event mappings: $\delta(w_1)=\emptyset$, $\delta(w_2)=\emptyset$, $\delta(w_3)=\emptyset$*
- *The Condition expressions mappings: $\alpha(w_1)=\emptyset$, $\alpha(w_2)=n_1$, $\alpha(w_3)=n_2$*
- *The transitions mappings:*

$\beta(w_1)=(p_1, k_1)$, $\beta(w_2)=(k_1, p_2)$, $\beta(w_3)=(k_1, p_3)$

4.3.2. Generating Artifact Classes

4.3.2.1. Artifact Classes Creation

Creating *Artifact Classes* is performed by examining the set of *Repositories P* of the *Conceptual Artifact Model G* and creating an associated set of *Artifact Classes C* of the *Artifact System W*.

For every *Repository $p \in P$,* if the associated *Artifact Class c_p* does not exist in the set of created *Artifact Classes C*, a corresponding *Artifact Class $(c, A, \gamma_{sim}, \gamma_{com}, \gamma_{ref}, \gamma_{str}, Q, s, F)$* is created and inserted into the set of *Artifact Classes C* with $c=c_p$ and the remaining elements specified in the following sections.

4.3.2.2. Generating Simple, Complex, and Stream Attributes

The *Information Model* of a created *Artifact Class* $c \in C$ is generated by inserting corresponding data attributes specified in the *Data Attribute Lists* attached to the set of *Tasks* K into the set of data attributes A of the *Artifact Class* c.

For every *Task* $k \in K$, we examine the data attributes of the attached *Data Attribute List* A_k. For every data attribute $a \in A_k$, if $\gamma_{art}(a)=c$ then a *is* inserted into the set of data attributes A of the *Artifact Class* c. Additionally, the data type functions γ_{sim}, γ_{com}, γ_{str} of the *Artifact Class* c are updated to reflect the data type of the data attribute a as specified by the data type functions γ_{sim}, γ_{com}, γ_{str} of the *Task* k.

4.3.2.3. Generating Reference Attributes

The *Tasks* creating child artifact instances (*Child Artifact Creation Pattern*) are identified by comparing their ingoing and outgoing *Flow Connectors*.

If a *Task* $k \in K$ has an outgoing *Flow Connector* $w_1 \in W$ such that $\beta(w_1)=(k, x_1)$ where x_1 is a *Repository* or a *Task* and there is not an ingoing read/write *Flow Connector* $w_2 \in W$ associated with the same *Artifact Class* as w_1 such that $\beta(w_2)=(x_2, k)$ and $w_1.c= w_2.c$ where x_2 is a *Repository* or a *Task,* then *Task* k creates an instance of a child *Artifact Class* c_{child}.

For every *Task* k that creates an instance of a child *Artifact Class* c_{child}, a stream data attribute a that references the child *Artifact Class* c_{child} is inserted into the set of data attributes A of the parent *Artifact Class* c such that $\gamma_{ref}(a)= c_{child}$.

4.3.2.4. Generating Lifecycle's States

The states of *Lifecycle* of an *Artifact Class* c are inserted by examining the set of *Repositories* P of the *Conceptual Artifact Model* G.

For every *Repository* $p \in P$, if $c_p=c$ then an associated state q is created and inserted into the set of *Lifecycle*'s states Q of the *Artifact Class* c.

Moreover, if *Repository* p has no ingoing *Flow Connector* w such that $\beta(w)=(x, p)$ where x is a *Task* or a *Repository*, then state q is the initial state of the *Lifecycle* of *Artifact Class* c and as a result $s= q$.

Furthermore, if *Repository* p has no outgoing *Flow Connector* w such that $\beta(w)=(p, x)$ where x is a *Task* or a *Repository*, then state q is a final state of the *Lifecycle* of the *Artifact Class* c and as a result s is inserted into the set of final states F.

Example 4.5 (Generating Artifact Classes) *Taking the Conceptual Artifact Model G illustrated in Figure 4.2 as an example, we generate the FireControlArtifact Class as a tuple* $(c, A, \gamma_{sim}, \gamma_{com}, \gamma_{ref}, \gamma_{str}, Q, s, F)$ *such that:*

- *The Artifact Class name:* $c = FireControlArtifact$
- *The set of data attributes:* $A = \{ FireControlArtifactId, House, Habitats, IndoorTemperature, IsAlarmTurnedOn, AreWaterEjectorsActivated \}$
- *The complex data type mappings:* $\gamma_{com}(House)=\{Address, Surface\}$, $\gamma_{com}(Habitats)=\{Name, PhoneNum\}$
- *The stream data type mappings:* $\gamma_{str}(IndoorTemperature)=\{Time, Tmp\}$
- *The reference data type mappings:* $\emptyset$
- *The set of Lifecycle's states:* $Q=\{ Normal, FireDetected, PrimaryProcedurePerformed, Failure\}$
- *The initial state:* $s= Normal$
- *The set of final states:* $F=\{ PrimaryProcedurePerformed, Failure \}$

4.3.3. Generating Services

4.3.3.1. Services Creation

Creating *Services* is performed by examining the set of *Tasks* K in the *Conceptual Artifact Model* G. For every *Task* $k \in K$, a corresponding *Service* (s, C_I, C_O, P, E) with $s= k$ is created and inserted into the set of *Services* S of the generated *Artifact System* W. The remaining elements of the created *Service* s are specified in the following sections.

4.3.3.2. Inputs Specification

The list of input *Artifact Classes* C_I of the created *Service* s is specified by examining the ingoing *Flow Connectors* into the corresponding *Task* k in the *Conceptual Artifact Model* G.

For every ingoing *Flow Connector* w of *Task* k such as $\beta(w)=(x, k)$ where x is a *Task* or a *Repository*, the associated *Artifact Class* $w.c$ is

inserted into the list of input *Artifact Classes* C_I of the corresponding *Service* s.

4.3.3.3. Outputs Specification

The list of output *Artifact Classes* C_O of the created *Service* s is specified by examining the outgoing *Flow Connectors* from the corresponding *Task* k in the *Conceptual Artifact Model* G.

For every outgoing *Flow Connector* w of *Task* k such as $\beta(w)=(k, x)$ where x is a *Task* or a *Repository*, the associated *Artifact Class* $w.c$ is inserted into list of output *Artifact Classes* C_O of the corresponding *Service* s.

4.3.3.4. Precondition Specification

Services' Precondition expression P of a *Service* s involves data attributes of *Data Attribute Lists* A_k attached to corresponding *Task* k. Only data attributes of the input *Artifact Classes* C_I are considered in *Services' Precondition* expression. The *Services' Precondition* expression is formed from the conjunction of *notdefined* or *closed* predicates over the selected data attributes. The *closed* predicate is used for stream data attributes.

For every data attribute $a \in A_k$, if $\gamma_{art}(a)=c$, $c \in C_I$ and $a \notin A_r$, then append to *Precondition* P a *notdefined* predicate over data attribute a such as $P = P \wedge notdefined(c.a)$.

On the other hand, , if $\gamma_{art}(a)=c$, $c \in C_I$ and $a \in A_r$, then append to *Precondition* P a *closed* predicate over data attribute a such as $P = P \wedge closed(c.a)$.

4.3.3.5. Effect Specification

Similarly, *Service's Effect* expression E of a *Service* s involves data attributes of *Data Attribute Lists* A_k attached to corresponding *Task* k. Only data attributes of the output *Artifact Classes* C_O are considered in *Service's Effect* expression. The *Service's Effect* expression is formed from the conjunction of *defined* or *opened* predicates over the selected data attributes. The *opened* predicate is used for stream data attributes.

For every data attribute $a \in A_k$, if $\gamma_{art}(a)=c$, $c \in C_O$ and $a \notin A_r$, then append to *Effect* E a *defined* predicate over data attribute a such as $E = E \wedge defined(c.a)$.

On the other hand, if $\gamma_{art}(a)=c$, $c \in C_O$ and $a \in A_r$, then append to *Effect* E a *defined* predicate over data attribute a such as $E = E \wedge defined(c.a)$.

Additionally, if *Task* k is creating a new artifact instance of a child *Artifact Class* c_{child} such that $c_{child} \in C_O$ but $c_{child} \notin C_I$, then append to *Effect* E a *new* predicate over *Artifact Class* c_{child} such as $E = E \wedge new(c.a)$ where a is the reference data attribute such that $a \in A_k$, $\gamma_{art}(a)=c$ and $\gamma_{ref}(a)=c_{child}$.

Example 4.6 (Generating Services) *Taking the Conceptual Artifact Model illustrated in Figure 4.14 as example, we generate an IssueFireStationAlert Service as the tuple* (s, C_I, C_O, P, E) *such that:*

- *The service name:* s= *IssueFireStationAlert*
- *The input list:* C_I={*FCA*}
- *The output list:* C_O={*FCA, FSAA*}
- *The precondition:* P=*notdefined(FCA.FireStationAlert)*
- *The effect:* E = *new(FCA.FireStationAlert)* $\wedge$ *defined(FSAA.FireStationAlertArtifactId)* $\wedge$ *defined(FSAA.House)*

4.3.4. Generating Artifact Rules

4.3.4.1. Artifact Rule Creation

Artifact Rules are created by examining the source of read/write *Flow Connectors*.

For every read/write *Flow Connector* $w \in W$ such that $\neg w.ro$ (not read-only), if the source of w is a *Repository* $p \in P$ such that $\beta(w)=(p,x)$ where x is a *Task* or a *Repository*, then an *Artifact Rule* r is created and inserted into the set of *Artifact Rules* R of the *Artifact System* W.

On the other hand, if the source of w is a *Task* $k \in K$ such that $\beta(w)=(k,x)$ where x is a *Task* or *Repository*, then for every ingoing read/write Flow Connector w' into *Task* k such that $\beta(w')=(x,k)$ and $\neg w'.ro$ where x is a *Task* or a *Repository,* an *Artifact Rule* r is created and inserted into the set of *Artifact Rules* R of the *Artifact System* W.

4.3.4.2. Generating Event Predicate

An *Event* is added to the condition part of an *Artifact Rule* if an *Event* construct is attached to the corresponding *Flow Connector*.

For every *Artifact Rule* $r \in R$, if the corresponding *Flow Connector* $w \in W$ has an attached *Event* $e \in E$ such that $\delta(w)=e$, then a corresponding *event* predicate is appended to the condition part Con_r of *Artifact Rule* r such that $Con_r = Con_r \wedge event(e)$.

4.3.4.3. Generating State Predicate

With the exception of *Flow Connectors* having no sources, a *state* predicate specifying the state of an artifact instance is inserted into the condition part of corresponding *Artifact Rules*.

For every *Artifact Rule* $r \in R$, if its corresponding *Flow Connector* $w \in W$ have a source such that $\beta(w)=(x, y)$ where x and y are *Tasks* or *Repositories*, we append a *state* predicate corresponding to the first source *Repository* p into the condition part Con_r of the *Artifact Rule* r such that $Con_r = Con_r \wedge state(c_p, p)$. If the source of the corresponding *Flow Connector* w is a *Repository* p such that $\beta(w)=(p, x)$ where x is a *Task* or a *Repository,* then in this case p is the source *Repository*. On the other hand, if the source of the corresponding *Flow Connector* w is a *Task* k such that $\beta(w)=(k, x)$ where x is a *Task* or a *Repository,* then we backtrack on the ingoing *Flow Connector* w' that was used to create the *Artifact Rule* r in Section 4.3.4.1 until we discover the first *Repository* which is in this case the source *Repository*.

Moreover, if a child artifact instance is created by a *Task,* then we use the *Initialized* state. For every *Artifact Rule* $r \in R$, if the corresponding *Flow Connector* w transports a child artifact instance that is created by its source *Task* k such that $\beta(w) = (k, x)$ where x is a *Task* or a *Repository*, then we append a state predicate to the condition part Con_r of *Artifact Rule* r such that $Con_r = Con_r \wedge state(w.c, Initialized)$.

4.3.4.4. Generating Notdefined Predicate

If the destination of a *Flow Connector* is a *Task,* then the *notdefined* predicate over the data attributes of the attached *Data Attribute List* is used in the condition of the corresponding *Artifact Rule*.

For every *Artifact Rule* $r \in R$, if the destination of the corresponding *Flow Connector* w is a *Task* k such that $\beta(w)=(x, k)$ where x is a *Task* or a *Repository*, then for every data attribute $a \in A_k$ we append a *notdefined* predicate over a to the condition part Con_r of the *Artifact Rule* r such that $Con_r = Con_r \wedge notdefined(c.a)$.

4.3.4.5. Generating Defined Predicate

If the source of a *Flow Connector* is a *Task,* then the *defined* predicate over the data attributes of the attached *Data Attribute List* is used in the condition of the corresponding *Artifact Rule*.

For every *Artifact Rule* $r \in R$, if the source of the corresponding *Flow Connector* w is a *Task* k such that $\beta(w)=(k, x)$ where x is a *Task* or a *Repository*, then for every data attribute $a \in A_k$ we append a *defined* predicate over a to the condition part Con_r of the *Artifact Rule* r such that $Con_r = Con_r \wedge defined(c.a)$.

4.3.4.6. User Defined Condition

An additional user defined condition is appended to the condition part of an *Artifact Rule* if a *Condition* construct is attached to the corresponding *Flow Connector*.

For every *Artifact Rule* $r \in R$, if the corresponding *Flow Connector* $w \in W$ has an attached *Condition* $n \in N$ such that $\alpha(w)=n$, then the user defined condition is appended to the condition part Con_r of *Artifact Rule* r such that $Con_r = Con_r \wedge n$.

4.3.4.7. Action Specification

The action part of an *Artifact Rule* is specified according to the destination of the corresponding *Flow Connector*.

For every *Artifact Rule* $r \in R$, if the destination of the corresponding *Flow Connector* $w \in W$ is a *Task* k, then the action part Act_r of the *Artifact Rule* r is an *invoke* predicate over the corresponding *Service* s such that $Act_r = invoke(s)$.

On the other hand, if the destination of the corresponding *Flow Connector* $w \in W$ is a *Repository* p, then the action part Act_r of the *Artifact Rule* r is a *state* predicate such that $Act_r = state(c_p, p)$.

Example 4.7 (Generating Artifact Rules) *Taking the Conceptual Artifact Model illustrated in Figure 4.17, the generated set of Artifact Rules is* $R=\{r_1, r_2, r_3\}$ *such that:*

- $r_1: state(FSAA, Located) \wedge \neg defined(FSAA.IsSuccessfullyAlerted) \rightarrow invoke(AlertFireStation)$

- r_2: $state(FSAA, Located) \wedge defined(FSAA.IsSuccessfullyAlerted) \wedge FSAA.IsSuccessfullyAlerted = true \rightarrow state(FSAA, Success)$
- r_3: $state(FSAA, Located) \wedge defined(FSAA.IsSuccessfullyAlerted) \wedge FSAA.IsSuccessfullyAlerted = false \rightarrow state(FSAA, Failed)$

4.4. Summary of Modeling Artifact Systems

In this chapter, we present the *Conceptual Artifact Modeling Notation (CAMN)* which is used to graphically model *Artifact Systems*. Moreover, the constructed *Conceptual Artifact Models (CAMs)* include all components of an *Artifact System* in the same model and provide a more representative and holistic model that supports process awareness. Furthermore, the proposed modeling approach combines the advantages of both procedural and declarative modeling approaches. Additionally, we propose modeling patterns that include data stream specific patterns required for modeling smart artifact-based processes. Finally, we define transformation semantics for generating valid *Artifact Systems* from *CAMs* based on the proposed formalism.

In the next chapter, we will present the *Artifact Integration System* that integrates heterogeneous *CAMs* into one global *CAM*. The global *CAM* serves as a *Unified View* for supervising, managing, and querying heterogeneous *artifacts*.

5

Integrating Conceptual Artifact Models

Chapter Outline

In Chapter 4, we proposed the *Conceptual Artifact Modeling Notation* (*CAMN)* to graphically model *Artifact Systems* as *Conceptual Artifact Models (CAMs)*. The *CAM* provides many benefits over existing conceptual models. First, the *CAM* is a representative and holistic model that includes all components of an *Artifact System*. It also combines advantages of both declarative and procedural modeling approaches. Finally, the *CAM* supports modeling patterns for situations that involve data streams. As a result, we aims at building artifact-centric process integration by considering corresponding *CAMs*.

In fact, existing integration semantics like *Data Integration* [Lenz02] and *Business Process Merging* [LDUD13] only deal with the integration of one aspect of a *Business Processes*. *Data Integration* deals with integrating data structures but ignores processes that manipulate data. *Business Process Merging* deals with integrating process activities or tasks but ignores the data aspect. As a result, existing integration semantic are not suitable for integrating *CAMs* which combine both data and their processes into same models.

In this chapter, we introduce the *Artifact Integration System* that aims at integrating heterogeneous *CAMs*. In this system, several local *CAMs* representing heterogeneous artifacts or artifact-based processes are integrated into one global *CAM* acting as a *Unified View of the integrated artifact*. In addition, we generate mapping rules that translate elements between global and local *conceptual artifact models*.

The semantics of the *Artifact Integration System* are based on three sub-phases:

1. *Matching Sub-phase*: Inspired by *S chema Matching* [BoVe11], the *Matching Sub-phase* deals with identifying correspondences between different elements of local *CAMs*.
2. *Merging Sub-phase*: Inspired by *Business Process Merging* [SuKY06], the *Merging Sub-phase* deals with merging local *CAMs* into one global *CAM* according to identified correspondences and by re-branching and re-connecting *Flow Connectors*.
3. *Mapping Sub-phase*: Inspired by *Schema Mapping* [Do06], the *Mapping Sub-phase* deals with transforming data between global and local models.

The generated global *CAM* acts as a *Unified View* that can be used in order to supervise, execute, and/or query local *CAMs*. Without loss of generality, the proposed integration semantics is presented in the context of

integrating two *CAMs*. However, integrating any number of *CAMs* can be performed by incrementally integrating *CAMs*.

The remainder of this chapter is organized as follow:

In Section 5.1, we introduce the *Artifact Integration System* and the three sub-phases of the *Integration Phase*.

In Section 5.2, we describe the *Matching Sub-Phase* of the *Integration Phase* and the semantics of identifying correspondence relationships.

In Section 5.3, we describe the *Merging Sub-Phase* of the *Integration Phase* and the semantics of generating a global *CAM*.

In Section 5.4, we describe the *Mapping Sub-Phase* of the *Integration Phase* and the semantics of generating mapping rules between local and global *CAMs*.

5.1. Artifact Integration System

The integration of *CAMs* is based on traditional *Data Integration Systems* [HuZh96a, PaSp00] in which several local schemas are integrated into one global schema acting as a *Unified View* for accessing local schemas. Moreover, in traditional *Data Integration Systems,* the global schema is generated based on identifying correspondences between elements of local schemas. The process of identifying correspondences between local schemas is known as *Schema Matching* [RaBe01]. Additionally, transformations between local and global schemas are achieved with mapping rules. The process of defining mapping rules between local and global schemas is known as *Schema Mapping* [BoVe11].

Similarly to traditional *Data Integration Systems*, we integrate several local *CAMs* into one global *CAM* that acts as a *Unified View*. The generation of global *CAMs* is based on the correspondences between local *CAMs* and mapping rules to transform queries between local and global *CAMs*.

Definition 5.1 (Artifact Integration System) *An Artifact Integration System I_{CAM} is a triplet (G, S, M), where G is the global CAM, S is the set of local CAMs, and M is the set of mapping rules between G and S.*

Figure 5.1 illustrates the *Artifact Integration System* where the *Artifact Manipulation Language (AML)* queries are sent to the global *CAM*. *Mapping Rules* are then used to translate the queries from the global *CAM* into local

CAMs. Returned results are then translated and combined from local *CAMs* back into the global *CAM*.

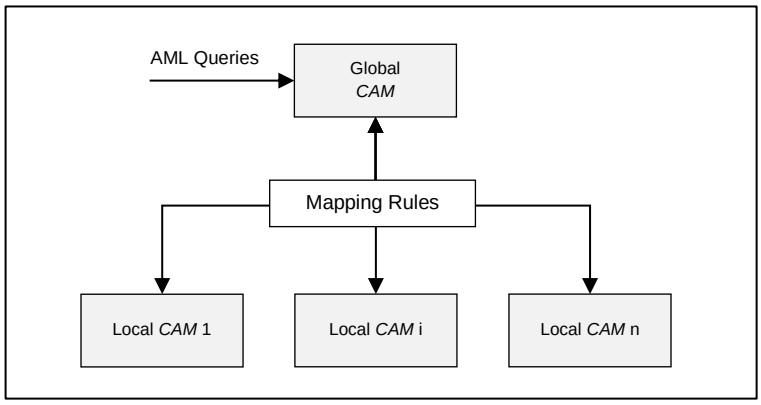


Figure 5.1 Artifact Integration System

The semantics of the *Artifact Integration System* are based on three sub-phases: *Matching Sub-Phase*, *Merging Sub-Phase*, and *Mapping Sub-Phase*.

First, the *Matching Sub-Phase* deals with identifying correspondences between different elements of local *CAMs* and is constituted of two incremental match operations. The first match operation identifies correspondences between *Artifacts*, *Tasks* and *Repositories*. The second match operation identifies correspondences between data attributes of both models. In later case, matching expressions over data attributes defining data transformation rules are specified based on a set of predefined functions. The result of the *Matching Sub-Phase* is a set of correspondences between different elements of the local *CAMs* and is used to guide sub-phases of the *Integration Phase*.

Second, the *Merging Sub-Phase* deals with generating the *Unified View* or global *CAM* by merging the local *CAMs* based on the identified correspondences of the *Matching Sub-Phase*. First, *Artifacts* are generated followed by *Repositories*, *Tasks*, *Data Attribute Lists*, and finally *Flow Connectors,* including *Events* and *Conditions*. Each step is based on the result of the previous steps as follows: The generation of *Tasks* and *Repositories* is based on the generation of *Artifacts*, the generation of *Data Attribute Lists* is based on the generation of *Tasks*, and the generation of *Flow Connectors* is based on the generation of *Artifacts*, *Tasks*, *Repositories*, and *Data Attribute List*.

Finally, the *Mapping Sub-Phase* defines mapping rules between local and global *CAMs*. Mapping rules are used to translate elements between local and global *CAMs.* Moreover, mapping rules are specified based on the

structure and relationship between the different *CAMN* constructs used to model *CAMs*.

In the remaining of the chapter, we describe the semantics of the *Artifact Integration System* using two local *CAMs*; G_1 and G_2 that are integrated into one global *CAM*; G_I. by such, G_1 is the tuple $(P_1, T_1, W_1, E_1, N_1, \delta_1, \alpha_1, \beta_1)$. G_2 is the tuple $(P_2, T_2, W_2, E_2, N_2, \delta_2, \alpha_2, \beta_2)$. And, G_I is the tuple $(P_I, T_I, W_I, E_I, N_I, \delta_I, \alpha_I, \beta_I)$. It is worth noting *CAMs* are defined based on the formalism described in Chapter 4.

An important difference between the global *CAM* G_I and the local *CAMs* is that in G_I, *Flow Connectors* can be associated with several *Events* and/or *Conditions*. This reflects the fact that two semantically equivalent *Flow Connectors* in the local *CAMs* can be triggered by two different *Events* and/or *Conditions*. As a result, δ_I and α_I respectively map W_I to E_I^n and N_I^n instead of E_I and N_I.

5.2. Matching Sub-Phase

Integration of *CAMs* is based on identified correspondences between their different constituting elements. Correspondences are acquired as a result of match operations [Do06]. Match operations can be of two types: manual operations, where the user specifies the corresponding elements using graphical interfaces, and semi-automatical operations using matching algorithms and ontologies [RaBe01]. In this thesis, we perform match operations based on graphical interfaces. However, the proposed approach can be enhanced with ontologies in order to perform semi-automatic match operations.

The *Matching Sub-Phase* consists of two operations: 1) *Artifacts*, *Tasks* and *Repositories* match operation, and 2) *Data Attributes* match operation. The result is a set of correspondences which are exposed as correspondence functions that maps the elements of the local *CAMs* to their corresponding elements, if any.

5.2.1. Artifacts, Tasks, and Repositories Match Operation

The first match operation identifies correspondences of *Artifacts*, *Tasks* and *Repositories*. Three correspondences relationships are involved: *uniqueness*, *equivalence*, and *composition* relationships.

The *uniqueness* relationship is a correspondence of *one-to-zero* or *zero-to-one* cardinalities and signifies that an element of one local *CAM* has no

corresponding element in the other local *CAM*. Figure 5.2 illustrates the *uniqueness* relationship for *Tasks*, *Repositories*, and *Artifacts* characterized by the absence of visual relations between the unique elements and any other element from the other *CAM*.

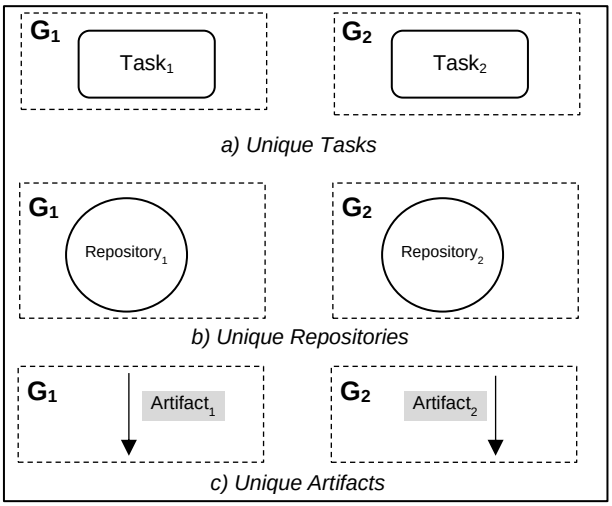


Figure 5.2 Uniqueness correspondence relationships for Tasks, Repositories, and Artifacts

The *equivalence* relationship is a correspondence of *one-to-one* cardinality and signifies that two elements of two different *CAMs* are semantically equivalent. The *equivalence* relationship is represented using double headed arrows as illustrated in Figure 5.3. The solid head points in the direction of the dominant element. The dominant element is the generated element in the global *CAMs* when merging the elements of the *equivalence* relationship.

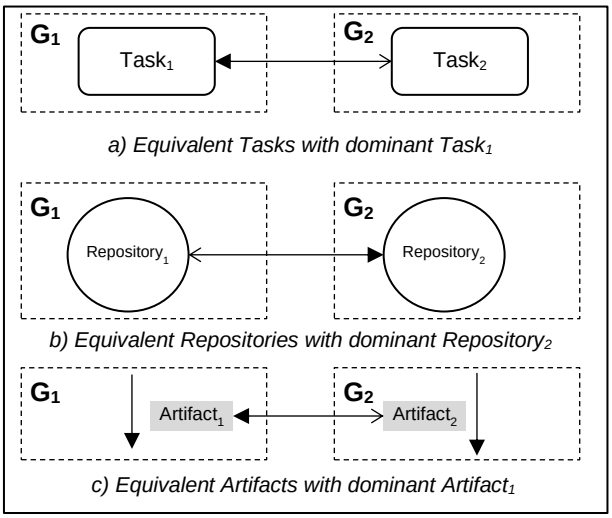


Figure 5.3 Equivalence correspondence relationships for Tasks, Repositories, and Artifacts

The *composition* relationship represents compositions of (business) functions where one (business) function is used to abstract several other (business) functions. As such, in a *composition* relationship a *Task* in one local *CAM* is used to aggregate *n elements; Tasks* and *Repositories,* in the other local *CAM*. The *composition* relationship is a correspondence of *one-to-many* or *many-to-one* cardinalities, and represented graphically as a circle shape with several outgoing arrows as illustrated in Figure 5.4. In *composition* relationship, the *Task* aggregating the *n* elements is the dominant element and the *n* elements must be on a single path.

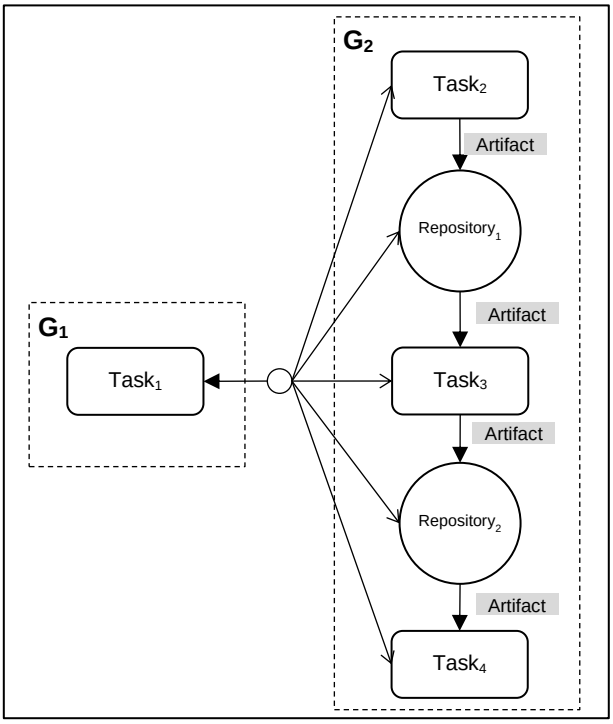


Figure 5.4 Composition correspondence relationship for Tasks

After the specification of correspondences between elements of the two local *CAMs* using the graphical notation, the correspondence relationships are exposed as a *Correspondence Function C.*

Definition 5.2 (Correspondence Function) *The Correspondence Function C exposes correspondence relationships between Artifacts, Tasks and Repositories from G_1 and G_2 and is defined as:*

$$C: Art_1 \cup Art_2 \cup P_1 \cup P_2 \cup K_1 \cup K_2 \rightarrow (Art_1 \cup Art_2 \cup P_1 \cup P_2 \cup K_1 \cup K_2)^n \times d$$

Where: Art_1 and Art_2 are respectively the sets of all used *Artifacts of G_1 and G_2. P_1 and P_1 are respectively the sets of Repositories of G_1 and G_2. K_1 and K_2 are respectively the sets of Tasks of G_1 and G_2. And $d \in \{ r, l \}$ defines the*

dominant element participating in the correspondence relationship; l (left-dominant) for the element belonging to the left-side e.g., G_1, and r (right-dominant) for the element belonging to the right-side e.g., G_2.

The semantics of the *Correspondence Function* C for *Artifacts*, *Tasks* and *Repositories* are defined as follow:

- *Uniqueness correspondences*: If an *Artifact*, *Task* or *Repository* in one *CAM* has no equivalent *Artifact*, *Task* or *Repository* in the other *CAM*, then $C(e) = \perp$ where $\perp$ is the null symbol and e is an *Artifact*, *Task* or *Repository* from G_1 *or* G_2.
- *Equivalence correspondences*: If two *Artifacts*, *Tasks* or *Repositories* belonging to the first and second *CAMs* have an *equivalence* relationship, then:
  - $C(e_1)=(e_2, l)$ if e_1 is the dominant element,
  - $C(e_1)=(e_2, r)$ if e_2 is the dominant element,

  where e_1 is an *Artifact*, *Task* or *Repository* from G_1 and e_2 is an *Artifact*, *Task* or *Repository* from G_2.
- *Composition correspondences*: If one *Task* in one *CAM* has a composition relationship with n *Tasks* and *Repositories* in the other *CAM*, then:
  - $C(e_0)=(e_1,...e_n, l)$ if e_0 is the dominant element, where e_0 is a *Task* from G_1 and $e_1,... e_n$ are *Tasks* and *Repositories* from G_2.
  - $C(e_0)=(e_1,...e_n, r)$ if e_0 is the dominant element, where $e_1,... e_n$ are *Tasks* and *Repositories* from G_1 and e_0 is a *Task* from G_2.

  And the *Task* aggregating the n element is always chosen as the dominant element.

5.2.2. Data Attributes' Match Operation

In the second match operation, correspondences between data attributes of G_1 and G_2 are identified. Data attributes characterizing *Tasks* of both models are automatically collected and presented in a graphical interface where the user specifies correspondence relationships. Similarly to *Artifacts*, *Tasks* and *Repositories* match operation, three correspondence relationships are proposed : *uniqueness*, *equivalence*, and *composition* relationships.

The *uniqueness* relationship is a correspondence of *one-to-zero* or *zero-to-one* cardinalities and signifies that a data attribute in one *CAM* has no

corresponding data attribute in the other *CAM*. Table 5.1 illustrates the *uniqueness* relationship for data attributes characterized by the absence of visual relations between the unique data attribute and any other data attribute from the other *CAM*.

The *equivalence* relationship is a correspondence of *one-to-one* cardinality and signifies that two data attributes belonging to the two *CAMs* are semantically equivalent. *Equivalence* relationships are graphically represented using the double headed arrow where the solid head points in the direction of the dominant attribute as illustrated in Table 5.1. Additionally, the user must specify a *matching expression* that defines how the two data attributes are related to each other in the *equivalence* relationship as illustrated in Table 5.1.

The *composition* relationship is a correspondence of *one-to-many* or *many-to-one* cardinalities and signifies that a data attribute in one *CAM* is the composition of several data attributes in the other *CAM*. Composition relationships are represented using the circle shape with outgoing arrows. The solid head arrow points to the composite data attribute in one *CAM*. The other normal head arrows points to the composite data attributes in the other *CAM* as illustrated in Table 5.1. In addition a matching expression defining how data attributes are related to each other in the *composition* relationship must be specified by the user as illustrated in Table 5.1.

Table 5.1 Data Attribute Correspondences Examples

Match Relationship	Graphical Notation	Matching Expression
Uniqueness correspondence	G1 - Fax G2 - Telephone	*NA*
Equivalence correspondence	G1 - Price ⟷ G2 - Cost	*Cost = Price * 100*
Composition correspondences (one-to-many)	G1 - Cost ⟵ G2 - Price, - Tax	*Cost = Price * (1 + Tax/100)*
Composition correspondences (many-to- one)	G1 - FirstName, - MiddleName, - LastName ⟶ G2 - FullName	*FullName = **concat**(FirstName, " ", MiddleName, " ", LastName)*

Matching expressions are user defined mathematical expressions involving data attributes written using arithmetic operators (equality, addition, subtraction, multiplication, division, and remainder) in addition to a list of predefined functions as listed in Table 5.2. *Matching expressions* serves two purposes:

1) To describe how data attributes involved in a correspondence relationship are related to each other, as in the case of composition relationships.
2) To achieve domain type transformations.

Table 5.2 Matching Expression Predefined Functions

Function	Description	Example	Result
concat(*string, string, string*)*	Returns the concatenation of the arguments.	• ***concat(****'John', ' ', 'Smith')*	• *"John Smith"*
substring(*string, number, number?)*	Returns the substring of the first argument starting at the position specified in the second argument and the length specified in the optional third argument.	• ***substring(****'John Smith', 5)* • ***substring(****'John A. Smith', 5, 2)*	• "Smith" • "A."
substring-after(*string, string)*	Returns the substring of the first argument string that follows the first occurrence of the second argument string in the first argument string, or the empty string if the first argument string does not contain the second argument string.	• ***Substring-after(****'John Smith', ' ')*	• Smith"
substring-before(*string, string)*	Returns the substring of the first argument string that precedes the first occurrence of the second argument string in the first argument string, or the empty string if the first argument string does not contain the second argument string.	• ***substring-before(****'John Smith', ' ')*	• *"John"*
average(*number, number, number*)*	Returns the average of the arguments.	• ***average(****10, 20, 30, 40)*	• 25
min(*number, number, number*)*	Returns the minimum of the arguments.	• ***min(****10, 20, 30, 40)*	• 10
max(*number, number, number*)*	Returns the maximum of the arguments.	• ***max(****10, 20, 30, 40)*	• 40
ceiling(*number)*	Returns the smallest integer that is not less than the argument.	• ***ceiling(****2.15)*	• 3
floor(*number)*	Returns the largest integer that is not greater than the argument.	• ***floor(****2.75)*	• 2
round(*number)*	Returns an integer closest in value to the argument.	• ***ceiling(****2.15)*	• 2
string(*number)*	Converts the argument to a string.	• ***string(****25.5)*	• "25.5"
number(*string|boolean)*	Converts the argument to a number.	• ***number(****false)* • ***number(****true)* • ***number(****'25.5')* • ***number(****'Abcd')*	• 0 • 1 • 25.5 • NaN
boolean(*number)*	Converts the argument to a Boolean.	• ***boolean(****-5)* • ***boolean(****1)* • ***boolean(****5)* • ***boolean(****0)* • ***boolean(****'Abcd')*	• *true* • *true* • *true* • *false* • *false*
not(*boolean)*	Returns true if the argument is false; otherwise false.	• ***not(****true)* • ***not(****false)*	• *false* • *true*

Correspondences of data attributes are exposed using a specialized version of the *Correspondence Function C* that involves *Matching Expressions*.

Definition 5.3 (Data Attributes Correspondence Function) *The Data Attributes Correspondence Function C_{da} is defined as:*

$$C_{da} : A_1 \cup A_2 \rightarrow (A_1 \cup A_2)^n \times d \times Exp$$

where A_1 and A_2 are the sets of all data attributes respectively from G_1 and G_2, d is the dominance, and Exp is the set of all matching expressions.

The semantic of the correspondence function C_{da} for data attributes is defined as follow:

- *Uniqueness correspondences*: If a data attribute in one *CAM* has no equivalent data attribute in the other *CAM*, then $C_{da}(a)=\perp$ where $\perp$ is the null symbol and $a \in A_1 \cup A_2$ is a data attribute from G_1 or G_2.
- *Equivalence correspondences*: If two data attributes from the two *CAMs* have an equivalence relationship, then:
  - $C_{da}(a_1)=(a_2, l, x)$ if a_1 is the dominant data attribute.
  - $C_{da}(a_1)=(a_2, r, x)$ if a_2 is the dominant data attribute.

  where $a_1 \in A_1$, $a_2 \in A_2$, and $x \in Exp$ is a matching expression.
- *Composition correspondences*: If a data attribute belonging to one *CAM* has a composition relationship with n data attributes belonging to the other *CAM*, then:
  - $C_{da}(a_0)=(a_1,...a_n, l, x)$ if a_0 is the dominant data attribute, where $a_0 \in A_1$, $a_1,...a_n \in A_2$, and $x \in Exp$ is a matching expression.
  - $C_{da}(a_0)=(a_1,...a_n, r, x)$ if a_0 is the dominant data attribute, where $a_1,...a_n \in A_1$, $a_0 \in A_2$, and $x \in Exp$ is a matching expression.

  And the dominant element is always the data attribute aggregating the other n data attributes.

5.3. Merging Sub-Phase

In the *Merging Sub-Phase*, we generate the global *CAM* G_I by merging the two local *CAMs* G_1 and G_2 according to the correspondences exposed by the *Correspondence Function C* and its specialization C_{da} from the *Matching Sub-Phase*.

Similarly to *Business Process Merging* [SuKY06], the *Merging Sub-Phase* re-branches and re-connects *Flow Connectors* and is composed of five incremental operations: *Artifact Generation*, *Repository Generation*, *Task Generation*, *Data Attribute List Generation*, and *Flow Connector Generation* including *Events* and *Conditions*. We formally define the semantics of the *Merging Sub-Phase* based on a set of mathematical rules that make use of an *Integration Function I*, and its specialization *Condition Integration Function* I_{con}. The *Integration Function I* is then used to generate the various elements of the integrated *CAM*. *The Condition Integration Function* I_{con} is used to generate the *Conditions* of the integrated *CAM*.

Definition 5.4 (Integration Function) *The Integration Function I takes as input one or more elements from one CAM and returns the integrated element(s) according to the correspondence relationship between the elements. The Integration Function I is defined as:*

$$I : (G_1 \cup G_2)^n \rightarrow (G_1 \cup G_2)^n$$

where G_1 and G_2 are the two *CAMs*.

Example 5.1 (Integration Function) *Suppose that a Task* SubmitOrder *from* G_1 *has an equivalence relationship with a Task* CreateOrder *from* G_2 *and* SubmitOrder *is the dominant Task, in other words,* C(SubmitOrder)=(CreateOrder, l) *then:* I(SubmitOrder) =SubmitOrder *and* I(CreateOrder)=SubmitOrder.

Definition 5.5 (Condition Integration Function) *The Condition Integration Function* I_{con} *function takes a condition expression as input and updates its data attributes to reflect their integrated form in the global CAM.* I_{con} *uses Matching Expressions from Exp to translate between domain types if they are different.* I_{con} *is defined as:*

$$I_{con} : N_1 \cup N_2 \rightarrow N_I$$

where N_1, N_2, *and* N_I *are the sets of Conditions from respectively* G_1, G_2, *and* G_I.

Example 5.2 (Condition Integration Function) *Suppose that a data attribute* a_1="Price" *originates from* G_1 *such that* a_1 *is expressed in the dollar unit, and a data attribute* a_2="Cost" *originates from* G_2 *such that* a_2 *is expressed in Cent unit, and that* $C_{DA}(a_1) = (a_2, r,$ "Cost = Price*100" $)$ *in other words* $I(a_1)=a_2$, $I(a_2)=a_2$, *and "Cost = Price*100" is the matching expression. Then if we have a Condition* n_1="Price ≥ 500" *such that* $n_1 \in N_1$, *then* $I_{con}(n_1)$="Cost ≥ 50000".

In the rest of this section, we define the semantics of the five generate operations using generation rules. These generation rules are divided into

five incremental sets that successively describe the generation of *Artifacts*, *Repositories*, *Tasks*, *Data Attribute Lists*, and *Flow Connectors* including *Events* and *Conditions*. The generation rules have the following general form:

"If correspondence **then update I and generate** *integratedElement* ***in*** G_I*"*

where ***If-then*** is represented using the logic implication symbol "→". "*correspondence*" is a conjunction of conditions (using logic AND symbol "∧") that tests for a particular correspondence relationship. If the "*correspondence*" holds, then the *Integration Function I* is updated with a set "*X*" of new entries of the form "$I(e_i)=e_j$", and "*integratedElement*" is generated in the global *CAM*. The "*integratedElement*" is defined using the "*there exist*" logic symbol; "$\exists e \in G_I[specification]$" where "*specification*" is a set of conjunctive conditions (using the logic AND symbol exprssed as "∧") that the generated element must match. The *Integration Function I* will be further used in the "*specification*" part of the "*integratedElement*" to integrate associated elements if existing.

5.3.1. Artifacts Generation

The generation of *Artifacts* is achieved based on the correspondences exposed by the *Correspondence Function C*. Since *Artifacts* are transmitted between *Tasks*, *Repositories* and *Flow Connectors*, the first-order rules update the *Integration Function I* with new entries corresponding to the integrated *Artifacts* but do not generate an element in G_I. Thus, the generation rules are reduced to: *"If correspondence* **then update I***"* for *Artifact* generation. The *Integration Function I* is then used in the remaining generate operations in order to generate *Repositories*, *Tasks*, *Data Attribute Lists*, and *Flow Connectors*. We recall that $w = (c, ro)$ represents a *Flow Connector*.

- *Unique Artifacts Integration Rule*:

  $\forall w_i \in W_1 \cup W_2 [\ C(w_i.c)=\perp \rightarrow I(w_i.c)=w_i.c\ ]$

- *Equivalent Artifacts Integration Rule (l-dominant)*:

  $\forall w_i \in W_1, \forall w_j \in W_2 [\ C(w_i.c)=(w_j.c, l) \rightarrow I(w_i.c)=w_i.c \wedge I(w_j.c)=w_i.c\ ]$

- *Equivalent Artifacts Integration Rule (r-dominant)*:

  $\forall w_i \in W_1, \forall w_j \in W_2 [\ C(w_i.c)=(w_j.c, r) \rightarrow I(w_i.c)=w_j.c \wedge I(w_j.c)=w_j.c\ ]$

5.3.2. Repositories Generation

Similarly to *Artifacts*, the generation of *Repositories* is achieved based on the correspondences exposed by the *Correspondence Function C.* The *Integration Function I* is updated with new entries corresponding to the integrated *Repositories*. Additionally, the *Integration Function I* is used to generate the associated *Artifacts* of generated *Repositories*. The generation of *Repositories* is defined using the following generation rules, where p represents the *Repository* (p, c_p).

- *Unique Repositories integration rule*:

$$\forall p_i \in P_1 \cup P_2 \, [\, C(p_i)=\perp \rightarrow I(p_i)=p_i \wedge \exists p_j \in P_I [\, p_j=p_i \wedge c_{pj}=I(c_{pi}) \,] \,]$$

- *Equivalent Repositories integration rule (l-dominant)*:

$$\forall p_i \in P_1, \forall p_j \in P_2 [\, C(p_i)=(p_j, l) \rightarrow I(p_i)=p_i \wedge I(p_j)=p_i \wedge \exists p_x \in P_I [\, p_x=p_i \wedge c_{px}=I(c_{pi}) \,] \,]$$

- *Equivalent Repositories integration rule (r-dominant)*:

$$\forall p_i \in P_1, \forall p_j \in P_2 [\, C(p_i)=(p_j, r) \rightarrow I(p_i)=p_j \wedge I(p_j)=p_j \wedge \exists p_x \in P_I [\, p_x=p_j \wedge c_{px}=I(c_{pj}) \,] \,]$$

5.3.3. Tasks Generation

Tasks are generated based on the correspondences exposed by the *Correspondence Function C.* On the other hand, *Data Attribute Lists* attached to *Tasks* are generated in the *Data Attribute Lists Generation*. The *Integration Function I* is updated with new entries corresponding to the integrated *Tasks*. We recall that a *Task k* is a tuple $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim})$ in which $A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim}$ represent the attached *Data Attribute List.*

- *Unique Tasks integration rule:*

$$\forall k_i \in K_1 \cup K_2 \, [\, C(k_i)=\perp \rightarrow I(k_i)=k_i \wedge \exists k_j \in K_I [k_j = k_i] \,]$$

- *Equivalent Tasks integration rule (l-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2 \, [\, C(k_i)=(k_j, l) \rightarrow I(k_i)=k_i$$

$$\wedge\, I(k_j)=k_i$$
$$\wedge\, \exists k_x \in K_I [\, k_x=k_i \,] \,]$$

- *Equivalent Tasks integration rule (r-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2 [\, C(k_i)=(k_j, r) \rightarrow I(k_i)=k_j$$
$$\wedge\, I(k_j)=k_j$$
$$\wedge\, \exists k_x \in K_I [\, k_x=k_j \,] \,]$$

- *Composite Tasks integration rule (l-dominant):*

$$\forall k_i \in K_1, \forall k_1,...k_n \in K_2 [\, C(k_i)=(k_1,...k_n, l) \rightarrow I(k_i)=k_i$$
$$\wedge\, I(k_1)=k_i$$
$$\wedge\, ...$$
$$\wedge\, I(k_n)=k_i$$
$$\wedge\, \exists k_j \in T_I [\, k_j=k_i] \,]$$

- *Composite Tasks integration rule (r-dominant):*

$$\forall k_1,...k_n \in K_1, \forall k_j \in K_2 [\, C(k_j)=(\, k_1,...k_n, r) \rightarrow I(k_j)=k_j$$
$$\wedge\, I(k_1)=k_j$$
$$\wedge\, ...$$
$$\wedge\, I(k_n)=k_j$$
$$\wedge \exists k_x \in T_I [\, k_x=k_j] \,]$$

5.3.4. Data Attribute Lists Generation

The generation of *Data Attribute Lists* is achieved by integrating the constituting data attributes. Two important factors are involved in the generation of *Data Attribute Lists*:

1) The correspondences between the data attributes that constitutes the lists exposed using the *Data Attributes Correspondence Function* C_{da}, and
2) The integrated *Tasks* exposed using the *Integration Function I.*

As a result, the generation of data attribute lists is performed in two further sub-steps: *Data Attributes Integration*, and *Data Attribute Lists Population*.

5.3.4.1. Data Attributes Integration

In the *Data Attributes Integration* sub-step, the *Integration Function I* is filled with integration entries related to data attributes. The generation rules that describe this sub-step are reduced to: *"**If** dataAttributeCorrespondence **then update I**"*. We recall that a *Task k* is a tuple $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim})$ in which $A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim}$ represent the attached *Data Attribute List.*

- *Unique data attributes integration rule:*

$$\forall k_i \in K_1 \cup K_2, \forall a_j \in A_{ki} [\ C_{da}(a_j) = \perp \rightarrow I(a_j) = a_j\ ]$$

- *Equivalent data attributes integration rule (l-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2, \forall a_m \in A_{ki}, \forall a_n \in A_{kj} [\ C_{da}(a_m) = (a_n, l, x) \rightarrow I(a_m) = a_m \wedge I(a_n) = a_m\ ]$$

- *Equivalent data attributes integration rule (r-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2, \forall a_m \in A_{ki}, \forall a_n \in A_{kj} [\ C_{da}(a_m) = (a_n, r, x) \rightarrow I(a_m) = a_n \wedge I(a_n) = a_n\ ]$$

- *Composite data attributes integration rule (l-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2, \forall a_m \in A_{ki}, \forall a_1, \ldots a_n \in A_{kj} [\ C_{da}(a_m) = (a_1, \ldots a_n, l, x) \rightarrow I(a_m) = a_m \wedge I(a_1) = a_m \wedge \ldots \wedge I(a_n) = a_m\ ]$$

- *Composite data attributes integration rule (r-dominant):*

$$\forall k_i \in K_1, \forall k_j \in K_2, \forall a_1, \ldots a_m \in A_{ki}, \forall a_n \in A_{kj} [\ C_{da}(a_n) = (a_1, \ldots a_m, r, x) \rightarrow I(a_n) = a_n \wedge I(a_1) = a_n \wedge \ldots \wedge I(a_m) = a_n\ ]$$

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

5.3.4.2. Data Attribute Lists Population

In the *Data Attribute Lists Population* sub-step, we populate *Data Attribute Lists* of integrated *Tasks* with the integrated data attributes. A test is performed that identifies integrated *Tasks* in the global *CAM* prior to the population of attached *Data Attribute Lists*. Thus, the generation rules describing this sub-step reflects the following logic:

"If integratedTask **then generate** *integratedAttribute* **in** G_I*."*

where "*integratedTask*" is a condition written using the *Integration Function I* that checks for integrated *Tasks* in the global *CAM*. Furthermore, the associated *Artifact* and *Data Type Functions* of the data attribute are also updated. We recall that a *Task k* is a tuple $(k, A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim})$ in which $A_k, \gamma_{art}, \gamma_{com}, \gamma_{str}, \gamma_{ref}, \gamma_{sim}$ represent the attached *Data Attribute List.*

- *Data Attribute Lists population rule:*

$$\forall k_i \in K_1 \cup K_2, \exists k_x \in K_I [\, k_x = I(k_i) \rightarrow \forall a_j \in A_{ki}, \exists a_l \in A_{kx} [\, a_l = I(a_j) \wedge \gamma_{art}(a_l) = I(\gamma_{art}(a_j)) \wedge \gamma_{com}(a_l) = \gamma_{com}(a_j) \wedge \gamma_{str}(a_l) = \gamma_{str}(a_j) \wedge \gamma_{ref}(a_l) = I(\gamma_{ref}(a_j)) \wedge \gamma_{sim}(a_l) = \gamma_{sim}(a_j) \,] \,]$$

5.3.5. Flow Connectors Generation

The generation of *Flow Connectors* is achieved by integrating their source, destination, associated *Artifact*, type, attached *Event*, and attached *Condition* if any. We recall that a *Flow Connector* is defined as a tuple $w = (c, ro)$ where c is the associated *Artifact*, and ro is the type of the *Flow Connector* (read-only or read/write). Additionally, δ, α are the partial functions that associate *Events* and *Condition*s respectively to *Flow Connectors* and β is the total function that specifies the source and destination of *Flow Connectors*. We also use the *Condition Integration Function* I_{con} that takes as input the *Condition* of a *Flow Connector* and updates the involved data attributes and values with their integrated counterparts using the *Integration Function I* in addition to the user defined matching expressions from the *Matching Sub-Phase*.

- *Flow Connectors integration rule:*

$$\forall w_i \in W_1 \cup W_2, \exists s, d \in K_1 \cup K_2 \cup P_1 \cup P_2 \; [\; \beta(w_i) = (s, d) \wedge I(s) \neq I(d) \rightarrow \exists w_x \in W_I \; [\; w_x.c = I(w_i.c) \wedge w_x.ro = w_i.ro \wedge \beta_I(w_x) = ((I(s), I(d)) \wedge \delta(w_i) \in \delta_I(w_x) \wedge I_{con}(\alpha(w_i)) \in \alpha_I(w_x)] \;]$$

For every Flow Connector w_i in G_1 or G_2, we first ensure that the integration of its source and destination are not equal; $I(s) \neq I(d)$. Otherwise the source and destination are part of a composited *Task* and in this case no *Flow Connector* is generated.

If the integration of w_i's source and destination are not equal, we generate a *Flow Connector* w_x in G_I such that:

- The associated *Artifact* is the integration of the associated *Artifact* of w_i; $w_x.c = I(w_i.c)$.
- The type (read-only or read/write) is the same type as w_i; $w_x.ro = w_i.ro$.
- The source and destination are respectively the integration of the source and destination of w_i ; $\beta_I(w_x) = ((I(s), I(d))$. This will ensure that correct routing is performed when equivalent elements, *branch pattern*, or *convergence pattern* are involved.
- One of the *Events* that can trigger w_x is the *Event* attached to w_i if any; $\delta(w_i) \in \delta_I(w_x)$.
- One of the *Conditions* of w_x is the integrated form of the *Condition* attached to w_i if any; $I_{con}(\alpha(w_i)) \in \alpha_I(w_x)$

Example 5.3 (Merging Sub-Phase) *We consider two local CAMs G_1 and G_2 and their Artifacts, Tasks, Repositories and data attributes correspondences illustrated in Figure 5.5 and Figure 5.6 where matching expressions are omitted.*

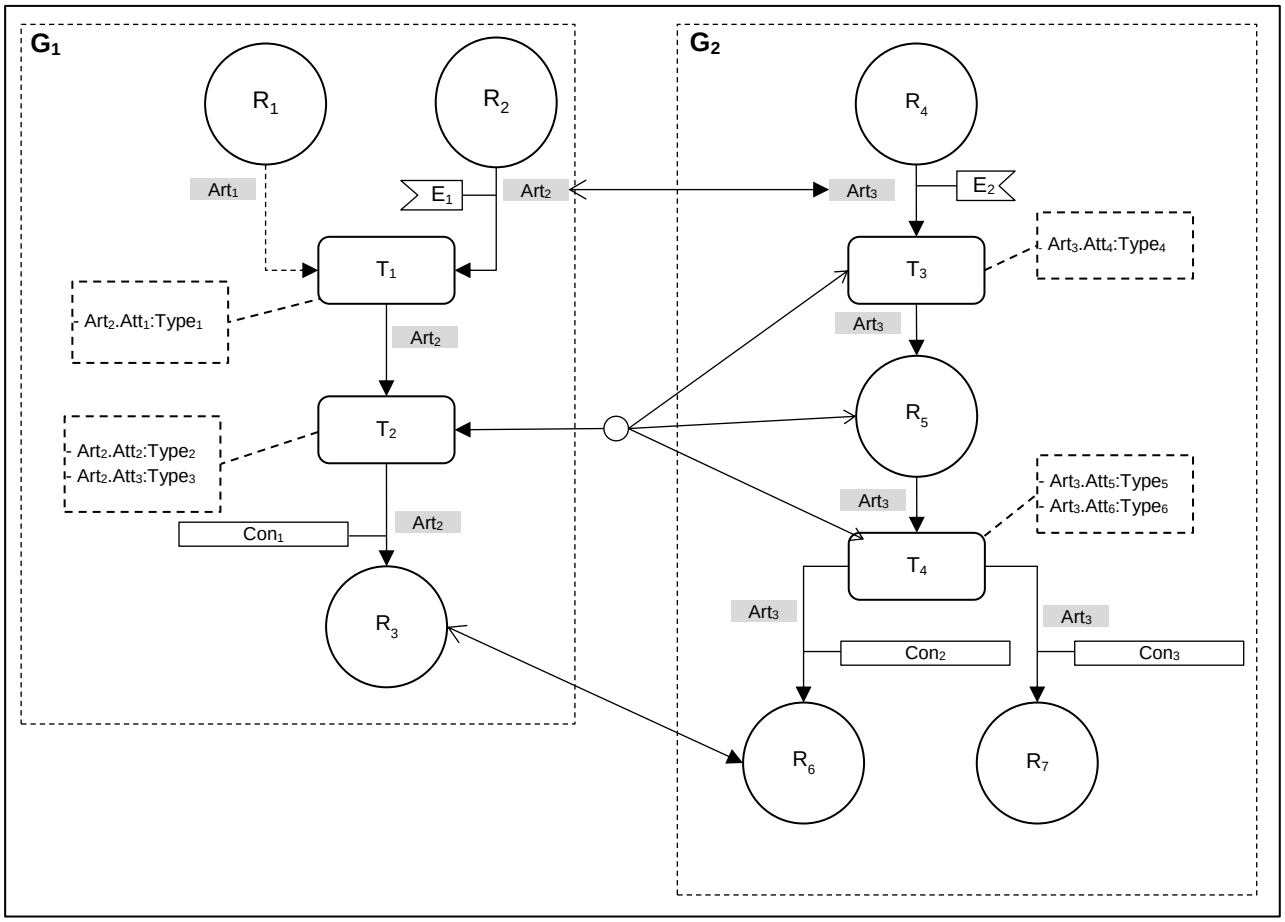


Figure 5.5 Merging Sub-Phase local *CAMs* example

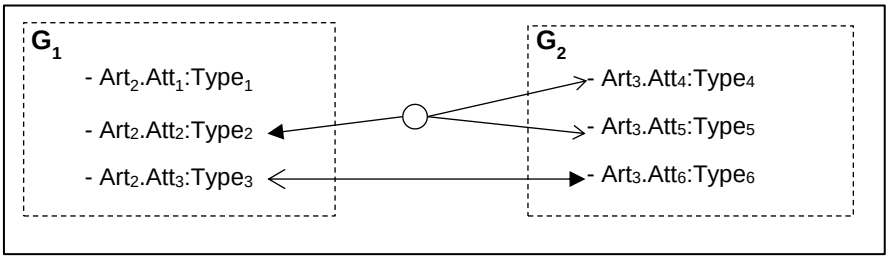


Figure 5.6 data attributes correspondences of local *CAMs* example

The graphical correspondences are exposed using the Correspondence Functions C and C_{DA} as follow:

- *Unique elements:*

  $C(Art_1) = \perp$, $C(R_1) = \perp$, $C(R_2) = \perp$, $C(R_4) = \perp$, $C(R_7) = \perp$, $C(T_1) = \perp$, $C_{DA}(Att_1) = \perp$.

- *Equivalent elements:*

$C(Art_2)=(Art_3, r)$, $C(R_3)=(R_6, r)$, $C_{DA}(Att_3)=(Att_6, r, x_1)$.

- *Composition elements:*

$$C(T_2)=(T_3,R_5,T_4, l),\ C_{DA}(Att_2)=(Att_4, Att_5, l, x_2)$$

The Integration Function I will have the following entries:

$I(Art_1)=Art_1$, $I(R_1)=R_1$, $I(R_2)=R_2$, $I(R_4)=R_4$, $I(R_7)=R_7$, $I(T_1)=T_1$, $I(Att_1)=Att_1$, $I(Art_2)=Art_3$, $I(Art_3)=Art_3$, $I(R_3)=R_6$, $I(R_6)=R_6$, $I(Att_3)=Att_6$, $I(Att_6)=Att_6$, $I(T_2)=T_2$, $I(T_3)=T_2$, $I(R_5)=T_2$, $I(T_4)=T_2$, $I(Att_2)=Att_2$, $I(Att_4)=Att_2$, $I(Att_5)=Att_2$.

Finally, the generated global CAM G_I is illustrated in Figure 5.7.

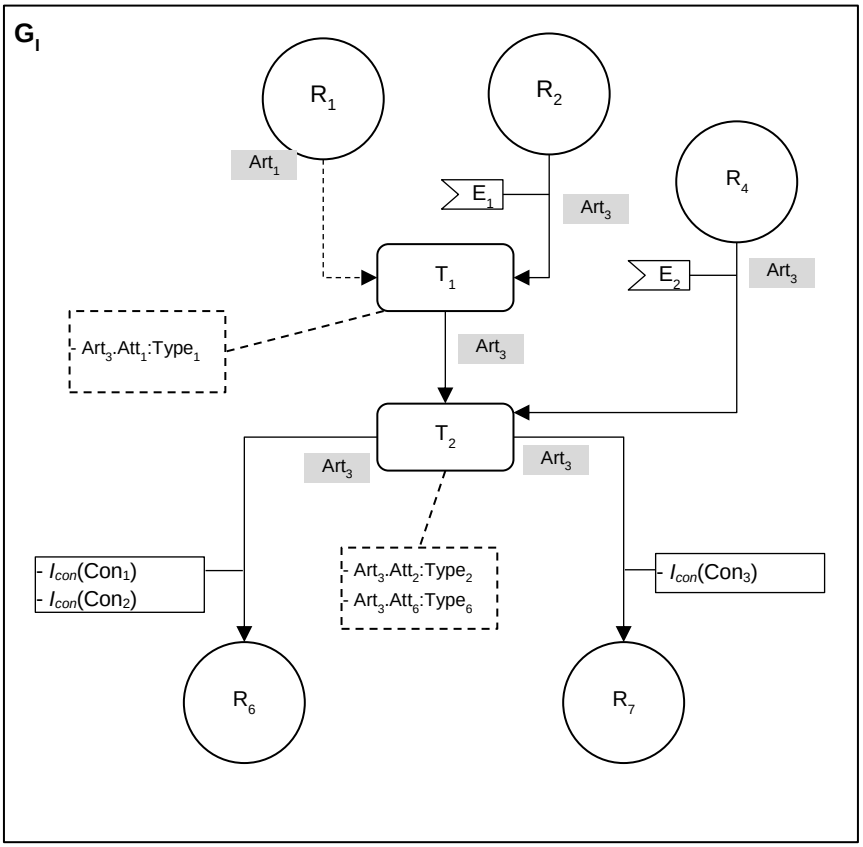


Figure 5.7 Generated global *CAMs*

5.4. Mapping Sub-Phase

Mappings describe relationships between elements of global *CAM* and local *CAMs* and are used to translate between them. Mappings are used for several applications including;

- *Query reformulation*: To translate queries posed at global *CAM* into queries compatible with local *CAMs*.
- *Centralized execution*: To translate execution instructions posed at global *CAM* into execution instruction compatible with local *CAMs*.
- *Supervising platform*: To translate processing details from local *CAMs* into global *CAM*.

In this section, we propose mapping specifications based on the structure of *CAMN* constructs. This structure can be resumed by the following three key relationships:

1) *Artifacts* are associated to *Repositories* and *Flow Connectors*.
2) *Tasks* are the sources or destinations of *Flow Connectors*.
3) *Repositories* are the sources or destinations of *Flow Connectors*.

Based on these relationships, we define five mapping functions that respectively map *Artifacts*, *Repositories*, *Tasks*, *Data Attributes* and *Flow Connectors* of global *CAM* into local *CAMs*.

Definition 5.6 (Artifact Mapping Function) *The Artifact Mapping Function M_{art} maps Artifacts of global CAM to Artifacts of a selected local CAM and is defined as:*

$$M_{art} : Art_I \times d \rightarrow Art_1 \cup Art_2$$

where Art_1, Art_2, and Art_I are respectively the sets of Artifacts of G_1, G_2, and G_I. $d \in \{l, r\}$ is the dominance singleton specifying the target local CAM; l signifies the left side CAM or G_1, r signifies the right side CAM or G_2.

Example 5.4 (Artifact Mapping Function) *Suppose that we have a FireStationAlertArtifact in the global CAM G_I. When applied to FireStationAlertArtifact, the Artifact Mapping Function M_{art} returns the results:*

M_{art}*("FireStationAlertArtifact", l) =* *"FireStationAlertArtifact", and* M_{art}*("FireStationAlertArtifact", r)* $= \perp$.

In other words, *FireStationAlertArtifact* is mapped to *FireStationAlertArtifact* in G_1, *and has no mapped Artifact in* G_2.

Definition 5.7 (Repository Mapping Function) *The Repository Mapping Function M_P maps Repositories associated to Artifacts of global CAM to Repositories associated to Artifacts of a selected local CAM. M_P is defined as:*

$$M_P : (P_I \times Art_I) \times d \rightarrow (P_1 \times Art_1) \cup (P_2 \times Art_2)$$

where: Art_1, Art_2, and Art_I are respectively the sets of Artifacts of G_1, G_2, and G_I. P_1, P_2, and P_I are respectively the sets of Repositories of G_1, G_2, and G_I. And $d \in \{l,r\}$ is the dominance singleton specifying the target local CAM as before.

Example 5.5 (Repository Mapping Function) *Suppose that we have a Repository "Normal" associated to an Artifact "FireControlArtifact" in G_I. Then, the Repository Mapping Function M_P returns the result:*

M_P*("Normal", "FireControlArtifact", l) = ("Normal", "FireControlArtifact")*

M_P*("Normal", "FireControlArtifact", r) = ("Idle", "ReactiveProcedureArtifact").*

In other words, the Repository "Normal" associated to the Artifact "FireControlArtifact" in G_I is mapped to the Repository "Normal" associated to the Artifact "FireControlArtifact" in G_1 and is mapped to the Repository "Idle" associated to the Artifact "ReactiveProcedureArtifact" in G_1.

Definition 5.8 (Task Mapping Function) *The Task Mapping Function M_K maps Tasks of global CAM to Tasks of a selected local CAM. In the case that the Task is a composite Task, then the Task Mapping Function M_K returns the composited path made of Tasks and Repositories. M_K is defined as:*

$$M_K: K_I \times d \rightarrow (K_1 \cup P_1)^n \cup (K_2 \cup P_2)^n$$

where K_1, K_2, and K_I are respectively the sets of Tasks of G_1, G_2, and G_I. P_1 and P_2 are respectively the sets of Repositories of G_1 and G_2 . And, $d \in \{l,r\}$ is the dominance singleton specifying the target local CAM as before.

Example 5.6 (Task Mapping Function) *Suppose that we have a Task "PerformPrimaryProcedure" in G_I. Then, the Task Mapping Function M_K returns the results:*

M_K*("PerformPrimaryProcedure", l) = ("TurnOnAlarm", "AlarmTurnedOn", "ActivateWaterEjectors", "WaterEjectorsActivated")*

M_K*("PerformPrimaryProcedure",r)=("PerformPrimaryProcedure").*

In other words, Task "PerformPrimaryProcedure" in G_I is mapped to the composited Tasks and Repositories; TurnOnAlarm", "AlarmTurnedOn", "ActivateWaterEjectors", "WaterEjectorsActivated" in G_1 and is mapped to Task "PerformPrimaryProcedure" in G_2.

Definition 5.9 (Data Attribute Mapping Function) *The Data Attribute Mapping Function M_{da} maps data attributes associated to Artifacts in global CAM to data attributes associated to Artifacts in a selected local CAM in addition to matching expressions originating from Data Attribute Correspondence Function C_{da}. M_{da} is defined as:*

$$M_{da}: A_I \times Art_I \times d \rightarrow (A_1^n \times Art_1 \times Exp) \cup (A_2^n \times Art_2 \times Exp)$$

where: A_1, A_2, and A_I are respectively the sets of data attributes of G_1, G_2, and G_I. Art_1, Art_2, and Art_I are respectively the sets of Artifacts of G_1, G_2, and G_I. Exp is the set of matching expressions. And, $d \in \{l,r\}$ is the dominance singleton specifying the target local CAM as before.

Example 5.7 (Data Attribute Mapping Function) *Suppose that a Habitats Artifact in G_I includes a data attribute "FullName". When G_1 is selected, the Data Attribute Mapping Function M_{da} returns the result:*

$$M_{da}(\text{"FullName"}, \text{"Habitats"}, l) = (\{\text{"FirstName"}, \text{"MiddleName"}, \text{"LastName"}\}, \text{"Habitats"}, \text{"concat('FirstName', ' ', 'MiddleName', ' ', 'LastName')"})$$

In other words, data attribute "FullName" in G_I is mapped to the composited data attributes "FirstName", "MiddleName", and "LastName" in G_1 according to the matching expression "concat(FirstName, ' ', MiddleName, ' ', LastName)".

Definition 5.10 (Flow Connector Mapping Function) *Flow Connector Mapping Function M_W maps sources, targets, and associated Artifacts of Flow Connectors of global CAM to sources, targets, and associated Artifacts of Flow Connectors of a chosen local CAM. M_W is defined as:*

$$M_W: (K_I \cup P_I) \times (K_I \cup P_I) \times Art_I \times d \rightarrow ((K_1 \cup P_1) \times (K_1 \cup P_1) \times Art_1) \cup ((K_2 \cup P_2) \times (K_2 \cup P_2) \times Art_2)$$

where the first $(K \cup P)$ represents the source, the second $(K \cup P)$ represents the target, Art represents the associated Artifact of G_1, G_2, and G_I.

Example 5.8 (Flow Connector Mapping Function) *Suppose that a Flow Connector in G_I is associated to a "FireControlArtifact" and its source and destination are respectively "FireDetected" and "PerformPrimaryProcedure". When G_2 is selected, the Flow Connector Mapping Function M_W returns the result:*

$$M_W(\text{"FireDetected"}, \text{"PerformPrimaryProcedure"}, \text{"FireControlArtifact"}, r) = (\text{"FireDetected"}, \text{"TurnOnAlarm"}, \text{"FireControlArtifact"})$$

In other words, the selected flow connector in G_1 is mapped to the flow connector having "FireDetected" and "TurnOnAlarm" as respectively its source and destination in G_2.

5.5. Summary of Integrating Conceptual Artifact Models

Since *CAMs* combines both process and data aspects into same models, we propose an *Artifact Integration System* based on specialized integration semantics for integrating heterogonous *CAMs.* The proposed integration semantic combines integration mechanisms from both *Data Integration* and *Business Process Merging,* and covering artifact-specific integration mechanisms. The proposed artifact integration semantics is based on three sub-phases: *Matching Sub-Phase, Merging Sub-Phase, and Mapping Sub-Phase*. The *Matching Sub-Phase* uses concepts from *Schema Matching* in order to identify correspondence relationships between elements of local *CAMs*. The *Merging Sub-Phase* uses concepts from *Business Process Merging* in order to merge local *CAMs* and generate global *CAMs*. Finally, the *Mapping Sub-Phase* uses concepts from *Schema Mapping* in order to define mapping functions that translate elements between global and local *CAMs*.

In the next chapter, we present the prototype implementing of the *Artifact Integration Framework*. Moreover, we illustrate an experimental scenario about a fire control smart process in the context of a smart city. Finally, we present experimental results in the context of a applied study performed on the developed prototype.

6

Prototyping and Experimentation

Chapter Outline

In order to validate our work, we have developed a prototype to demonstrate key concepts of the *Artifact Integration Framework,* including covering *Integration*, *Modeling*, *Specification*, and *Execution functionalities.*

The prototype implements the four phases of the *Artifact Integration Framework* using a modular architecture. Modules communicate with each other using input and output messages. The prototype relies on several programming languages, including; *HTML5* [00b], *XML* [00c], *Java* [00d], *Xtend* [00e], and *JavaScript* [00f]. it also deploys several programming frameworks such as *Eclipse Rich Client Platform (Eclipse RCP)* [00g], *Xtext Framework* for the development of *Domain Specific Languages (DSL)* [00h], the *Java Architecture for XML Binding (JAXB) Framework* [00i], *JointJS Javascript Diagramming Library* [00j], and *Apache Derby Database Management System* [00k].

In this chapter, we describe the prototype implementation and an experimental scenario about fire control smart processes in the context of a futuristic smart city. The remainder of the chapter is organized as follow:

In Section 6.1, we describe the prototype implementation of the *Artifact Integration Framework*.

In Section 6.2, we describe the fire control scenario.

Finally, Section 6.3 is a summary of the prototyping and experimentation chapter.

6.1. Prototype Architecture

The prototype implementation of the *Artifact Integration Framework* covers the four phases using six software modules as illustrated in Figure 6.1.

The *AQL Processor* module implements the *Execution Phase* and is responsible for processing, executing, and managing *AQL* queries and *Artifact Systems*. The *AQL Processor* takes *AQL* queries as input from the *AQL Generator,* the *CAM Mapper,* or the built-in *Graphical User Interface*, executes queries, and produces the result using the built-in *Graphical User Interface.*

The *AQL Generator* module implements the *Specification Phase* and generates *Artifact Definition Language (ADL)* queries from *CAMs*. The *AQL Generator* takes as input a *CAM* from the *CAM Modeler* and outputs *Artifact Definition Language (ADL)* queries of corresponding *Artifact System*.

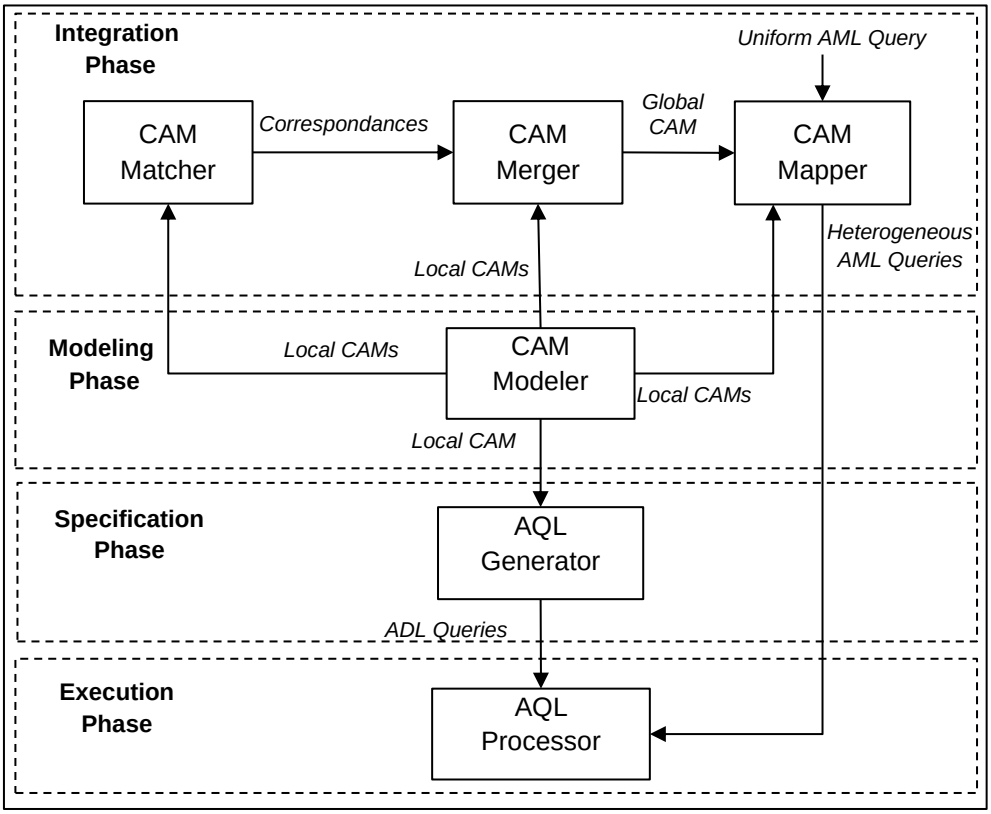


Figure 6.1 Main Modules of Artifact Integration Framework

The *CAM Modeler* module implements the *Modeling Phase* and provides graphical editor for modeling *CAMs*. The *CAM Modeler* takes input from the user and generates *CAMs*.

The *Integration Phase* is implemented by three modules:

- The *CAM Matcher* module implements the *Matching Sub-phase* of the *Integration Phase*. The *CAM Matcher* is a graphical editor that is used in order to capture correspondences between two local *CAMs*. The *CAM Matcher* takes two *CAMs* specifications as inputs and outputs correspondence relationships.
- The *CAM Merger* module implements the *Merging Sub-phase* of the *Integration Phase*. The *CAM Merger* takes two local *CAMs* specifications and their correspondences as input and generates one global *CAM* specification as output.
- The *CAM Mapper* module implements the *Mapping Sub-phase* of the *Integration Phase*. The *CAM Mapper* takes global and local *CAMs* in addition to a uniform *Artifact Manipulation Language (AML)* query

and translates the query to heterogeneous queries corresponding to the local *CAMs*.

6.1.1. AQL Processor Module

The *AQL Processor* module deals with processing, executing, and managing *AQL* queries and *Artifact Systems*. It provides a complete *Integrated Development Environment (IDE)* based on the *Eclipse Rich Client Platform (Eclipse RCP)* and offers several views, dialogs and windows to work with *AQL* and *Artifact Systems*.

The *AQL Processor* module is composed of eight sub-modules as illustrated in Figure 6.2.

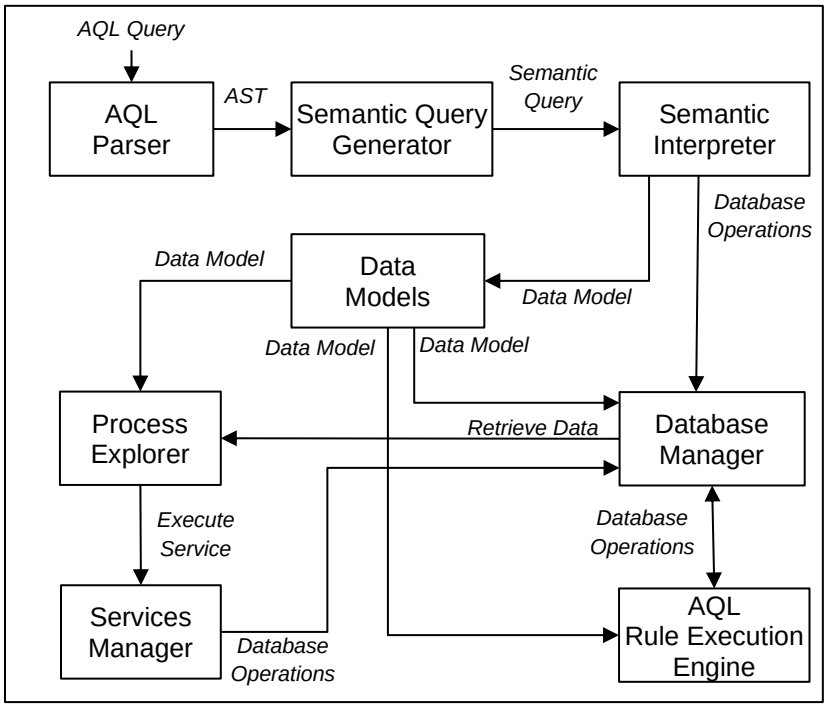


Figure 6.2 AQL Processor Architecture

6.1.1.1. AQL Parser Sub-Module

The *AQL Parser* sub-module is based on the *Xtext Framework* and reads *AQL* queries, parses them, and outputs an *Abstract Syntax Tree (AST)*. Additionally, it makes use of the *AQL Editor* for directly reading *AQL* queries from the user. In this module, the *"Aql.xtext"* source file consists of 484 lines of *Xtext* code specifying the *AQL* grammar. Figure 6.3 illustrates the *AQL Editor* generated by the *Xtext* framework.

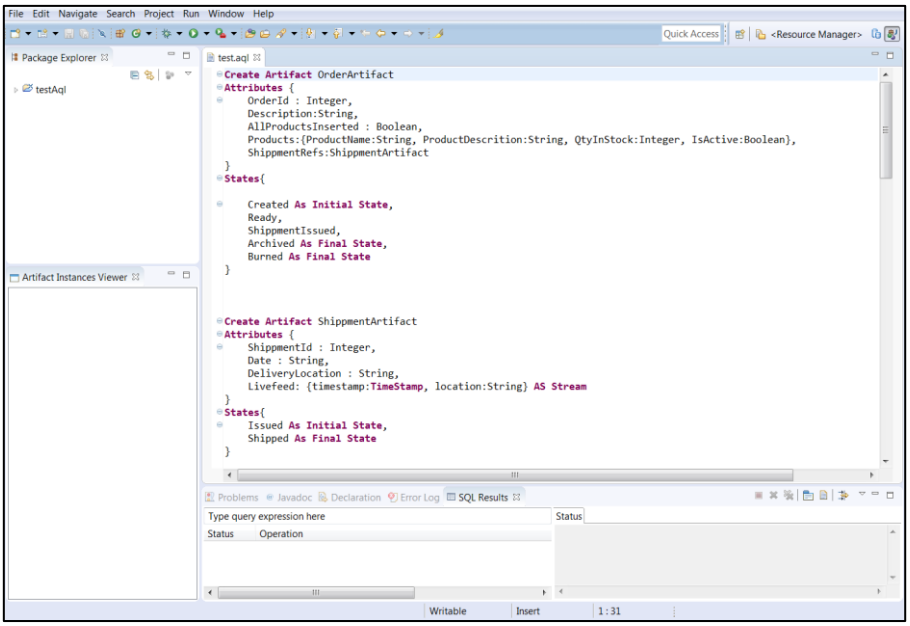


Figure 6.3 Graphical Interface of the AQL Editor

6.1.1.2. Semantic Query Generator Sub-Module

The *Semantic Query Generator* sub-module is an *Xtend* based generator that takes the *AST* of an *AQL* query as input and generates an *XML*-based semantic query as output. In this module, the *"AqlGenerator.xtend"* consists of 693 lines of *Xtend* code that generates an *XML* file representing the semantic form for every *AQL* query in an *AQL* file. The generated *XML* files are then inserted into the *"src-gen"* folder of the current project. Figure 6.4 illustrates an example of a generated *XML*-based semantic query.

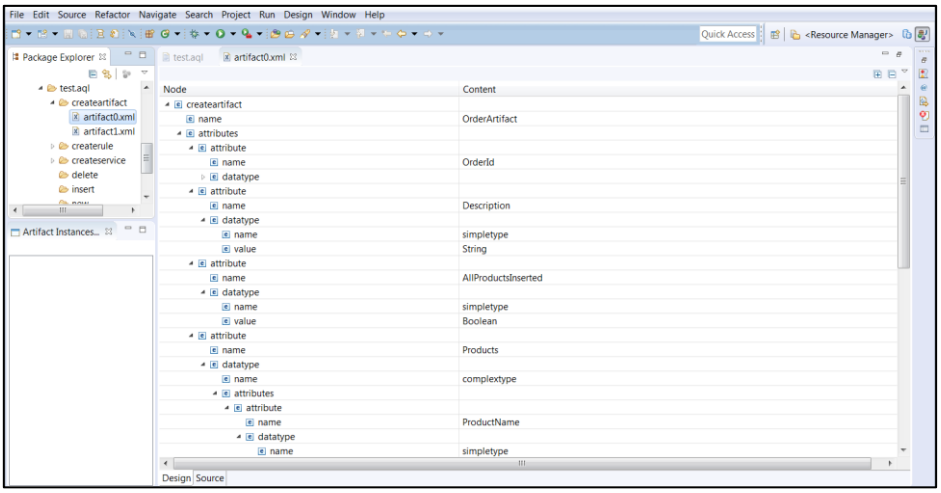


Figure 6.4 Generated XML Semantic Query

6.1.1.3. Semantic Query Interpreter Sub-Module

The *Semantic Query Interpreter* sub-module executes *AQL* queries. It takes an *XML*-based semantic query as an input, generates corresponding data models and performs necessary database operations. It consists of two *Java* classes: *SemanticInterpreter* and *RunAqlQueriesHandler*.

The *SemanticInterpreter* class executes *XML*-based semantic *AQL* queries. Figure 6.5 illustrates the *UML* class of the *Semantic Query Interpreter* sub-module.

Figure 6.5 UML Class of SemanticInterpreter

The *runAqlQueries(IWorkbenchWindow, ISelection)* method is invoked by the *RunAqlQueriesHandler* class and aims to interpret *AQL* queries stored in an *AQL* file. References to the active *WorkbenchWindow* and *Selection* are passed as parameters in order to retrieve the selected *AQL* file.

The *addRuleEngineAsDatabaseListener()* method is used in order to register the *AQL Rule Execution Engine* as a listener on the *Database Manager*. As a result, the *AQL Rule Execution Engine* will receive events whenever the *Database Manager* performs database operations.

The *createSystemTables()* method invokes methods from the *Database Manager* in order to create three system tables. The created three system tables are internally used by the *AQL Processor* in order to invoke services and receive user-generated external events. The created three system tables are: *InvokedServicesSysTable*, *InvokedServicesInputSysTable*, and *GeneratedEventsSysTable*.

The *setDatabaseSettings()* method is used in order to retrieve and set necessary database settings from the created project properties. The database settings are then used by the *Database Manager* whenever database operations are performed.

The *interpretCreateRuleQuery(IFile)*, *interpretCreateServiceQuery(IFile)*, *interpretCreateArtifactQuery(IFile)*, *interpretDeleteQuery(IFile)*, *interpretNewQuery(IFile)*, *interpretRetrieveQuery(IFile)*, *interpretUpdateQuery(IFile)*, *interpretInsertIntoQuery(IFile)*, and *interpretRemoveQuery(IFile)* methods respectively interpret and execute *Create Rule*, *CreateService*, *Create Artifact*, *Delete*, *New*, *Retrieve*, *Update*, *InsertInto*, and *Remove* queries. Every method takes a reference to the *XML* semantic query file as an *IFile* parameter.

The *createDatabase(CreateArtifact)* method invokes the *Database Manager* in order to create database tables corresponding to a *Create Artifact* query.

The *RunAqlQueriesHandler* class is a handler that is used in order to invoke the *SemanticInterpreter* from the *Eclipse Workbench*. Figure 6.6 illustrates the *UML* class of *RunAqlQueriesHandler* consisting of one method.

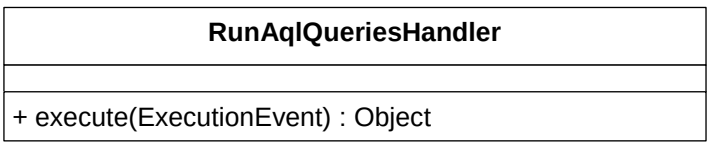


Figure 6.6 UML Class of RunAqlQueriesHandler

The *execute(ExecutionEvent)* method retrieves the active *WorkbenchWindow* and *Selection* and invokes the *runAqlQueries* method from the *SemanticInterpreter* class.

6.1.1.4. Data Models Sub-Module

The *Data Models* sub-module contains data models corresponding to *AQL* statements and *Artifact Systems*. It consists of packages automatically generated from *XML Schema Definition (XSD)* files corresponding to the *XML* semantic queries using the *Java Architecture for XML Binding (JAXB)* framework. Moreover, the *Data Models* contains one additional package called *"Common"* that contains classes: *ArtifactSystem*, *AttIdentification*, and *Helper*. The *Data Models* are populated by the *SemanticInterpreter* class and used by the *DatabaseManager*, *ProcessExplorer*, and *AQL Rule Execution Engine*.

The *ArtifactSystem* class is used to store instances of *Artifact Systems*. It consists of three static collections: *createArtifactList*, *createServiceList*, and *createRuleList* that respectively are used to store the interpreted *Create Artifact*, *Create Service*, and *Create Rule* semantic queries. Figure 6.7 illustrates the *UML* class of *ArtifactSystem*.

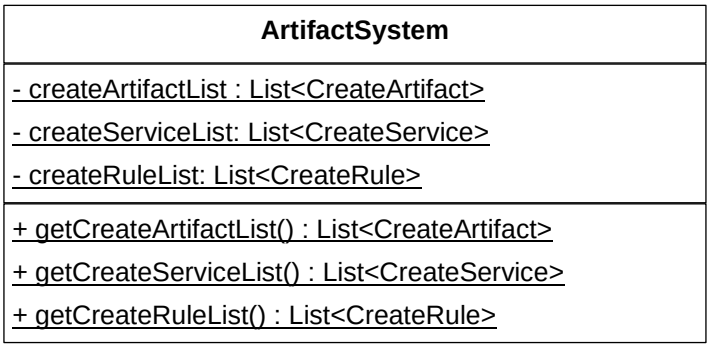


Figure 6.7 UML Class of ArtifactSystem

The *AttIdentification* class is used to identify data attributes of *Information Models.* The *AttIdentification* class provides several members to set and return the data attribute, parent data attribute, and containing *Artifact Class* names. Figure 6.8 illustrates the *UML* class of *AttIdentification*.

The *Helper* class provides methods for working with the *ArtifactSystem* class. Some of these methods return *Create Artifact*, *Create Service*, and *Create Rules* objects, other methods return data attributes of *Information Models* according to types, test types of data attributes, return *Stream* or *Ad-hoc Services*, or test types of *Services* or *Artifact Rules*.

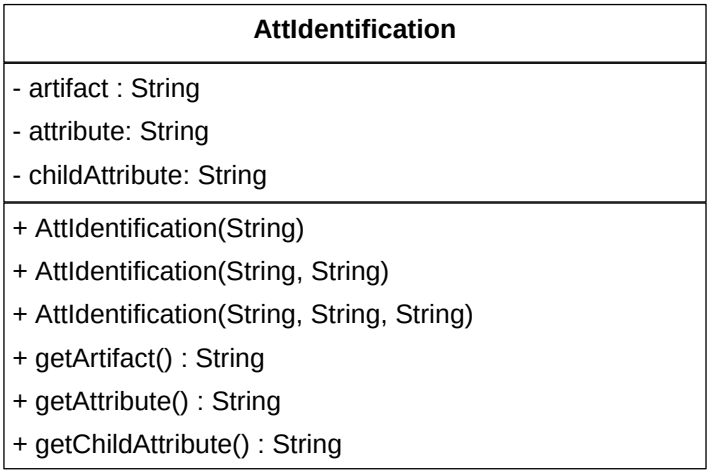


Figure 6.8 UML Class of AttIdentification

6.1.1.5. Database Manager Sub-Module

The *Database Manager* sub-module is responsible of performing database operations and generating necessary events that are used by the *AQL Rule Execution Engine*. The *Database Manager* consists of three classes; *DatabaseManager*, *DatabaseOperationEvent*, and a *DatabaseListener* interface. The *Database Manager* is used by the *Semantic Interpreter*, *Process Explorer*, *Service Manager*, and *AQL Rule Execution Engine.*

The *DatabaseListener* interface declares one method "*onDatabaseOperation(DatabaseOperationEvent) that* must be implemented by all classes of the *DatabaseListener* interface.

The *DatabaseOperationEvent* class is used to represent database operation events generated by the *DatabaseManager* class. Three attributes that characterizes a database operation are involved:

- *artName* holds the name of the *Artifact Class* targeted by the database operation.
- *artPK* holds the primary key of the *Artifact* instance targeted by the database operation. And,
- *dbOperation* specifies the type of the performed database operation: *insert, update,* or *delete.*

Figure 6.9 illustrates the *UML* class of *DatabaseOperationEvent.*

The *DatabaseManager* class provides static methods for creating and managing an underlying database. The *DatabaseManager* class performs three main functionalities:

1) Create and manage database tables corresponding to *Artifact Classes*, *complex attributes*, *stream attributes*, *reference attributes,* and *states*. the *DatabaseManager* class provides methods for creating, populating, manipulating and querying these tables.

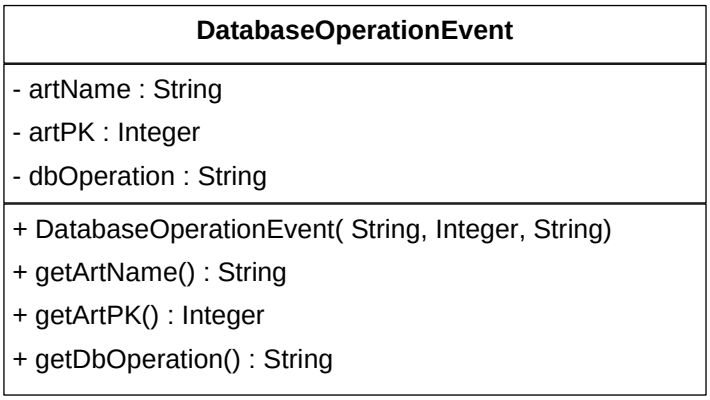


Figure 6.9 UML Class of DatabaseOperationEvent

2) Create and manage system tables that are responsible for executing services and receiving user-generated or timely events. The *DatabaseManager* class creates three system tables:
   - *InvokedServicesSysTable* table keeps track of the invoked services by the *AQL Rule Execution Engine*.
   - *InvokedServicesInputSysTable* table keeps track of a list of input artifact instances of the invoked services.
   - *GeneratedEventsSysTable* table keeps track of the user-generated or timely events.

3) Implement a variant of the *observer design pattern* responsible for registering listeners and sending database operation events to registered listeners. This functionality is useful for the *AQL Rule Execution Engine* that listens for database operation events and reevaluates *Artifact Rules* when these events are received.

Figure 6.10 illustrates the *UML* class of *DatabaseManager*.

6.1.1.6. AQL Rule Execution Engine Sub-Module

The *AQL Rule Execution Engine* sub-module is responsible for evaluating and executing *Artifact Rules* every time a database operation is performed. In

order to be notified when database operations are performed, the *AQL Rule Execution Engine* is registered as a listener to the *DatabaseManager* class. The *DatabaseManager* class will then generate and send database operation events every time database operations are performed.

The *AQL Rule Execution Engine* consists of two classes: *RuleEngineDatabaseListener* and *RuleEngine*.

The *RuleEngineDatabaseListener* class implements the *DatabaseListener* interface. Specifically, the *RuleEngineDatabaseListener* implements the *onDatabaseOperation(DatabaseOperationEvent)* method in order to invoke the *evaluateRules(DatabaseOperationEvent)* method of the *RuleEngine* class and passes a *DatabaseOperationEvent* object as parameter.

The *RuleEngine class* provides methods that evaluate and execute *Artifact Rules*. Figure 6.11 illustrates the *UML* class of *RuleEngine*.

The *evaluateRules(DatabaseOperationEvent)* method is invoked by the *RuleEngineDatabaseListener* and passed a *DatabaseOperationEvent* object as a parameter. When a database operation is performed, the *evaluateRules()* method reevaluates the registered *Artifact Rules* and executes the matching rules. However, the *performInvocationAction()* method is used to execute the action of service invoking *Artifact Rules*. The *performTransitionAction()* method is used to execute the action of state transitioning *Artifact Rules*. Finally, the *checkIfInstanceMatchCondition()* method checks if an *Artifact* instance matches the condition of an *Artifact Rule*.

6.1.1.7. Process Explorer Sub-Module

The *Process Explorer* sub-module provides views and editors for executing *Artifact Systems* and manipulating *Artifact* instances. It consists of two packages: *ArtifactInstanceView*, and *ArtifactProcessExplorer.*

The *ArtifactInstanceView* package consists of five classes: *ArtifactInstanceNode*, *ArtifactInstancesContentProvider*, *ArtifactInstancesLabelProvider*, *ArtifactInstancesView*, and *ArtifactNode*. they provide a graphical user interface, called the *Artifact Instance Viewer,* and lists *Artifact* instances registered in the *Artifact System*.

DatabaseManager
- databaseListeners : List<DatabaseListener>
+ getDatabaseListeners() : List<DatabaseListener> + addDatabaseListener(DatabaseListener) : void + removeDatabaseListener(DatabaseListener) : void - fireDatabaseOperationEvent(DatabaseOperationEvent) : void + createConnection() : void + shutdown() : void + createArtifactTable(CreateArtifact) : void + createComplexAttributesTables(CreateArtifact) : void + createReferenceAttributesTables(CreateArtifact) : void + createStreamAttributesTables(CreateArtifact) : void + createAndPopulateStatesTable(CreateArtifact) : void + getArtInstancesPKs(String) : List<String> + getArtSimpleAttributesRow(String, Integer) : Map<String, String> + populateArtComplexAttributeTable(String, Integer, String, Table) : void + populateArtReferenceAttributeTable(String, Integer, String, String, Table) : void + createInvokedServicesSysTable() : void + createInvokedServicesInputSysTable() : void + hasInvokedService(String, Integer) : void + retrieveInvokedService(String, Integer) : void + createArtifactInstance(String) : Integer + insertChildArtifactReference(String, Integer, String, int) : void + updateArtSimpleAttribute(String, Integer, String, String, String) : void + updateArtState(String, Integer, String) : void + deleteArtifactInstance(CreateArtifact, String, Integer) : void + deleteInvokedService(Integer) : void + insertInvokedService(String) : Integer + insertInvokedServiceInput(Integer, String, Integer, Boolean) : void + createGeneratedEventsSysTable() : void + getArtifactState(String, Integer) : String + insertRuleEvent(String, Integer, String) : void + insertStreamTuple(String, Integer, String, List<Attribute>, Map<String, String>) : void + populateArtStreamAttributeTable(String, Integer, String, Table) : void + getLastStreamRow(String, Integer, String) : Map<String, String> + getArtInstancesMatchingCondition(String, List<Predicate>) : List<Integer>

Figure 6.10 UML Class of DatabaseManager

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

RuleEngine
+ evaluateRules(DatabaseOperationEvent) : void - performTransitionAction(CreateRule) : void - performInvocationAction(CreateRule) : void - checkIfInstanceMatchCondition(CreateRule, String, Integer) : Boolean

Figure 6.11 UML Class of RuleEngine

The *Artifact Instance Viewer* (see Figure 6.12) includes a context menu for refreshing, creating and deleting *Artifact* instances, and opens the *Artifact Process Explorer* when double clicking an *Artifact* instance.

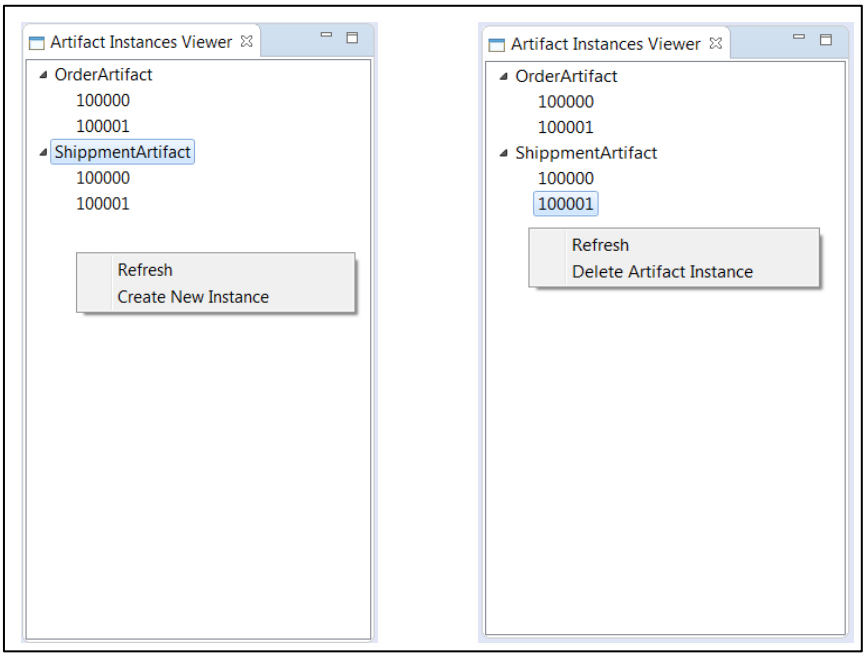


Figure 6.12 Artifact Instance Viewer

The *ArtifactProcessExplorer* package consists of three classes: *ArtifactProcessExplorer*, *CallArtifactProcessExplorer*, and *ProcessExplorerInput*. They provide a graphical user interface -- the *Artifact Process Explorer* (see Figures 6.13 and 6.14), which allows users to explore the *Information Model* and *Lifecycle* of an *Artifact* instance. It also executes invoked *Services*, and creates user-generated or timely events.

The *Artifact Process Explorer* is composed of several panels. The *Simple Attributes* panel displays simple attributes, states and primary key of the *Artifact* instance. Complex, reference, and stream attributes are

represented as tables. In addition stream attributes' tables are continuously updated in order to continuously display incoming tuples.

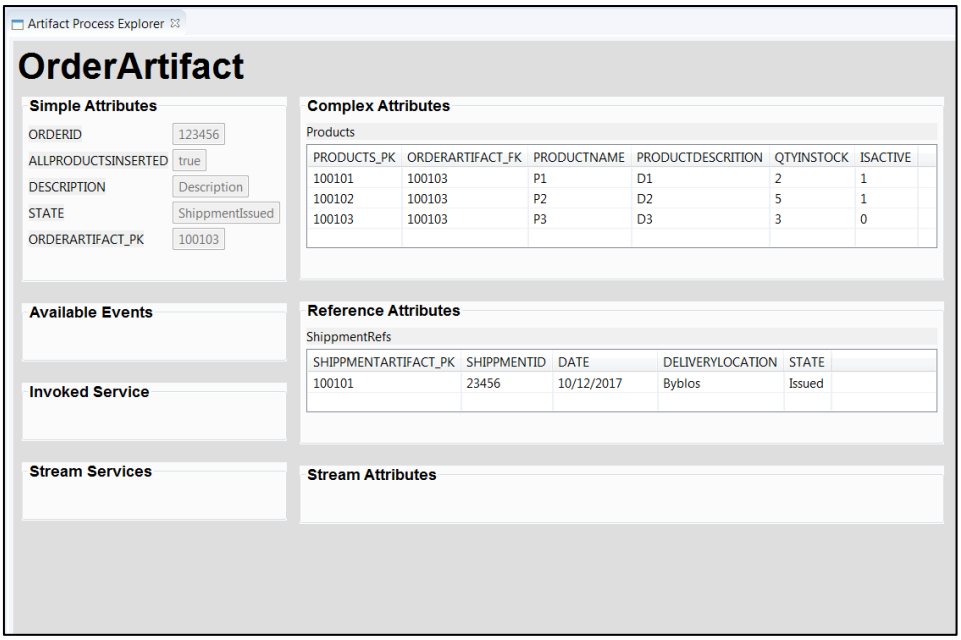


Figure 6.13 Artifact Process Explorer 1

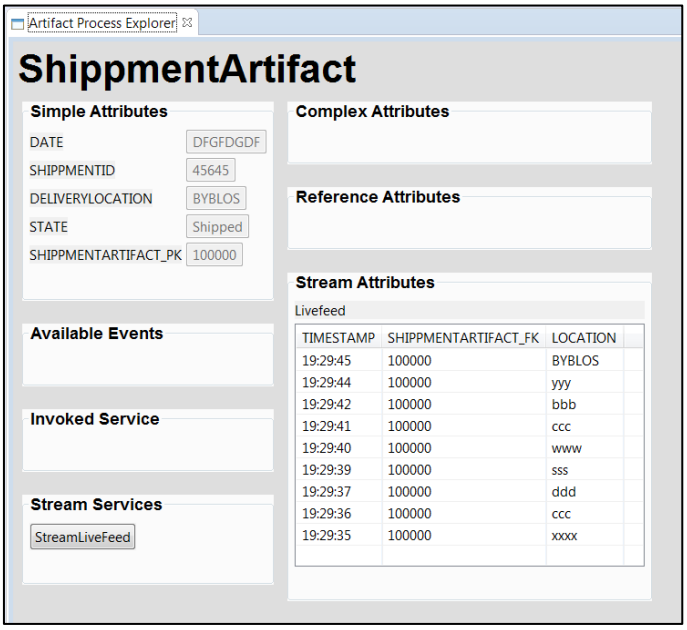


Figure 6.14 Artifact Process Explorer 2

The *Available Events* panel displays user-generated or timely events that can be created at a particular state of the *Artifact* instance. Available events are represented as a list of buttons that when clicked will create the corresponding event.

The *Invoked Services* panel displays the ad-hoc *Services* that are invoked by *Artifact Rules*. The invoked *Services* are represented as a list of buttons that when clicked will execute the corresponding *Service* using the *Services Manager*.

The *Stream Services* panel displays the stream services available to an artifact instance. The stream services are represented as lists of buttons that when clicked will open the stream *Service* configuration dialog using the *Services Manager*.

6.1.1.8. Service Manager Sub-Module

The *Service Manager* sub-module is responsible for managing ad-hoc and stream *Services*. It generates dialogs used to configure and execute *Services*. Three configurations are available for ad-hoc *Services*:

1) Automatically generated *GUI Forms* that collect needed data attribute values directly from users.
2) *Web Service Calls* that return needed data attribute values from *Web Services*. And,
3) *User Defined Functions* that perform user-defined computations and return needed data attribute values.

Two configurations are available for stream *Services*:

1) *File Readers* that continuously read input streams from *Comma-Separated Values (CSV)* files.
2) *Web Service Calls* that continuously receive input streams from *Web Services*.

The *Service Manager* consists of five classes: *ServicesManager, AdHocServiceDialog, StreamServiceDialog*, *StreamServiceThread*, and *StreamServiceThreadProvider*, and performs three main tasks:

1) Invoke ad-hoc *Services* using the *AdHocServiceDialog* class.
2) Configure stream *Services* using the *StreamServiceDialog* class. And,
3) Update database with *Services'* returned values using the *DatabaseManager* class.

Figure 6.15 illustrates the *UML* class of the *ServicesManager*.

ServicesManager
+ invokeAdHocService(CreateService, Map<String, Integer>, IWorkbenchWindow) : void + configureStreamService(CreateService, String, Integer, IWorkbenchWindow) : void - updateDatabaseWithServiceOutput(CreateService, Map<String, Integer>, HashMap<AttIdentification, String>, IWorkbenchWindow) : void

Figure 6.15 UML class of ServicesManager

The *AdHocServiceDialog* class automatically generates dialogs for configuring and executing ad-hoc *Services* (see Figure 6.16).

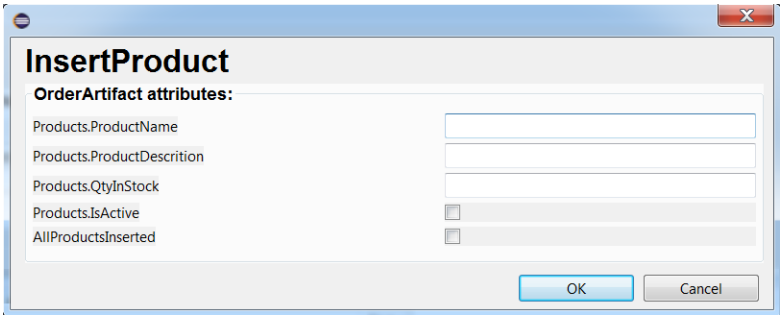


Figure 6.16 Automatically generated dialog of an ad-hoc Service

The *StreamServiceDialog* class automatically generates dialogs for configuring stream *Services*. The *StreamServiceThread* class and the *StreamServiceThreadProvider* class are used to create threads that continuously receive data streams from configured input streams (see Figure 6.17).

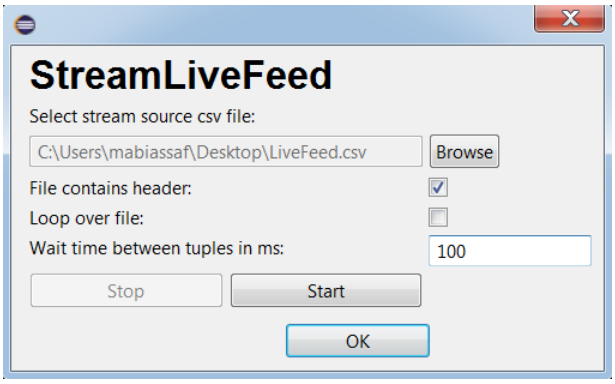


Figure 6.17 Automatically generated dialog of a stream Service

6.1.2. CAM Modeler Module

The *CAM Modeler* module is a graphical editor that implements the *Conceptual Artifact Modeling Notation (CAMN)* and allows the modeling of *Conceptual Artifact Models (CAMs)*. It is based on *HTML5*, *JavaScript*, and the *JointJS JavaScript Diagramming Library*. Figure 6.18 illustrates the graphical interface of the *CAM Modeler*.

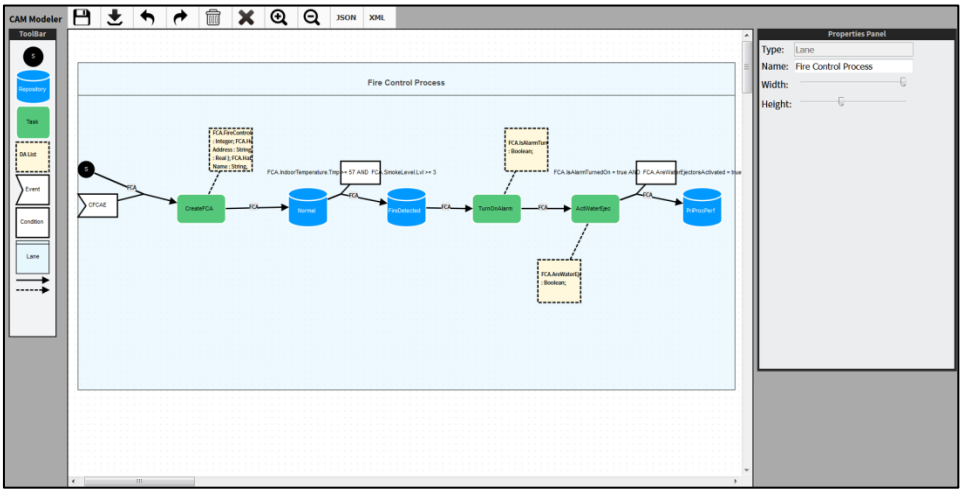


Figure 6.18 CAM Modeler Graphical Interface

A *Toolbar* allows the user to select and insert *CAMN* into the *Drawing Canvas*. Additionally, a *Properties Panel* allows the specification of properties for a selected element. The *CAM* can then be saved as an array of elements in an *XML* file. Every element has an *ID* attribute that uniquely identifies it and a *Construct* attribute that specifies its *CAMN* type. The *Construct* attribute can be *Task, Repository, Flow Connector, Data Attribute List, Event,* or *Condition*. Moreover, every construct type has additional attributes that describe it. *Flow Connector* has attributes for storing the ids of its source and destination elements in addition to attached *Event* and *Condition* elements if any. *Repository* has attributes for storing its name and the name of the associated *Artifact*. *Data Attribute List* has an attribute for specifying a list of *Data Attributes*. Finally, *Task* has an attribute for storing its name.

6.1.3. AQL Generator Module

The *AQL Generator* module is a *JavaScript* based module that implements the semantics described in Section 4.3. It takes as input an *XML* file created

by the *CAM Modeler*, generates an equivalent *Artifact System*, and returns as output the corresponding *AQL* queries.

6.1.4. CAM Matcher Module

The *CAM Matcher* module is a graphical editor that is used to graphically capture correspondences relationships between two local *CAMs*. It is based on *HTML5*, *JavaScript*, and *JointJS JavaScript Diagramming Library*. The captured correspondences can be registered in an *XML* file alongside the two local *CAMs*. It consists of two graphical interfaces: the main graphical interface captures correspondences relationships between *Tasks*, *Repositories*, and *Artifacts* (Figure 6.19) and the *Attribute Matcher* graphical interface, which captures correspondences relationships between data attributes (Figure 6.20). The data attributes displayed in the *Attribute Matcher* graphical interface are automatically generated from the two local *CAMs*.

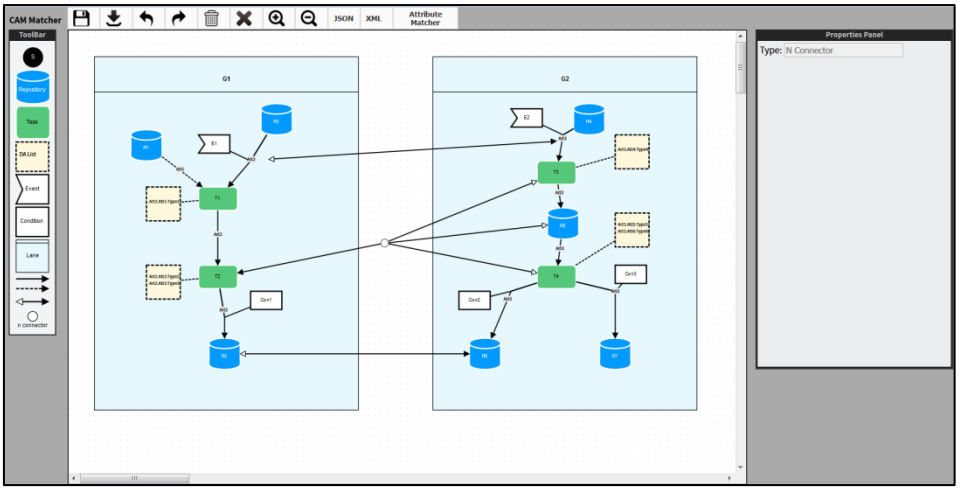


Figure 6.19 CAM Matcher Main Graphical Interface

6.1.5. CAM Merger Module

The *CAM Merger* module is a graphical editor that takes as input two local *CAMs* and their correspondences relationships as *XML* files, merges the two *CAMs* according to the semantics described in Section 5.3, and displays the generated global *CAM*. It is based on *HTML5*, *JavaScript*, and *JointJS JavaScript Diagramming Library*.

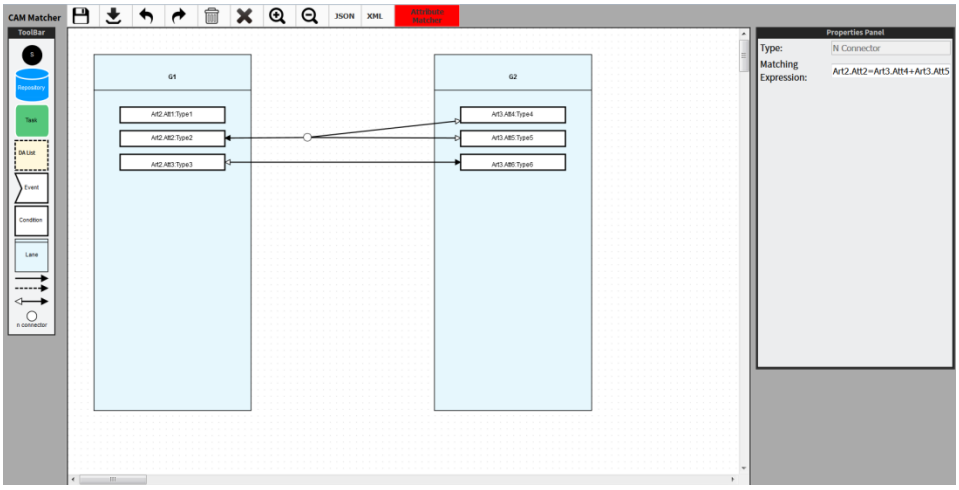


Figure 6.20 Attribute Matcher Graphical Sub-Interface

6.1.6. CAM Mapper Module

The *CAM Mapper* module is a *JavaScript* module that implements the mapping functions described in Section 5.4. It takes as input a global *CAM*, in addition to two local *CAMs* and their correspondences relationships. For a given element from the global *CAM*, the *CAM Mapper* returns equivalent elements in the two local *CAMs*.

6.2. Experimentation

6.2.1. Fire Control Process Scenario

In order to validate our contributions, we illustrate an experimental scenario about the detection and control of house fires in the context of a smart city.

We assume that every house in a smart city is equipped with temperature and smoke sensors in addition to alarm and water ejectors (i.e., actuators). Moreover, every house is wirelessly connected to the city control center that remotely detects fire incidents and controls responses. Additionally, the city control center manages, in its databases, information about every house, like its location, surface, habitats, fire station addresses and can remotely cut off the power grid in every house.

The city control center detects a fire incident when house temperature sensor values become higher than 57°C and smoke sensor levels exceed a

threshold of 3 in a range of values between 0 and 5. When the fire incident occurs, the control system turns on alarms and activates water ejectors. Then the control system alerts the closest fire station with information regarding the house's address and its related data as extracted from its database. If the fire station is not able to respond in case of insufficient resources, then the control center will identify another close fire station and alert it. This process is repeated until an available fire station is successfully responding. In addition, the city control system informs the house habitats about the fire incidents by automatically sending SMS messages to their mobile phones.

If the fire has not been extinguished and water has been depleted from ejectors, water pumps are remotely activated to refill the ejectors. Water levels are detected using water level sensors and water pumps are activated using water pump actuators that are also installed in every house of the smart city.

Finally, when the house temperature becomes less than 50°C and the smoke level value is less than or equal to 1, the fire is considered to be extinguished. The fire incident is then archived in the fire control database for any possible future references and analytics.

6.2.2. Analysis of Fire Control Artifact System

6.2.2.1. Artifact Classes

Since artifacts are goal-oriented, every artifact is designed to perform and reach a particular goal. To this end, we design an *Artifact System* for the fire control smart process based on two artifacts:

1) The *FireControlArtifact* deals with detecting house fires and performing reactive procedures.
2) The *FireStationAlertArtifact* deals with locating fire stations close to the house under fire and successfully alerting one of them.

The two artifacts are related in a parent to child relationship; The *FireControlArtifact* is the parent artifact that emits the *FireStationAlertArtifact* child artifact.

The *FireControlArtifact* collects information about the house, the fire, and the reactive procedures. Its *Information Model* is illustrated listed in Table 6.1.

Table 6.1 FireControlArtifact Attributes

Attribute	Data Type	Category
FireControlArtifactId	Integer	simple
FireDate	Date	simple
IsAlarmTurnedOn	Boolean	simple
AreWaterEjectorsActivated	Boolean	simple
AreHabitatsNotified	Boolean	simple
AreWaterPumpsActivated	Boolean	simple
House(Address, Surface)	(String, Integer)	complex
Habitats(Name, PhoneNum)	(String, Integer)	complex
FireStationAlert	FireStationAlertArtifact	reference
IndoorTemperature(Time, Tmp)	(TimeStamp, Integer)	stream
SmokeLevel(Time, Lvl)	(TimeStamp, Integer)	stream
WaterLevel(Time, Lvl)	(TimeStamp, Lvl)	stream

The *FireControlArtifact Lifecycle* includes the following states:

- *Normal* is the initial state. It signifies that an artifact instance corresponding to a particular house is created and no fire is detected yet.
- *FireDetected* signifies that a fire is detected.
- *PrimaryProcedurePerformed* signifies that alarm and water ejectors are activated.
- *ClosestFireStationAlerted* signifies that a close fire station is successfully located and alerted.
- *HabitatsInformed* signifies that all of the house's habitats are informed by sending them SMS messages.
- *EjectorsDepleted* signifies that water ejectors are depleted.
- *EjectorsRefilled* signifies that water pumps are activated in order to refill water ejectors.
- *FireExtinguished* signifies that the fire has been extinguished.
- *Archived* is the final state. It signifies that fire information is registered and the artifact instance has been archived for future references.

Figure 6.21 illustrates the transitions that can be performed between the *FireControlArtifact* states.

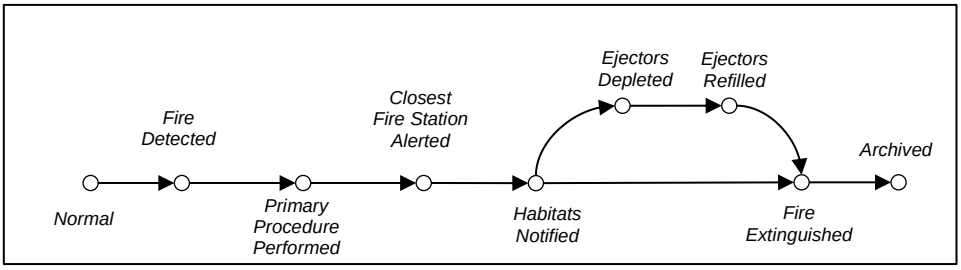


Figure 6.21 State Transitions of FireControlArtifact

On the other hand, the *FireStationAlertArtifact* collects information about nearest fire stations and alerts incidents. Its *Information Model* includes the attributes listed in Table 6.2.

Table 6.2. FireStationAlertArtifact attributes

Attribute	Data Type	Category
FireStationAlertArtifactId	Integer	simple
FireStationAddress	String	simple
House(Address, Surface)	(String, Real)	complex
IsSuccessfullyAlerted	Boolean	simple

The *FireStationAlertArtifact Lifecycle* includes the following states:

- *Issued* is the initial state. It signifies that a new artifact instance corresponding to the parent artifact is created.
- *Located* signifies that a fire station has been located.
- *Failed* signifies that the located fire station could not be alerted.
- *Alerted* is the final state. It signifies that the located fire station was successfully alerted.

Figure 6.22 illustrates the state transitions to be performed between the the *FireStationAlertArtifact* states.

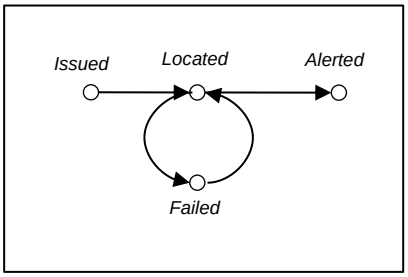


Figure 6.22 State Transitions of *FireStationAlertArtifact*

6.2.2.2. Services

Services manipulating data artifact in the fire control smart process are divided into ad-hoc and stream *Services*. Ad-hoc *Services* include:

- *CreateFCA* service initializes a new instance of the *FireControlArtifact* (*FCA*) class and defines its *FireControlArtifactId, House,* and *Habitats* attributes.
- *TurnOnAlarm* service activates the alarm actuator of the corresponding *FireControlArtifact* and defines its *IsAlarmTurnedOn* attribute.
- *ActivateWaterEjectors* service activates water ejectors of the corresponding *FireControlArtifact* and defines its *AreWaterEjectorsActivated* attribute.
- *IssueFireStationAlert* service creates a new instance of the *FireStationAlertArtifact* (*FSAA*) as a child artifact of the corresponding *FireControlArtifact.* It defines its *FireStationAlertArtifactId* and *House* attributes.
- *NotifyHabitats* service sends SMS messages using an SMS Web Service to the habitats of the corresponding *FireControlArtifact* in order to notify them about the fire.
- *ActivateWaterPumps* service activates the water pump actuator in order to refill the depleted ejectors of the corresponding *FireControlArtifact*. It defines its *AreWaterPumpsActivated* attribute.
- *RegisterFireData* service registers data about the extinguished fire corresponding to the *FireControlArtifact*.
- *LocateFireStation* service locates a close fire station to the house under fire of the corresponding *FireStationAlertArtifact*.
- *AlertFireStation* service alerts the located fire station of the corresponding *FireStationAlertArtifact*.

The stream *Services* which are invoked when an artifact instance is created include the following *Services*:

- *StreamIndoorTemperature* service streams temperature readings from the temperature sensor of the corresponding *FireControlArtifact* into its *IndoorTemperature* attribute.
- *StreamSmokeLevel* service streams smoke level readings from the smoke sensor of the corresponding *FireControlArtifact* into its *SmokeLevel* attribute.

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

- *StreamWaterLevel service* streams water level readings from the water level sensor of the corresponding *FireControlArtifact* into its *WaterLevel* attribute.

6.2.2.3. Artifact Rules

Artifact Rules automate the fire control smart process by invoking ad hoc *Services* and performing lifecycle state transitions on artifact instances. *Artifact Rules* for the fire control smart process include the following:

- Rule 1 invokes the *CreateFCA* service when a *Create Fire Control Artifact Event* is received.
- Rule 2 changes the state of *FireControlArtifact* to the *Normal* state if it is initialized and its *FireControlArtifactId*, *House*, and *Habitats* attributes are set.
- Rule 3 changes the state of *FireControlArtifact* to the *FireDetected* state if it is in the *Normal* state, house temperature sensor values are higher than 57°C, and smoke sensor levels exceed a threshold of 3.
- Rule 4 invokes the *TurnOnAlarm* service if a *FireControlArtifact* is in the *FireDetected* state and its *IsAlarmTurnedOn* attribute is not set.
- Rule 5 invokes the *ActivateWaterEjectors* services if a *FireControlArtifact* is in the *FireDetected* state, its *IsAlarmTurnedOn* attribute is set, and *its AreWaterEjectorsActivated* attribute is not set.
- Rule 6 changes the state of a *FireControlArtifact* to the *PrimaryProcedurePerformed* state if it was in the *FireDetected* state and its alarm and water ejectors are activated.
- Rule 7 invokes the *IssueFireStationAlert* service if a *FireControlArtifact* is in the *PrimaryProcedurePerformed* state and its *FireStationAlert* attribute is not set.
- Rule 8 changes the state of a *FireControlArtifact* to the *ClosestFireStationAlerted* if it is in the *PrimaryProcedurePerformed* and its child *FireStationAlertArtifact* is created.
- Rule 9 invokes the *NotifyHabitats* service if a *FireControlArtifact* is in the *ClosestFireStationAlerted* state and its *AreHabitatsNotified* attribute is not set.
- Rule 10 changes the state of a *FireControlArtifact* to the *HabitatsInformed* state if it is in the *ClosestFireStationAlerted* state and its house's habitats are sent SMS notification messages.

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

- Rule 11 changes the state of a *FireControlArtifact* to the *EjectorsDepleted* state if it is in the *HabitatsInformed* state and water level drops to zero.
- Rule 12 invokes the *ActivateWaterPumps* service if a *FireControlArtifact* is in the *EjectorsDepleted* state and its *AreWaterPumpsActivated* attribute is not set.
- Rule 13 changes the state of a *FireControlArtifact* to the *EjectorsRefilled* state if it is in the *EjectorsDepleted* state and its water pumps are activated.
- Rule 14 changes the state of a *FireControlArtifact* to the *FireExtinguished* state if it is in the *HabitatsInformed* state and the house's temperature becomes less than 50°C and the smoke level is less than or equal to 1.
- Similarly, rule 15 changes the state of a *FireControlArtifact* to the *FireExtinguished* state if it is in the *EjectorsRefilled* state and the house's temperature becomes less than 50°C and the smoke level is less than or equal to 1.
- Rule 16 invokes the *RegisterFireData* service if a *FireControlArtifact* is in the *FireExtinguished* state and its fire date and duration are not registered yet.
- Rule 17 changes the state of a *FireControlArtifact* to the final *Archived* state if it is in the *FireExtinguished* state and its fire incident data are registered.
- Rule 18 changes the state of an initialized *FireStationAlertArtifact* to the *Issued* state if its *FireStationAlertArtifactId* and *House* attributes are set.
- Rule 19 invokes the *LocateFireStation* service if a *FireStationAlertArtifact* is in the *Issued* state and its *FireStationAddress* attribute is not set.
- Rule 20 changes the state of a *FireStationAlertArtifact* to the *Located* state if it is in the *Issued* state and its fire station address is located.
- Rule 21 invokes the *AlertFireStation* service if a *FireStationAlertArtifact* is in the *Located* state and its *IsSuccessfullyAlerted* attribute is not set.
- Rule 22 changes the state of a *FireStationAlertArtifact* to the *Failed* state if it is in the *Located* state and the located fire station was not successfully alerted.

- Rule 23 invokes the *LocateFireStation* service if a *FireStationAlertArtifact* is in the *Failed* state and its *FireStationAddress* attribute is not set.
- Rule 24 changes the state of a *FireStationAlertArtifact* to the *Located* state if it is in the *Failed* state and its fire station address is located.
- Rule 25 changes the state of a *FireStationAlertArtifact* to the *Success* state if it is in the *Located* state and the located fire station was successfully alerted.

6.2.3. Modeling of Fire Control Artifact System

Implementing the fire control *Artifact System* by directly writing corresponding *AQL* queries is a time consuming, error prone, and not a visual-oriented approach. A more practical and user-friendly approach is to model the fire control artifact system using the *CAM Modeler* as *a Conceptual Artifact Model (CAM)* which is then automatically translated by the *AQL Generator* into *AQL* queries of the corresponding *Artifact System*. The generated *AQL* queries can then be executed by the *AQL Processor*. Figure 6.23 illustrates the *CAM* of the fire control process while the generated *AQL* queries are listed in Appendix A.

6.2.4. Integrating Local Fire Control CAMs

In addition to the smart process modeled in Section 6.2.3, we have modeled a variant fire control smart process in the context of another smart city. In this context, the detection and control of house fires are performed in a slightly different manner so that we have similar and different elements with the first fire control smart process. Our goal is to integrate both smart processes in order to provide a *Uniform View* for supervising and querying both smart processes in a uniform manner.

Using the *CAM Matcher*, *CAM Merger*, and *CAM Mapper*, we have integrated the local *CAMs* of both fire control smart processes and generated a global *CAM* in addition to mapping rules that translates elements between global and local *CAMs*. Figure 6.24 illustrates some of the correspondences captured by the *CAM Matcher* for a part of the local *CAMs* where *Data Attribute Lists* are omitted for readability concerns. Figure 6.25 illustrates part of the generated global *CAM*.

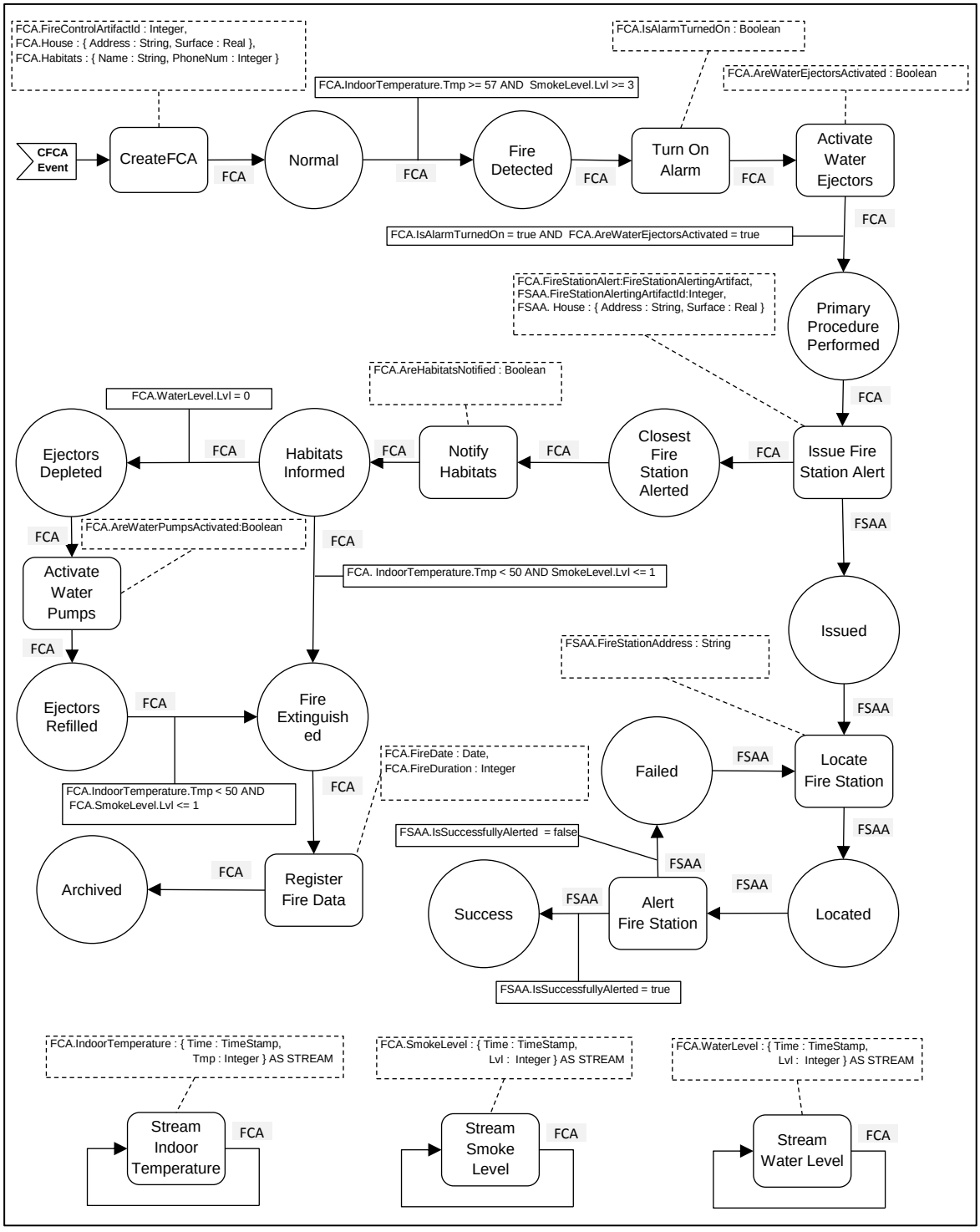


Figure 6.23 Fire Control Conceptual Artifact Model

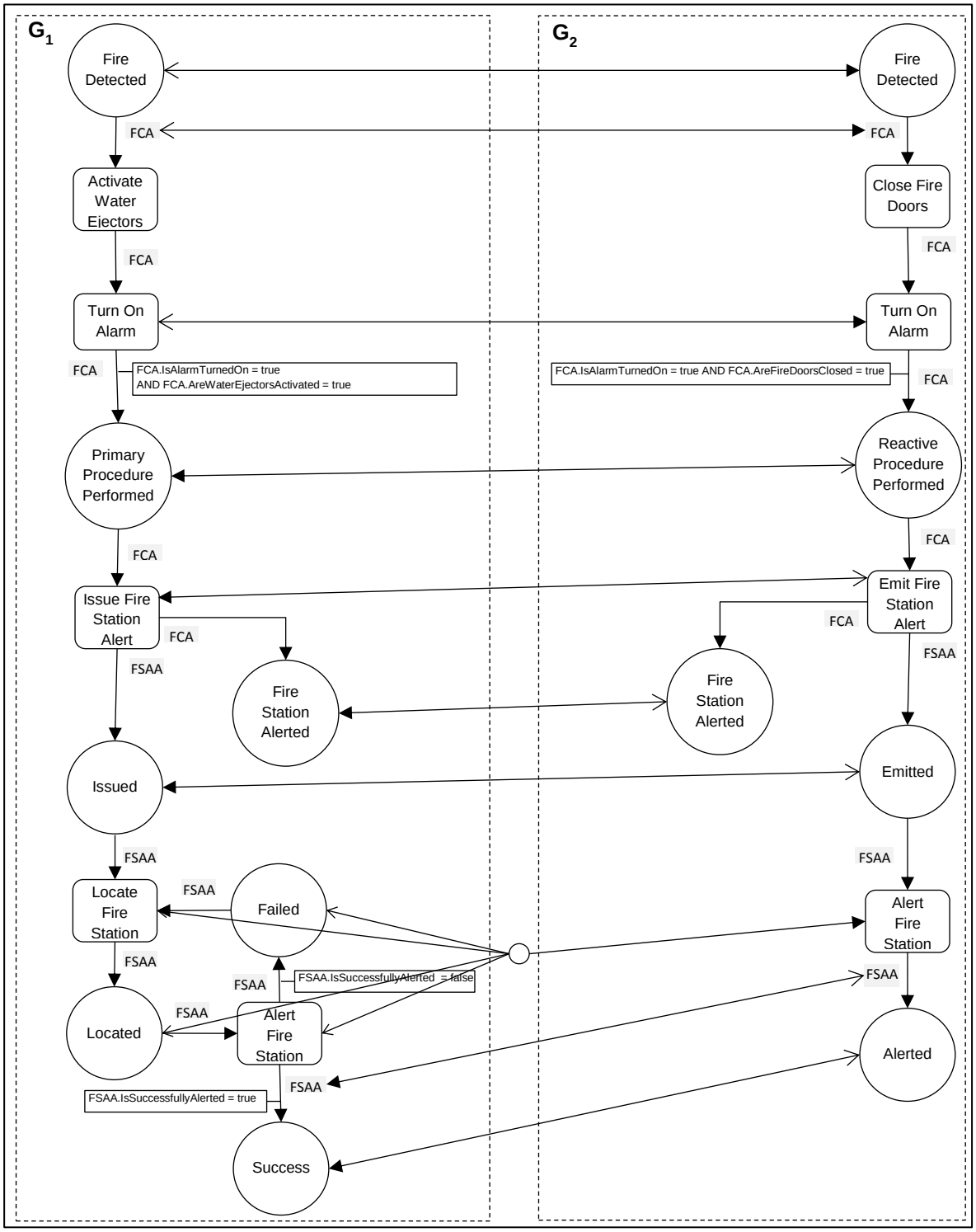


Figure 6.24 Fire Control Local CAMs Correspondences

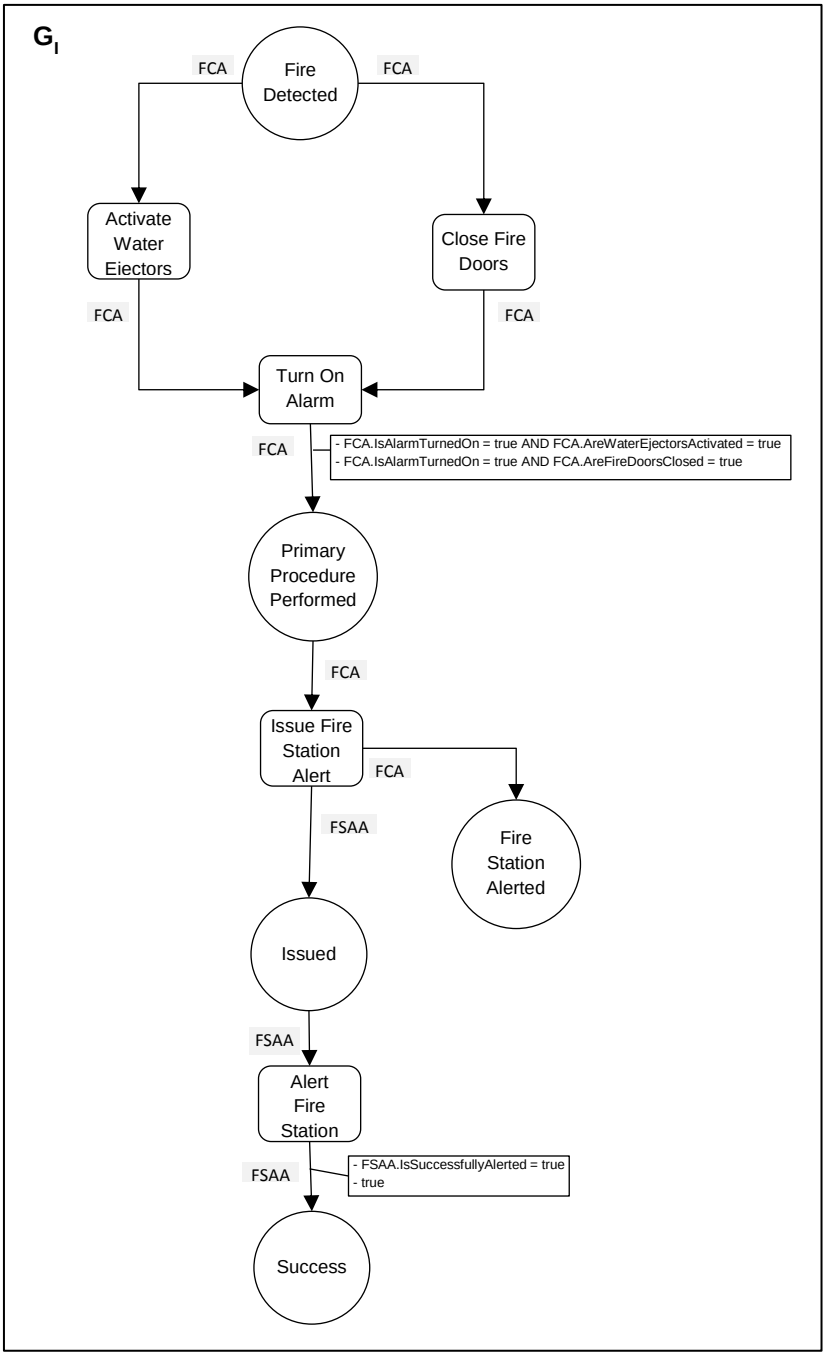


Figure 6.25 Fire Control Global CAM

6.3. Summary of Prototyping and Experimentation

In this chapter, we describe the architecture of the developed prototype for validating our contributions. The prototype covers the four phases of the *Artifact Integration Framework*: *Execution Phase*, *Specification Phase*,

Modeling Phase, and *Integration Phase.* The prototype is based on several programming languages including; *HTML5* [00b], *XML* [00c], *Java* [00d], *Xtend* [00e], and *JavaScript* [00f] languages, and programming frameworks; *Eclipse Rich Client Platform (Eclipse RCP)* [00g], *Xtext Framework* for the development of *Domain Specific Languages (DSL)* [00h].

Moreover, we present and implement an experimental scenario about the detection and control of house fires in the context of a smart city.

7

Conclusion

Chapter Outline

Artifact-centric process modeling offers an alternative approach to traditional activity-centric process modeling, and has been demonstrated to provide many benefits and advantages. In this thesis, we address the problem of modeling, integration and execution of artifact-centric processes which are suitable for implementing various types of processes including smart processes in the *Internet of Things (IoT)*.

In this chapter, we discuss about different design choices we made, the limits of our work, and provide future works perspectives.

7.1. Summary of Contributions

The main goal of this thesis is to integrate heterogeneous artifact-centric processes. We have proposed a complete *Artifact Integration Framework* that not only deals with integrating artifact-centric processes but also with modeling, querying, specifying, and executing artifact-centric processes. The proposed *Artifact Integration Framework* is based on four phases, covering *Modeling*, *Specification*, *Integration*, and *Execution*.

7.1.1. Modeling Phase

In the *Modeling Phase*, we model *Artifact Systems* using conceptual models that we refer to as *Conceptual Artifact Models (CAMs)*. We have proposed the *Conceptual Artifact Modeling Notation (CAMN),* a minimalistic graphical notation which is used to graphically model *Artifact Systems* without writing complex and error prone *AQL* queries. The constructed *Conceptual Artifact Models (CAMs)* include all components of an *Artifact System* into the same model thus providing a more representative and holistic model than existing works. Furthermore, the proposed modeling approach combines the advantages of both procedural and declarative modeling approaches. Additionally, we have proposed modeling patterns that include data stream specific patterns required for modeling smart processes which are not supported by existing works. Finally, we have defined transformation semantics for generating valid and functional *Artifact Systems* from *CAMs* based on a proposed formalism.

7.1.2. Specification Phase

In the *Specification Phase*, we generate *Artifact Systems* from *CAMs* that are modeled in the previous phase. We have proposed a formal model for *Artifact Systems* that is specifically adapted to model modern day smart processes that are based on the *Internet of Things (IoT)* in a simple and

intuitive manner. The formal model supports data streams, and represents sensors, and actuators which are needed in *IoT* based processes. Moreover, unlike existing works, the proposed formal model allows the definition, manipulation, and querying of *Artifact Systems* at a high-level without dealing with the underlying model of relations and streams. The formal model makes use of four high-level attribute categories: *Simple*, *Complex*, *Reference*, and *Stream*. Stream attributes are used to represent data streams. Complex attributes are used to represent complex relationships between artifacts and various data objects in the process. And, reference attributes are used to represent *Parent-Child* relationships between artifacts. Moreover, the formal model presents two types of *Services*: *Ad-hoc* and *Stream*. Ad-hoc services can be used to perform actions on actuators. Stream services are used to stream data from stream sources like sensors.

Moreover, we have also proposed the *Artifact Query Language (AQL)* to declaratively define *Artifact Systems* and manipulate their artifact instances. In order to reduce its learning curve, the *AQL* is based on syntax similar to *SQL*, but unlike *SQL*, the *AQL* does not deal with low level relations and streams. Instead, the *AQL* provides simple and declarative statements that hide the underlying model of relations and streams. Furthermore, the *AQL* statements are grouped into two parts; *Artifact Definition Language (ADL)*, and *Artifact Manipulation Language (AML)*. *ADL* provides statements to define *Artifact Classes*, *Services*, and *Rules*. *AML* provides statements to manipulate and query artifact instances. Additionally, the *AML* is used in order to uniformly query *Unified Views* in the *Artifact Integration Framework*. Finally, the *AQL* supports *Data Streams* and *Continuous Query* capabilities and allows *Complex Event Processing* over *Data Streams* through the use of *Artifact Rules*.

7.1.3. Integration Phase

In the *Integration Phase*, we have integrated several local *CAMs* in order to generate one global *CAM,* acting as a unified view. Since *CAMs* combines both process and data aspects into the same model, we have proposed an *Artifact Integration System* based on specialized integration semantics for integrating heterogonous *CAMs.* The proposed integration semantics combines integration mechanisms from both *Data Integration* and *Business Process Merging* in addition to artifact-specific integration mechanisms. The proposed artifact integration semantics is based on three sub-phases: *Matching, Merging*, and *Mapping*.

The *Matching Sub-Phase* relies concepts from *Schema Matching* in order to identify correspondence relationships between elements of local

CAMs. Three types of correspondence relationships are supported; *Unique*, *Equivalent*, and *Composition.*

The *Merging Sub-Phase* uses concepts from *Business Process Merging* in order to merge local *CAMs* and generate global *CAMs*. The merging semantics are based on the identified correspondences and on *Flow Connector*'s re-branching and re-connecting.

Finally, the *Mapping Sub-Phase* uses concepts from *Schema Mapping* in order to define mapping functions that translate elements and data between global and local *CAMs*.

7.1.4. Execution Phase

In the *Execution Phase*, the *Execution Engine* executes *Artifact Systems* by translating *AQL* queries into *Semantic* queries. *Semantic* queries are then executed on a *Database Management System* to perform relational and stream operations. The *Execution Engine* is also responsible for invoking *Ad-hoc* and *Stream* services.

7.1.5. Prototype Implementation

In order to validate our various contributions, we have also developed a prototype that implements the artifact-centric process integration framework. The prototype covers the four phases of the *Artifact Integration Framework*: *Execution*, *Specification*, *Modeling*, and *Integration* using six main modules in addition to eight sub-modules. Moreover, we have illustrated an experimental scenario about the detection and control of house fires in the context of a smart city.

7.2. Perspective and Future Works

This thesis deals with specifying *Artifact Systems*. We have chosen a declarative approach based on *ECA (Event-Condition-Action) Rules* in order to support process transformation and customization, and perform *Complex Event Processing (CEP)* that is needed in smart process scenarios.

In order to be relevant for modeling of smart processes, we have also included support of *Data Streams* incoming from sensors and the ability to invoke *Ad-hoc Services* that can perform actions on actuators. Moreover, *Complex Event Processing (CEP)* is performed using the proposed *Artifact Rules*. In its current manifestation, *Artifact Rules* support the fundamental event operators; *OR, AND*, and *SEQ*. Future works seek to extend *Artifact*

Rules with additional event operators such as *Not*, *Any*, *Aperiodic*, and *Periodic* operators.

Moreover, we seek to improve the execution strategies of *Artifact Rules* in order to incorporate conditions on two or more unrelated *Artifact Classes*. Furthermore, we intend to investigate techniques for the automatic discovery of *Web Services* in the context of *Artifact Systems*.

The second part of this thesis dealt with modeling of *Artifact Systems* as conceptual models, generating *Artifact Systems* based on these models, and integrating these conceptual models in order to generate unified views.

In order to enable the *Modeling Phase* and perform a holistic integration, we have proposed a conceptual model and notations that are based on modeling all of the components of an *Artifact System* into the same model. Moreover, the chosen model supports the automatic generation of declarative *Artifact Systems* that are valid and fully functional.

Since the integration of process models in this thesis was limited to heterogeneous *CAMs,* future works seek to integrate heterogeneous process models based on different models and notations including *CAMN*, *GSM*, *BPMN*, *ArtiFlow*, and *Database Schemas*. As a prerequisite, we intend to devise semantics and algorithms that achieve a transformation between the proposed *CAM* and the different process models.

From a different research perspective, the artifact integration semantics that we proposed are based on matching, merging and mapping of *CAMs*. The matching of *CAMs* are performed using semi-automatic algorithms and a graphical editor. Future works should take advantages of the progress achieved in the field of *Sematic Integration* in order to employ ontologies and automatically perform matching between *CAMs*.

Finally, we intend to extend the proposed *Artifact* meta-model with capabilities to consider constraints on functional and non-functional properties, consumable resources and feedback loops in the context of connected devices.

7.3. Final Words

In this thesis, we cover a large spectrum of research topics on *Smart Processes* and *Business Artifacts*, *Data Integration* and *Business Processes.* Our theoretical contributions have led to the definition of a new *Artifact* meta-model with its architecture, graphical notation, query language, and integration semantics. From a technological perspective, the prototype

models, integrates, and executes the proposed *Artifact System*. Beyond these points, many interesting research perspectives remain unsolved with open challenges to handle and adapt to the *Internet of Things*.

A

Fire Control Process AQL Queries

In this Appendix, we list the AQL queries generated by the AQL Generator from the CAM of the fire control process illustrated in Figure 6.23.

1. Create Artifact Queries

- Fire Control Artifact:

```
Create Artifact FCA
Attributes (
  FireControlArtifactId : Integer,
  FireDate : Date,
  FireDuration : Integer,
  IsAlarmTurnedOn : Boolean,
  AreWaterEjectorsActivated : Boolean,
  AreHabitatsNotified : Boolean,
  AreWaterPumpsActivated : Boolean,
  House : { Address : String, Surface : Real },
  Habitats : { FullName : String, PhoneNum : Integer },
  FireStationAlert : FireStationAlertArtifact,
  IndoorTemperature : { Time : TimeStamp,
                        Tmp : Integer } As Stream,
  SmokeLevel : { Time : TimeStamp,
                 Lvl :  Integer } As Stream,
  WaterLevel : { Time : TimeStamp,
                  Lvl :  Integer } As Stream
)
States (
  Normal as initial state,
  FireDetected,
  PrimaryProcedurePerformed,
  ClosestFireStationAlerted,
  HabitatsInformed,
  EjectorsDepleted,
  EjectorsRefilled,
  FireExtinguished,
  Archived as final state
)
```

- Fire Station Alert Artifact:

```
Create Artifact FSAA
Attributes (
  FireStationAlertArtifactId : Integer,
  House : { Address : String, Surface : Real },
  FireStationAddress : String,
  IsSuccessfullyAlerted : Boolean
)
States (
  Issued as initial state,
  Located,
  Failed,
  Alerted as final state
)
```

Maroun Abi Assaf
Thèse en informatique / 2017
Institut national des sciences appliquées de Lyon

2. Create Service Queries

- *CreateFCA* Service:

```
Create Service CreateFCA
Input  -
Output FCA
Precondition -
Effect new(FCA)
  And defined(FCA.FireControlArtifactId)
  And defined(FCA.House)
  And defined(FCA.Habitats)
```

- *TurnOnAlarm* Service:

```
Create Service TurnOnAlarm
Input FCA
Output FCA
Precondition notdefined(FCA.AlarmTurnedOn)
Effect defined(FCA.AlarmTurnedOn)
```

- *ActivateWaterEjectors* Service:

```
Create Service ActivateWaterEjectors
Input FCA
Output FCA
Precondition notdefined(FCA.WaterEjectorsActivated)
Effect defined(FCA.WaterEjectorsActivated)
```

- *IssueFireStationAlert* Service:

```
Create Service IssueFireStationAlert
Input FCA
Output FCA, FSAA
Precondition notdefined(FCA.FireStationAlert)
Effect new(FCA. FireStationAlert)
  And defined(FSAA.FireStationAlertArtifactId)
  And defined(FSAA.House)
```

- *NotifyHabitats* Service:

```
Create Service NotifyHabitats
Input FCA
Output FCA
Precondition notdefined(FCA.AreHabitatsNotified)
Effect defined(FCA.AreHabitatsNotified)
```

- *ActivateWaterPumps* Service:

```
Create Service ActivateWaterPumps
Input FCA
Output FCA
Precondition notdefined(FCA.AreWaterPumpsActivated)
Effect defined(FCA.AreWaterPumpsActivated)
```

- *RegisterFireData* Service:

```
Create Service RegisterFireData
Input FCA
```

```
Output FCA
Precondition notdefined(FCA.FireDate)
  And notdefined(FCA.FireDuration)
Effect defined(FCA.FireDate)
  And defined(FCA.FireDuration)
```

- *LocateFireStation* Service:

```
Create Service LocateFireStation
Input FSAA
Output FSAA
Precondition notdefined(FSAA.FireStationAddress)
Effect defined(FSAA.FireStationAddress)
```

- *AlertFireStation* Service:

```
Create Service AlertFireStation
Input FSAA
Output FSAA
Precondition notdefined(FSAA.IsSuccessfullyAlerted)
Effect defined(FSAA.IsSuccessfullyAlerted)
```

- *StreamIndoorTemperature* Service:

```
Create Service StreamIndoorTemperature
Input FCA
Output FCA
Precondition closed(FCA.IndoorTemperature)
Effect opened(FCA.IndoorTemperature)
```

- *StreamSmokeLevel* Service:

```
Create Service StreamSmokeLevel
Input FCA
Output FCA
Precondition closed(FCA.StreamSmokeLevel)
Effect opened(FCA.StreamSmokeLevel)
```

- *StreamWaterLevel* Service:

```
Create Service StreamWaterLevel
Input FCA
Output FCA
Precondition closed(FCA.StreamWaterLevel)
Effect opened(FCA.StreamWaterLevel)
```

3. Create Rule Queries

- Rule 1:

```
Create Rule r1
On CreateFireControlArtifactEvent
Invoke CreateFCA()
```

- Rule 2:

```
Create Rule r2
If state(FCA, Initialized)
  And defined(FireControlArtifactId)
  And defined(House)
  And defined(Habitats)
```

```
Change State Of FCA To Normal
```

- Rule 3:

```
Create Rule r3
If state(FCA, Normal)
  And FCA.IndoorTemperature.Tmp >= 57
  And FCA.SmokeLevel.Lvl >= 3
Change State Of FCA To FireDetected
```

- Rule 4:

```
Create Rule r4
If state(FCA, FireDetected)
  And notdefined(FCA.IsAlarmTurnedOn)
Invoke TurnOnAlarm(FCA)
```

- Rule 5:

```
Create Rule r5
If state(FCA, FireDetected)
  And defined(FCA.IsAlarmTurnedOn)
  And notDefined(FCA.AreWaterEjectorsActivated)
Invoke ActivateWaterEjectors(FCA)
```

- Rule 6:

```
Create Rule r6
If state(FCA, FireDetected)
  And defined(FCA.IsAlarmTurnedOn)
  And defined(FCA.AreWaterEjectorsActivated)
  And FCA.IsAlarmTurnedOn = true
  And FCA.AreWaterEjectorsActivated = true
Change State Of FCA To PrimaryProcedurePerformed
```

- Rule 7:

```
Create Rule r7
If state(FCA, PrimaryProcedurePerformed)
  And notdefined(FCA.FireStationAlert)
Invoke IssueFireStationAlert(FCA)
```

- Rule 8:

```
Create Rule r8
If state(FCA, PrimaryProcedurePerformed)
  And defined(FCA.FireStationAlert)
Change State Of FCA To ClosestFireStationAlerted
```

- Rule 9:

```
Create Rule r9
If state(FCA, ClosestFireStationAlerted)
  And notdefined(FCA.AreHabitatsNotified)
Invoke NotifyHabitats(FCA)
```

- Rule 10:

```
Create Rule r10
```

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

```
If state(FCA, ClosestFireStationAlerted)
  And defined(FCA.AreHabitatsNotified)
  And FCA.AreHabitatsNotified = true
Change State Of FCA To HabitatsInformed
```

- Rule 11:

```
Create Rule r11
If state(FCA, HabitatsInformed)
  And FCA.WaterLevel.Lvl = 0
Change State Of FCA To EjectorsDepleted
```

- Rule 12:

```
Create Rule r12
If state(FCA, EjectorsDepleted)
  And notdefined(FCA.AreWaterPumpsActivated)
Invoke ActivateWaterPumps(FCA)
```

- Rule 13:

```
Create Rule r13
If state(FCA, EjectorsDepleted)
  And defined(FCA.AreWaterPumpsActivated)
  And FCA.AreWaterPumpsActivated = true
Change State Of FCA To EjectorsRefilled
```

- Rule 14:

```
Create Rule r14
If state(FCA, HabitatsInformed)
  And FCA.IndoorTemperature.Tmp < 50
  And FCA.SmokeLevel.Lvl <= 1
Change State Of FCA To FireExtinguished
```

- Rule 15:

```
Create Rule r15
If state(FCA, EjectorsRefilled)
  And FCA.IndoorTemperature.Tmp < 50
  And FCA.SmokeLevel.Lvl <= 1
Change State Of FCA To FireExtinguished
```

- Rule 16:

```
Create Rule r16
If state(FCA, FireExtinguished)
  And notdefined(FCA.FireDate)
  And notdefined(FCA.FireDuration)
Invoke RegisterFireData(FCA)
```

- Rule 17:

```
Create Rule r17
If state(FCA, FireExtinguished)
  And defined(FCA.FireDate)
  And defined(FCA.FireDuration)
Change State Of FCA To Archived
```

- Rule 18:

```
Create Rule r18
If state(FSAA, Initialized)
  And defined(FSAA.FireStationAlertArtifactId)
  And defined(FSAA.House)
Change State Of FSAA To Issued
```

- Rule 19:

```
Create Rule r19
If state(FSAA, Issued)
  And notdefined(FSAA.FireStationAddress)
Invoke LocateFireStation(FSAA)
```

- Rule 20:

```
Create Rule r20
If state(FSAA, Issued)
  And defined(FSAA.FireStationAddress)
Change State Of FSAA To Located
```

- Rule 21:

```
Create Rule r21
If state(FSAA, Located)
  And notdefined(FSAA.IsSuccessfullyAlerted)
Invoke AlertFireStation(FSAA)
```

- Rule 22:

```
Create Rule r22
If state(FSAA, Located)
  And FSAA.IsSuccessfullyAlerted = false
Change State Of FSAA To Failed
```

- Rule 23:

```
Create Rule r23
If state(FSAA, Failed)
  And notDefined(FSSA.FireStationAddress)
Invoke LocateFireStation(FSAA)
```

- Rule 24:

```
Create Rule r24
If state(FSAA, Failed)
  And defined(FSAA.FireStationAddress)
Change State Of FSAA To Located
```

- Rule 25:

```
Create Rule r25
If state(FSAA, Located)
  And defined(FSAA.IsSuccessfullyAlerted)
  And FSAA.IsSuccessfullyAlerted = true
Change State Of FSAA To Alerted
```

B

Résumé Long en Français

1. Introduction

La modélisation des processus centrée sur l'artéfact est une approche de modélisation des processus métiers qui vise à unifier explicitement les données et les processus, et élimine par conséquent la dichotomie qui sépare les communautés de base de données et de gestion des processus métier.

Les processus centrés sur les artéfacts ont été introduits par IBM en 2003 [NiCa03]. L'approche centrée sur les artéfacts, plutôt que la modélisation des schémas relationnels dans les bases de données [AbHV95] ou la modélisation des *Workflows* dans la gestion des processus métier [DTKB03], combine les données et les processus en entités autonomes, appelées artéfacts qui servent de blocs de construction de base à partir desquels les modèles de processus (métier) sont construits.

En général, un processus centré sur les artéfacts appelé *Système d'Artéfact* [BGHL07] est formé de trois composants principaux:

1. Les *Classes d'Artéfacts* incluant des *modèles d'information* pour les données relatives aux artéfacts et des *cycles de vie* basés sur des états décrivant les étapes possibles,
2. Les *Services* qui sont les unités de travail de base manipulant les artéfacts,
3. Les *Règles* (*Métiers*) décrivant comment les *Services* peuvent être invoqués sur les *artéfacts* en suivant les transitions issus de leurs *cycles de vie*.

Un *Système d'Artéfacts* est donc une combinaison de données et de processus formant des entités dynamiques dont la sémantique d'exécution est dictée par les c*ycles de vie* spécifiés.

La Figure 1 illustre l'exemple de *l'*artéfact « commande » (*Order Artifact*). Le *modèle d'information* de la *Classe d'Artéfacts* « commande » possède des attributs pour la représentation des données telles que l'identificateur de la commande, le numéro de produit, la quantité commandée, le numéro du client, l'adresse d'expédition, la disponibilité du produit, la date de récupération, la date de livraison. Le *cycle de vie* comprend des états pour représenter les différentes étapes d'un artéfact de commande parmi lesquels : *Created*, *NotAvailable*, *Available*, *Retrieved*, et *Delivered*. La liste des *Services* agissant sur l'artéfact de commande comprend :

Maroun Abi Assaf
Thèse en Informatique / 2018
Institut National des Sciences Appliquées de Lyon

1. *Create Order*: pour créer une nouvelle instance d'artéfact de commande et enregistrer les informations nécessaires.
2. *Check Availability*: pour vérifier si le produit est disponible en stock en quantité suffisante.
3. *Retrieve Product*: pour récupèrer le produit du stock.
4. *Deliver Product*: pour expédier le produit à l'adresse de livraison.

Les *Règles* sont des règles déclaratives de la forme « événement, condition, action *» (ECA Rules)*. Elles sont représentées par des flèches dans la Figure 1. Les *Règles* sont responsables de l'invocation des *Services* et de la modification de l'état des instances d'artéfact.

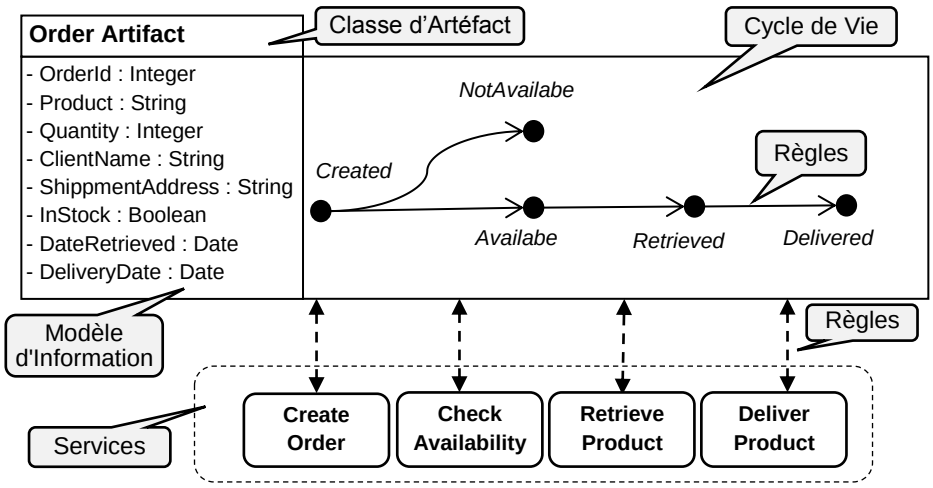


Figure 1. Exemple d'un Système d'Artéfacts

En exploitant les modèles de processus d'une manière sémantique, les processus centrés sur les artéfacts fournissent un cadre intuitif et flexible pour l'exécution et la gestion des processus pilotés par les données. Comme indiqué dans [CoHu09, Hull08], l'approche centrée sur les artéfacts a été appliquée avec succès à la gestion des processus et à la gestion des cas (*Case Handling*) et a montré de nombreux avantages comme: 1) la modularité naturelle, 2) la simplicité de transformation de processus, 3) la disponibilité d'un cadre de différents niveaux d'abstraction, et 4) la comprehension de l'interaction entre les données et les processus d'une manière qui n'est pas supportée par les abstractions plus « traditionnelles ». En conséquence, les utilisateurs finaux peuvent gérer, contrôler et transformer les processus centrés sur les artéfacts

quasiment sans l'intervention de spécialistes et d'experts en technologie de l'information.

Au cours des dernières années, les processus centrés sur les artéfacts ont proliféré à un rythme important grâce à une gamme d'applications, relevant principalement de la finance, la surveillance et l'organisation virtuelle. Une autre application prometteuse des artefacts est l'Internet des objets (*IoT*) où les objets reliés à des réseaux de capteurs et d'actionneurs deviennent « intelligents ». Dans ce contexte, ces objets peuvent être modélisés comme des artéfacts auto-évolutifs rassemblant des flux de données provenant de divers capteurs, détectant des événements complexes et effectuant des actions grâce à des actionneurs. L'Internet des objets, où de nombreux objets connectés sont intégrés avec le protocole Internet afin de créer des services métier de haut niveau, peut être divisé en trois couches, comme illustré dans la Figure 2:

1. *Couche des Objets* : c'est la couche de plus bas niveau qui consiste en un ensemble de capteurs et d'actionneurs interagissant avec leur environnement physique.
2. *Couche Passerelle :* c'est la couche intermédiaire, qui est capable de fournir un point d'accès unifié à la variété d'objets connectés.
3. *Couche des Services Métier* : l'Internet des objets présente des opportunités commerciales considérables, non seulement du point de vue de la fabrication des objets, mais aussi du point de vue des entreprises, à travers les services métier et les applications de haut niveau.

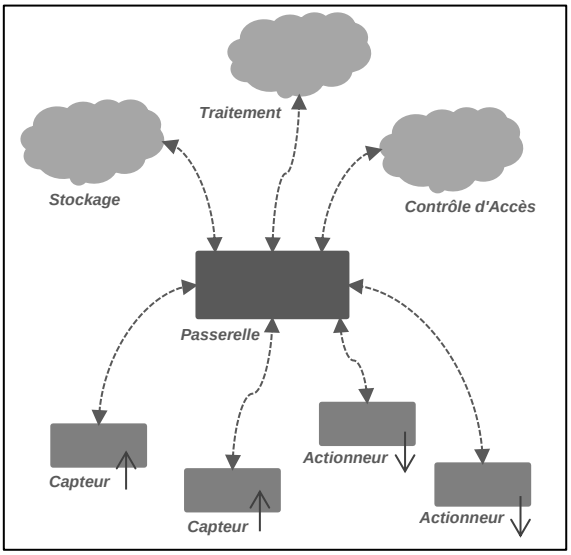


Figure 2. Architecture de l'Internet des objets

L'approche centrée « artéfacts » peut fournir une abstraction au-dessus de l'architecture à trois couches de l'Internet des objets et de ses différents composants. Grâce à l'utilisation des *Classes d'Artéfacts*, *Services* et *Règles d'Artéfact*, un *Système d'Artéfacts* peut représenter tous les composants de bas niveau de l'Internet des objets, notamment les capteurs, les actionneurs, les unités de stockage, le traitement, le contrôle d'accès et les passerelles.

En conséquence, l'approche centrée sur les artéfacts démontre de nombreux avantages, y compris; une modularité naturelle des entités autonomes et un cadre de différents niveaux d'abstraction afin de développer des composants orientés vers les objectifs au lieu de composants orientés vers les fonctions dans le cas des services Web.

2. Description du Problème

De nombreuses entreprises se constituent par fusion et acquisition et, par conséquent, les gestionnaires sont face à des processus et des bases de données hétérogènes fonctionnant de manière similaire ou différente (processus de fabrication, processus de vente) [PaSp00]. La même situation est applicable à l'Internet des objets dans lequel un grand nombre d'objets connectés nécessitent la fusion de données ou la construction d'une vue unifiée pour une gestion plus simple évitant, par exemple, de traiter un grand nombre de tables dans une base de données distribuée.

Comme décrit dans [Lenz02], une approche pratique pour gérer des processus et des bases de données hétérogènes consiste à utiliser une vue unifiée qui centralise l'accès aux informations et aux tâches disponibles de ces processus. Une vue unifiée est un modèle de données virtuel qui peut être utilisé pour superviser, exécuter et interroger des processus et des entités de données distribués et hétérogènes indépendamment de la complexité et des différences de représentations des données et des traitements. Par conséquent, une requête basée sur une vue unifiée est transformée en utilisant des règles de mapping en des requêtes hétérogènes correspondantes aux entités de données ou des processus distribuées. Les ensembles de résultats des requêtes hétérogènes sont ensuite transformés et fusionnés en utilisant d'autres règles de mapping en un ensemble de résultats compatible avec la vue unifiée. Les avantages de l'utilisation de ce type de vues sont; i) la gestion d'un grand nombre d'entités à une grande échelle, ii) la facilitation de l'évaluation et l'analyse des données et de leurs comportements, et iii) le support d'un point de l'accès centralisé pour les administrateurs et les utilisateurs occasionnels. La Figure 3 illustre les mécanismes de l'intégration.

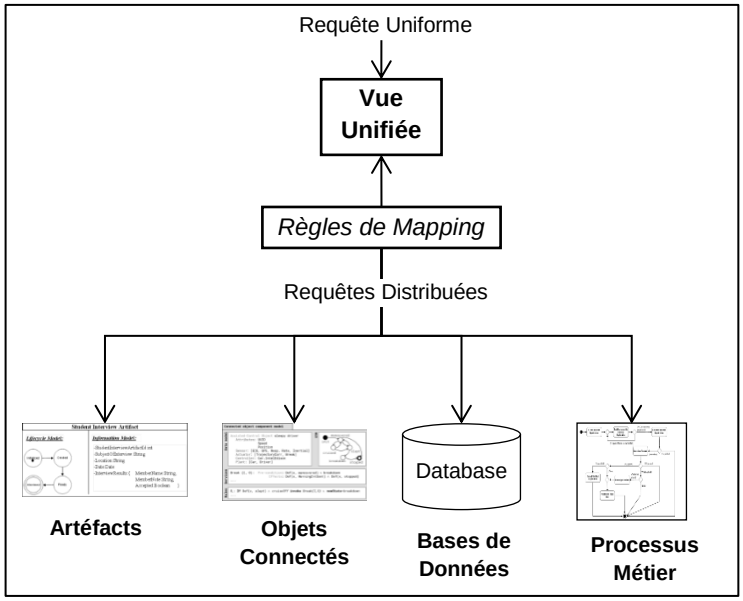


Figure 3. Mécanisme d'intégration

Puisque les processus centrés sur les artéfacts ont émergé comme un nouveau paradigme de modélisation et ont fourni des applications intéressantes dans le contexte de l'Internet des objets pour modéliser des objets connectés basés sur des artefacts, **le travail de cette thèse se propose de prolonger ce paradigme en se concentrant sur l'intégration de processus hétérogènes centrés sur les artéfacts**. L'intégration de processus centrés sur les artéfacts est un problème important en raison de la complexité de mapping de deux ou plusieurs artéfacts au niveau de leurs composants (les *Modèles d'Information*, les *Cycles de Vie*, les *Services* et les *Règles*). Par conséquent, les solutions et techniques traditionnelles d'intégration de données et de fusion de processus telles que [ChTr12, KuYY14, PaSp00] ne permettent pas de traiter la complexité de l'intégration de processus centrée sur les artéfacts.

De plus, étant donné un objet connecté basé sur des artéfacts, différentes variantes peuvent exister pour gérer différentes utilisations dans des différents domaines d'application. Des variations peuvent survenir dans le *Modèle d'Information* (données semi-structurées), les *Etats* (nouveaux états intermédiaires), les *Services* (nouveaux services ou mêmes services avec différentes signatures...) et les *Règles* (spécifiquement adaptées à un domaine d'application).

En conséquence, les variantes d'artéfacts entraînent des processus hétérogènes centrés sur les artéfacts. L'intégration de ces processus issus de différentes sources devient un défi majeur lorsque nous devons fournir des

vues unifiées pour gérer, interroger et exécuter un très grand nombre d'artéfacts répartis sur l'Internet des objets.

Puisque les processus centrés sur les artéfacts combinent trois composants principaux; *Classes d'Artéfacts*, *Services* et *Règles*, le problème d'intégration de deux artéfacts ou plus nécessite simultanément l'intégration de leurs *composants respectifs*. Cependant, cette classe de problème a été largement étudiée dans des disciplines telles que l'intégration de données, les bases de données, et la fusion de processus métier, mais la complexité et la richesse des structures d'artéfacts nécessitent des approches d'intégration nouvelles. Les défis liés à l'intégration de processus centrée sur les artéfacts peuvent ainsi être classés en trois niveaux différents :

1) **Le niveau de la sémantique d'intégration :** l'intégration d'artéfacts nécessite la définition d'une nouvelle sémantique d'intégration. Cette nouvelle sémantique doit prendre en charge différents types de relations (*uniques*, *équivalentes* et *compositions*) entre les éléments des différents composants des *Systèmes d'Artéfacts*.
2) **Le niveau de la modélisation conceptuelle d'Artéfacts** : afin de définir une sémantique d'intégration efficace, les *Systèmes d'Artéfacts* doivent être représentés en utilisant des modèles conceptuels qui capturent graphiquement leurs trois composantes. Ces modèles conceptuels doivent non seulement être simples, intuitifs et holistiques, mais peuvent également être utilisés pour générer des *Systèmes d'Artéfacts* fonctionnels.
3) **Le niveau du langage spécifique aux Artéfacts** : afin de créer, exécuter, manipuler et interroger efficacement les *Systèmes d'Artéfacts*, un langage spécifique aux artéfacts doit être défini pour cibler les artéfacts et profiter pleinement de leur nature sémantique. De plus, ce langage doit être utilisé pour interroger les vues unifiées générées.
4) **Le niveau de l'Artéfact Etendu** : ayant été uniquement appliqués aux processus métier traditionnels, les artéfacts doivent être étendus avec des capacités de flux de données afin de soutenir les processus émergents de l'Internet des objets.

3. Contributions

Dans cette thèse, nous nous focalisons sur le problème de l'intégration des processus centrés sur les artéfacts dans le contexte de l'Internet des objets à travers la représentation des *Systèmes d'Artéfacts* en utilisant des modèles conceptuels et la fusion de modèles selon les relations de correspondance entre leurs différents éléments.

Nous proposons une méthodologie ou cadre d'intégration des processus centré sur les artéfacts basé sur quatre phases principales comme illustré sur la Figure 4.

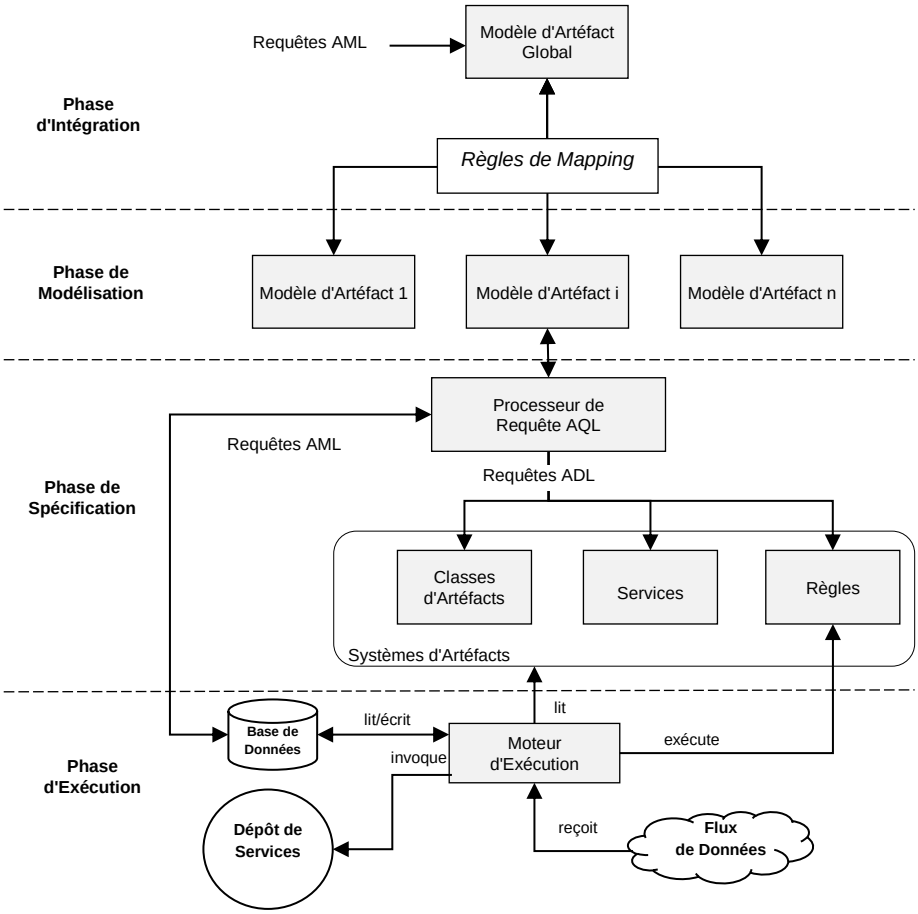


Figure 4. Cadre d'intégration de processus centré sur les artéfacts

3.1 Phase de Modélisation

Dans la phase de modélisation, nous modélisons les *Systèmes d'Artéfacts* en utilisant des modèles conceptuels que nous appelons *Conceptual Artifact Models (CAM)*. Nous proposons une notation graphique minimaliste, *Conceptual Artifact Modeling Notation (CAMN),* que nous utilisons pour modéliser les *CAMs*. *CAMN* est composée de six primitives de modélisation comme indiqué dans le Tableau 1 : *Repository*, *Task*, *Flow Connector*, *Data Attribute List*, *Event*, et *Condition*.

Tableau 1. Conceptual Artifact Modeling Notation (CAMN)

Primitive de Modélisation	Notation Graphique	Description
Task	Task	Les tâches sont des unités de travail opérant sur les artéfacts.
Repository	Repository	Les dépôts sont des emplacements de stockage basés sur les états des artéfacts.
Read/write Flow Connector	Artifact →	Les connecteurs de flux en mode lecture/écriture transfèrent les artéfacts entre les tâches et les dépôts.
Read-only Flow Connector	Artifact - - - →	Les connecteurs de flux en mode lecture seule accèdent au contenu d'artéfact d'un dépôt.
Data Attribute List	$Att_1 : type_1$; $Att_2 : type_2$; ... $Att_n : type_n$;	Liste des paires d'*attribut-type* manipulées et attachées à une tâche.
Condition	Condition	Conditions attachées aux connecteurs de flux.
Event	Event	Événement associé aux connecteurs de flux.

En utilisant les six primitives de *CAMN*, un *CAM* peut être construit comme illustré sur la figure 5 illustrant une partie d'un processus de contrôle d'incendie. Les *CAM*s ne sont pas seulement des conteneurs de toutes les informations requises pour générer des *Systèmes d'Artéfacts* fonctionnels, mais ils constituent également la base de l'intégration et de la génération des vues unifiées.

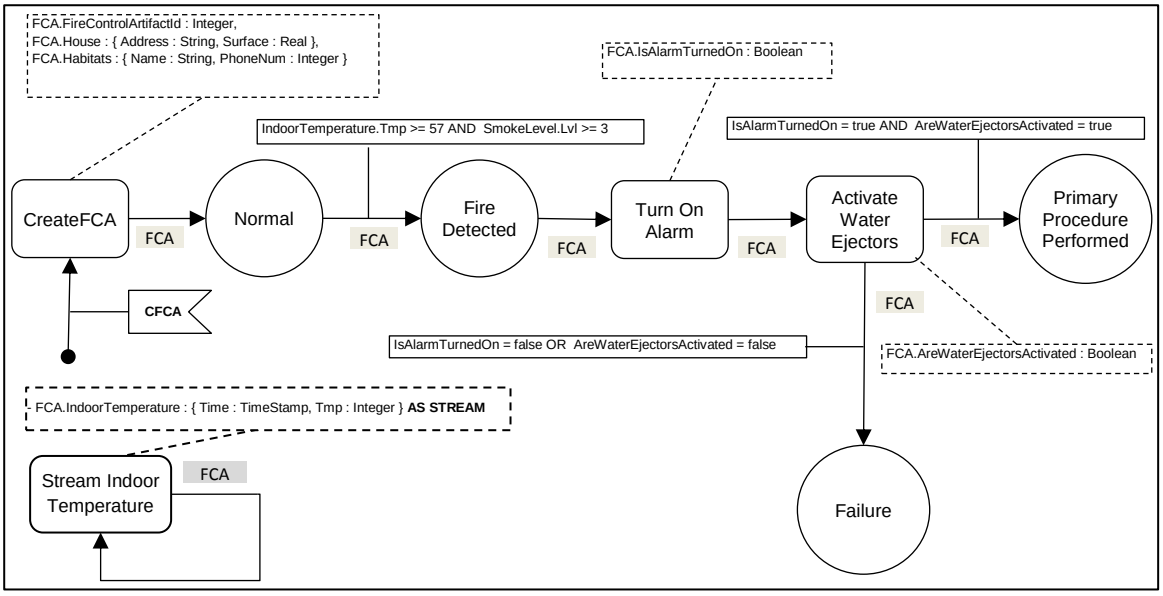


Figure 5. Partie du modèle d'artéfact conceptuel d'un contrôle d'incendie

3.2 Phase de Spécification

Dans la phase de spécification, nous définissons et générons des *Systèmes d'Artéfacts* à partir des *CAMs* modélisés dans la phase précédente. Nous proposons l'*Artifact Query Language (AQL)*, un langage spécifique aux artéfacts que nous utilisons pour exprimer et implémenter les *Systèmes d'Artéfacts*. *AQL* est un langage déclaratif de haut niveau similaire à *SQL* qui cible spécifiquement les artéfacts et profite pleinement de leur nature sémantique. *AQL* est composé de deux parties : *Artifact Definition Language (ADL)* et *Artifact Manipulation Language (AML)*. *ADL* contient trois déclarations *Create Artifact*, *Create Service*, et *Create Rule* pour définir respectivement les *Classes d'Artéfacts*, les *Services* et les *Règles d'Artéfacts*. *AML* contient six déclarations *New*, *Update*, *Insert Into*, *Remove From*, *Delete,* and *Retrieve* pour instancier, manipuler et interroger les instances d'artéfacts. De plus, *AQL* prend en charge les capacités de flux de données et de requêtes continues et permet le traitement des événements complexes (*CEP*) sur les flux de données grâce à l'utilisation de *Règles d'Artéfact*. Le Tableau 2 énumère les neuf déclarations d'*AQL* et leur syntaxe.

La Figure 6 illustre des exemples de requêtes pour *ADL*. La requête 1 définit un *Classe d'Artéfact, FireControlArtifact (FCA)* responsable du contrôle des incendies. Les requêtes 2 et 3 définissent des *Règles d'Artéfact*. Enfin, les requêtes 4 et 5 définissent des *Services*.

La Figure 7 illustre des exemples de requêtes pour *AML*. La requête 1 instancie un *FCA*. Les requêtes 2 et 3 récupèrent des données à partir d'artéfacts en fonction de la condition et la fenêtre sur les flux de données. Les requêtes 4 à 7 manipulent des instances d'artéfacts.

3.3 Phase d'Intégration

Dans la phase d'intégration, nous intégrons plusieurs *CAM* locaux afin de générer un *CAM* global utilisé comme vue unifiée. Nous proposons et définissons un *Système d'Intégration d'Artéfacts* et une sémantique d'intégration basée sur un protocole qui enchaine trois sous-phases : *Correspondance*, *Fusion*, et *Mapping*.

Tableau 2. Déclarations d'AQL

Déclaration	Syntaxe
Create Artifact	**Create Artifact** <name> **Attributes** <list of attributes> **States** <list of states>
Create Service	**Create Service** <name> **Input** <list of artifacts> **Output** <list of artifacts> **Precondition** <expression> **Effect** <expression>
Create Rule	**Create Rule** <name> (**On** <event> | **On** <event> **If** <condition> | **If** <condition>) (**Change State Of** <artifact> **To** <state> | **Invoke** <list of services>)
New	**New** <artifact> <list of simple attributes> **Values** <list values> {<complex attribute> **Include** <list of tuples>} {<reference attribute> **Having** <condition>} {<stream attribute> **Using** <stream service>} [**Set state to** <state>]
Retrieve	**Retrieve** <list of attributes> **From** <list of artifacts> [**Where** <condition>] [**Within** <range>]
Update	**Update** <artifact> **Set** <list of assignments> [**Where** <condition>]
Insert Into	**Insert** <attribute> **Into** <artifact> <list of tuples> [**Where** <condition>]
Remove From	**Remove** <attribute> **From** <artifact> [**Where** <condition>]
Delete	**Delete** <artifact> [**Where** <condition>]

```
//Requête 1
Create Artifact FCA
Attributes (
  FireControlArtifactId : Integer,
  FireDate : Date,
  FireDuration : Integer,
  House : { Address : String,
            Surface : Real } As One,
  Habitats : { Name : String,
               PhoneNum : Integer } As Many,
  IndoorTemperature : { Tmp : Integer,
                        Time : TimeStamp } As Stream,
  SmokeLevel : { Lvl :  Integer,
                 Time : TimeStamp } As Stream,
  WaterEjectorsActivated : Boolean,
  AlarmTurnedOn : Boolean
)
States (
  Normal As Initial State,
  FireDetected,
  PrimaryProcedurePerformed,
  FireExtinguished As Final State
)

//Requête 2
Create Rule r1
If state(FCA, FireDetected)
Invoke  TurnOnAlarm(FCA),
        ActivateWaterEjectors(FCA)

//Requête 3
Create Rule r2
If state(FCA, Normal)
  And FCA.IndoorTemperature.Tmp > 57
  And FCA.SmokeLevel.Lvl >= 3
Change State Of FCA To FireDetected

//Requête 4
Create Service StreamIndoorTmp
Input FCA
Output FCA
Precondition closed(FCA.IndoorTemperature)
  And defined(FCA.FireControlArtifactId)
  And defined(FCA.House)
Effect opened(FCA.IndoorTemperature)

//Requête 5
Create Service ActivateWaterEjectors
Input FCA
Output FCA
Precondition
  FCA.WaterEjectorsActivated = false
  And defined(FCA.FireControlArtifactId)
  And defined(FCA.House)
Effect FCA.WaterEjectorsActivated = true
```

Figure 6. Exemples de requêtes ADL

```
//Requête 1
New FCA
(FireControlArtifactId)Values(100235)
House Include ("20 Av. Albert Einstein", 64)
Habitats Include { ("John", 00330675839457),
                   ("Sam", 00330625374883)}
IndoorTemperature Using StreamIndoorTemperature(this)
SmokeLevel Using StreamSmokeLevel(this)
Set State To Normal

//Requête 2
Retrieve * From FCA
Where state(FCA, FireDetected)
  And FCA.IndoorTemperature.Tmp > 100
Within 10 Seconds

//Requête 3
Retrieve IndoorTemperature From FCA
Where state(FCA, Normal)
Within 3 Seconds

//Requête 4
Update FCA
Set Habitats.PhoneNum = 0033763423758
Where Habitats.Name = "John"
  And FCA.FireControlArtifactId = 100325

//Requête 5
Insert Habitats Into FCA
{ ("Sebastien", 0033823459876),
  ("Nicole", 003357643214) }
Where FCA.FireControlArtifactId = 100325

//Requête 6
Remove Habitats From FCA
Where FCA.FireControlArtifactId = 100325
  And Habitats.Name = "John"

//Requête 7
Delete FCA
Where FCA.FireControlArtifactId = 100325
```

Figure 7. Exemples de requêtes AML

Dans la sous-phase de correspondance, trois types de relations de correspondance (*unique*, *équivalent* et *composition*) sont identifiés entre les différents éléments des *CAMs*. Tout d'abord, les correspondances entre les artéfacts, les tâches et les dépôts sont identifiées manuellement à l'aide d'une interface utilisateur graphique, comme illustré sur la Figure 8. Ensuite, les correspondances entre les attributs de données sont également identifiées manuellement en utilisant une interface utilisateur graphique et en spécifiant des expressions de correspondance, comme illustré dans le Tableau 3.

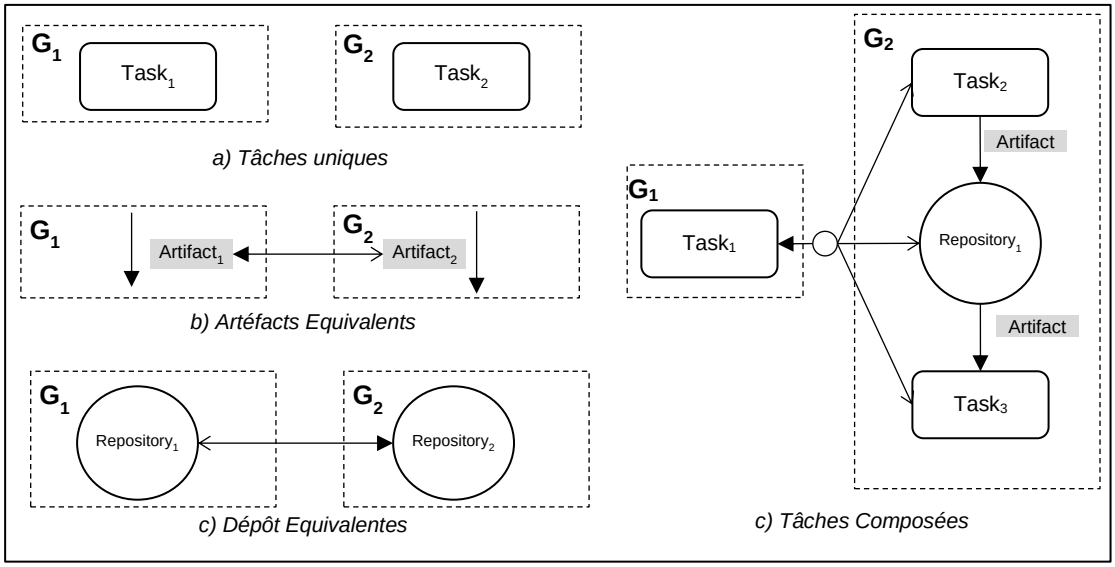


Figure 8. Exemples de correspondance entre les artéfacts, les tâches et les dépôts

Tableau 2. Exemples de correspondance entre les attributs de données

Relation de Correspondance	Notation Graphique	Expression de Correspondance
Attributs de Données Uniques	G1 - Fax G2 - Telephone	*NA*
Attributs de Données Equivalents	G1 - Price ⟷ G2 - Cost	*Cost = Price * 100*
Attributs de Données Composés	G1 - Cost ⟵ G2 - Price, - Tax	*Cost = Price * (1 + Tax/100)*

Dans la sous-phase de fusion, un *CAM* global est généré en fusionnant les *CAM* locaux en fonction des correspondances identifiées. La sémantique de la fusion est définie à l'aide des règles algorithmiques qui rebranchent et reconnectent les connecteurs de flux. Ces règles sont divisées en cinq ensembles incrémentiels qui décrivent successivement la génération d'artéfacts, des dépôts, des tâches, des listes d'attributs de données et des connecteurs de flux, y compris les événements et les conditions. Les règles algorithmiques ont la forme générale suivante:

"If correspondence ***then update I and generate*** *integratedElement* ***in*** G_I*"*

Si la relation de correspondance définie par "*correspondance*" est valide, une fonction d'intégration $\boldsymbol{I}$ est mise à jour et un élément défini par "*integratedElement*" est généré dans le *CAM* global G_I.

Finalement, dans la sous-phase de mapping, des règles de mapping sont spécifiées afin de traduire les requêtes *AML* et les données entre les *CAM* globaux et locaux en fonction du type d'éléments de *CAM*. La sémantique de mapping est définie à l'aide de cinq fonctions de mapping qui réalisent des correspondances des artéfacts, des dépôts, des tâches, des attributs de données et des connecteurs de flux dans le *CAM* global avec éléments respectifs dans les *CAM* locaux.

3.4 Phase d'Exécution et Prototype

Dans la phase d'exécution, nous exécutons les *Systèmes d'Artéfacts* en utilisant un moteur d'exécution basé sur la traduction des requêtes *AQL* en requêtes sémantiques. Les requêtes sémantiques sont ensuite exécutées sur un système de gestion de base de données afin d'effectuer des opérations relationnelles. Le moteur d'exécution est également responsable de l'appel des services.

Nous avons également développé un prototype mettant en œuvre le cadre complet d'intégration des artéfacts, y compris ses quatre phases: *Intégration*, *Modélisation*, *Spécification* et *Exécution*. Le prototype met en œuvre le cadre d'intégration des artéfacts en utilisant une architecture modulaire. Chaque module est responsable de la mise en œuvre d'un aspect du cadre d'intégration des artéfacts. Le prototype est basé sur plusieurs langues, y compris; HTML5 [00a], XML [00b], Java [00c], Xtend [00d] et JavaScript [00e], en plus de plusieurs cadres de programmation et systèmes, y compris; *Eclipse Rich Client Platform (Eclipse RCP)* [00f], *Xtext Framework for the development of Domain Specific Languages (DSL)* [00g], *Java Architecture for XML Binding (JAXB) Framework* [00i], *JointJS Javascript Diagramming Librar*y [00j], et *Apache Derby Database Management System* [00h].

Le prototype est composé de six modules principaux; *CAM Modeler*, *CAM Matcher*, *CAM Merger*, *CAM Mapper*, *AQL Generator*, et *AQL Processor* comme illustré sur la Figure 9. Les différents modules communiquent entre eux à l'aide de messages d'entrée et de sortie.

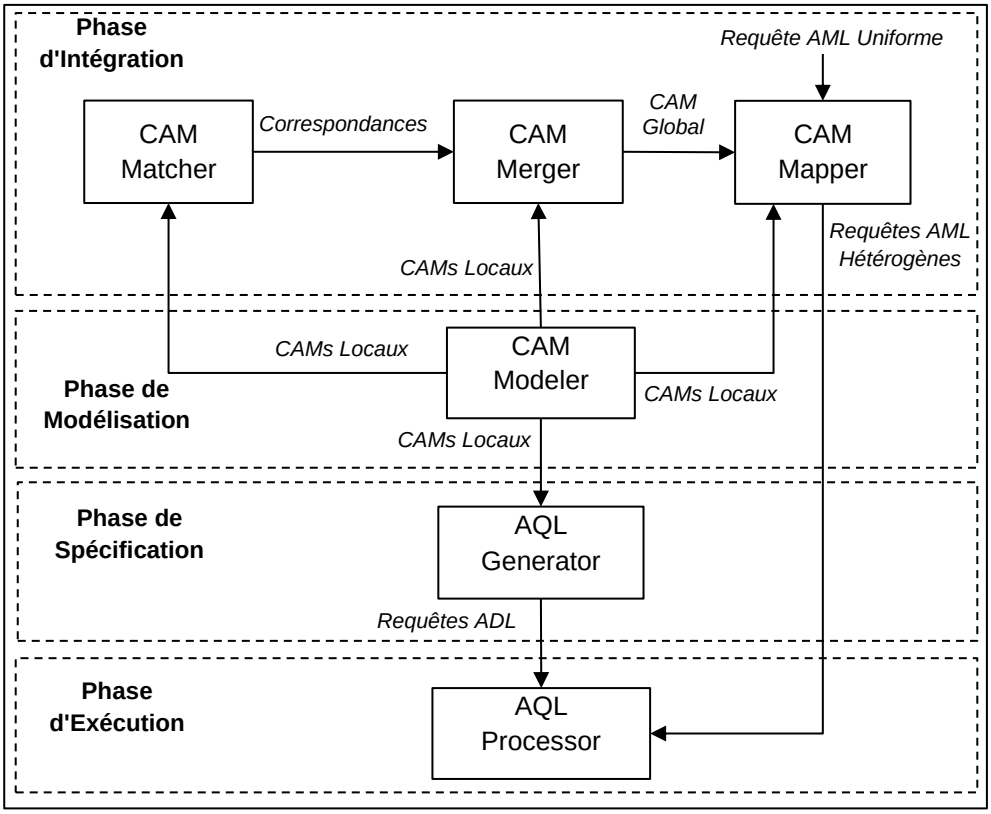


Figure 9. Modules principaux du prototype

4. Conclusion

Dans cette thèse, nous avons abordé le problème de l'intégration des processus métier hétérogènes centrés sur les artéfacts. Nous avons proposé un cadre d'intégration d'artéfacts complet qui traite non seulement l'intégration des processus, mais aussi leur modélisation, interrogation, spécification et exécution. Le cadre d'intégration des artéfacts proposé est basé sur quatre phases principales : i) *Modélisation*, ii) *Spécification*, iii) *Intégration*, et iv) *Exécution*.

Dans la phase de modélisation, nous modélisons des *Systèmes d'Artéfacts* en utilisant des modèles conceptuels que nous appelons des *Conceptual Artifact Models (CAM)*. Nous avons proposé la *Conceptual Artifact Modeling Notation (CAMN)*, une notation graphique minimaliste qui est utilisée pour modéliser graphiquement Les *Systèmes d'Artéfacts* sans écrire des requêtes *AQL* complexes et sujettes aux erreurs.

Dans la phase de spécification, nous générons des *Systèmes d'Artéfacts* à partir de *CAM* qui sont modélisés dans la phase précédente. Nous avons proposé un modèle formel pour les *Systèmes d'Artéfacts* qui est spécifiquement adapté pour modéliser des processus intelligents basés sur l'Internet des objets *(IoT)* d'une manière simple et intuitive. Le modèle formel prend en charge les flux de données, les capteurs et les actionneurs, et permet la définition, la manipulation et l'interrogation de *Systèmes d'Artéfacts* à un niveau élevé sans traiter le modèle sous-jacent des relations. Nous avons également proposé *l'Artifact Query Language (AQL)* utilisé pour définir de manière déclarative les *Systèmes d'Artéfacts* et manipuler leurs instances d'artéfacts.

Dans la phase d'intégration, nous intégrons plusieurs *CAM* locales afin de générer une *CAM* globale qui agit comme une vue unifiée. Puisque les *CAMs* combinent les aspects de processus et données dans le même modèle, nous avons proposé un *Système d'Intégration d'Artéfacts* basé sur une sémantique d'intégration spécialisée pour l'intégration de *CAM*s hétérogènes. La sémantique d'intégration proposée combine des mécanismes d'intégration issus à la fois de l'intégration de données et de la fusion de processus métier, en plus de mécanismes d'intégration spécifiques aux artéfacts.

Dans la phase d'exécution, nous exécutons les *Systèmes d'Artéfacts* en utilisant un moteur d'exécution basé sur la traduction des requêtes *AQL* en requêtes sémantiques. Finalement, afin de valider nos différentes contributions, nous avons également développé un prototype complet qui implémente les différentes phases du cadre d'intégration des processus centré sur les artéfacts.

Les travaux futurs cherchent à étendre les règles d'artéfact avec des opérateurs d'événements tels que les opérateurs : *Not*, *Any*, *Aperiodic* et *Periodic*. De plus, nous cherchons à améliorer les stratégies d'exécution des *Règles d'Artéfact* afin d'incorporer des conditions sur deux ou plusieurs *Classes d'Artéfacts* non apparentées. De plus, nous avons l'intention d'étudier des techniques pour la découverte automatique de services *Web* dans le contexte des *Systèmes d'Artéfacts*. Puisque l'intégration de modèles de processus dans cette thèse se limitait à des *CAM* hétérogènes, des travaux futurs cherchent à intégrer des modèles de processus hétérogènes basés sur différents modèles et notations, notamment *CAMN*, *GSM*, *BPMN*, *ArtiFlow* et schémas de base de données (*Database Schemas)*. De plus, nous avons l'intention de profiter de progrès réalisés dans le domaine de l'intégration sémantique afin d'utiliser des ontologies qui découvrent automatiquement les relations de correspondance entre les *CAMs*.

Bibliography

[00a] *Odysseus, Data Stream Management System*. URL https://odysseus.informatik.uni-oldenburg.de, *last visited:24/05/2018*

[00b] *Hyper Text Markup Language 5 (HTML5)*. URL https://www.w3.org/TR/html/xhtml.html, *last visited:24/05/2018*

[00c] *Extensible Markup Language (XML)*. URL https://www.w3.org/XML/, *last visited:24/05/2018*

[00d] *Java Programming Language*. URL https://www.oracle.com/java/, *last visited:24/05/2018*

[00e] *Xtend Programming Language*. URL http://www.eclipse.org/xtend/, *last visited:24/05/2018*

[00f] *JavaScript (JS) Programming Language*. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript, *last visited:24/05/2018*

[00g] *Eclipse Rich Client Platform*. URL http://wiki.eclipse.org/Rich_Client_Platform, *last visited:24/05/2018*

[00h] *Xtext Framework for the development of Domain Specific Languages (DSL)*. URL https://eclipse.org/Xtext/, *last visited:24/05/2018*

[00i] *Java Architecture for XML Binding (JAXB) Framework*. URL http://jaxb.java.net/, *last visited:24/05/2018*

[00j] *JointJS Javascript Diagramming Library*. URL https://www.jointjs.com/opensource, *last visited:24/05/2018*

[00k] *Apache Derby Database Management System*. URL https://db.apache.org/derby/, *last visited:24/05/2018*

[ABBC16] Arasu, Arvind ; Babcock, Brian ; Babu, Shivnath ; Cieslewicz, John ; Datar, Mayur ; Ito, Keith ; Motwani, Rajeev ; Srivastava, Utkarsh ; u. a.: Stream: The stanford data stream management system. In: *Data Stream Management* : Springer, pp. 317–336, 2016

[ABGM09] Abiteboul, Serge ; Bourhis, Pierre ; Galland, Alban ; Marinoiu, Bogdan: The AXML artifact model. In: *16th International Symposium on Temporal Representation and Reasoning* : IEEE, pp. 11–17, 2009

[AbHV95] Abiteboul, Serge ; Hull, Richard ; Vianu, Victor: *Foundations of databases: the logical level* : Addison-Wesley Longman Publishing Co., Inc., 1995

[ACÇC03] Abadi, Daniel J ; Carney, Don ; Çetintemel, Ugur ; Cherniack, Mitch ; Convey, Christian ; Lee, Sangdon ; Stonebraker, Michael ; Tatbul, Nesime ; u. a.: Aurora: a new model and architecture for data stream management. In: *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, 2003

[ADMR05] Aumueller, David ; Do, Hong-Hai ; Massmann, Sabine ; Rahm, Erhard: Schema and ontology matching with COMA++. In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* : Acm, pp. 906–908, 2005

[AKKK03] Arenas, Marcelo ; Kantere, Vasiliki ; Kementsietsidis, Anastasios ; Kiringa, Iluju ; Miller, Renée J ; Mylopoulos, John: The hyperion project: from data integration to data coordination. In: *ACM SIGMOD Record*, vol. 32, no. 3, pp. 53–58, 2003

[AnVa04] Antoniou, Grigoris ; Van Harmelen, Frank: Web ontology language: Owl. In: *Handbook on ontologies* : Springer, pp. 67–92, 2004

[ArBW06] Arasu, Arvind ; Babu, Shivnath ; Widom, Jennifer: The CQL continuous query language: semantic foundations and query execution. In: *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006

[BGHL07] Bhattacharya, Kamal ; Gerede, Cagdas ; Hull, Richard ; Liu, Rong ; Su, Jianwen: Towards formal analysis of artifact-centric business process models. In: *International Conference on Business Process Management* : Springer, pp. 288–304, 2007

[BhHS09] Bhattacharya, Kamal ; Hull, Richard ; Su, Jianwen: A data-centric design methodology for business processes. In: *Handbook of Research on Business Process Modeling* : IGI Global, pp. 503–531, 2009

[BoVe11] Bonifati, Angela ; Velegrakis, Yannis: Schema matching and mapping: from usage to evaluation. In: *Proceedings of the 14th International Conference on Extending Database Technology* : ACM, pp. 527–529, 2011

[CaGo12] Cabot, Jordi ; Gogolla, Martin: Object constraint language (OCL): a definitive guide. In: *Formal methods for model-driven engineering* : Springer, pp. 58–90, 2012

[CCDF03] Chandrasekaran, Sirish ; Cooper, Owen ; Deshpande, Amol ; Franklin, Michael J ; Hellerstein, Joseph M ; Hong, Wei ; Krishnamurthy, Sailesh ; Madden, Samuel R ; u. a.: TelegraphCQ: continuous dataflow processing. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* : ACM, pp. 668–668, 2003

[CDHP08] Cohn, David ; Dhoolia, Pankaj ; Heath Iii, Fenno ; Pinel, Florian ; Vergo, John: Siena: From powerpoint to web app in 5 minutes. In: *International Conference on Service-Oriented Computing* : Springer, pp. 722–723, 2008

[CDLR04] Calvanese, Diego ; De Giacomo, Giuseppe ; Lenzerini, Maurizio ; Rosati, Riccardo: Logical foundations of peer-to-peer data integration. In: *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* : ACM, pp. 241–251, 2004

[ChPe17] Cheatham, Michelle ; Pesquita, Catia: Semantic Data Integration. In: *Handbook of Big Data Technologies* : Springer, pp. 263–305, 2017

[ChTr12] Chinosi, Michele ; Trombetta, Alberto: BPMN: An introduction to the standard. In: *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 124–134, 2012

[CKAK94] Chakravarthy, Sharma ; Krishnaprasad, Vidhya ; Anwar, Eman ; Kim, Seung-Kyum: Composite events for active databases: Semantics, contexts and detection. In: *VLDB*. vol. 94, pp. 606–617, 1994

[CoHu09] Cohn, David ; Hull, Richard: Business artifacts: A data-centric approach to modeling business operations and processes. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 32, no. 3, pp. 3–9, 2009

[CrXi05] Cruz, Isabel F ; Xiao, Huiyong: The role of ontologies in data integration. In: *Engineering intelligent systems for electrical engineering and communications*, vol. 13, no. 4, pp. 245, 2005

[DaHV13] Damaggio, Elio ; Hull, Richard ; Vaculín, Roman: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard–Stage–Milestone lifecycles. In: *Information Systems*, vol. 38, no. 4, pp. 561–584, 2013

[DGPR07] Demers, Alan J ; Gehrke, Johannes ; Panda, Biswanath ; Riedewald, Mirek ; Sharma, Varun ; White, Walker M: Cayuga: A General Purpose Event Monitoring System. In: *CIDR*. vol. 7, pp. 412–422, 2007

[DHPV09] Deutsch, Alin ; Hull, Richard ; Patrizi, Fabio ; Vianu, Victor: Automatic verification of data-centric business processes. In: *Proceedings of the 12th International Conference on Database Theory* : ACM, pp. 252–267, 2009

[DLLP18] De Giacomo, Giuseppe ; Lembo, Domenico ; Lenzerini, Maurizio ; Poggi, Antonella ; Rosati, Riccardo: Using ontologies for semantic data integration. In: *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years* : Springer, pp. 187–202, 2018

[DMVF00] Decker, Stefan ; Melnik, Sergey ; Van Harmelen, Frank ; Fensel, Dieter ; Klein, Michel ; Broekstra, Jeen ; Erdmann, Michael ; Horrocks, Ian: The

semantic web: The roles of XML and RDF. In: *IEEE Internet computing*, vol. 4, no. 5, pp. 63–73, 2000

[Do06] Do, Hong-Hai: *Schema matching and mapping-based data integration*, 2006

[DoMR03] Do, Hong-Hai ; Melnik, Sergey ; Rahm, Erhard: Comparison of schema matching evaluations. In: *Web, Web-Services, and Database Systems*, pp. 221–237, 2003

[DuTe01] Dumas, Marlon ; Ter Hofstede, Arthur HM: UML activity diagrams as a workflow specification language. In: *International Conference on the Unified Modeling Language* : Springer, pp. 76–90, 2001

[EQST12] Estanol, Montserrat ; Queralt, Anna ; Sancho, Maria Ribera ; Teniente, Ernest: Artifact-centric business process models in UML. In: *International Conference on Business Process Management* : Springer, pp. 292–303, 2012

[EsWi03] Eshuis, Rik ; Wieringa, Roel: Comparing Petri net and activity diagram variants for workflow modelling-a quest for reactive Petri nets. In: *Petri Net Technology for Communication-Based Systems*, vol. 2472, pp. 321–351, 2003

[FoFo09] Fortis, Alexandra ; Fortis, Florin: Workflow patterns in process modeling. In: *arXiv preprint arXiv:0903.0053*, 2009

[Fort09] Fortis, Alexandra: Business Process Modeling Notation-An Overview. In: *arXiv preprint arXiv:0904.3633*, 2009

[GeBS07] Gerede, Cagdas E ; Bhattacharya, Kamal ; Su, Jianwen: Static analysis of business artifact-centric operational models. In: *International Conference on Service-Oriented Computing and Applications* : IEEE, pp. 133–140, 2007

[GeHS95] Georgakopoulos, Diimitrios ; Hornick, Mark ; Sheth, Amit: An overview of workflow management: From process modeling to workflow automation infrastructure. In: *Distributed and parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995

[GeSu07] Gerede, Cagdas E ; Su, Jianwen: Specification and verification of artifact behaviors in business process models. In: *International Conference on Service-Oriented Computing* : Springer, pp. 181–192, 2007

[GoGL12] Gonzalez, Pavel ; Griesmayer, Andreas ; Lomuscio, Alessio: Verifying GSM-based business artifacts. In: *19th International Conference on Web Services* : IEEE, pp. 25–32, 2012

[GuMO95] Gupta, Ashish ; Mumick, Inderpal Singh ; others: Maintenance of materialized views: Problems, techniques, and applications. In: *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 3–18, 1995

[Hale01] Halevy, Alon Y: Answering queries using views: A survey. In: *The VLDB Journal*, vol. 10, no. 4, pp. 270–294, 2001

[HaOt07] Hai, Do ; others: *Schema matching and mapping-based data integration: Architecture, approaches and evaluation* : VDM Verlag, 2007

[HBGV13] Heath III, Fenno Terry ; Boaz, David ; Gupta, Manmohan ; Vaculín, Roman ; Sun, Yutian ; Hull, Richard ; Limonad, Lior: Barcelona: A design and runtime environment for declarative artifact-centric BPM. In: *International Conference on Service-Oriented Computing* : Springer, pp. 705–709, 2013

[HCDD11] Hariri, Babak Bagheri ; Calvanese, Diego ; De Giacomo, Giuseppe ; De Masellis, Riccardo ; Felli, Paolo: Foundations of Relational Artifacts Verification. In: *BPM*. vol. 6896 : Springer, pp. 379–395, 2011

[HDDF11a] Hull, Richard ; Damaggio, Elio ; De Masellis, Riccardo ; Fournier, Fabiana ; Gupta, Manmohan ; Fenno Terry Heath, III ; Hobson, Stacy ; Linehan, Mark ; u. a.: A Formal Introduction to Business Artifacts with Guard-Stage-Milestone Lifecycles 2011

[HDDF11b] Hull, Richard ; Damaggio, Elio ; De Masellis, Riccardo ; Fournier, Fabiana ; Gupta, Manmohan ; Heath III, Fenno Terry ; Hobson, Stacy ; Linehan, Mark ; u. a.: Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In: *Proceedings of the 5th ACM international conference on Distributed event-based system* : ACM, pp. 51–62, 2011

[HIST03] Halevy, Alon Y ; Ives, Zachary G ; Suciu, Dan ; Tatarinov, Igor: Schema mediation in peer data management systems. In: *Proceedings of the 19th International Conference on Data Engineering* : IEEE, pp. 505–516, 2003

[Hull08] Hull, Richard: Artifact-centric business process models: Brief survey of research results and challenges. In: *On the Move to Meaningful Internet Systems Confederated International Conferences* : Springer, pp. 1152–1163, 2008

[Hull97] Hull, Richard: Managing semantic heterogeneity in databases: A theoretical perspective. In: *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* : ACM, pp. 51–61, 1997

[HuZh96a] Hull, Richard ; Zhou, Gang: A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '96*. New York, NY, USA : ACM — ISBN 0-89791-794-4, pp. 481–492, 1996

[HuZh96b] Hull, Richard ; Zhou, Gang: Towards the study of performance trade of between materialized and virtual integrated views. In: *Workshop on Materialized Views*. vol. 91, 1996

[JaMS95] Jagadish, Hosagrahar V ; Mumick, Inderpal Singh ; Silberschatz, Abraham: View maintenance issues for the chronicle data model. In: *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* : ACM, pp. 113–124, 1995

[JiAC07] Jiang, Qingchun ; Adaikkalavan, Raman ; Chakravarthy, Sharma: MavEStream: Synergistic integration of stream and event processing. In: *Digital Telecommunications, 2007. ICDT'07. Second International Conference on* : IEEE, pp. 29–29, 2007

[JoBa14] Joseph, Harry Raymond ; Badr, Youakim: Business artifact modeling: A framework for business artifacts in traditional database systems. In: *Enterprise Systems Conference (ES), 2014* : IEEE, pp. 13–18, 2014

[KaPa09] Katsis, Yannis ; Papakonstantinou, Yannis: View-based data integration. In: *Encyclopedia of Database Systems* : Springer, pp. 3332–3339, 2009

[KGFE08] Küster, Jochen Malte ; Gerth, Christian ; Förster, Alexander ; Engels, Gregor: A Tool for Process Merging in Business-Driven Development. In: *CAiSE Forum*. vol. 344 : Citeseer, 2008

[KlCa06] Klyne, Graham ; Carroll, Jeremy J: Resource description framework (RDF): Concepts and abstract syntax 2006

[KuLW08] Kumaran, Santhosh ; Liu, Rong ; Wu, Frederick Y: On the duality of information-centric and activity-centric models of business processes. In: *International Conference on Advanced Information Systems Engineering* : Springer, pp. 32–47, 2008

[KuSu10] Kucukoguz, Esra ; Su, Jianwen: On lifecycle constraints of artifact-centric workflows. In: *International Workshop on Web Services and Formal Methods* : Springer, pp. 71–85, 2010

[KuYY14] Kunchala, Jyothi ; Yu, Jian ; Yongchareon, Sira: A survey on approaches to modeling artifact-centric business processes. In: *International Conference on Web Information Systems Engineering* : Springer, pp. 117–132, 2014

[LaWB08] Langegger, Andreas ; Wö\s s, Wolfram ; Blöchl, Martin: A semantic web middleware for virtual data integration on the web. In: *European Semantic Web Conference* : Springer, pp. 493–507, 2008

[LDUD10] La Rosa, Marcello ; Dumas, Marlon ; Uba, Reina ; Dijkman, Remco: Merging business process models. In: *On the Move to Meaningful Internet Systems Confederated International Conferences* : Springer, pp. 96–113, 2010

[LDUD13] La Rosa, Marcello ; Dumas, Marlon ; Uba, Reina ; Dijkman, Remco: Business process model merging: An approach to business process consolidation. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 2, pp. 11, 2013

[Lenz02] Lenzerini, Maurizio: Data integration: A theoretical perspective. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* : ACM, pp. 233–246, 2002

[LiBW07] Liu, Rong ; Bhattacharya, Kamal ; Wu, Frederick Y: Modeling business contexture and behavior using business artifacts. In: *International Conference on Advanced Information Systems Engineering* : Springer, pp. 324–339, 2007

[LLQS10] Liu, Guohua ; Liu, Xi ; Qin, Haihuan ; Su, Jianwen ; Yan, Zhimin ; Zhang, Liang: Automated realization of business workflow specification. In: *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops* : Springer, pp. 96–108, 2010

[LoNy11] Lohmann, Niels ; Nyolt, Martin: Artifact-centric modeling using BPMN. In: *International Conference on Service-Oriented Computing* : Springer, pp. 54–65, 2011

[LoWo10] Lohmann, Niels ; Wolf, Karsten: Artifact-centric choreographies. In: *International Conference on Service-Oriented Computing* : Springer, pp. 32–46, 2010

[MaHV12] Marin, Mike ; Hull, Richard ; Vaculín, Roman: Data centric bpm and the emerging case management standard: A short survey. In: *International Conference on Business Process Management* : Springer, pp. 24–30, 2012

[Maie83] Maier, David: *The theory of relational databases*. vol. 11 : Computer science press Rockville, 1983

[MaPl06] Manakanatas, Dimitris ; Plexousakis, Dimitris: A Tool for Semi-Automated Semantic Schema Mapping: Design and Implementation. In: *DISWEB* : Citeseer, 2006

[Mend09] Mendelson, Elliott: *Introduction to mathematical logic* : CRC press, 2009

[MeWe13] Meyer, Andreas ; Weske, Mathias: Activity-centric and artifact-centric process model roundtrip. In: *International Conference on Business Process Management* : Springer, pp. 167–181, 2013

[MSMP05] Meena, Hemant Kr ; Saha, Indradeep ; Mondal, Koushik Kr ; Prabhakar, TV: An approach to workflow modeling and analysis. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* : ACM, pp. 85–89, 2005

[Mura89] Murata, Tadao: Petri nets: Properties, analysis and applications. In: *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989

[NgOt13] Ngamakeur, Kan ; others: On realization of artifact-centric model for business processes 2013

[NgYL12] Ngamakeur, Kan ; Yongchareon, Sira ; Liu, Chengfei: A framework for realizing artifact-centric business processes in service-oriented architecture. In: *International Conference on Database Systems for Advanced Applications* : Springer, pp. 63–78, 2012

[NiCa03] Nigam, Anil ; Caswell, Nathan S: Business artifacts: An approach to operational specification. In: *IBM Systems Journal*, vol. 42, no. 3, pp. 428–445, 2003

[NKMH10] Nandi, Prabir ; Koenig, Dieter ; Moser, Simon ; Hull, Richard ; Klicnik, Vlad ; Claussen, Shane ; Kloppmann, Matthias ; Vergo, John: Data4BPM, part 1: Introducing business entities and the business entity definition language (BEDL). In: *IBM Corporation, Riverton*, 2010

[NPKM11] Nandi, Prabir ; Pinel, Florian ; Koenig, Dieter ; Moser, Simon ; Hull, Richard: Data4BPM, Part 2: BPEL4Data: Binding WS-BPEL to Business Entity Definition Language (BEDL). In: *IBM Corporation, Riverton*, 2011

[PaSp00] Parent, Christine ; Spaccapietra, Stefano: *Database integration: the key to data interoperability.*, 2000

[Pasq08] Pasquier, Claude: Biological data integration using Semantic Web technologies. In: *Biochimie*, vol. 90, no. 4, pp. 584–594, 2008

[PoDu12] Popova, Viara ; Dumas, Marlon: From Petri Nets to Guard-Stage-Milestone Models. In: *Business Process Management Workshops*, pp. 340–351, 2012

[PoFD15] Popova, Viara ; Fahland, Dirk ; Dumas, Marlon: Artifact lifecycle discovery. In: *International Journal of Cooperative Information Systems*, vol. 24, no. 01, pp. 1550001, 2015

[RaBe01] Rahm, Erhard ; Bernstein, Philip A: A survey of approaches to automatic schema matching. In: *the VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001

[Ritt04] Rittgen, Peter: Workflows in UML. In: *Innovations Through Information Technology*, pp. 755–758, 2004

[RuJB04] Rumbaugh, James ; Jacobson, Ivar ; Booch, Grady: *The unified modeling language reference manual* : Pearson Higher Education, 2004

[Slaz04] Slazinski, Erick D: Structured Query Language (SQL). In: *The Internet Encyclopedia*, 2004

[SPAM91] Schreier, Ulf ; Pirahesh, Hamid ; Agrawal, Rakesh ; Mohan, C: Alert: An architecture for transforming a passive DBMS into an active DBMS. In: *Proceedings of the 17th International Conference on Very Large Data Bases* : Morgan Kaufmann Publishers Inc., pp. 469–478, 1991

[StHa04] Störrle, Harald ; Hausmann, JH: semantics of uml 2.0 activities. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2004

[StHa05] Störrle, Harald ; Hausmann, Jan Hendrik: *Towards a Formal Semantics of UML 2.0 Activities*, 2005

[SuKY06] Sun, Shuang ; Kumar, Akhil ; Yen, John: Merging workflows: A new perspective on connecting business processes. In: *Decision Support Systems*, vol. 42, no. 2, pp. 844–858, 2006

[TGNO92] Terry, Douglas ; Goldberg, David ; Nichols, David ; Oki, Brian: *Continuous queries over append-only databases*. vol. 21 : ACM, 1992

[TrAS08] Trcka, Nikola ; van der Aalst, Wil ; Sidorova, Natalia: Analyzing control-flow and data-flow in workflow processes in a unified way. In: *Computer science report*, no. 08-31 2008

[Ullm00] Ullman, Jeffrey D: Information integration using logical views. In: *Theoretical Computer Science*, vol. 239, no. 2, pp. 189–210, 2000

[VaPe06] Van Der Aalst, Wil MP ; Pesic, Maja: DecSerFlow: Towards a truly declarative service flow language. In: *International Workshop on Web Services and Formal Methods* : Springer, pp. 1–23, 2006

[VaTe05] Van Der Aalst, Wil MP ; Ter Hofstede, Arthur HM: YAWL: yet another workflow language. In: *Information systems*, vol. 30, no. 4, pp. 245–275, 2005

[VaVa04] Van Der Aalst, Wil ; Van Hee, Kees Max: *Workflow management: models, methods, and systems* : MIT press, 2004

[VTKB03] Van Der Aalst, Wil MP ; Ter Hofstede, Arthur HM ; Kiepuszewski, Bartek ; Barros, Alistair P: Workflow patterns. In: *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003

[Whit04] White, Stephen A: Process modeling notations and workflow patterns. In: *Workflow handbook*, vol. 2004, pp. 265–294, 2004

[Whit09] White, Michael: Case management: Combining knowledge with process. In: *BPTrends, July*, 2009

[WiCe96] Widom, Jennifer ; Ceri, Stefano: *Active database systems: Triggers and rules for advanced database processing* : Morgan Kaufmann, 1996

[WVVS01] Wache, Holger ; Voegele, Thomas ; Visser, Ubbo ; Stuckenschmidt, Heiner ; Schuster, Gerhard ; Neumann, Holger ; Hübner, Sebastian: Ontology-based integration of information-a survey of existing approaches. In: *IJCAI-01 workshop: ontologies and information sharing*. vol. 2001 : Citeseer, pp. 108–117, 2001

[Xiao06] Xiao, Huiyong: *Query processing for heterogeneous data integration using ontologies*. vol. 68, 2006

[XSYY11] Xu, Wei ; Su, Jianwen ; Yan, Zhimin ; Yang, Jian ; Zhang, Liang: An artifact-centric approach to dynamic modification of workflow execution. In: *On the Move to Meaningful Internet Systems Confederated International Conferences* : Springer, pp. 256–273, 2011

[YaKL97] Yang, Jian ; Karlapalem, Kamalakar ; Li, Qing: Algorithms for materialized view design in data warehousing environment. In: *VLDB*. vol. 97, pp. 136–145, 1997

[YoLi10] Yongchareon, Sira ; Liu, Chengfei: A process view framework for artifact-centric business processes. In: *On the Move to Meaningful Internet Systems Confederated International Conferences* : Springer, pp. 26–43, 2010

[YoLZ11] Yongchareon, Sira ; Liu, Chengfei ; Zhao, Xiaohui: An artifact-centric view-based approach to modeling inter-organizational business processes. In: *International Conference on Web Information Systems Engineering* : Springer, pp. 273–281, 2011

[YYZO15] Yongchareon, Sira ; Yu, Jian ; Zhao, Xiaohui ; others: A view framework for modeling and change validation of artifact-centric inter-organizational business processes. In: *Information systems*, vol. 47, pp. 51–81, 2015

[ZHKF95] Zhou, Gang ; Hull, Richard ; King, Roger ; Franchitti, Jean-Claude: Data Integration and Warehousing Using H2O. In: *IEEE Data Eng. Bull.*, vol. 18, no. 2, pp. 29–40, 1995

[ZYLL11] Zhang, Da-Wei ; Ying, WANG ; Li, Hui-Fang ; LIU, Guo-hua: ArtiFlow Designer: A Tool towards Artifact-Centric Business Process Designing. In: *Journal of Qingdao Technological University*, vol. 4, pp. 019, 2011

FOLIO ADMINISTRATIF

THESE DE L'UNIVERSITE DE LYON OPEREE AU SEIN DE L'INSA LYON

NOM : ABI ASSAF

DATE de SOUTENANCE : 09/07/2018

Prénoms : Maroun

TITRE : Integration Framework for Artifact-centric Processes in the Internet of Things

NATURE : Doctorat

Numéro d'ordre : 2018LYSEi059

Ecole doctorale : INFOMATHS

Spécialité : Informatique

RESUME : The emergence of fixed or mobile communicating objects poses many challenges regarding their integration into business processes in order to develop smart services. In the context of the Internet of Things, connected devices are heterogeneous and dynamic entities that encompass cyber-physical features and properties and interact through different communication protocols. To overcome the challenges related to interoperability and integration, it is essential to build a unified and logical view of different connected devices in order to define a set of languages, tools and architectures allowing their integrations and manipulations at a large scale.
Business artifact has recently emerged as an autonomous (business) object model that encapsulates attribute-value pairs, a set of services manipulating its attribute data, and a state-based lifecycle. The lifecycle represents the behavior of the object and its evolution through its different states in order to achieve its business objective. Modeling connected devices and smart objects as an extended business artifact allows us to build an intuitive paradigm to easily express integration data-driven processes of connected objects. In order to handle contextual changes and reusability of connected devices in different applications, data-driven processes (or artifact processes in the broad sense) remain relatively invariant as their data structures do not change. However, service-centric or activity-based processes often require changes in their execution flows.
This thesis proposes a framework for integrating artifact-centric processes and their application to connected devices. To this end, we introduce a logical and unified view of a "global" artifact allowing the specification, definition and interrogation of a very large number of distributed artifacts, with similar functionalities (smart homes or connected cars, ...). The framework includes a conceptual modeling method for artifact-centric processes, inter-artifact mapping algorithms, and artifact definition and manipulation algebra. A declarative language, called AQL (Artifact Query Language) aims in particular to query continuous streams of artifacts. The AQL relies on a syntax similar to the SQL in relational databases in order to reduce its learning curve. We have also developed a prototype to validate our contributions and conducted experimentations in the context of the Internet of Things.

MOTS-CLÉS : Business Process Modeling and Merging, Query Languages, Data Integration, Smart Processes, Internet of Things

Laboratoire (s) de recherche : LIRIS

Directeur de thèse: BADR, Youakim

Présidente de jury : LAFOREST, Frédérique

Composition du jury :
LAFOREST, Frédérique - Professeure (Université Jean Monet) - Présidente
VERDIER, Christine - Professeure (Université de Grenoble Alpes) - Rapporteure
LAURENT, Anne - Professeure (Université de Montpellier) - Rapporteure
CAUVET, Corine - Professeur (Université Aix-Marseille) - Examinatrice
BADR, Youakim - Maître de Conférences - HDR - Directeur de thèse
AMGHAR, Youssef - Professeur (INSA-Lyon) - Co-directeur de thèse
BARBAR, Kablan - Professeur (Université Libanaise) - Co-directeur de thèse