Journal of King Saud University – Computer and Information Sciences 30 (2018) 478–497

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

An UML profile for representing real-time design patterns

Hela Marouane [a,b,*], Claude Duvallet [a], Achraf Makni [b], Rafik Bouaziz [b], Bruno Sadeg [a]

[a] LITIS laboratory, University of Le Havre, France
[b] MIRACL laboratory, University of Sfax, Tunisia

ARTICLE INFO

Article history:
Received 15 November 2016
Revised 20 June 2017
Accepted 25 June 2017
Available online 29 June 2017

Keywords:
UML profile
Design patterns
Object Constraint Language
Real-time database

ABSTRACT

Systems which manipulate important volumes of data need to be managed with Real-Time (RT) databases. These systems are subject to several temporal constraints related to data and to transactions. Thus, their design remains a complex task. To remedy this complexity, it is necessary to integrate design methods to support data and transactions temporal constraints. Among the design methods, those based on patterns have been widely used in several fields. However, despite their advantages, these patterns present some shortcomings. Indeed, they do not manage efficiently the patterns variability and they do not specify the pattern elements when they are instantiated. To overcome these limitations, we propose, in this paper, a new UML profile to (i) express the variability in patterns and (ii) to identify the pattern elements in its instance. Besides, in order to well-capture the knowledge of the domain, the proposed profile extends UML with concepts related to real-time databases and integrates OCL (Object Constraint Language) to enforce the variation points consistency. Finally, we give an example of a RT pattern that illustrates these UML extensions, where we implement the proposed profile and we validate the pattern diagrams using the constraints we have proposed.

1. Introduction

Recently, many Real-Time (RT) systems (*e.g.* driver assistance systems and control traffic systems) need to store large amounts of data and to process them in order to operate efficiently. Therefore, an RT database should be used. This database must have not only have the same features of conventional databases (*e.g.,* efficient management of accesses to structured data), but requires also efficient management of data and transactions timing constraints (Ramamritham, 1993). The design methods already proposed for traditional databases cannot be directly applied to model RT database applications since there is no mechanism to deal with the representation of time constraints. Besides, RT database applications become more complex, leading to an extensive conceptual description and to a prohibitive complexity from the practical point of view. Therefore, the design of these applications is a hard process which requires the development of new design methods to support both data structures and the dynamic behavior of RT applications, based on RT databases. In order to successfully design such applications, we believe that a powerful design method (*e.g.* design patterns (Gamma et al., 1995)) may improve the quality of development process.

The design patterns are reusable abstract design elements that can reduce the difficulty of systems modeling. These patterns present mechanisms which successfully capture and promote best practices in the software design. However, despite the benefits of design patterns, the designer may spend a lot of time to understand and to instantiate them. Thus, many researchers proposed several notations in order to facilitate the specification of the patterns and the documentation of their instances. These notations facilitate the understanding of complex concepts. There exist in the litterature different notations for documenting the patterns. For instance, we can mention the natural languages, which are imprecise and too ambigus, and the visual languages (*e.g.* the Unified Modeling Language (UML)). UML language is a standard object-oriented modeling language for general-purpose software. It is a commonly used language for visualizing, specifying and cumenting artifacts of systems by providing a precise semantics of its concepts (Rumbaugh et al., 1999). It provides a set of graphical notations to capture different aspects of the developed system.

* Corresponding author at: MIRACL laboratory, University of Sfax, Tunisia.
E-mail addresses: marouane.hela@gmail.com (H. Marouane), claude.duvallet@univ-lehavre.fr (C. Duvallet), Achraf.Makni@fsegs.rnu.tn (A. Makni), Raf.bouaziz@fsegs.rnu.tn (R. Bouaziz), bruno.sadeg@univ-lehavre.fr (B. Sadeg).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

http://dx.doi.org/10.1016/j.jksuci.2017.06.005
1319-1578/

Among the benefits, these notations (such as diagrams and icons) bring are (i) enhancing the communication between designers (facilitating communication between all participants in the development process), (ii) making easier the understanding of concepts and models. Indeed, it is easier to learn the means to draw diagrams than how to write text, because the formers are more significant and more concrete than text written. Finally, these notations can help people to grasp a lot of information more quickly than written text.

Despite their benefits, graphical notations have some shortcomings. Indeed, they lack clarity and expressive power. In addition, they are sometimes imprecise and ambiguous since they do not offer a well-defined semantics for different UML concepts. For example, UML parameterized collaborations are too limited (they do not explain precisely how to specify patterns with their usage (Sunyé et al., 2000)). Furthermore, in the instantiation level, the graphical notations do not identify accurately the pattern elements and their roles in the pattern. Moreover, the notations are not expressive enough to model specific domains. Indeed, in a particular domain design, UML should take into account the domain specificities. For instance, if we consider RT databases applications, we find that these applications are complex and they have several details that should be considered by the UML notations. They must specify RT database features, such as the time-constrained data and transactions. To overcome the shortcomings of the graphical notations, UML defines a new package, named *Profile*, to extend its syntax and its semantics. This profile provides three extension mechanisms with specific names in order to annotate UML diagrams with quantitative information:

- ≪stereotype-name≫: a stereotype which allows the definition of extensions to the UML vocabulary. It is possible to associate a *tagged value* and constraints to a stereotype.
- A tagged value: which is an attribute associated to a modeling element in order to extend its properties with a certain kind of details.
- A constraint: which is a semantic restriction added to a model element. Usually, constraints are written in OCL (OMG, 2003).

The definition of a profile is very important as it allows adding semantics as well as constraints to UML concepts (*e.g.,* classes, attributes and lifelines). Besides, it provides new vocabulary for a particular domain by giving specific notations related to it. Thus, we define, in this paper, a new profile, which represents a specialized variant of the UML 2.1.2. This profile provides UML extensions to support RT database requirements. In addition, it aims at extending UML with concepts in accordance with design patterns representation. The development of this profile has three motivations:

1. It represents the RT database applications concepts. A RT database has two main features: the notions of (i) data temporal consistency and (ii) the transactions RT constraints (Ramamritham and Pu, 1995). Some of its data must not only be logically consistent, but also temporally consistent, *i.e.* a data must be used during its validity interval and two correlated data must be used within their relative validity interval (a certain temporal window). RT data are classified into: (i) sensor data collected from sensors, and (ii) derived data calculated using the sensor data (Amirijoo et al., 2006). RT data are updated through update transactions which can be executed either periodically (to update sensor data) or sporadically (to update derived data). Therefore, we can deduce that RT databases have their own specificities. Thus, their design need to have appropriate concepts in order to consider factors, such as sensor data, derived data, the quality of data management, temporal semantics of transactions and concurrency control mechanisms in order to meet the timing constraints of RT applications (Idoudi et al., 2008). For this reason, in our work, we define UML extensions to take into account all these concepts.
2. It represents the patterns at the specification level. At this level, our profile is beneficial, as: (i) it offers flexible patterns that allow distinguishing between the fundamental elements and the variable elements, and (ii) it facilitates the comprehension of patterns instantiation.
   It expresses the patterns at the instantiation level. At this level, our profile has two advantages: (i) it ensures the traceability of patterns elements since it identifies clearly the elements belonging to each pattern, and (ii) it avoids ambiguity when composing patterns by identifying the role played by each pattern element.

Moreover, the proposed profile includes OCL constraints that ensure the design patterns diagrams consistency and correctness, *i.e.* the diagram respects all constraints, specified by the designer. In this paper, we focus on the intra-diagram consistency (for a given UML diagram, we check the consistency between its elements). For this end, we propose OCL constraints to deal with the dependence of variable elements in each pattern diagram.

Furthermore, we evaluate the proposed profile based on some criteria and we compare it with other existing profiles. Besides, we illustrate our profile through a RT design pattern defined in Marouane et al. (2012). We also implement the proposed profile using MagicDraw UML tool and we incorporate it into the design pattern diagrams and their instances. After that, we verify if the elements of these diagrams are in accordance with the OCL constraints.

The remainder of this paper is organized as follows. Section 2 overviews recent proposed UML profiles. Section 3 describes our UML profile. This section defines also a set of well-formed rules written in OCL in order to verify the patterns consistency and correctness. Section 4 describes the case study methodology that shows how we have organized and conducted our research. Section 5 illustrates the profile, using a RT sensing pattern defined in Marouane et al. (2012). It gives also two examples of system models instantiating the pattern. Section 6 depicts the implementation of the proposed profile and the verification of the OCL rules on the UML diagrams of the pattern. In Section 7, we give some concluding remarks and outline future work.

2. Related work

2.1. UML Profiles for patterns representation

Several UML profiles have been proposed in the literature to represent design patterns. They can be classified into three categories. The first one reveals design patterns at the specification level (*e.g.,* the profile proposed in Arnaud et al. (2008)). The second category proposes extensions to show patterns at the instantiation level (*e.g.,* the profiles proposed in Dong et al. (2007) and Loo et al. (2012)). In the third category, the extensions are proposed to present patterns at both the two previously-mentioned levels (*e.g.,* the profiles proposed in Reinhartz-Berger and Sturm (2009) and Rekhis et al. (2010)). These UML profiles are evaluated according to a set of criteria for patterns specification (Table 1) and patterns instantiation (Table 2). The criteria used are those defined in Rekhis et al. (2010) (variability, consistency, expressivity, composition and traceability), together with the completeness of the patterns solution.

Table 1
Criteria for the representation of patterns at the specification level.

Criteria for patterns specification	Signification
Variability	Variability in a model is the representation of UML elements that can differ from one system to another. Indeed, variability is classified into optional and alternative features. Thus, it is necessary to show the optional elements which can be omitted in a pattern instance according to the context.
Consistency	The introduction of variability using extensions can generate some inconsistencies (*e.g.,* if a fundamental actor is associated to an optional use case, then the resulting model is incoherent). Then, some rules need to be specified to ensure the pattern correctness and consistency. These rules are generally expressed in OCL. Therefore, the correct patterns instantiation depends on whether or not the defined rules inherent to the variability consistency are respected.
Expressivity	In order to have more flexible and understandable patterns, it is necessary to define expressive visual notations based on UML. It is also essential to differentiate between the notations used for pattern specification and those employed for pattern instantiation in order to improve the expressivity of design languages.
Completeness	A pattern solution is complete if the proposed solution contains the functional, the structural and the behavioral views. It is important to note that completeness of a pattern is a key factor for understanding the context of the pattern use. In addition, the completeness criterion guarantees that all the required information is present. So, we propose to present the pattern solution as a mini-system composed of the functional view (a UML use case diagram), the structural view (a UML class diagram) and the behavioral view (a UML sequence diagram). Thus, it is necessary to define UML extensions for each diagram.

Table 2
Criteria for patterns representation at the instantiation level.

Criteria for patterns instantiation	Signification
Traceability	This criterion consists of easily identifying the pattern elements in a model instantiation of the pattern. Explicit representation of the key pattern elements can assist the traceability of the pattern, because it allows tracing back the pattern from a complex model (Bouassida et al., 2006).
Composition	A pattern may be composed or integrated with other patterns to resolve design problems in a software design. Therefore, if different patterns are combined in a model, UML extensions must clearly differentiate the elements belonging to each design pattern. This is mainly important when there are overlapping elements between the composed patterns.

2.1.1. UML Profiles for patterns specification

Arnaud defined in Arnaud et al. (2008) UML extensions for the explicit representation of the pattern variation points in the functional, static and dynamic views. Indeed, the author proposed extensions to express the variation points and the variants in the use case diagram and in the sequence diagram. In addition, he introduced extensions for the use case diagram to represent the fundamental and the optional use cases. The author extended also the class diagram with a UML extension to indicate that this diagram is associated with a use case.

The use case diagram represents the input diagram for the instantiation process where the designer selects the functionality variants based on the proposed extensions. Thus, we think that the extensions are very useful, but their application to the use case diagram is limited. Indeed, the use case diagram is too abstract and cannot allow the designer identifying, for example, the optional attributes or methods according to the requirements of the modeled application. Thus, the proposed UML profile fulfills partially the variability criterion. Moreover, it does not cover the traceability and the composition criteria. In fact, the suggested extensions do not allow the visualization of the pattern elements in a design diagram modeled using patterns instantiation. Besides, the author proposed to represent the pattern static view with elementary separated packages that refer to a functionality variant. Therefore, the profile is not full expressive.

2.1.2. UML Profiles for patterns instantiation

The profile introduced by Dong et al. (2007) focuses on the patterns instantiation level. The authors suggested extending the UML class diagram to allow designers to visualize the patterns elements in the design model created using the patterns instantiation. This profile includes the stereotypes ≪PatternClass≫, ≪PatternAttribute≫ and ≪PatternOperation≫ which identify respectively the classes, the attributes and the operations belonging to each instance of a pattern. Each stereotype has an associated tagged value to display the pattern name and the role names of each element (classe, attribute and operation). Therefore, the proposed profile satisfies the traceability and the composition criteria in the static view. However, it does deal neither with the behavioral view nor with the functional view. Besides, this profile fails to satisfy the criteria required to the pattern specification.

Unlike the profile proposed in Dong et al., Loo et al. introduced extensions for the UML sequence diagram to visualize the pattern roles and the different types of interaction groups for a design pattern (Loo et al., 2012). In fact, the authors defined the following stereotypes: (i) ≪PatternRole≫, used to determine the pattern role of a message and a lifeline instantiated from a pattern, (ii) ≪PatternInteractionFragment≫, specifies that a pattern role exists in a particular pattern, (iii) ≪PatternDisengage≫, indicates that a pattern role is removed from a design pattern, and (iv) ≪PatternEngage≫ applied to specify that a new pattern role is added to a particular design pattern. However, the defined profile does not propose extensions for the pattern specification, which decreases the extensions expressivity. Moreover, this profile expresses partially the variability in the sequence diagram using the interaction operators (*e.g.,* using the operator *alt*). But, it does not represent explicitly the variable elements (*i.e.,* messages and lifelines) in this diagram. Besides, this profile does not propose extensions to identify the variations points in the static view.

2.1.3. UML Profiles for specification and instantiation of patterns

Reinhartz-Berger et al. defined an Application based on DOmain Modeling approach (ADOM-UML) that provides new extensions to denote how many times a model element can appear in a particular context (Reinhartz-Berger and Sturm, 2009). Indeed, authors proposed four extensions, named ≪optional single≫, ≪optional many≫, ≪mandatory single≫ and ≪mandatory many≫. Each extension has the *min* and the *max* tagged values which define respectively the smallest and the largest multiplicity boundary. These extensions allow expressing the variability in domain-specific design patterns. They extended the UML use case diagram (functional view), the UML class diagram (static view) and the UML sequence diagram (behavioral view) to differentiate between fundamental and optional elements in a pattern.

Besides, the defined profile proposed extensions to visualize explicitly the patterns elements in a particular design model. In fact, for each model role defined in a design pattern, a corresponding stereotype is created with the same name of this role. The design model element playing this role owns this stereotype. Thus, this profile improves the readability of the design model, but it

does not keep trace of the pattern name when instantiated. This profile needs new stereotypes for each defined model role, which makes the number of stereotypes infinite and thus uncontrollable. In addition, such stereotype is ambiguous and confusing, especially, when model roles have the same names. In fact, the ADOM-UML approach fails to satisfy the composition criterion since it does not keep trace of the pattern name when instantiated. Each pattern diagram normally contains different participants, such as classes, attributes and methods. These participants play certain roles materialized by their names. When the design pattern is instantiated, the role names of its participants may be adapted to reflect the application domain. Therefore, pattern-related information represented by the role names is lost, which makes difficult to determine in which patterns a modeling element (*e.g.,* class, attribute and method) takes part in an application design. In this case, the designers are not able to take trace of this information in the application design.

In order to fill these gaps, Rekhis et al. presented a profile, named UML-RTDP (UML-RT Design Pattern) (Rekhis et al., 2010), which uses some extensions introduced in MARTE profile (OMG, 2011) to specify the RT domain aspects. In addition, the UML-RTDP profile proposes extensions focusing on the RT design patterns specification and the patterns instantiation. In fact, this profile allows distinguishing the extensions used in the specification of the patterns from those used in the patterns instantiation, which allows ensuring the expressivity criterion.

At the specification level, this profile introduces extensions to distinguish explicitly between the fixed part and the variable parts, which expresses the variability of the patterns. In fact, this profile defines the following extensions: (i) ≪mandatory≫, used to represent fundamental elements (*i.e.,* classes, association and lifelines), (ii) ≪optional≫, applied to specify optional elements (*i.e.,* classes, association, attributes and methods), (iii) ≪extensible≫, used to indicate if a class is extensible by adding new attributes and/or operations, and (iv) ≪variable≫, used to show that the method implementation varies according to the pattern instance. However, the UML-RTDP profile has limits in representing the variability criterion. First, it proposes extensions to express the variability in the static view (UML class diagram) and in the behavioral view (UML sequence diagram), but it does not define extensions that express the variability in the functional view. Second, it does not give extensions for certain pattern participants (*e.g.,* attributes and messages). Indeed, it does not provide extensions (i) to represent explicitly the fundamental attributes and operations, and (ii) to differentiate between optional and fundamental messages in a sequence diagram. This makes the pattern instantiation a difficult task since the designer is not able to distinguish between fundamental elements and variable elements.

Besides, the UML-RTDP profile integrates some OCL constraints ensuring the consistency in the static view of patterns. However, authors did not validate these constraints; they did not verify the consistency of the patterns diagrams using the OCL. Furthermore, the UML-RTDP profile did not include constraints to check the consistency of the other views, such as the behavior view. This can generate some inconsistencies which may result in incorrect pattern instances. For this reason, it is necessary to define constraints that enable the verification of consistencies of UML diagrams composing a pattern.

At the instantiation level, the defined profile proposes extensions to express explicitly the information about the role played by a design model element participating in the pattern instance. Indeed, authors suggested the following stereotypes to extend the class and the sequence diagrams: ≪PatternClass≫, ≪PatternInteraction≫ and ≪PatternLifeline≫. The first one is used to specify that the class is instantiated from a design pattern. The two other stereotypes are used to define respectively the pattern role of an interaction and a lifeline in a specific sequence diagram instantiated the design pattern. Each stereotype has two associated tagged values which are named *PatternName* (to indicate the name of the pattern) and *ParticipantRole* (to show the role played by the element in the pattern). Therefore, these extensions fulfill the traceability and composition criteria. However, this profile is limited because it defines extensions allowing only specifying classes, lifelines and interactions that play a role in a pattern, but it does not specify essential elements used to represent the properties and the behavioral features of an object (attributes, operations and messages). This means that it is difficult (i) to trace these pattern elements, and (ii) to differentiate between the pattern elements and the added elements.

Park et al. developed a PICUP (Pattern Instance Changes with UML Profiles) design method with the DPUPs (Design pattern in UML profiles) (Park et al., 2013). This design method helped preserving the quality of UML pattern-based design during perfective and corrective design maintenance for software systems by reducing the number of design pattern defects. The authors specified the design pattern using the UML profile extended from UML metamodel (M2). This profile was used to verify the conformance of design patterns instances to design patterns. Conformance checking can be achieved with UML class diagrams and OCL constraints.

The authors distinguished between a stereotype and a stereotype instance. Actually, each stereotype was declared with the stereotype keyword above or in the front of the name of the pattern element. These stereotypes did not allow differentiating between the fundamental elements and the variable elements. Thus, the UML profile did not express the variability in the pattern. However, the stereotype instance was expressed with the stereotype keyword *string*, surrounded by a pair of guillemets above or in front of the model element name. Moreover, the associations, the properties and the methods instantiated from the pattern were defined respectively with the following stereotypes: ≪PatternAssociation≫, ≪PatternProperty≫ and ≪PatternOperation≫. The *"Pattern"* at the name of the stereotype was substituted by a particular pattern name. The stereotypes used at the instantiation level allowed determining easily the pattern elements. Hence, the UML profile can express adequately the traceability and the composition criteria.

To conclude, none of the above-studied profiles satisfies all the criteria of the patterns specification and instantiation. We note that the UML profiles satisfying both patterns specification and instantiation criteria have some limitations. For instance, the UML-RTDP profile has limitations in expressing the variability in the patterns. Furthermore, the above-studied profiles do not deal with the RT design patterns since they do not provide extensions to represent RT features. The UML-RTDP profile is the only one that provides extensions to fulfill the RT domain requirements. But, these extensions are not very expressive since they do not express clearly the RT databases requirements (*e.g.,* the two kinds of RT attributes: sensor attributes and derived attributes). In order to overcome these weaknesses, we define, in this paper, a UML profile (i) that is used for the design patterns representation, and (ii) that takes into account the already-presented criteria as well as the specificities of the RT database applications. This profile allows us obtaining understandable, complete, expressive and consistent patterns.

2.2. RT UML profiles

Several researches defining UML extensions for RT applications have been proposed. For instance, we can mention TURTLE (Timed UML and RT-LOTOS Environment) profile (Apvrille et al., 2004) and RT-UML profile (Douglass, 2004). The basic concepts of RT-UML were integrated in the UML standard through the UML profile for

Schedulability, Performance and Time (denoted SPT profile) (OMG et al., 2005). The SPT profile which is extended from UML 1.4 does not specify a full methodology. It has some limitations related to expressive power and flexibility. For this reason, this profile is replaced by the MARTE profile which is an UML OMG standard for the Modeling and Analysis of RT Embedded systems (OMG, 2011). This profile extends UML 2.0 to support the aspects of time, hardware and software resources and Non-Functional Properties (NFP). However, the notions of RT data and RT transactions which are the basic features of RT databases do not taken into account in MARTE. Thus, Idoudi et al. introduced the UML-RTDB (UML-Real-Time DataBase) profile to take into account the RT database requirements only in the structural model (class diagram) (Idoudi et al., 2008). This profile specifies RT classes and two kinds of RT attributes: sensor and derived attributes in order to satisfy the requirements of current RT applications. Moreover, this profile represents explicitly the type of RT operations through the stereotypes ≪periodic≫, ≪sporadic≫ and ≪aperiodic≫. Nevertheless, the UML-RTDB does not mention how to specify some properties of RT databases, especially non-functional properties despite their importance in enhancing the software development quality.

The UML/MARTE profile was introduced to express RT databases features in a structural model (Louati et al., 2012). In Louati et al. (2012), some concepts were inspired by the UML-RTDB and the MARTE profiles. The UML/MARTE profile supports all RT databases features, *i.e.* time-constrained data, time-constrained transactions, concurrency, schedulability, non-functional properties, etc. However, it does not supply extensions for the behavioral view. Moreover, it does not mention how to specify the values of certain constraints (*e.g.,* the triggered time of a sporadic transaction). From the previous UML RT profiles description, we conclude that a new profile should be introduced to represent the RT databases features and the non-functional properties in both structural and behavioral models.

Nevertheless, the only UML extensions, defined for representing RT databases characteristics, remains insufficient to specify design patterns. Indeed, they must be generic designs intended to be instantiated in order to model any system that needs RT database environment. For this reason, additionally to the UML notations, expressing RT databases features, we need notations distinguishing the commonalities and differences between systems in the pattern domain. We need also extensions for the explicit representation of the pattern elements roles in a design model instantiating the pattern.

3. DP-RTDB profile

In this section, we propose DP-RTDB (Design Pattern-Real Time DataBase), a UML profile for patterns. It represents an extension of UML 2.1.2. Our profile takes into account not only the RT databases aspects, but also the variability and the aspects relative to the patterns. Indeed, the DP-RTDB profile provides extensions that allow (i) expressing the variability in a pattern, (ii) specifying the roles played by each pattern element in its instance and (iii) representing RT databases requirements as well as the non-functional properties.

To define our profile, we imported some extensions from the existing profiles. We proposed also new extensions. Each stereotype and its tagged values are presented on the UML diagrams using *notes*.

3.1. UML extensions for modeling RT database features

This sub-section summarizes the UML extensions supporting the modeling of RT database features. Some stereotypes were inspired by the UML-RTDB profile (Idoudi et al., 2008) used in the UML class diagram. We applied these stereotypes also on the UML sequence diagram. Moreover, the stereotype ≪nfp≫ was imported from the MARTE profile (OMG, 2011). Besides, we defined a new stereotype that represents explicitly the database system storing RT data and managing RT operations. We describe below the purpose of using each stereotype:

- ≪sensor≫ stereotype. We used this stereotype to extend *Class* and *Lifeline* meta-classes. It indicates that the measurement is a sensor data (data issued from sensors). This stereotype has the same meaning of the ≪sensor≫ stereotype defined in Idoudi et al. (2008), which extends only the*Property* meta-class.
- ≪derived≫ stereotype. We used this stereotype to extend the *Class* and *Lifeline* meta-classes. It indicates that the measurement is a derived data *i.e.,* data is calculated using sensor data. This stereotype has the same signification of the ≪derived≫ stereotype defined in Idoudi et al. (2008), which extends only the *Property* meta-class.
  We used these two stereotypes to extend the *Class* and the *Lifeline* meta-classes since the sensor data and the derived data represent objects characterized by the attributes: value, unit, timestamp and validity duration.
  We did not consider the tagged values associated to the ≪sensor≫ and ≪derived≫ stereotypes introduced in Idoudi et al. (2008) since these attributes are represented as fields associated to each RT data stored in the database.
- ≪rtDatabase≫ stereotype. A RT database is, by definition, a database system having several components, such as transactions, concurrency control and storage management (Stankovic et al., 1999). Thus, the design of a RT database is to consider these components. So, we define the ≪rtDatabase≫ stereotype which extends the meta-class *Class* and the meta-class *Lifeline*. This stereotype specifies the database system storing RT data (sensor and derived data) and managing RT operations (*e.g.,* concurrency control and updating the data).
- ≪periodic≫ and ≪sporadic≫ stereotypes. We used them to extend *Operation* and *Message* meta-classes. The ≪periodic≫ stereotype indicates that a transaction is executed periodically. The ≪sporadic≫ stereotype shows that a transaction is executed sporadically (the method is executed in the demand). These stereotypes have the same signification of the ≪periodic≫ and ≪sporadic≫ stereotypes (Idoudi et al., 2008) which extend only the *Operation* meta-class.

The *deadline* and *period* attributes were inspired by the UML-RTDB profile (Idoudi et al., 2008). The *deadline* attribute associated to ≪periodic≫ and ≪sporadic≫ stereotypes specifies the time by which the transaction execution must be achieved. The *period* attribute which is related to the ≪periodic≫ stereotype determines the periodicity of the transaction.

Besides, we defined the following attributes:

- *triggeredTime* attribute: The sporadic transactions have to derive a new data item from sensor data items when the latter are updated; that is why we introduced the *triggeredTime* attribute. We associated it to the stereotype ≪sporadic≫ in order to define the time at which the transaction is executed.
- *policy* attribute: A transaction has to read or write data items. A reader transaction may read RT data items and read or write non-RT data items. However, a writer transaction has only to write RT data items. Hence, we defined the *policy* attribute to specify the type of each transaction. We associated this attribute to both ≪periodic≫ and ≪sporadic≫ stereotypes.
- *priority* attribute: A transaction is scheduled according to its priority, which is assigned to the transaction based on its deadline

(Earliest Deadline First Scheduling Policy). Thus, a high priority transaction is executed before a lower priority transaction. This is true only if a high priority transaction is ready for execution. If no higher priority transaction is ready for execution, then a lower priority transaction is executed, if it is ready. As a consequence, data conflicts resolution should be based on transaction priorities. So, the priority assignment policy is important to control the concurrency between transactions. In fact, we defined the *priority* attribute which takes an integer parameter. We associated this attribute to both ≪periodic≫ and ≪sporadic≫ stereotypes.

- ≪nfp≫ stereotype: It is imported from NFP Modeling subprofile of MARTE (OMG, 2011). This stereotype extends the *Property* meta-class to show that the attributes are used to satisfy non-functional requirements, such as duration and frequency.

3.2. UML extensions for patterns specification

Design patterns are generic designs intended to be instantiated to model different systems. Thus, it is necessary to define UML extensions expressing the variability in patterns in order to assist the designer in distinguishing the commonalities and differences between the different systems. For this reason, we extended the functional, structural and behavioral views with UML extensions representing explicitly the optional and fundamental elements. Actually, the class diagram, the sequence diagram and the use case diagram were extended with the following stereotypes: (a) ≪mandatory≫, used to specify the fundamental elements (*i.e.* classes, attributes, methods, relations, use cases, actors, lifelines, messages and combined fragments) that must be instantiated when the model is applied to a specific application, and (b) ≪optional≫, used to express optional features (*i.e.,* classes, attributes, operations, relations, use cases, actors, lifelines, messages and combined fragments) that can be omitted in a pattern instance. These stereotypes have the same meaning with the stereotypes ≪optional≫ and ≪mandatory≫ defined in Rekhis et al. (2010). The stereotypes which are introduced in Rekhis et al. (2010) extended only the UML class diagrams, especially the classes and the relations.

Besides, we extended the use case diagram and the class diagram with the ≪variant≫ and ≪variationPoint≫ stereotypes. For the use case diagram, the latter is used to specify an abstract use case which can be expressed differently, while the former is applied to designate that the use case represents a specific realization of the variation point.

For the class diagram, we modeled variation point using UML inheritance and stereotypes (the stereotypes were used to give more semantics to the UML classes): each variation point was expressed by an abstract class and a set of sub-classes. The abstract class was stereotyped ≪variationPoint≫ and each sub-class was defined with the stereotype ≪variant≫. These two stereotypes have the same signification with the stereotypes defined in Ziadi et al. (2003).

Moreover, we proposed the ≪alternative≫ stereotype to extend the sequence diagram. This stereotype was used to express alternative messages. In fact, it indicates that the designer must choose at least one message or a sequence of messages according to a given condition when the pattern is instantiated.

3.3. UML extensions for patterns instantiation

The design patterns were specialized and instantiated by a specific application. Thus, we needed new extensions distinguishing between the pattern participants and the specific added elements. For this reason, we extended our profile to support instantiation of structural, behavioral and functional views of design patterns. For instantiation of structural design patterns, our stereotypes aim at visualizing the basic pattern elements (classes, attributes and operations) in an application model. To define notations for classes, attributes and operations, we based our work on the UML profile proposed by Dong et al. (2007). Thus, we imported the following stereotypes: (a) ≪PatternClass≫, associated with a class to indicate that this element is an instantiated pattern class and is not added by the designer, and (ii) ≪PatternAttribute≫ and ≪PatternOperation≫, which show respectively that an attribute and an operation play a role in the design pattern. In the design of a specific application, apart from the instantiation of the elements that play a role in the pattern, the designer can add some specific elements relative to the application. Thereby, it is important to distinguish between the pattern elements and the specific elements. Thus, we defined the new ≪ApplicationClass≫, ≪ApplicationAttribute≫ and ≪ApplicationOperation≫ stereotypes which indicate respectively that the class, attribute and operation are added by the designer and they constitute specific application elements.

In order to identify a model element which plays a role in the pattern sequence diagram, we imported the stereotype ≪PatternLifeline≫ from the profile defined in Rekhis et al. (2010). This stereotype was used to distinguish between the instantiated pattern lifelines and the specific lifelines. The UML profile proposed in Rekhis et al. (2010) does not deal with the messages and the combined fragments which represent essential elements in the sequence diagrams. Thus, we defined the ≪PatternMessage≫ and ≪PatternFragment≫ stereotypes. The former stereotype was used to indicate that the message takes part in a sequence pattern instance. The latter is associated with a combined fragment to reveal that this element is instantiated from the pattern sequence diagram.

Besides, we proposed the ≪ApplicationLifeline≫ and ≪ApplicationMessage≫ stereotypes to show respectively that a lifeline and a message are added by the designer as specific elements according to the modeled application.

As the use case diagram is a part of the pattern solution, we proposed new extensions to represent explicitly the patterns element roles in the functional view. We introduced new stereotypes called ≪PatternUseCase≫ and ≪PatternActor≫. The first one was used to specify that the use case is instantiated from the pattern, while the second stereotype, it was used to indicate that the actor is instantiated from the use case diagram of the pattern. We proposed also the ≪ApplicationActor≫ and ≪ApplicationUseCase≫ stereotypes to specify respectively that an actor and a use case are specific elements for the modeled application which instantiates the pattern.

The proposed stereotypes allow distinguishing easily the pattern elements and the specific elements, which makes easier to validate the patterns instances using tools.

For each of these stereotypes, we adapted the tagged value *pattern* proposed in Dong et al. (2007) to specify not only the exact role of an element in a pattern class diagram, but also the role of an element in a pattern sequence diagram and in a use case diagram. This tagged value has the following format: {*patternName, role*}, where *patternName* is the name of the pattern and *role* is the name of the role played by the element in the pattern. ≪PatternAttribute≫ and ≪PatternOperation≫ notations may be confused when two different operation or attribute roles defined within different class roles have the same name. This ambiguity can be confusing for tools during validation of a pattern instance. In order to overcome this limitation, we kept information about the class name of an attribute or an operation. Thus, we added the *className* for the ≪PatternAttribute≫ and ≪PatternOperation≫ stereotypes. So, these stereotypes have the

following tagged value {*patternName, className.role*} to indicate that the attribute or the operation is associated with the class whose name is *className* in the pattern. The information *className* allows eliminating the ambiguity when two different operation or attribute roles of different class roles have the same name.

We proposed to extend the class, the sequence and the use case meta-models by the above described stereotypes as shown in Fig. 1 (for structural view), Fig. 2 (for behavioral view) and Fig. 3 (for functional view). These figures depict the relationships between the UML meta-classes in order to facilitate the understanding of OCL rules explained in the next section. In fact, Fig. 1 illustrates the UML class diagram meta-model. The latter provides the essential elements (class, classifier, attribute, operation and association) linked through the associations and the inheritance relationships. The structural elements are described by properties (defined with the *Property* meta-class). The concept of *Property* has two object properties with cardinality 1. At the instance layer, it is determined if an instance of the class *Property* is representing an attribute (contained by a class) or a non-attribute (contained by an association). Besides, the meta-model supports the definition of operations (the *Operation* meta-class) representing the functions and the methods of a class.

Fig. 2 displays the UML sequence diagram meta-model. The *Lifeline* meta-class represents a specific object. Lifelines communicate with each other through messages (*Message* meta-class). Each message triggers two events: send event and receive event. The *Interaction* meta-class refers to the unit of behavior that focuses on the exchanges of information between a set of objects in the diagram. *InteractionFragment* is a piece of an interaction. The aggregation between the *Interaction* and the *InteractionFragment* specifies composite interaction, in the sense that an interaction can enclose other sub-interactions.

Fig. 3 shows the UML use case diagram meta-model. This meta-model covers the basic elements of a use case diagram: *Actor*, *UseCase*, *Extend* and *Include*. The *Actor* meta-class denotes the roles adopted by external entities that interact with the system directly. The *UseCase* meta-class refers to a set of actions representing the behavior of the system. The use cases can have relationships between them. Basically, a use case can include (or may be included in) other use cases and can extend (or may be extended by) other use cases.

These meta-models contain concepts that suffer from a lack of a precise semantics. Therefore, we extended the UML concepts by the above described stereotypes. These stereotypes enhance conceptual semantics onto the various UML concepts. In other word, we added the stereotypes to the UML elements in order to assist the designer in differentiating between (i) the fundamental elements and optional elements, (ii) the pattern elements and the specific elements and (iii) the RT concepts and the non-RT concepts.

3.4. Definition of OCL constraints

The introduction of variability using stereotypes makes easier the understanding of patterns, but, it can generate some inconsistencies. For example, if a mandatory class inherits from an optional class, the resulting model is incoherent. Thus, we proposed to define constraints expressed in OCL in order to reduce the impact of variable elements, ensuring then the consistency and the correctness of patterns diagrams. In fact, these constraints can be applied to ensure the consistency between the (i) elements of the static view, (ii) the elements of the functional view and (iii) the elements of the behavioral view of design patterns. The expression of these constraints is based on the auxiliary operation (*isStereotyped (S)*) which indicates whether or not an element is stereotyped by a string *S*. It is formalized in Ziadi et al. (2003) using OCL as follows:

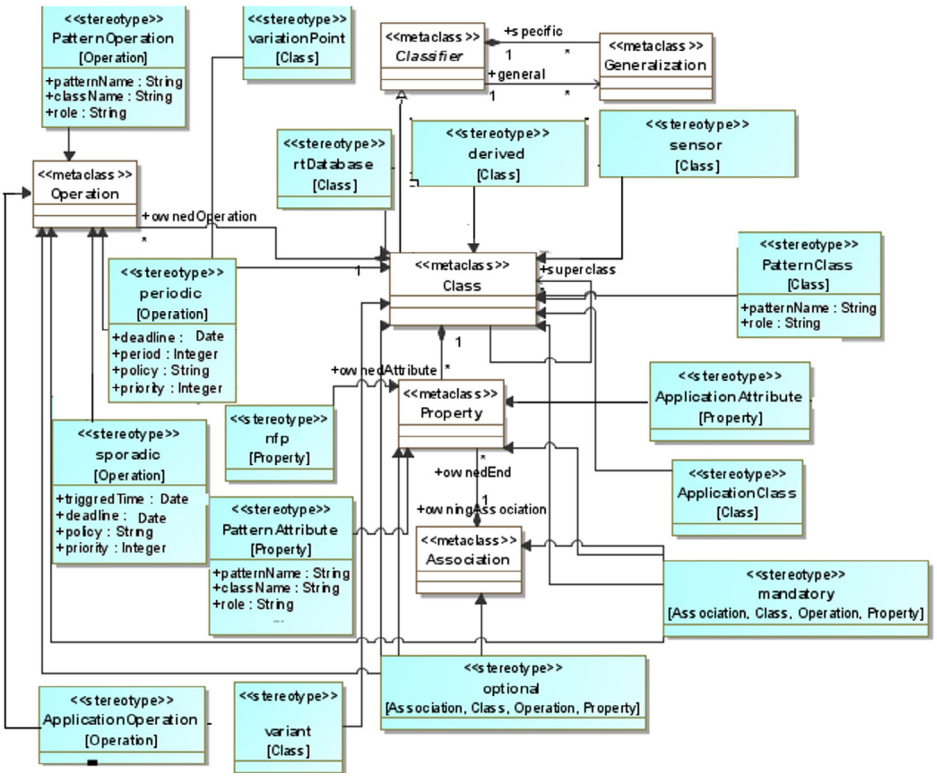

Fig. 1. Stereotypes for extension of class meta-model.

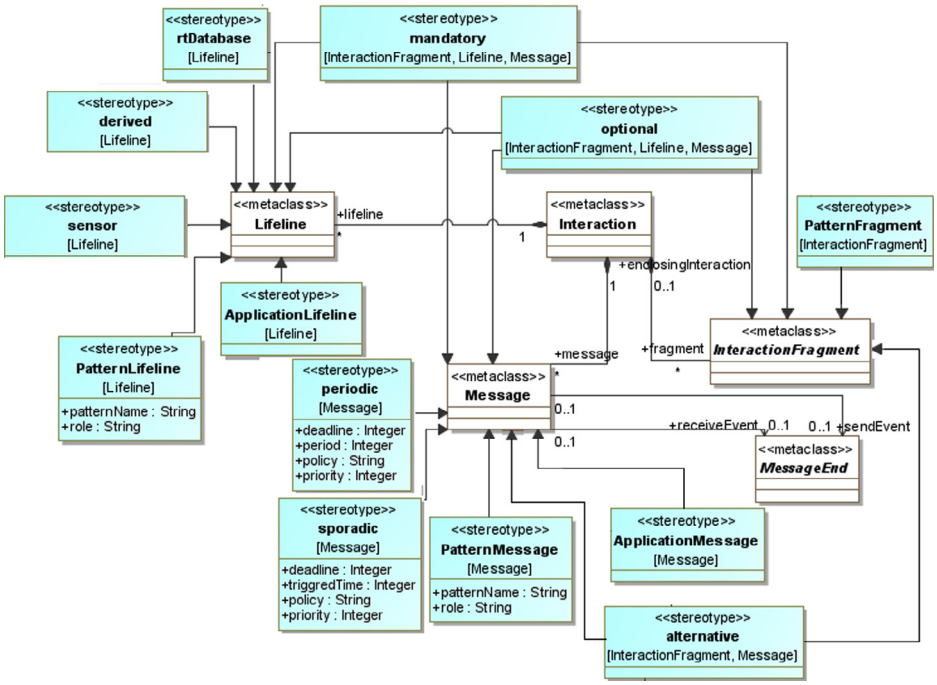

Fig. 2. Stereotypes for extension of sequence meta-model.

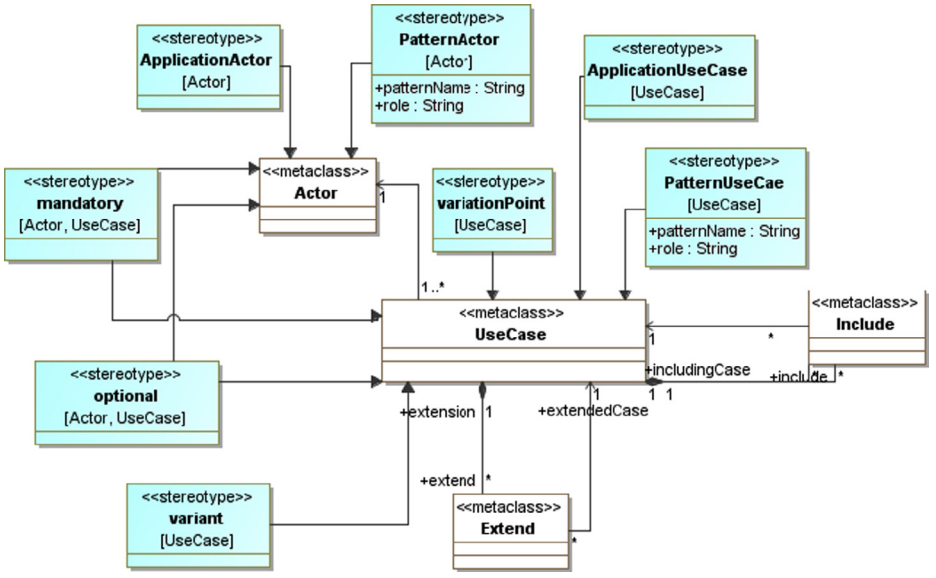

Fig. 3. Stereotypes for extension of use case meta-model.

Context Construct::Class::isStereotyped(S:string):Boolean
isStereotyped = self.extensions -> exists(E | E. ownedEnd.type.name = S)

Context Class inv:
self.isStereotyped ('optional') implies self.ownedAttribute -> forAll(a |.
a.owingAssociation.isStereotyped ('optional')).

In order to ensure the consistency of the static view of design patterns, we applied only the following constraints defined in Rekhis et al. (2013):

Constraint CC1: Each association related to an optional class must be stereotyped ≪optional≫.

In the UML 2.2 meta-model, an inheritance relationship is modeled by the *Generalization* meta-class related to the *Classifier* meta-class by two associations representing the generic class and the specialized class. Thus, the second constrqint is written as follow:

Constraint CC2: Each class which inherits the feature of an optional super class must be stereotyped ≪optional≫.

```
Context Classifier inv:
self.generalization − > forAll (c | c.general.isStereotyped ('optional')).
implies self.isStereotyped ('optional').
```

In addition to the two previously-mentioned constraints, we proposed the following:

Constraint CC3: Each class, which inherits the feature of a super class defined with the≪variationPoint≫ stereotype, must be stereotyped ≪variant≫.

```
Context Classifier inv:
self.generalization − > forAll (c | c.general.isStereotyped ('variationPoint')) implies self.isStereotyped ('variant').
```

Constraint CC4: Each attribute and each operation associated with an optional class must be stereotyped ≪optional≫.

```
Context Class inv:
self.isStereotyped ('optional') implies self.ownedAttribute − > forAll (p |.
p.isStereotyped ('optional')) and self.ownedOperation − > forAll (op |op.isStereotyped ('optional')).
```

In addition to the static view, the functional and behavioral views represent a part of the pattern solution. Thus, it is necessary to verify the consistency and the correctness of the UML diagrams representing these views. Therefore, we proposed to define OCL constraints for the UML use case diagram and for the UML sequence diagram.

For the functional view, we defined the following OCL:

Constraint CU1: If an included use case is mandatory, the including use case must be mandatory, too.

```
Context UseCase inv:
self.isStereotyped ('mandatory') implies self.including Case.isStereotyped ('mandatory').
```

Constraint CU2: If a use case is stereotyped ≪variant≫, it should have one extended use case stereotyped ≪variationPoint≫.

```
Context UseCase inv:
self.isStereotyped ('variant') implies self.extendedCase.is Stereotyped ('variationPoint').
```

Constraint CU3: each use case which inherits an optional super use case must be stereotyped ≪optional≫.

```
Context redefinableElement inv:
self.isStereotyped ('optional) implies self.redefinedEle ment.isStereotyped ('optional').
```

For the behavioral view, we proposed the following constraints:

Constraint CS1: Messages which are associated to an optional lifeline must be optional.

```
Context Lifeline inv:
self.isStereotyped ('optional') implies self.interaction. message − > forAll(m | m.isStereotyped ('optional')).
```

Constraint CS2: Messages related to an optional combined fragment must be optional.

```
Context InteractionFragment inv:
self.isStereotyped ('optional') implies self.EnclosingInter action.message − > forAll(m | m.isStereotyped ('optional')).
```

Constraint CS3: If a message is mandatory, then the sender and the receiver lifelines must be mandatory. In addition, if this message is related to a combined fragment, the latter must be mandatory.

```
Context Message inv: self.isStereotyped ('mandatory') implies.
self.sendEvent.Message.Interaction.lifeline − > forAll(l | l. isStereotyped.
('mandatory')) and self.receiveEvent.Message.Interaction. lifeline − > forAll(l1 | l1.isStereotyped ('mandatory')) and.
self.Interaction.InteractionFragment − > forAll(f | f. isStereotyped ('mandatory')).
```

3.5. Evaluation of the DP-RTDB profile

Our DP-RTDB profile is evaluated with regard to the design pattern specification and instantiation criteria. In Table 3, we compared this profile with the already-studied UML profiles. This table shows that none of the UML profiles, presented in Section 2.1, satisfies all these criteria. In fact, some studied profiles proposed extensions that allow covering either the patterns specification criteria (*e.g.* the profile proposed by Arnaud et al. (2008)) or the patterns instantiation criteria (*e.g.* the profile proposed by Dong et al. (2007)).

Furthermore, previous works focused mainly on modeling the pattern structure and the pattern behavior. However, they have some limitations related to expressing variability and representing pattern participant roles, essentially in the functional view. Obviously, the ADOM-UML approach is the only one that expresses correctly the variability criterion since it proposes extensions distinguishing all fundamental and optional elements in the static, functional and behavioral views. In addition, this approach represents the unique profile that satisfies the traceability criterion because it defines extensions to retrieve all pattern elements (*i.e.,* actors, use cases, classes, attributes, operations, lifelines, messages and interaction fragments). However, this profile fails to satisfy the composition criterion since it does not trace the name of the pattern when the latter is instantiated.

Besides, the profiles studied in Section 2.1 have limits in ensuring the variation point consistency. They do not enable the designer to verify the correctness of each pattern diagram. The ADOM-UML approach, the profile proposed in Park et al. (2013)

Table 3
Comparative analysis of the UML profiles.

	Arnaud	Dong et *al.*	Loo et al.	Reinhartz-Berger et al.	Rekhis et al.	Park et al.	DP-RTDB
Variability	+[1]	–[2]	+	++[3]	+	–	++
Expressivity	+	–	+	++	+	+	++
Consistency	–	–	–	+	+	+	++
Completeness	+	+	+	++	+	+	++
Traceability	–	+	+	++	+	+	++
Composition	–	+	+	–	+	+	++

[1] Partially verified.
[2] Not verified.
[3] Verified.

and the UML-RTDP profile are the only ones that define constraints to validate the models. The first approach verifies that the domain constraints are satisfied to validate the application model. But, these constraints do not deal with the dependence of the variable participants in a model. The ADOM-UML approach defines OCL constraints to check the conformance of design patterns instances to patterns. However, these constraints do not allow ensuring the consistency of UML class diagrams of patterns. The UML-RTDP profile defines constraints to ensure the consistency in the static view of patterns. However, these constraints are not validated. In addition, the UML-RTDP profile does not define constraints to deal with the dependencies between the variable elements of the use case diagram, and those between the variable elements of the sequence diagram of the considered pattern. These weaknesses can generate some inconsistencies that may be propagated directly into implementation errors. Besides, inconsistencies can result in incorrect pattern instance. Thus, it is necessary to give great importance to define constraints that enable the verification of consistencies of UML patterns diagrams.

Unlike the profiles proposed in literature, our DP-RTDB profile satisfies conveniently all the pattern specification and instantiation criteria. In fact, it provides expressive extensions based on UML to specify all participants of the class diagram, all participants of the use case diagram and all participants of the sequence diagram. These extensions deal with design patterns representation at the specification and the instantiation levels.

At the specification level, the DP-RTDB profile has to show clearly the variable participants through the ≪optional≫ and ≪alternative≫ stereotypes. It has also to indicate the fundamental participants through the ≪mandatory≫ stereotype. Expressing explicitly the variable elements and the fundamental elements helps the designer choosing the adapted variation in a pattern instantiation. Moreover, this profile defines a set of well-formed rules written in OCL language in order to verify the patterns consistency and correctness. These rules allow constraining dependencies between pattern variable elements in three views (class diagram, use case diagram and sequence diagram). They allow validating the consistency and the correctness of each diagram (see Section 6).

At the instantiation level, our DP-RTDB profile allows easily identifying design patterns when they are instantiated by using extensions (*e.g.,* ≪PatternClass≫ and ≪PatternLifeline≫) for explicit representation of patterns participants in software designs. These extensions allow tracing back the patterns participants (*e.g.,* classes, attributes and methods in the class diagram) from a complex design diagram. Besides, the DP-RTDB profile supports the composition of patterns. Indeed, it proposes extensions showing the pattern name and the role names of each participant (classes, uses cases, lifelines, attributes, etc.) which makes easier to recognize the pattern instance when it is composed or integrated with other patterns in a particular design diagram.

4. Case study methodology for UML profile evaluation

To evaluate the effects of using pattern annotated with UML profile design method, we present the case study research method proposed by Yin et al. (2014).

4.1. Determining and elaborating the research questions

In this case study, the main question to be answered in order to support or reject the main proposition of the case study is as follows:

- Is the UML profile an improved method ensuring structural conformance of UML pattern to the corresponding applications models?

The following sub-questions are issued from the main preceeding question:

- How UML profile can improve the design patterns?
- Does the UML profile design method result in fewer design defects than the conventional UML design method without profile?
- How UML profile can facilitate the pattern instantiation?
- Does the design patterns with UML profile result in fewer design defects than the design patterns without UML profile?

4.2. Conducting the case study

The researcher collects the requirement specification of each application and some unification rules. An expert designer conducts this case study and produces the UML diagrams annotated with UML profile.

4.2.1. The analysis unit

The analysis unit in the case study is the requirements specification documents of each application and some unification rules defined in Marouane et al. (2012). These documents are provided by contacting the organizations and the companies.

4.2.2. Questionnaire

The case study researcher asks each designer about his/her background related to their experience in information system, especially in design patterns. In Table 4, there are two different types of questions given in this case study questionnaire: (1) yes-or-no questions, and (2) short-descriptive-answer questions.

The results of questionnaire are collected from a group of ten professors on software engineering. These participants had substantial software development experience ranging from 8 to 12 years. In fact, from the answers of question 1 (Q1), the designers agreed that the applications models resulting from the pattern instantiation with the respect to the UML profile are conform to

Table 4
Questionnaire for designers.

Questions	Answer types
Q1. Does the applications models resulting from the pattern instantiation with the respect to the UML profile conform to the pattern?	Yes or No
Q2. How is the UML profile used to facilitate and to guide the pattern instantiation?	short descriptive answer
Q3. How constraints can help the designer to instantiate the pattern?	short descriptive answer
Q4. Is it easy to understand the design pattern annotated with UML profile?	Yes or No
Q5. What would be the advantage of using design patterns annotated with UML profile to model applications? Please explain.	short descriptive answer
Q6. Are the mandatory pattern elements present in the applications models?	Yes or No

the pattern. They also agreed that it easy to understand the pattern annotated with UML profile. This result is derived from the answers of the question 4 (Q4) (the designers answered by Yes).

In addition, as shown in the answers from the question 6 (Q6), the designers proved that the fundamental pattern elements are present in the application model (the designers answered by Yes). Indeed, each pattern instance is annotated with stereotypes which indicate whether or not fundamental elements are present in the application model. These stereotypes also allow distinguishing easily between the elements instantiated from the pattern (elements defined with stereotype declared with the *Pattern* keyword, *e.g.* ≪PatternClass≫ stereotype) and the specific elements (elements defined with stereotype declared with the *Application* keyword, *e.g.* ≪ApplicationClass≫ stereotype).

The answers from question 3 (Q3) show that some designers experienced a little difficulty in using the UML profile. The pattern instantiation using the OCL is not easy. That is why a textual description of each constraint is needed as shown in the answers from question 3.

The answers from question 2 (Q2) and question 5 (Q5) show that the UML profile facilitates the pattern instantiation and has several advantages to model applications. Indeed, the designers agreed that the profile helps identify easily the fundamental parts (elements defined with the ≪mandatory≫ stereotype), the optional parts (elements defined with the ≪optional≫ stereotype) and the RT concepts.

4.2.3. Design solution

The case study researcher collects the requirements specification documents of the different applications and derives a design pattern annotated with the UML profile. The modeling of this pattern is performed by applying some unification rules defined in Marouane et al. (2012), with the respect to the OCL constraints.

Before a designer conducts the pattern instantiation, the case study researcher explains the design pattern. In addition, the case study researcher provides the requirements specification documents of the different applications. Each designer uses the documents and models the applications by instantiating the pattern annotated with the UML profile.

The UML diagrams (use cases, class and sequence diagrams) and the questionnaire answers are collected from each designer.

4.3. Data analysis and evaluation

The case study researcher analyzes the applications models provided by each designer. He/she checks whether or not each application model contains the mandatory pattern elements that must be instantiated and the optional elements that can be omitted in the pattern instance. An application model is correct and conforms to the design pattern if all the mandatory pattern elements are present.

5. Case study example

The number of vehicles on the road has greatly increased in recent years, which leads to the rise in number of accidents. For example, according to the European Commission survey, 26 000 people lost their lives on European roads in 2015 (E.C.P., 2016). To reduce the risk of accidents, the safety programs have helped lower the number of deaths by increasing seat belt use. Vehicle improvements, including Driver Assistance Systems (DAS), have also contributed to reducing traffic fatalities. These systems are embedded systems in the automotive domain. Several examples of DAS (*e.g.* Adaptive Cruise Control system (Tass BV, 2012) and Lane Departure Warning system (Tass BV, 2012)) have already been introduced to the market. Indeed, DAS assist the driver by delivering warning information and/or providing automatic actions in order to reduce risks and improve road safety.

DAS are complex systems, then the design reuse may be a judicious choice to model such systems since it allows to benefit from the expertise of developers gained in the design of particular system. Since these systems must be able to meet data and transactions RT constraints, it is necessary to give a great importance to DAS design.

DAS consist mainly of three modules for sensing, data processing and making actions (Amditis et al., 2010). The sensing module is responsible for data acquisition from the environment through a set of sensors. This module includes a series of sensors (*e.g.,* radar and vehicle sensors) and a sensor data fusion unit allowing the computation of the appropriate sensors data. The sensing module is the first module which provides the necessary data for the driving situation analysis and the decision making. For this reason, we mainly focus, in this paper, on modeling the functional, the static and the behavior views of the sensing module through the definition of the *Sensing* pattern (Marouane et al., 2012). We used this pattern to illustrate the DP-RTDB extensions. This pattern is created by the application of the development process defined in Marouane et al. (2012). This process is composed of four steps:

- The study of several driver assistance documents provided by the automotive companies (*e.g.,* BMW (Prestl et al., 2000), Volvo (Amditis et al., 2010), etc.).
- The identification of DAS functionalities *i.e.,* data acquisition, data control and taking actions.
- The identification of the requirements of the data acquisition functionality. This step consists in identifying the domain concepts (*e.g.,* objects, their attributes and their relationships) related to the data acquisition functionality as well as the associated constraints. For example, the *sensor* represents a concept belonging to this functionality.
- The decomposition of applications: The already-identified concepts help to decompose the applications. Indeed, this step consists in finding the relation between these concepts in order to determine the applications fragments (*i.e.,* class diagrams, sequence diagrams and use case diagrams).
- The diagrams unification: In order to unify the different applications fragments and to derive the design pattern, we applied some unification rules defined in Marouane et al. (2012). Thus, we fused the common concepts to all DAS in order to derive the fundamental elements of the pattern. Then, we added the appropriate concepts to the applications as variable elements.

Additional information about the development process and the unification rules can be found in Marouane et al. (2012).

5.1. Pattern specification

The *Sensing* pattern is described by the following fields: (a) *name*, indicating the name of the pattern, (b) *context*, showing a design situation giving rise to a design problem, (c) *problem*, specifying a design problem at which addresses the pattern, (d) *solution*, which is a form applied to resolve the problem, and (e) *consequences*, showing the benefits and the drawbacks of the pattern.

Name: *Sensing.*

Context: This pattern is used to design the sensing module. The designer has to define the various kinds of sensors related to the developed driver assistance system and their properties.

Problem: How to specify the kinds of sensors? How driver assistance system can manage efficiently the large quantities of RT data and RT transactions?.

Solution: The solution comes in both "push" and "pull" mechanisms. In the push mechanism, each active sensor initiates the transmission of its value to the fusion unit. An active sensor sets the data values related to the observed element (*i.e.,* the controlled vehicle and its environment). In the pull mechanism, each passive sensor transmits its value only if the fusion unit queries it, *i.e.* a passive sensor gets the data values related to the observed element.

- **Functional view:** Fig. 4 shows the functional view of the *Sensing* pattern using the UML use case diagram. This diagram describes the shared and the varying functions of the sensing module. *Exteroceptive sensor*, *Proprioceptive sensor* and *Fusion unit* are the actors of the sensing module. The use cases describe five main functions: observe the vehicle, observe the vehicle environment, process data, merge data and save data in a database. We know from our discussions with domain experts in driver assistance domain that (i) observe the vehicle, observe the vehicle environment, process data and save data in a database functionalities and (ii) exteroceptive sensors, proprioceptive sensors and fusion unit actors are fundamental. However, merging data functionality is optional, then they can be omitted in several driver assistance systems since it is considered only if the system uses sensors that provide redundant or complementary data. Thus, the use cases *Observe vehicle*, *Observe vehicle environment*, *Process data* and *Save data in database* are stereotyped ≪mandatory≫, whereas the use case *Merge data* is stereotyped ≪optional≫, which expresses explicitly the variability in the pattern use case diagram.
- **Structural view:** Fig. 5 illustrates the structural view of the *Sensing* pattern *i.e.* the participants represented by the UML class diagram.

  Sensor. The sensors can be classified into exteroceptive and proprioceptive sensors. These categories of sensors constitute the variants of *Sensor* generic class. They are expressed through the generalization relationship and the stereotypes. The *Sensor* class is defined with the ≪variationPoint≫ stereotype, while the sub-classes *ExteroceptiveSensor* and *ProprioceptiveSensor* are stereotyped ≪variant≫. These sensors constitute common elements in all driver assistance systems. Thus, they are fundamental elements in the class diagram and they are defined with the stereotype ≪mandatory≫.

  The *Sensor* class has the following attributes: (i) *type*, which represents the type of a sensor (*i.e.,* passive or active), (ii) *accuracy*, indicating that the sensor can acquire the measurement with a certain error rate and (iii) *periodicity*, which defines the period within the sensor should update the measurement. This class has also the *transmit()* method to indicate that the sensor transmits periodically the measurement to the fusion unit. So, this method is stereotyped ≪periodic≫. The attributes *accuracy* and *periodicity* are defined with the stereotype ≪nfp≫ (they represent non-functional properties).

  ExteroceptiveSensor. This sub-class defines exteroceptive sensors (*e.g.,* a laser, a GPS and a radar). This class has (i) the *range* attribute, showing the maximal distance needed to detect an object, and (ii) the *observeElement()* operation, which indicates that the sensor observes periodically the external vehicle environment (*e.g.,* road and obstacles). Thus, this operation is defined with the stereotype ≪periodic≫.

  ProprioceptiveSensor. This sub-class defines the proprioceptive sensors which observe the vehicle state, such as speed sensor and yaw rate sensor. This class has the *observeVehicle()* operation which indicates that the sensor acquires periodically the measures related to the vehicle. Thus, this operation is defined with the stereotype ≪periodic≫.

  TrackedElement. This class represents the tracked elements (*e.g.,* road, fixed obstacle, driver and so on) observed by the exteroceptive sensors. The tracked elements are common between all driver assistance systems. Therefore, the *TrackedElement* class is defined with the stereotype ≪mandatory≫.

  ControlledVehicle. This class defines the controlled vehicle observed by the proprioceptive sensors. It is stereotyped ≪mandatory≫ since the vehicle is a common element between all driver assistance systems.

  Measurement. This class represents the database which stores RT data (sensor data and derived data) and manages RT transactions. It is stereotyped ≪rtDatabase≫. It specifies the common characteristics of sensor data and derived data stored in the database and it has the following attributes: (i) *value*, representing the value of the data (ii) *timestamp*, showing the time at which the attribute value is updated, (iii) *duration*[1], which is the time interval at which data is considered valid, (iv) *type*, representing the type of data (*e.g.* speed and position), and (v) *unit* of the measure. The *duration* attribute is defined with the ≪nfp≫ stereotype since it represents a non-functional property. Besides, the *Measurement* class has the *saveData()* operation, defined with the stereotype ≪sporadic≫ since the data are stored in the database only if the difference between the stored data value and its last updated value (*i.e.,* the value collected from the sensor) is greater than the maximum data error (Amirijoo et al., 2006).

  The RT measurements are classified into sensor data and derived data. These two kinds of data constitute the variants of *Measurement* generic class. Therefore, the *Measurement* class is defined with the ≪variationPoint≫ stereotype, while the *SensorData*

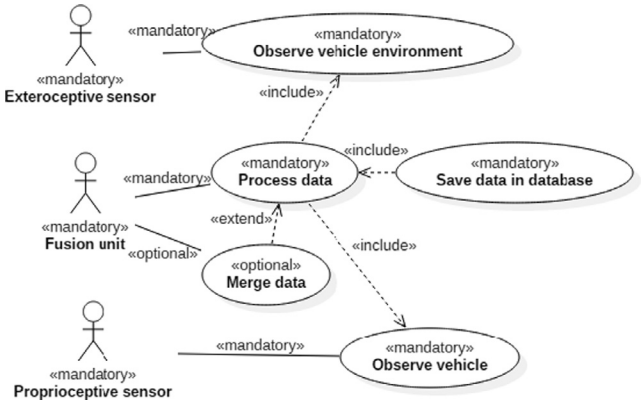

Fig. 4. UML use case diagram of *Sensing* pattern.

[1] The duration represents the duration between the timestamp and the current time.

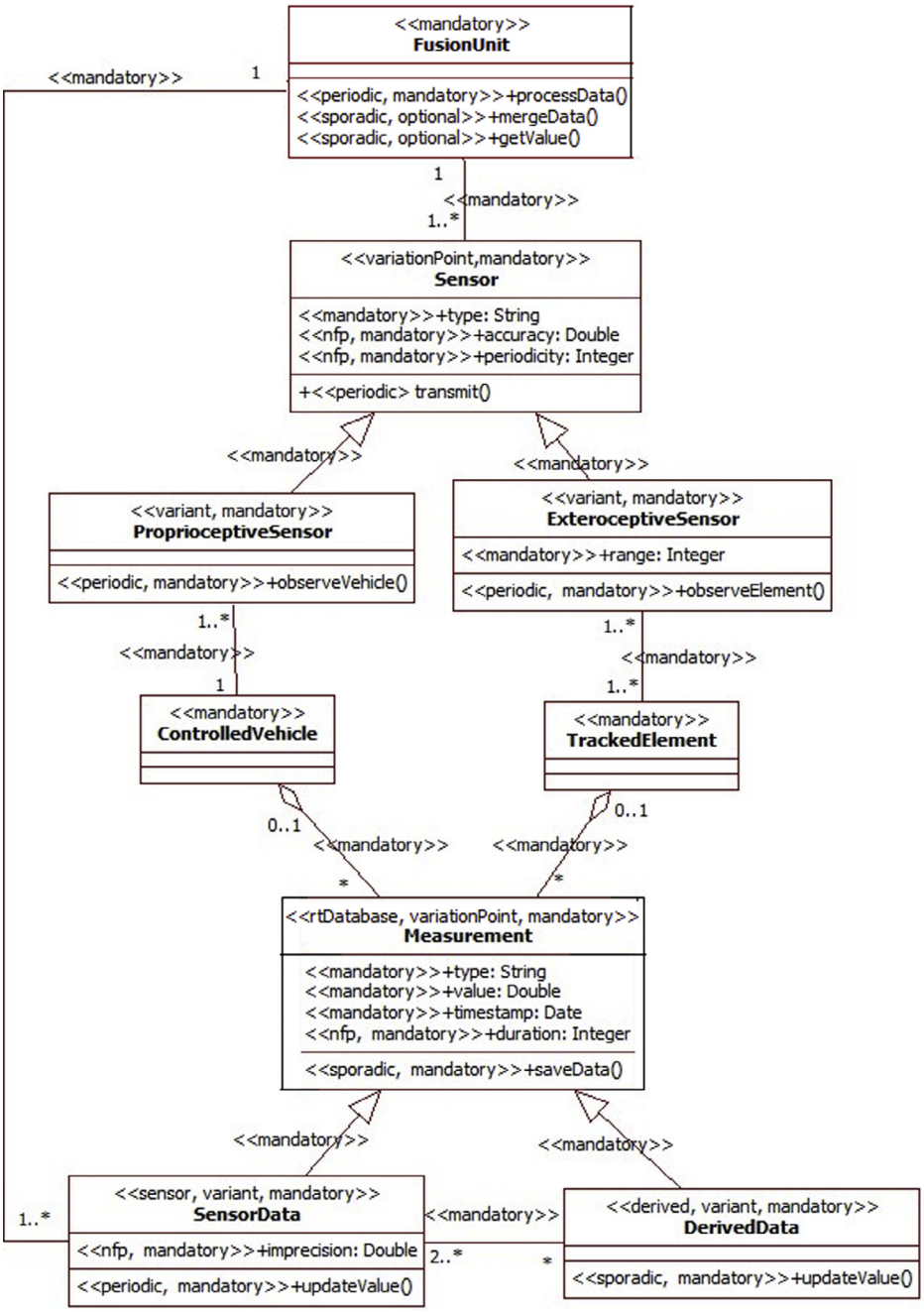

Fig. 5. UML class diagram of *Sensing* pattern.

and the *DerivedData* sub-classes are stereotyped ≪variant≫.

SensorData, DerivedData. The sensor data and derived data represent fundamental elements in driver assistance systems. Therefore, they are stereotyped ≪mandatory≫. In addition, *SensorData* sub-class is stereotyped ≪sensor≫ to show that the data collected from the sensor. *DerivedData* sub-class is stereotyped ≪derived≫ to express that the data is calculated using sensor data.

The *SensorData* sub-class has the attribute *imprecision* defined with the ≪nfp≫ stereotype. This attribute represents the maximum amount of imprecision associated with a data value (Amirijoo et al., 2006). In addition, this sub-class has the operation *updateValue()* stereotyped ≪periodic≫ since the data value is updated periodically by the sensor. However, the *DerivedData* sub-class has the operation *updateValue()* which is stereotyped ≪sporadic≫ because the data value is updated sporadically (when the value of one sensor data used in its calculation is updated).

FusionUnit. This unit allows processing periodically the collected data from a single sensor (*e.g.,* reducing noise in sensor reading, processing data with sensor parameters and extracting information from images). Thereby, the *processData()* method is stereotyped ≪periodic≫. This unit allows also fusing data issued from multiple sensors in order to produce a consistent estimation of the controlled driving situation. In fact, the *mergeData()* operation is stereotyped ≪sporadic≫ as this operation is executed only if the value of one of the used sensor data is updated in the database. In addition, this operation is stereotyped ≪optional≫ since there are applications that do not use sensors providing redundant or complementary data.

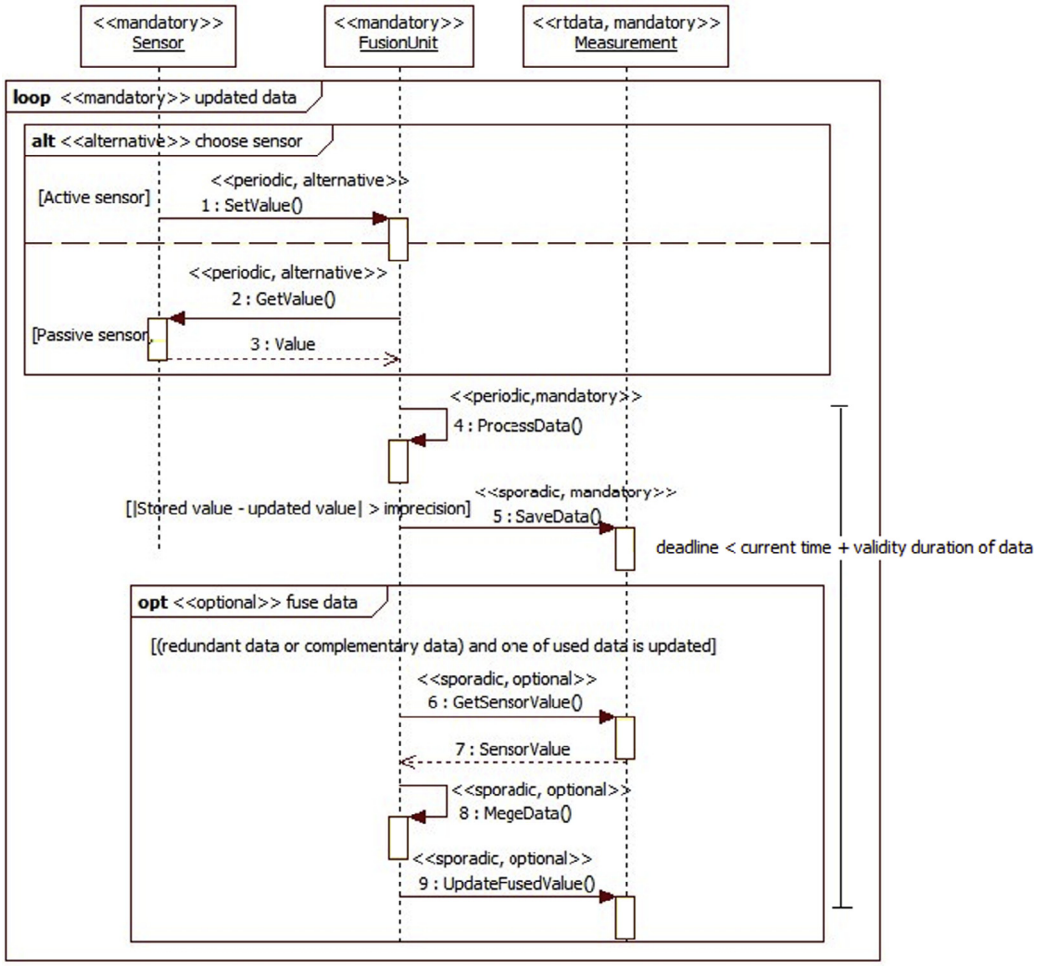

Fig. 6. UML sequence diagram of *Sensing* pattern.

- **Behavioral view:**
Fig. 6 represents the UML sequence diagram of the *Sensing* pattern. In this diagram, we modeled the interactions between the components of the sensing module of driver assistance systems and the updating of the RT data. First, the *FusionUnit* receives data from passive sensors through the *GetValue()* operation or from active sensors by the *SetValue()* signal. The *GetValue()* operation and the *SetValue()* signal are defined with the ≪alternative≫ stereotype since the designer should select at least one type of sensor (passive and/or active) when instantiating the pattern. In addition, they are defined with the stereotype ≪periodic≫ since the sensors continuously acquire data from the environment. Second, the *FusionUnit* processes periodically data (*e.g.,* converts signal and extracts information from images) to obtain a precise environment state. So, the *ProcessData()* operation is defined with the stereotype ≪periodic≫. Then, the data are stored in the database (if the difference between the stored values and the updated values exceeds the maximum data error of each data (Amirijoo et al., 2006)). Therefore, the *SaveData()* operation is stereotyped ≪sporadic≫. Afterwards, if the sensors provide redundant or complementary data, then the *FusionUnit* merges the data in order to produce a consistent estimation of the controlled vehicle and an accurate description of the external vehicle environment. Therefore, the operation *MergeData()* is stereotyped ≪optional≫. Finally, the *FusionUnit* updates the value of the fused data only if the value of one of the used sensor data is updated. Thus, the *UpdateFusedValue()* operation is stereotyped ≪sporadic≫.

Consequence: The *Sensing* pattern has the following benefits:

- Sensor classes have a common interface. Thus, the complexity of the system is potentially reduced.
- Sensor division using the generalization relationship makes the system more understandable and manageable.
- The model can be extended by adding new types of sensors.

The Sensing pattern has the following drawback:

- Using a single component for processing sensor data implies a single point of failure.

5.2. Sensing Pattern instantiation

As said previously, a RT design pattern contains a set of variation points. Thus, to obtain a particular RT system model via pattern instantiation, some choices associated with these variation points are needed. The refinement of pattern model and the choice of the suitable optional elements represent the first step of pattern instantiation. The second step is based on model transformation by renaming pattern elements and adding some specificities of a given system. It automatically generates the system model deploying the used pattern.

In this section, we propose to illustrate the *Sensing* pattern instantiation through the design of Adaptive Cruise Control (ACC) system (Tass BV, 2012) and Lane Departure Warning (LDW) system

(Tass BV, 2012), which represent two examples of driver assistance systems. We focus mainly on modeling the sensing module of these systems. We also explain how this design issue can be facilitated by the instantiation of the *Sensing* pattern.

5.2.1. Example 1: Adaptive Cruise Control System

ACC system is an automotive application integrated and tested in modern luxury cars such as Audi A8 (Tass BV, 2012). It aims at reducing the risk of accidents by controlling the longitudinal velocity of a car and keeping safe distance to the preceeding vehicle.

ACC system uses vehicle sensors and two radars (long range radar and short range radar). The radars are used to detect the presence of a lead vehicle and to measure its speed and the distance between ACC-vehicle and the forward vehicle. The sensors are used to measure sensor data (*e.g.,* vehicle speed, acceleration and throttle/brake positions). The sensor data are periodically updated to maintain consistency with the current state of the environment. The merge unit processes periodically the data provided by sensors to estimate the consistent state of a vehicle and its environment. Moreover, this unit fuses the redundant data provided by the two radars to estimate the accuracy state of the detected vehicle. These data are then stored in a RT database to guarantee the temporal and the logical constraints. The controller reads the stored data and calculates the desired acceleration or deceleration to maintain the safety distance. Thus, these data constitute the derived data.

Fig. 7 illustrates the *Sensing* pattern static view instantiation. It shows that the fundamental elements defined with the ≪mandatory≫ stereotype are instantiated. Thus, the *FusionUnit*, *Sensor*, *ExteroceptiveSensor*, *ProprioceptiveSensor*, *ControlledVehicle*, *TrackedElement*, *Measurement*, *SensorData* and *DerivedData* classes are instantiated respectively by *MergeUnit*, *Sensor*, *Radar*, *VehicleSensor*, *Car*, *Vehicle*, *Measurement*, *SensorData* and *DerivedData* classes.

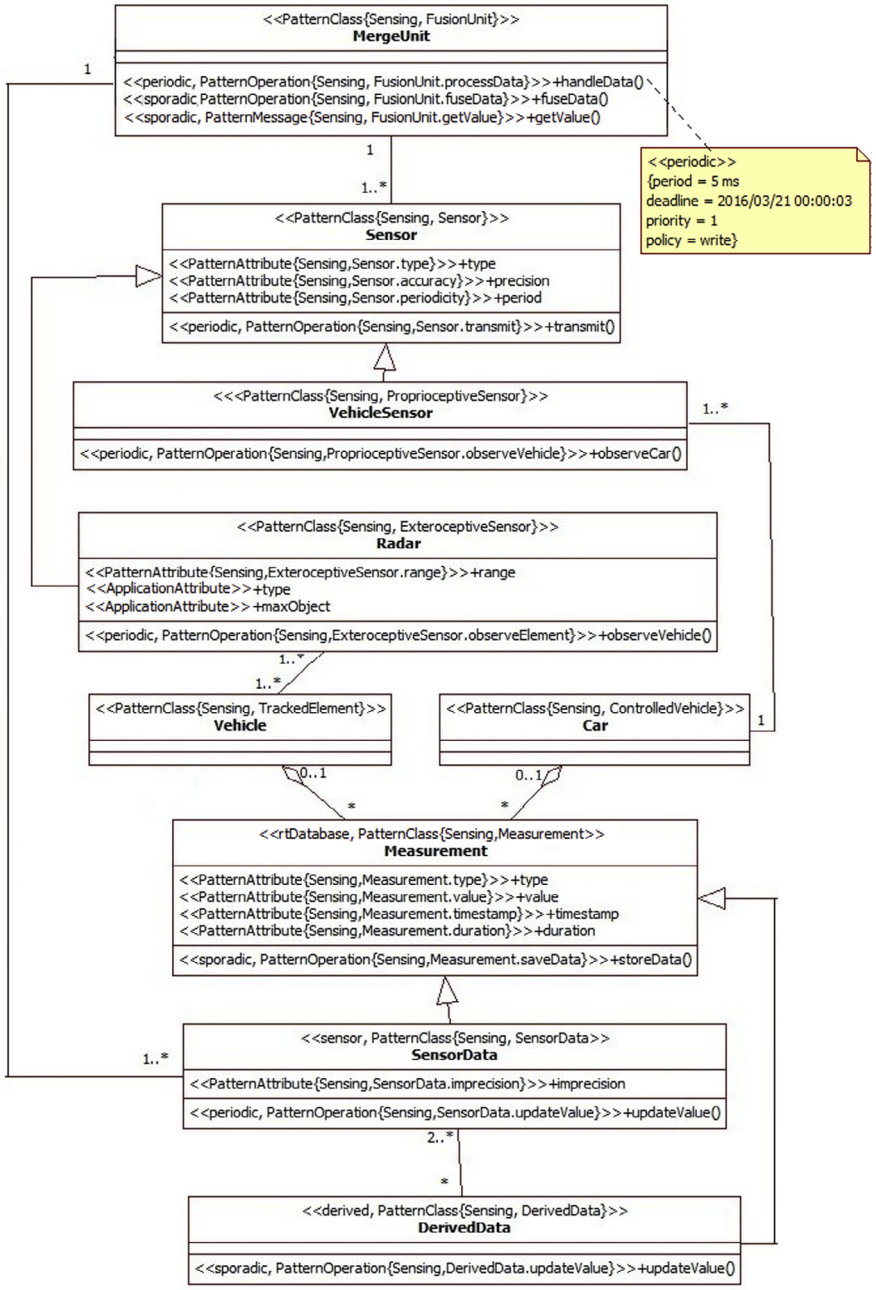

Fig. 7. UML class diagram of ACC system.

Moreover, Fig. 7 shows that the optional elements are instantiated since the ACC system uses sensors that provide redundant data. Therefore, the operations *mergeData()* and *getValue()* are instantiated respectively by the operations *fuseData()* and *getValue()*. Each class instantiated from the pattern is stereotyped automatically ≪PatternClass≫. For example, *Radar* class plays the role of an exteroceptive sensor in the *Sensing* pattern. Thus, the *patternName* is *Sensing*, while the *role* is *ExteroceptiveSensor* (*i.e.*, the stereotype has the following format: ≪*PatternClass{Sensing, ExteroceptiveSensor*}≫).

Finally, the specific elements, related to ACC system, are added: the *type* and *maxObject* attributes relative to the *Radar* class. Thus, they are defined with the ≪ApplicationAttribute≫ stereotype. As for the other attributes, they are instantiated from the pattern and they are stereotyped automatically ≪PatternAttribute≫. For example, the attribute *accuracy*, which is associated to the *Sensor* class, is instantiated by the attribute *precision*. This attribute is defined with the ≪PatternAttribute {Sensing, Sensor.accuracy}≫ stereotype. In the same way, all operations are instantiated from the pattern. So, they are stereotyped ≪PatternOperation≫.

Fig. 8 represents the pattern behavioral view instantiation. It shows that all lifelines and messages are instantiated from the pattern and are respectively stereotyped ≪PatternLifeline≫ and ≪PatternMessage≫.

The instantiation of the RT *Sensing* pattern functional view is represented in Fig. 9 in which all actors and use cases are instantiated from the use case diagram of the pattern. Therefore, they are respectively labeled with the stereotypes ≪PatternActor≫ and ≪PatternUseCase≫.

5.2.2. Example 2: lane departure warning system

LDW system is a vehicle lateral guidance system which warns the driver when the vehicle is unintentionally drifting out of its lane (Tass BV, 2012). This system maintains the vehicle position by detecting lane markings using a camera sensor.

The data acquisition of LDW system consists of vehicle sensors and a camera. The sensors are used to determine the vehicle state (*e.g.* vehicle speed and steering angle), while the camera, it is used to register a monochrome view of the road in front of the vehicle. The processing algorithms extract the desired information, such as lane width and vehicle lateral deviation. These data are stored in a RT database taking into account the RT data and transactions constraints.

The controller calculates the distance between the lines and the vehicle using the information collected by the camera and sensors in order to determine the appropriate action. If the distance between the lines and the vehicle is less than 0.3 m and the light indicator is not activated, then the system alerts the driver by a visual warning. The LDW system controls the sensors state based on a data flag component.

The instantiation of the functional view of the RT *Sensing* pattern is represented in Fig. 10. This figure shows that the *Merge data* use case is omitted since the used sensors do not provide redundant data or complementary data.

Fig. 11 represents the instantiation of the behavior view of the LDW data acquisition module. It illustrates that the optional messages relative to the data fusion are omitted since the *mergeData()* and *getValue()* methods are not instantiated in the corresponding system class diagram (Fig. 12).

Fig. 12 reveals that the *DataFlag* class with the *controlSensorState()* method is added as a specific class in the class diagram of LDW system. Therefore, the *DataFlag* class is specified with the ≪ApplicationClass≫ stereotype. In addition, the *controlSensorState()* method is defined with the ≪ApplicationOperation≫ stereotype.

6. DP-RTDB profile implementation

Several techniques were developed for model validation. In this paper, we focus on approaches that evaluate UML diagrams with OCL invariants, that are supported by tools. Currently, there are

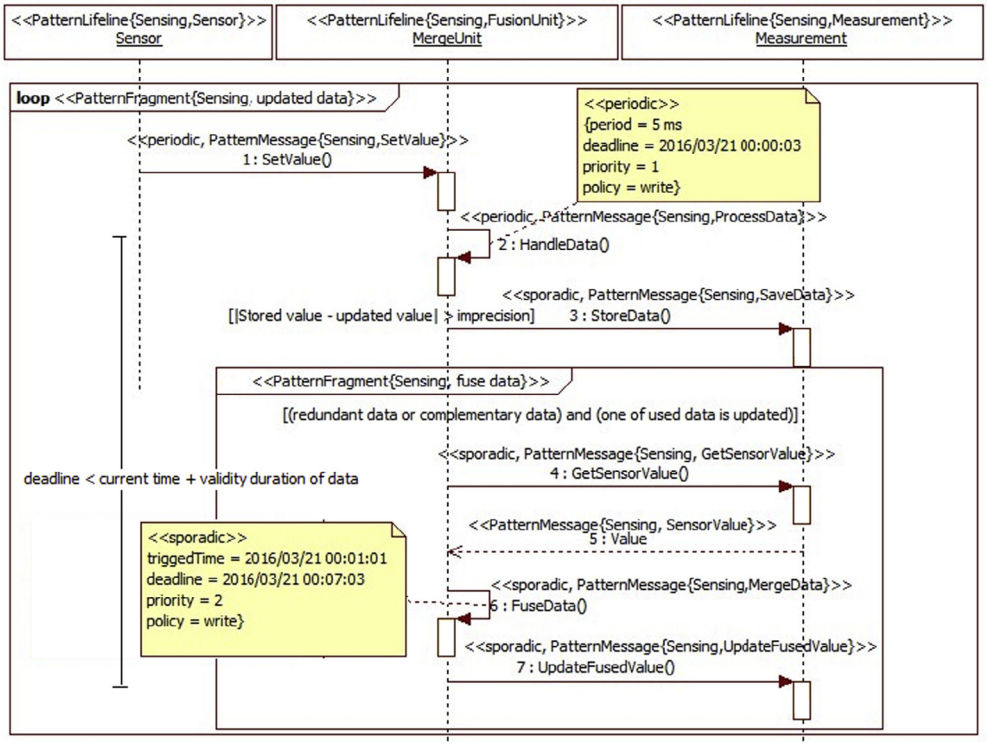

Fig. 8. UML sequence diagram of ACC system.

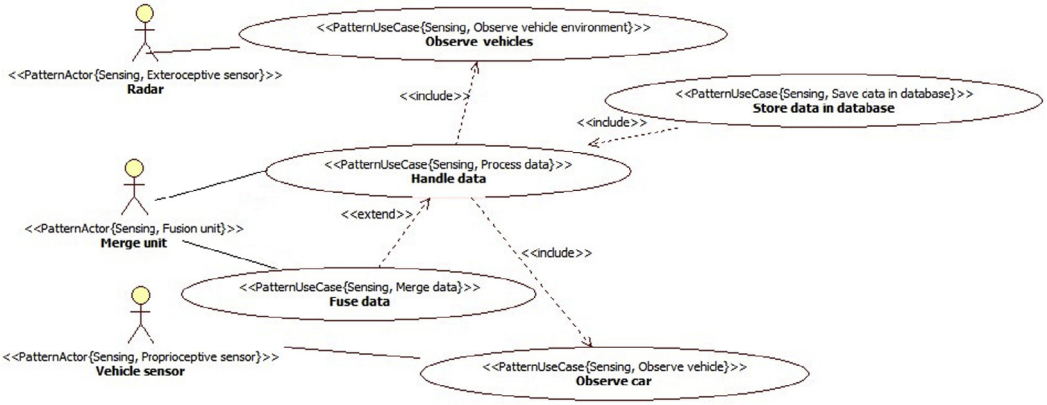

Fig. 9. UML use case diagram of ACC system.

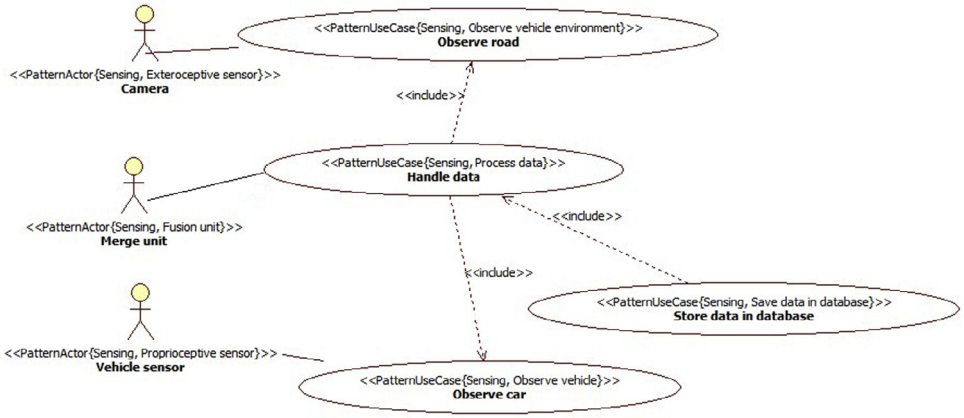

Fig. 10. UML use case diagram of LDW system.

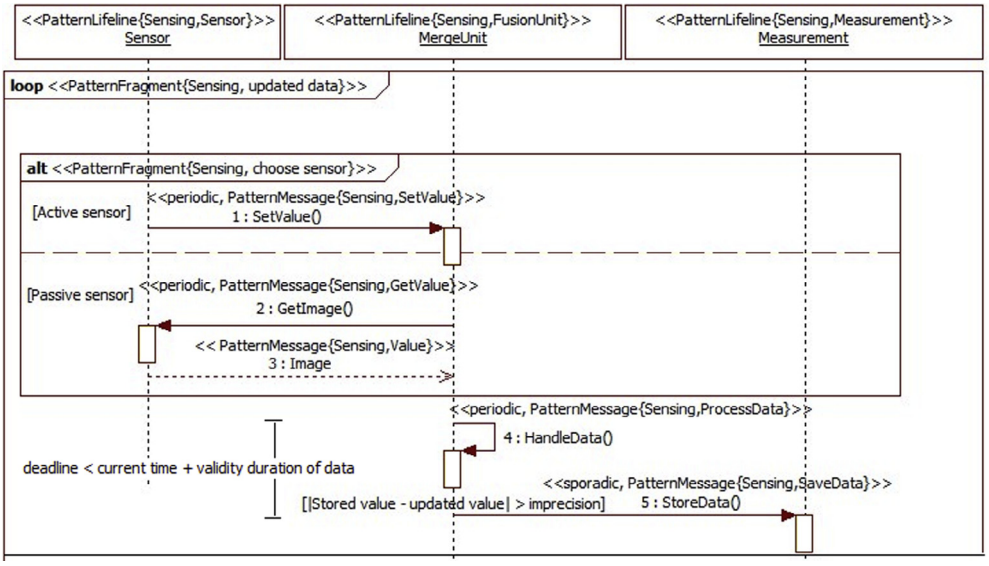

Fig. 11. UML sequence diagram of LDW system.

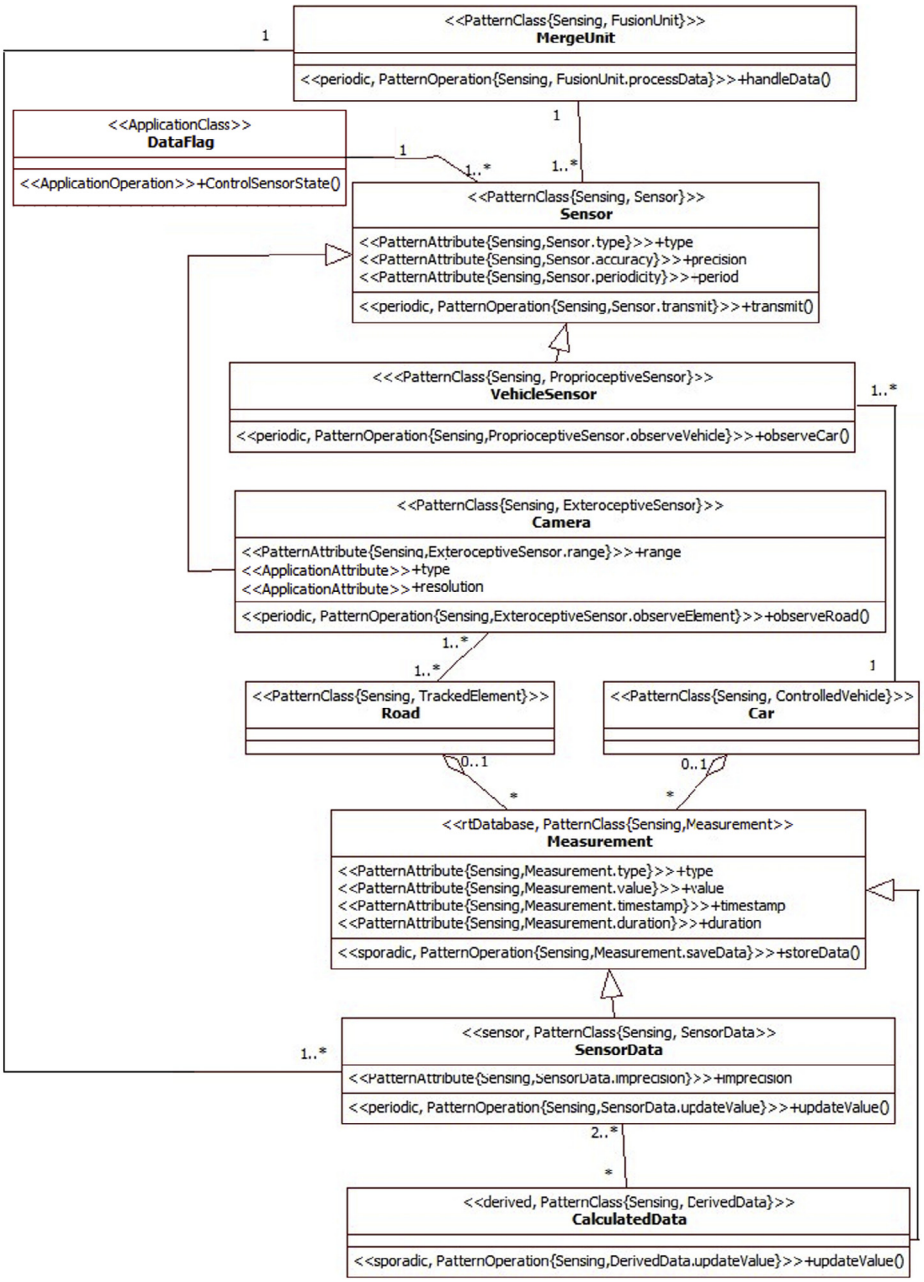

Fig. 12. UML class diagram of LDW system.

few tools used to analyze UML models and OCL constraints, such as USE (UML Specification Environment) (Ziemann and Gogolla, 2003), Papyrus[2] and MagicDraw[3]. The USE tool does not support the UML profiles specification, while Papyrus tool allows the profiles management. These two tools are dedicated to check UML class models with OCL constraints. MagicDraw tool provides extensive support for UML profiles and allows checking UML models (*e.g.* class, sequence and use case models) with OCL invariants. For these reasons, we propose to implement our DP-RTDB profile using MagicDraw UML tool. Indeed, MagicDraw includes all the facilities for defining and applying UML profiles efficiently (*e.g.,* import UML meta-classes and modeling stereotypes). This tool specifies constraints on model elements, constraints on stereotypes and constraints on elements of UML meta-model. In addition, it analyzes the chosen model and evaluates the corresponding OCL expressions on each model element. For example, if a constraint is defined in the context of the meta-class *Class*, it will be evaluated for each class in the model.

When designing our profile, we customized the UML2 diagrams (*i.e.* class, use case and sequence diagrams) by adding the relevant stereotypes defined in the DP-RTDB profile. The creation of the proposed stereotypes is illustrated in Fig. 13.

When the DP-RTDB profile is successfully defined, it is embedded within MagicDraw UML profiles list (Fig. 14) in order to simplify the manipulation of the stereotypes and their related properties.

Afterwards, we specified the proposed OCL constraints for UML elements at the profile meta-model level in order to verify

[2] http://www.papyrusuml.org.
[3] http://www.magicdraw.com.

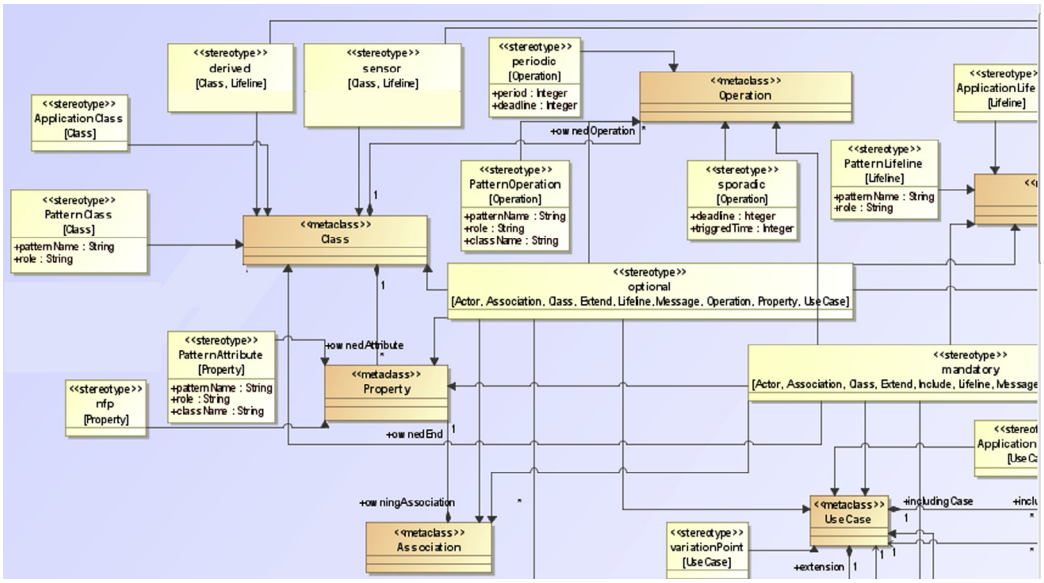

Fig. 13. Creation of the DP-RTDB profile meta-model within MagicDraw UML tool.

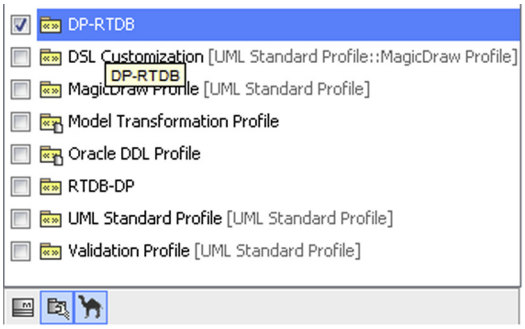

Fig. 14. DP-RTDB profile within MagicDraw UML profiles list.

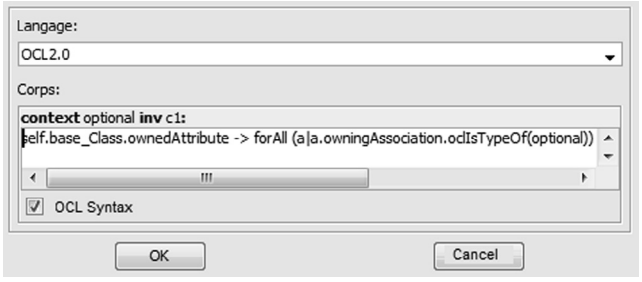

Fig. 15. Example of an OCL constraint.

variation points coherence. Fig. 15 shows an example of an OCL constraint that allows verifying if an association related to an optional class is optional.

Then, we defined several validation rules using OCL constraints defined for covering all the main aspects of model correctness and consistency. These validation rules will be defined in the context of stereotypes. After that, we applied ≪ValidationSuite≫ stereotype to the profile itself to enable using all its validation rules to check the pattern diagrams. Fig. 16 presents a screenshot from MagicDraw UML tool displaying a result of the validation of the UML class diagram of *Sensing* pattern. This validation result is not conforming to the validation rule shown in Fig. 15. In fact, this figure demonstrates that if we have a mandatory class associated to an

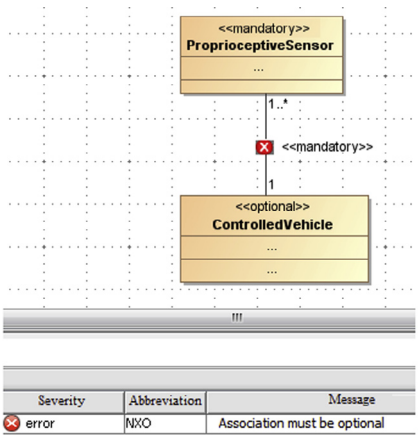

Fig. 16. Screenshot indicating validation error of an OCL constraint.

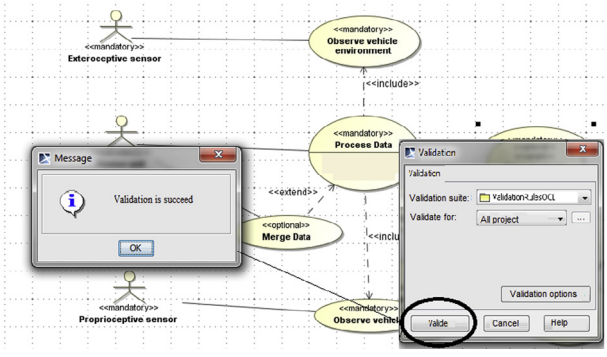

Fig. 17. Screenshot indicating the correctness of the pattern use case diagram.

optional class with an association stereotyped ≪mandatory≫, then an error is displayed. However, Fig. 17 reveals that the pattern use case diagram is consistent and correct since it is conform to the defined OCL constraints. Similarly, we validated the consistency and the correctness of the UML class diagram and the UML sequence diagram of the pattern.

7. Conclusion and future work

Designing RT database applications is a delicate task since data and transactions timing constraints must be taken into account during the design process. Therefore, it is necessary to benefit from the existing reusable design techniques, such as patterns. In order to design more comprehensible and more flexible RT design patterns, we proposed, in this paper, a UML profile for RT design patterns representation. This profile includes various extensions of UML diagrams (*i.e.* class diagram, use case diagram and sequence diagram). These extensions allow distinguishing between the fundamental elements and the variable elements. They also allow identifying easily pattern elements in the system model. Moreover, this profile provides extensions that specify explicitly the RT features of RT database applications. Besides, we introduced some OCL constraints that may be used as validation rules to check the pattern diagrams correctness. Then, we illutrated the defined profile through the specification of a RT design pattern and its instantiation. To develop our profile, we used MagicDraw UML tool. Finally, we checked the correctness and the consistency of each pattern diagram by using a set of OCL constraints.

In our future works, we plan to define OCL constrains for checking the inter-diagrams consistency (*i.e.,* these constraints allow maintaining consistency between the different views describing the pattern). These constraints will be implemented and validated, then added to the DP-RTDB profile. We also plan to build a catalog of RT-DB patterns and develop a methodology to build RT-DB applications using patterns.

References

Amditis, A., Bimpas, M., Thomaidis, G., Tsogas, M., Netto, M., Mammar, S., Beutner, A., Möhler, N., Wirthgen, T., Zipser, S., Etemad, A., Lio, M.D., Cicilloni, R., 2010. A situation-adaptive lane-keeping support system: overview of the SAFELANE approach. IEEE Intelligent Transp. Syst. Soc. 11 (3), 617–629.

Amirijoo, M., Hansson, J., Son, S.H., 2006. Specification and management of QoS in real-time databases supporting imprecise computations. IEEE Trans. Comput. 55 (3), 304–319.

Apvrille, L., Courtiat, J.-P., Lohr, C., Saqui-Sannes, P.D., 2004. TURTLE: A real-time UML profile supported by a formal validation toolkit. IEEE Trans. Software Eng. 30, 473–487.

Arnaud, N., 2008. Fiabiliser la rutilisation des patrons par une approche oriente compltude, variabilit et gnricit des spcifications, Ph.D. thesis, Universit Joseph Fourier-Grenoble I.

Bouassida, N., Ben-Abdallah, H., 2006. Extending UML to guide design pattern reuse. In: Sixth Arab International Conference On Computer Science Applications, Dubai.

Dong, J., Sheng, Y., Zhang, K., 2007. In: Visualizing Design Patterns in Their Applications and Compositions. IEEE Transactions on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 433–453.

Douglass, B.P., 2004. Real Time UML. Addison Wesley Longman Publishing, Redwood City, CA, USA.

E.C.P. release, Road safety: new statistics call for fresh efforts to save lives on EU roads (2016). http://europa.eu/rapid/press-releaseIP-16-863en.htm.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co. Inc, Boston, MA, USA.

Idoudi, N., Louati, N., Duvallet, C., Bouaziz, R., Sadeg, B., Gargouri, faiez, 2008. A Framework to model real-time databases. Int. J. Comput. Inf. Sci. 6 (1), 19–28.

Loo, K.N., Lee, S.P., Chiew, T.K., 2012. UML extension for defining the interaction variants of design patterns. J. IEEE Software 29 (5), 64–72.

Louati, N., Bouaziz, R., Duvallet, C., Sadeg, B., 2012. A UML/MARTE profile for real-time databases. IEEE/ACIS 11th International Conference on Computer and Information Science. IEEE, Shanghai, pp. 664–668.

Marouane, H., Makni, A., Bouaziz, R., Duvallet, C., Sadeg, B., 2012. A real-time design pattern for advanced driver assistance systems. Proceedings of 17th European conference on Pattern Languages of Programs (EuroPLoP 2012). Springer, Kloster Irsee, Germany. C6:1–C6:11.

OMG, 2003. Uml 2.0 OCL specification, available from: https://www.lri.fr/wolff/teach-material/2008-09/IFIPS-VnV/UML2.0OCL-specification.pdf.

OMG, A UML Profile for MARTE (2011). http://www.omg.org/spec/MARTE/1.1/.

OMG, Uml Profile for Schedulability, Performance, and Time Specification, Version 1.1 (2005). http://www.omg.org/cgi-bin/doc?formal/2005-01-02.

Park, J., Lee, S.-W., Rine, D.C., 2013. Uml design pattern metamodel-level constraints for the maintenance of software evolution. Software: Practice Exp. 43 (7), 835–866.

Prestl, W., Sauer, T., Steinle, J., Tschernoster, O., 2000. The BMW active cruise control ACC. SAE Trans. 109 (7), 119–125.

Ramamritham, K., 1993. Real-time database. Int. J. Distributed Parallel Databases 1 (2), 199–226.

Ramamritham, K., Pu, C., 1995. A formal characterization of epsilon serialisability. IEEE Trans. J. Knowl. Data Eng. 7 (6), 997–1007.

Reinhartz-Berger, I., Sturm, A., 2009. Utilizing domain models for application design and validation. Inf. Softw. Technol. 51 (8), 1275–1289.

Rekhis, S., Bouassida, N., Duvallet, C., Bouaziz, R., Sadeg, B., 2010. A UML-profile for domain specific patterns: application to real-time. DE@CAISE'10: the Domain Engineering workshop of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE'10), Hammamet, Tunisia, pp. 32–46.

Rekhis, S., Bouassida, N., Bouaziz, R., Duvallet, C., Sadeg, B., 2013. Domain Engineering: Product Lines, Languages, and Conceptual Models. Springer, Berlin Heidelberg. Ch. Modeling Real-Time Design Patterns with the UML-RTDP Profile, pp. 59–82..

Rumbaugh, J., Jacobson, I., Booch, G., 1999. The Unified Modeling Language Reference Manual, Pearson Higher Education.

Stankovic, J.A., Son, S.H., Hansson, J., 1999. Misconceptions about real-time databases. IEEE Comput. 32 (6), 29–36.

Sunyé, G., Guennec, A.L., Jézéquel, J.-M., 2000. Design patterns application in uml. 14th European Conference Sophia Antipolis and Cannes. Springer, Berlin Heidelberg, pp. 44–62. http://dx.doi.org/10.1007/3-540-45102-13.

T. Tass BV, Prescan help (2012). https://tass-safe.com/en/downloads.

Yin, R.K., 2014. Case Study Research: Design and Methods. In: 5th Edition, SAGE Publications, p. 312.

Ziadi, T., Helouet, L., Jezequel, J.-M., 2003. Towards a uml profile for software product lines. Proceedings of the 5th International Workshop on Product Family Engineering. Springer, Berlin Heidelberg, New York, pp. 129–139. http://dx.doi.org/10.1007/978-3-540-24667-110.

Ziemann, P., Gogolla, M., 2003. Validating OCL specifications with the use tool an example based on the BART case study. Electron. Notes Theor. Comput. Sci. 80, 157–169. http://dx.doi.org/10.1016/S1571-0661(04)80816-8.