Journal of King Saud University – Computer and Information Sciences (2016) xxx, xxx–xxx

King Saud University

Journal of King Saud University – Computer and Information Sciences

www.ksu.edu.sa
www.sciencedirect.com

A new method for constructing and reusing domain specific design patterns: Application to RT domain

Saoussen Rekhis [a,*], Nadia Bouassida [a], Rafik Bouaziz [a], Claude Duvallet [b], Bruno Sadeg [b]

[a] MIRACL, Pôle Technologique de Sfax, BP 242, Sfax 3021, Tunisia
[b] LITIS, UFR des Sciences et Techniques, BP 540, 76 058 Le Havre Cedex, France

Received 18 September 2015; revised 19 April 2016; accepted 20 April 2016

KEYWORDS

Design pattern engineering;
Model transformation;
Real-time application modeling

Abstract Domain specific design patterns capture domain knowledge and provide solutions of non trivial design problems in a specific domain. Their application improves considerably the quality of software design. In order to benefit from these advantages and to reinforce the application of these patterns, we provide, in this paper, new processes and tools for the development and the instantiation of domain specific design patterns, especially those intended for real-time domain.

Initially, we propose a pattern development process that guides pattern developers in the construction of patterns. The proposed process defines unification rules that apply a set of comparison criteria on various applications in the pattern domain. This process is illustrated through the design of the controller pattern. Moreover, we propose a process guiding the application designers in pattern instantiation based on model transformation. Finally, the proposed RT patterns and their development process are evaluated by calculating quality metrics and comparing the applications designed with our RT patterns and others developed by experts without the use of our patterns.

1. Introduction

Design patterns (Gamma et al., 1994) represent solutions to common design problems in a given context. They improve substantially software quality and reduce the development cost. Nowadays, their use is wide spread since they capture and promote best practices in software design. However, the design patterns of GoF (Gamma et al., 1994) do not focus on a particular domain, thus they need a great adaptation effort since it is hard to determine in which context or in which part of the system these patterns can be used (Port, 1998). These reasons motivated several works on domain-specific patterns which encapsulate the essence of a certain domain

* Corresponding author.
E-mail addresses: saoussen.rekhis@fsegs.rnu.tn (S. Rekhis), nadia.bouassida@isimsf.rnu.tn (N. Bouassida), raf.bouaziz@fsegs.rnu.tn (R. Bouaziz), claude.duvallet@univ-lehavre.fr (C. Duvallet), 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.2016.04.004
1319-1578

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

(e.g., human computer interaction, security and real-time systems).

A domain-specific design pattern offers a flexible architecture with clear boundaries, in terms of well-defined and highly encapsulated parts that are in alignment with the natural constraints of the considered domain (Port, 1998). One of the domains where reuse through patterns will bring many benefits is the Real Time (RT) domain since it is a complex and evolving domain. The RT patterns help designers in developing applications that express time-constrained data and time-constrained methods. They provide various solutions to addressing the fundamental RT scheduling, communications and synchronization problems (Boukhelfa and Belala, 2015).

Design patterns specific to the RT domain are similar to any domain specific pattern; they need a representation language that shows the specificities of the domain. They need also a design process that helps in their construction and finally an instantiation process that will guide their reuse (Boukhelfa and Belala, 2015).

Nowadays, representation of domain-specific design patterns and their reuse receives special attention from many researchers (e.g., Kim et al., 2004; Kim, 2007). They proposed modeling languages in order to take into account pattern variability. In fact, when representing domain-specific patterns, the design language, e.g., UML has to support not only the flexibility characteristic of patterns, but also the specificities of the domain itself (Port, 1998). Nevertheless, in the example of real-time (RT) domain, the standard UML remains insufficient for expressing all features of RT applications. Therefore, different extensions to the UML language have been proposed to take into account RT application characteristics (Douglass, 2004; Lanusse et al., 1999; OMG, 2008). However, the proposed modeling languages are not suitable to patterns. That is, RT patterns must be generic designs intended to be specialized and reused by any application in RT domain. For this reason, in addition to the UML extensions representing RT aspects, we need new notations distinguishing the commonalities and differences between applications in the pattern domain.

On the other hand, the difficulty of the domain specific pattern development and specification slows their expansion. This is due essentially to the fact that they have to incorporate flexibility and variability in order to be instantiated for various applications in the domain. As a result there is a need for a design process that guides the domain-specific patterns development and defines rules to find similarities between a set of applications in the considered domain and their possible variations. This need is crucial in the RT domain since it is an evolving domain where the variety of applications is quite large and the reuse is very important. In RT domain, we distinguish the case where applications use a lot of RT data that must be stored in a database. We call this case "real-time domain with intensive data". Our contribution aims to guide the development of RT design patterns specific to RT domain with intensive data through the definition of unification rules that facilitate their specification.

Finally, note that, even when assisting the pattern developers in expressing and building design patterns, there is no certitude that these patterns would be correctly instantiated, by application designers. Thus, an ultimate assistance in validating the pattern instantiation would be of a valuable benefit. Several researchers (e.g., Kim and Carrington, 2006; Kajsa, 2013; Hammouda et al., 2009; Koskinen et al., 2010) were interested in the validation of pattern instances, however their approaches are not adapted to the RT domain. They do not take into account the UML extensions differentiating between passive resources and RT active resources and specifying RT features such as concurrency and deadline. As a consequence, there is still a need for an efficient guidance process for RT patterns reuse.

In our approach, we are interested in providing assistance, not only for pattern designers to support RT design pattern representation and development, but also for application designers to instantiate RT design patterns. This assistance is provided through:

(1) A UML profile intended for RT design pattern specification and reuse. This profile establishes a relation between the different systems that need similar facilities and provides a good ground to build on in order to establish a real pattern language for RT systems.
(2) A pattern development process using unification rules to determine the fundamental elements and the variation points of a pattern. This process facilitates the pattern developers' work, in order to specify patterns with a better quality. It addresses the structural and behavioral aspects describing how the different roles of a pattern interact.
(3) A guidance process for building and validating applications reusing patterns. When a pattern is deployed in an application design, some constraints must be preserved to make the pattern instantiation valid. The pattern instantiation process guides the application designers in patterns reuse and prevents them from violating pattern constraints.

The first contribution of this paper consists in showing how our pattern development process, initially presented in Rekhis et al. (2010), can be improved, fine-tuned and automated in order to define the structure of RT design patterns. Moreover, in this paper the process was augmented with the specification of the behavior of RT design patterns. A second contribution of this paper consists in proposing a new pattern instantiation process guiding the application designers in RT patterns reuse and preventing them from violating pattern constraints. We implement the instantiation process using an existing modeling framework, EMF, and incorporate the implementation as plug-in to the Eclipse modeling environment. The developed plug-in can interpret automatically the properties of the patterns since they are described in a precise manner using the UML-RTDP profile (Rekhis et al., 2013) that we defined previously. A third contribution of this paper consists in defining new metrics and using the CK metrics proposed in Chidamber and Kemerer (1994) to assess experimentally the efficiency of our design process. For the evaluation of the RT patterns obtained with our proposed development process, we propose some projects to experimented designers, while dividing them into two groups, one group models RT applications using our patterns approach, and the other will not use them. Then, we calculate some quality metrics for the produced projects (e.g., number of classes, average number of attributes, etc). The evaluation answers two questions: do the RT patterns have a good design quality? And do the RT patterns encapsulate really the concepts tied to RT domain?

The remainder of the paper is structured as follows. Section 2 discusses related work. Section 3 presents the pattern

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

development process. Section 4 proposes a process guiding the reuse and the validation of domain specific design patterns, and describes its supporting CASE tool we have developed. Section 5 presents the evaluation of the proposed RT patterns. Section 6 concludes the paper and discusses future works.

2. Related work

In this section, we firstly present the previously proposed profile (Rekhis et al., 2013) that extends UML 2.2 with new stereotypes representing design patterns for RT domain. Secondly, we present, briefly, works interested in the development of patterns through the unification of existing applications or through domain analysis. Finally, we present works that provide support and guidance for design patterns instantiation.

2.1. Design patterns representation with UML-RTDP

In our previous work (Rekhis et al., 2013), we have proposed a UML profile, named UML-RTDP, including MARTE stereotypes describing RT features. This profile facilitates the pattern comprehension and instantiation thanks to the stereotypes described below:

- The ⟨⟨***optional***⟩⟩ stereotype allows representing an optional element (i.e., class, association, attribute, method, interface, association Class).
- The ⟨⟨***mandatory***⟩⟩ stereotype allows representing a fundamental element that must be instantiated at least once.
- The ⟨⟨***variable***⟩⟩ stereotype indicates that the method implementation varies according to the pattern instantiation.
- The ⟨⟨***extensible***⟩⟩ stereotype indicates that a pattern class may be extended by adding new attributes and/or methods.
- The ⟨⟨***patternClass***⟩⟩ stereotype is used to differentiate between the instantiated pattern classes and the original classes added by the designers in an application model and to check the existence of any conflicts with pattern properties.

In addition to the above stereotypes, the specification of RT design patterns needs the use of UML extensions to model RT aspects. Thus, we import from MARTE profile stereotypes which deal with quantitative and qualitative features related to behavior, communication and concurrency.

From HLAM (High Level Application Modeling) sub-profile, we import the following stereotypes:

- The ⟨⟨***rtFeature***⟩⟩ (real-time feature) stereotype is used to model temporal features such as deadline.
- The ⟨⟨***ppUnit***⟩⟩ (protected passive Unit) stereotype is used to model the shared data requiring the specification of concurrency policy. Protected passive units specify their concurrency policy either globally for all their provided services through their concPolicy attribute, or locally through the concPolicy attribute of ⟨⟨***RtService***⟩⟩ stereotype.
- The ⟨⟨***rtUnit***⟩⟩ (real-time Unit) stereotype models a real-time unit that may be seen as an autonomous execution resource, able to handle different messages at the same time. A real-time unit can manage concurrency and real-time constraints attached to incoming messages.
- The ⟨⟨***rtService***⟩⟩ (real-time service) stereotype is used to specify the services concurrency policy (reader, writer or parallel) provided by real-time units and protected passive units.

Moreover, we import from NFP (Non Functional Properties) sub-profile two stereotypes: ⟨⟨***Nfp***⟩⟩ and ⟨⟨***NfpType***⟩⟩. ⟨⟨***Nfp***⟩⟩ stereotype describes the attributes satisfying non functional requirements. ⟨⟨***NfpType***⟩⟩ stereotype specifies a composite type that contains value of an attribute and its unit. There is a set of pre-declared *NFP_Types* which are useful for specifying NFP values, such as NFP_Duration and NFP_Frequency.

In Fig. 1, we illustrate the usage of our UML-RTDP profile through the design of RT sensor pattern, where mandatory and optional elements are highlighted. As shown in Fig. 1, there are three fundamental classes: the *sensor* class, the *measure* class and the *observedElement* class. The sensors are classified into passive or active. An active sensor takes the transmission initiative of its current value (push mechanism). While a passive sensor transmits its value only on demand (pull mechanism). Depending to the designer's problem in a specific problem situation, the designer may decide which implementation is suitable to solve the problem he/she is working on.

The *Measure* class models the RT data, which are classified into either base data or derived data. Base data are issued from sensors, whereas derived data are calculated from base data. The derived data have the same characteristic of base data (timestamp, validity duration,...). The relation between base and derived RT data is represented by an optional and reflexive association defined on the *Measure* class. This class is stereotyped ⟨⟨***ppUnit***⟩⟩ since it represents a passive resource, which needs to be set and controlled by active resources (like controller). It provides a RT service called updateValue. This operation carries the concurrency kind (writer) and the execution kind (remoteImmediate) indicating that the execution is performed immediately with the called active object (Controller). Besides, the active controller resource creates dynamically schedulable resources to handle the execution of its services needing to be achieved before a deadline.

2.2. Overview of development processes

In the literature many researchers were interested in pattern extraction and not like in our case patterns construction. Their approaches were based on graphs (e.g., Tsantalis et al., 2006; Liamwiset et al., 2013), other works used XML (e.g., Satvinij and Vatanawood, 2011; Bouassida et al., 2013), and others were based on ontology (e.g., Alnusair and Zhao, 2010; Robles et al., 2012). However, they are different from our context since they are related to reverse-engineering context. (e.g., Tsantalis et al., 2006; Liamwiset et al., 2013). Moreover, all these approaches are based on the structures of the expected design patterns. That is the UML class diagrams of design patterns are the input of the automatic tool used to extract the patterns from the analyzed applications models. Our contribution consists in determining the structure and the behavior of design patterns specific to a particular domain.

On the other hand, we note that existing development processes of reusable elements (e.g., frameworks, patterns) can be classified into either bottom-up or top-down. A bottom-up

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

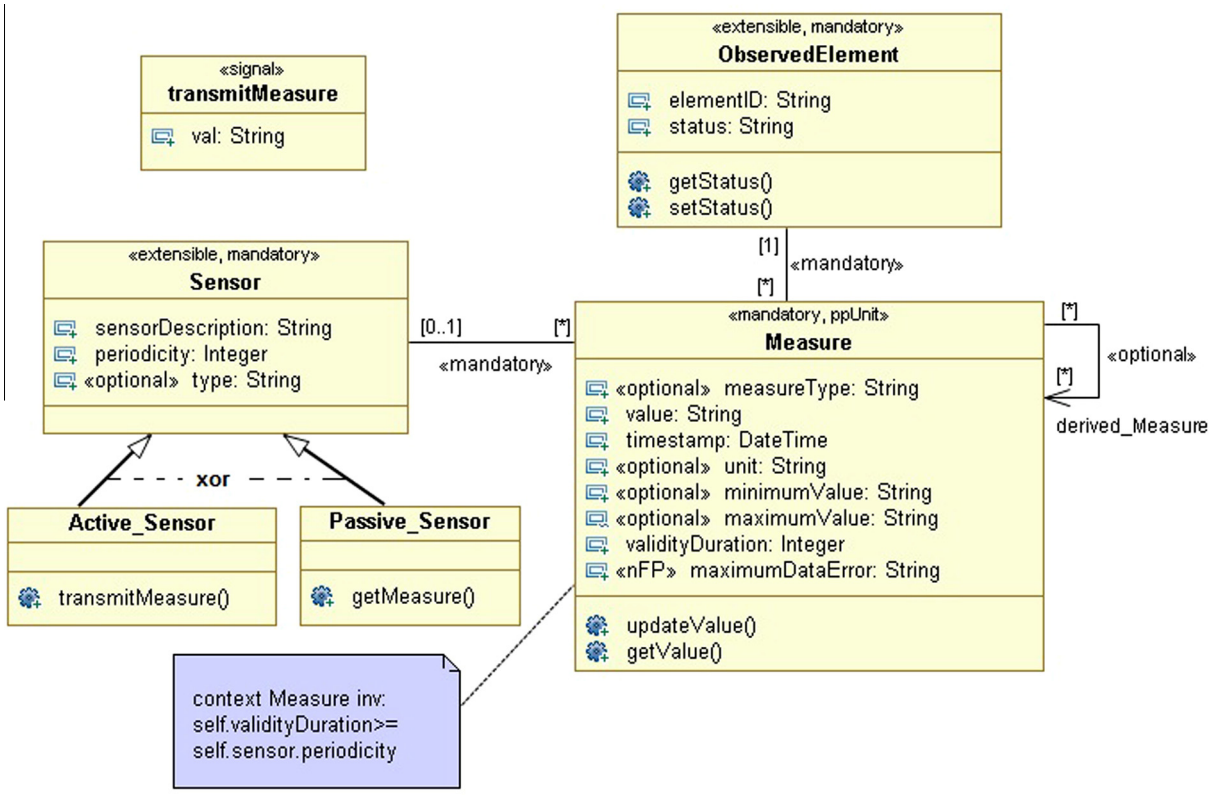

Figure 1 RT sensor pattern.

process starts from a set of applications representing the domain and identifies their common and variable elements. The purpose of examining sample applications is to create a generic reusable component that is understandable and easy to reuse. The bottom-up design process works well when a domain is already well-understood, for example, after some initial evolutionary cycles (Ben-Abdallah et al., 2004). However, there is no guarantee that all domain requirements are met. Moreover, this process is known to be highly iterative, thus it becomes not efficient in the case of large or complex applications design.

A top-down process starts from a domain analysis and then constructs the reusable element. As a result, the design is driven starting from functional requirements toward solution alternatives. Top-down development processes represent the best solution when the domain has not yet been sufficiently explored (Ben-Abdallah et al., 2004). However, this type of processes is time consuming and it lacks guidelines for the domain requirements analysis. As an example, Raminhos et al. (2006) and Caeiro et al. (2004) describe the steps a developer must follow, in order to build a pattern through domain requirements analysis. However, the proposed processes do not provide an efficient assistance for pattern development. In addition, they do not guide the developer in finding the pattern fundamental and variable elements.

To conclude, we think that it is necessary to define a process that merges both the bottom-up and the top-down processes. The pattern development process has to guide designers in capturing domain requirements and in determining common properties of application models belonging to a specific domain in order to create design patterns.

2.3. Overview of patterns instantiation and validation works

Several works have been interested in providing assistance for patterns instantiation and validation (e.g., Kim and Carrington, 2006; Kajsa, 2013). They define the necessary steps for the instantiation of patterns and, then, check the validity of the resulting application. The validation of the reusable components is accomplished, either, through the use of a formal specification or through model transformation.

Recently, Kajsa (2013) has proposed a process for design pattern instantiation. This process consists in defining (i) a UML profile and (ii) model transformations based on semantics. This profile proposes new stereotypes to depict the design pattern participants and their relations in a specific application model. Model transformations support and automate the deployment of design patterns into specific applications models. This approach requires the definition of appropriate stereotypes for each design pattern. Thereby, the developer must know the different stereotypes related to each applied design pattern and must also understand the semantic of each stereotype meta-attributes (i.e., tagged values).

Kim and Carrington (2006) proposed a formal approach, based on Object-Z (Graeme, 2000), which allows defining and reusing design patterns. They formalized the role metamodel that constitutes the modeling language used to define design patterns and consistency constraints. The role metamodel defined in Object-Z is automatically transformed to an Ecore model and then implemented using the Eclipse Modeling Framework (EMF) (Steinberg et al., 2008). These authors used also Object-Z to formalize the binding meta-

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

model that maps pattern entities to application model entities and to define the constraints that must be preserved to make the pattern deployment valid.

3. Development process for RT design patterns

Our process guides the development of domain-specific patterns, in general, and it is particularly adapted to the RT design patterns specific to RT domain with intensive data. It adopts a top-down approach that allows the identification of domain requirements, on one hand, and a bottom-up approach that generates patterns from a given set of applications using unification rules, on the other hand. The generated patterns are represented with the UML-RTDP profile.

The design pattern development process is decomposed into five main phases as shown in Fig. 2. It is illustrated through the modeling of RT controller pattern. This illustration is provided in Appendix A.

3.1. Identification of domain functionalities

The first step aims to delimit the domain boundaries and it consists in identifying the most important domain sub-problems. Each sub-problem has one main functional goal to achieve, called *domain functionality*. The most important functionalities related to the domain are identified through the collection of information brought by experts and stakeholders. The functionalities identified in this step help the developers to determine the context of each created pattern.

In RT domain with intensive data, all applications share a common behavior: they monitor and control an environment via values reported by sensors. By examining these applications, we distinguish three main domain functionalities: (i) acquisition of RT data from the environment via sensors, (ii) RT data analysis and production of results within time constraints, and (iii) sending of orders to the environment via actuators. For each functionality, we define a reusable design pattern that captures RT domain knowledge and design expertise. We focus in this paper, only, on the modeling of data control functionality through the definition of the RT controller pattern.

3.2. Identification of requirements

This step consists of refinement of the domain functionalities. Each identified functionality is decomposed iteratively into functions until reaching a level at which the functions become elementary. This step identifies all the domain concepts and constraints associated to each functionality. The functional domain requirements at this step are represented as a pair $\langle F, Ef \rangle$ where F is a domain functionality and Ef is an elementary function.

a. The RT data acquisition functionality is decomposed into four elementary functions:
   Ef_1 {the sensors (e.g., radar, camera,...) observe physical elements in the environment, e.g., an aircraft, a vehicle, a patient, and so on}.
   Ef_2 {the sensors acquire measures (e.g., speed, temperature, pressure, pulse,...)}.
   Ef_3 {the active sensors transmit periodically the acquired measures to the compute unit in order to be stored in a RT database (push mechanism)}.
   Ef_4 {the passive sensors are solicited periodically to transmit the acquired measures to the compute unit in order to be stored in a RT database (pull mechanism)}.Note that Ef_3 and Ef_4 are the two possible alternatives of RT data transmission. Moreover, a set of domain concepts, such as Sensor, Active Sensor, Passive Sensor, Measure, and Observed Element, are identified during the refinement step. Active Sensor and Passive Sensor are domain concepts that represent a variation of a general concept which is Sensor. These concepts will be used in the next step, which is decomposition.
b. The RT data control functionality is decomposed into five elementary functions:
   Ef_1 {a controller monitors the state of each observed element}.
   Ef_2 {a controller calculates derived value (i.e., derived data) from captured value (i.e., base data)}.
   Ef_3 {a controller checks if an action misses its deadline}.
   Ef_4 {a controller monitors the captured values of an observed element to check if a measure's value is between the minimum and the maximum values defined for this data}.
   Ef_5 {a controller initiates corrective actions when it detects abnormal situation}.
   Ef_6 {a controller notifies the surveillance operator of any detected abnormal situation}.

Note that Ef_2, Ef_3, Ef_4, Ef_5 and Ef_6 are optional functions that describe the possible scenarios of RT data control functionality. The refinement of the data control functionality allows the identification of domain concepts, such as Controller, Observed Element, and so on.

3.3. Decomposition of applications

This step goes through the decomposition of the different applications according to the already identified domain concepts. It aims to determine the relations between the classes of the application and the domain concepts related to the different functionalities according to the following two Decomposition Rules (DRi). The first rule adds to the different applications fragments the classes related to the domain concepts. The second rule adds to the different fragments the other classes (i.e., which are not related to the domain concepts).

DR1. For each class $C \in \{CA_i\}$, if the class name is identical or synonymous to a domain concept $CD \in \{CD_j\}$, then the class C is transferred to the fragment F_{ij}, where

$\{CA_i\}$ is the set of classes belonging to the application i.
$\{CD_j\}$ is the set of domain concepts relative to the functionality j.
F_{ij} is the fragment of application i relative to the functionality j.

If the application class C is not a synonym to any domain concept, then the pattern designer has to verify if C plays the role of a domain concept $CD \in \{CD_j\}$ (e.g., the *RoadSegment* class of COMPASS system plays the role of *ObservedElement*

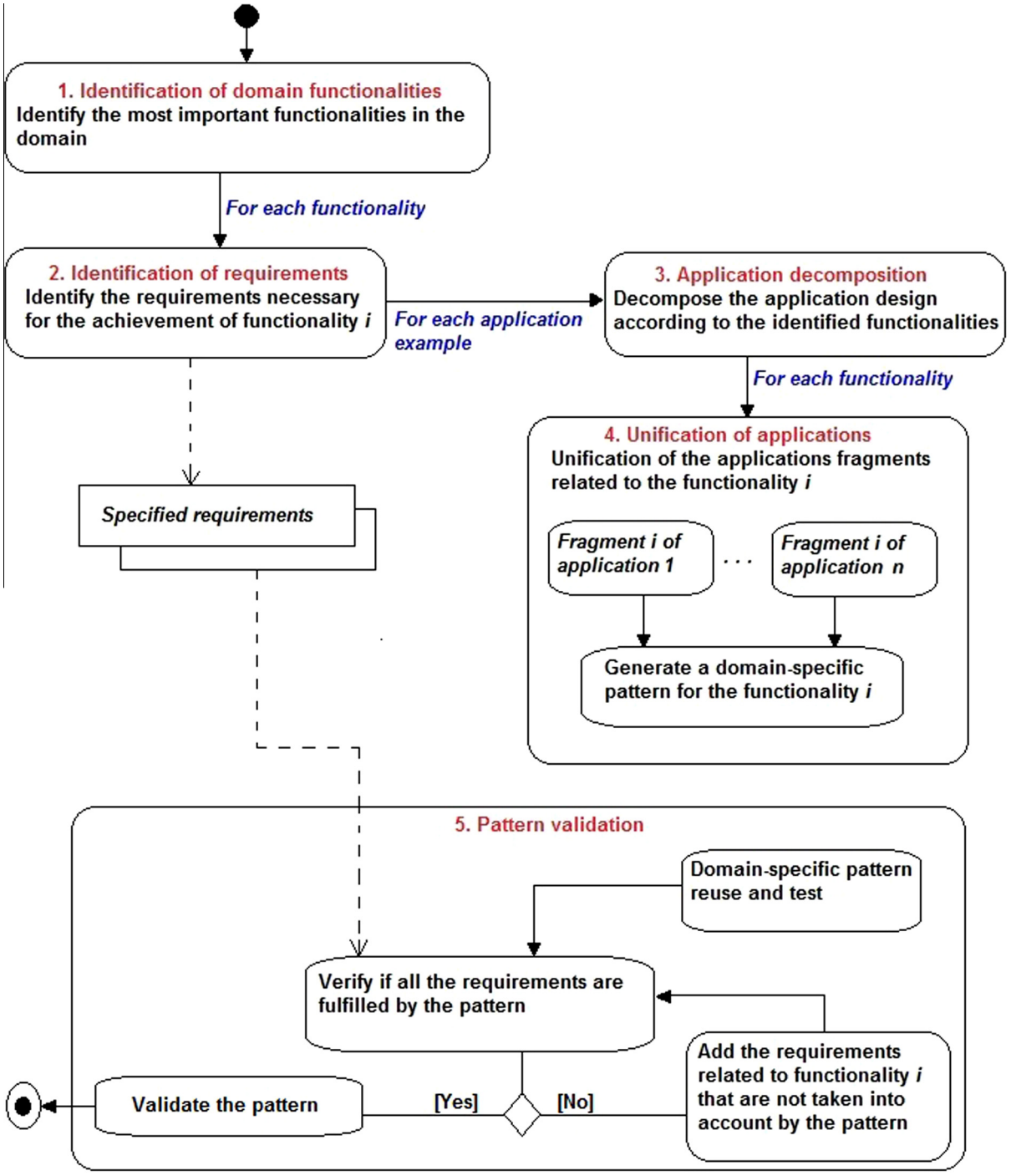

Figure 2 Pattern development process.

domain concept). In this case, the class C is added to the fragment F_{ij}.

DR2. For each pair of classes $(C, C') \in \{CA_i\}$, if $C \in F_{ij}$, $C' \notin F_{ij}$ (i.e., C' is not related to any domain concept) and there is an association between C and C', then the class C' is transferred to the fragment F_{ij}.

The decomposition phase generates an initial version of a domain dictionary containing the class names which are synonymous or equivalent to RT domain concepts. It contains also the class names which are identified by the designer as playing the role of the domain concept.

3.4. Unification of application designs

The unification rules allow the unification of the different application fragments and, then, the generation of RT design patterns. The unification consists in finding the common classes of all the applications and deriving the fundamental elements of the patterns. Then, the classes specific to the applica-

tions are extracted as variable elements. The unification rules are based on semantic comparison criteria. These latter rely on linguistic relationships to define semantic equivalence and variation between class, attribute and operation names. The determination of the linguistic/semantic relations is handled through either the lexical database WordNet or the domain dictionary that we have created (cf. Fig. 3). Designers are asked to verify whether the names of model elements (i.e., classes, attributes, operations, etc.) for which there is no linguistic relations according to the WordNet dictionary, can be synonyms, antonyms or hyponyms in the RT domain.

The unification rules of the class diagrams have already been defined in an initial form in a previous work (Rekhis et al., 2010). In this section we improve them and present new unification rules which are relevant to sequence diagram unification.

Before the unification of applications fragments, a pretreatment step for class names, attribute names and operation names is necessary. For example, if the name of an attribute is composed of several words separated by dashes or containing words whose first letter is capitalized, we propose to treat each word separately. This allows to check if there is a linguistic relationship between the words that compose the attribute name with the words of other attribute names. For example, the decomposition of the attribute names *VehicleStatus* and *SegmentStatus*, belonging respectively to the classes *Vehicle* and *Segment*, allows the identification of two identical words (*Status*). Therefore, these two attributes are treated as two equivalent elements since the words *Vehicle* and *Segment* are related to the class names containing these attributes.

3.4.1. Unification of class diagram fragments

The comparison criteria of class names, attribute names and operation names use a set of linguistic relations. The class name comparison criteria consist of the following three relations:

- N_equiv($CA_{1j},\ldots,CA_{nj}$) means that the names of the classes are either identical or synonymous.
- Note that the class C in the fragment j of an application Ai is represented by CA_{ij} where the fragment j is the part of application model relative to functionality j.
- N_var($CA_{1j},\ldots,CA_{nj}$) means that the names of the classes are a variation of a concept, e.g., mobile-sensor, passive-sensor, active-sensor.
- N_dist($CA_{1j},\ldots,CA_{nj}$) means that none of the above relations holds.

The attribute comparison criteria use the following three relationships to compare the attribute names and types:

- Att_equiv($CA_{1j},\ldots,CA_{nj}$) means that the classes have either identical or synonymous attribute names with the same types.

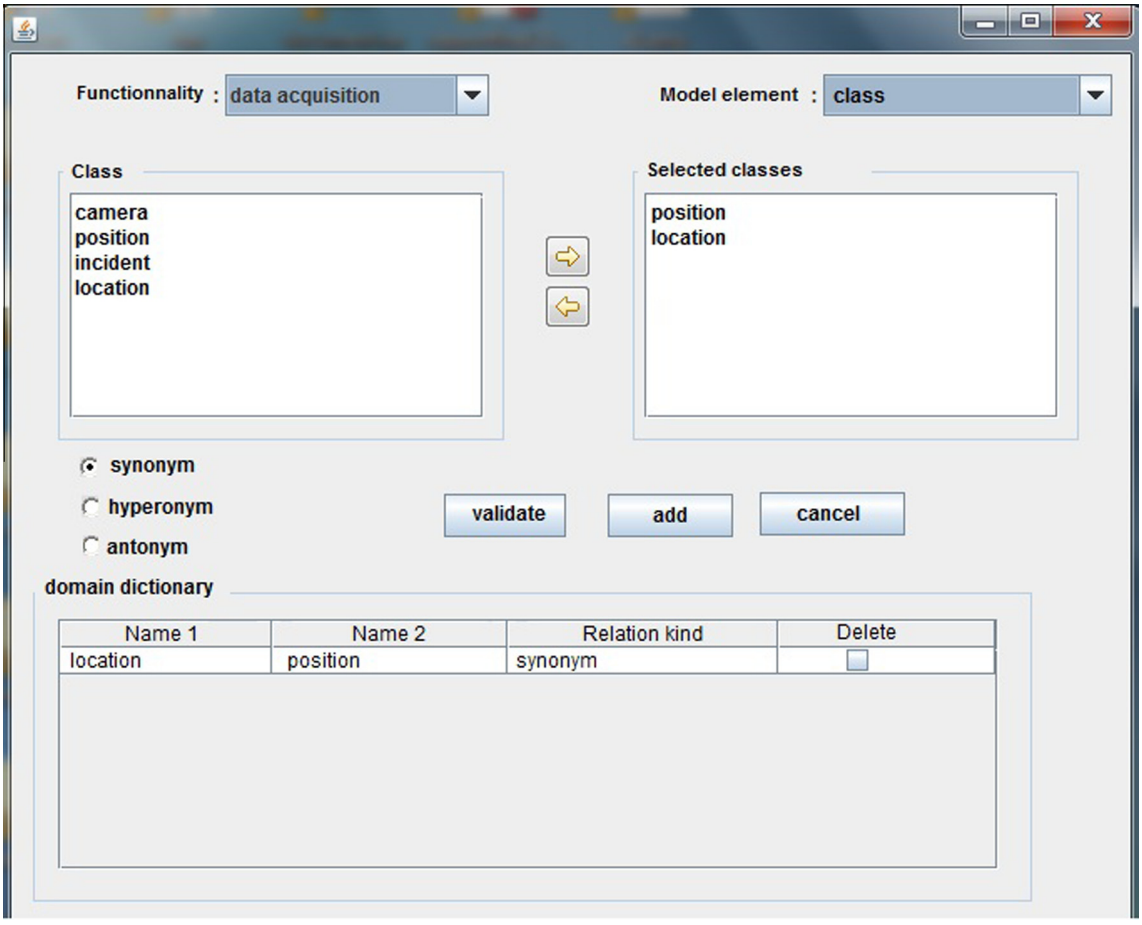

Figure 3 Domain dictionary creation.

- Att_int($CA_{1j},\ldots,CA_{nj}$) means that the classes $CA_{1j},\ldots,CA_{nj}$ have common attributes.
- Att_dist($CA_{1j},\ldots,CA_{nj}$) means that none of the above relations holds.

The operation comparison criteria use three relations (Op_equiv($CA_{1j},\ldots,CA_{nj}$), Op_int($CA_{1j},\ldots,CA_{nj}$), and Op_dist($CA_{1j},\ldots,CA_{nj}$)) to compare the operation names and signatures (parameter types and returned types). These relations are defined in a similar manner to attribute comparison relations.

In order to ease the understanding of the unification rules, we define, in the following, some new concepts:

- A model element (class, attribute, method or message) is pertinent to RT domain if it is present in more than 50% of the applications. For this purpose we define the domain coverage ratio (*Rdc*) which is computed as follows:

$$Rdc(E) = \frac{\text{Number of occurrences of a model element } E \text{ in the applications}}{\text{Number of applications}}$$

Note that if domain coverage ratio of a model element E is less than ø, then this element is not pertinent to RT domain and it is too application-specific. Thus, if it is added to the pattern, it may complicate unnecessarily the pattern comprehension. The arbitrary threshold value (50%) could be changed by the pattern developers according to their need. Note that, this threshold will be determined in a future work thanks to experiments that can help to find the ideal threshold.

- A model element (class or object) plays a role of a domain concept if in the decomposition phase it has been identified by the designer as playing a role of a domain concept (observed element, sensor, etc.) and it has already been stored in the domain dictionary.

The design of a pattern class diagram is guided by the Unification Rules (URi):

UR1. If a set of classes $\{CA_{1j},\ldots,CA_{nj}\}$ are present in all the applications with equivalent attributes, i.e., Att_equiv ($CA_{1j},\ldots,CA_{nj}$), and methods, i.e., Op_equiv($CA_{1j},\ldots,CA_{nj}$), then a class is added to the pattern as a fundamental class stereotyped ⟪***mandatory***⟫.

UR2. If a set of classes $\{CA_{1j},\ldots,CA_{nj}\}$ share equivalent attributes and/or methods, i.e., Att_int($CA_{1j},\ldots,CA_{nj}$) and/or Op_int($CA_{1j},\ldots,CA_{nj}$), then there are three cases:

Case 1: If N_equiv($CA_{1j},\ldots,CA_{nj}$) is held, then a fundamental class is added to the pattern with common attributes and methods. The distinct attributes and/or methods are added as optional elements if they are pertinent to the domain.

Case 2: If N_var($CA_{1j},\ldots,CA_{nj}$), then a class with common attributes and methods is added to the pattern as a fundamental class and two cases are possible:

1. If there exists a sub set from the classes ($CA_{1j},\ldots,CA_{nj}$) that play the same role of domain concepts (e.g., active sensor, passive sensor) and that constitute a variation of a general domain concept (e.g., sensor), then a set of sub-classes inheriting from the fundamental class and containing the pertinent attributes and/or methods is added.For example, the classes *CameraSensor*, *InductanceLoopSensor*, *BoundaryStickSensor* and *ActimetrySensor* presented in Fig. 4 share common attributes and methods and their names represent a variation of a concept. *CameraSensor* and *BoundaryStickSensor* play the role of the domain concept *passive sensor* whereas *ActimetrySensor* and *InductanceLoopSensor* play the role of the domain concept *active sensor*. Since active sensor and passive sensors represent a variation of the domain concept *sensor*, then two subclasses inherited from the super class named sensor are added.
2. If there does not exist a sub set from the classes ($CA_{1j},\ldots,CA_{nj}$) playing the same role of a domain concept, then the pertinent attributes and/or methods existing in the classes ($CA_{1j},\ldots,CA_{nj}$) are added to the fundamental class as optional elements.For example, the classes *WaterController*, *PatientController* and *TrafficController* presented in Fig. 5 share common methods and their names represent a variation of one concept which is *Controller*. Then a class *Controller* is added to the pattern with *notify()* method as optional element.

Case 3: If N_dist($CA_{1j},\ldots,CA_{nj}$) and the classes ($CA_{1j},\ldots,CA_{nj}$) play a role of the same domain concept, then a fundamental class is added to the pattern with common attributes and methods and it will take the name of the domain concept. In this case, the two possible alternatives explained in case 2 of this rule are also applied. For example, the classes RoadSegment, Vehicle and Patient play the same role which is ObservedElement. These classes share common attributes and methods. Therefore, a class named ObservedElement is added to the pattern.*Note*: For all the cases of rule UR2, if there exists distinct attributes and/or methods that are not pertinent for the considered domain, then the fundamental class is added to the pattern with the common attributes and/or methods and it is stereotyped ⟪***extensible***⟫ in order to indicate that the class can be extended when the pattern is reused.

UR3. If there is a set of classes which are not common to all applications, they share equivalent attributes and/or methods and they are pertinent to RT domain, then an optional class is added to the pattern with common attributes and methods.

UR4. If a method exists in all the applications with the same name but with different signatures, then it will have a corresponding method in the pattern with an undefined signature and it is stereotyped ⟪***variable***⟫.

UR5. If an attribute exists in all the applications with the same name but with different types, then there are two cases:

Case 1: If these types are compatible (real, integer, etc.), then it will have a corresponding attribute in the pattern with the most general type.

Case 2: If these types are not compatible, then it will have a corresponding attribute in the pattern with the

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

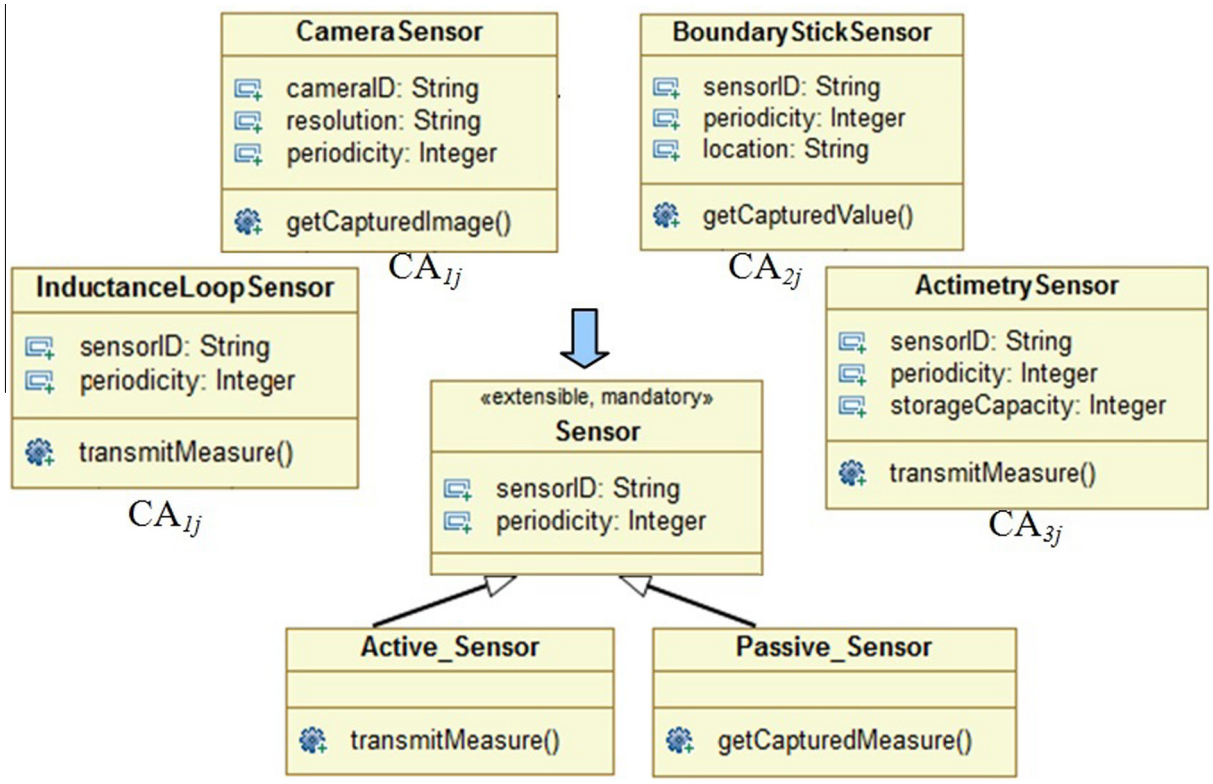

Figure 4 Example of classes representing a variation of concepts with inheritance relationship.

same name and the enumeration type which includes different types of conflict attributes.

UR6. If two or more classes are transferred in the pattern, then all their relations (aggregation, inheritance, association) will be maintained in the pattern.

3.4.2. *Unification of sequence diagrams*

The unification rules are applied to identify the common elements as well as the variable elements between different sequence diagrams, relative to the same domain functionality.

The unification of sequence diagrams is based on linguistic comparison criteria for objects and messages names. The comparison criteria of objects are similar to those of classes. They are based on the following three relations:

- N_equiv($OA_{1j},\ldots,OA_{nj}$) means that the names of the classes relative to the objects ($OA_{1j},\ldots,OA_{nj}$) are either identical or synonymous.
- Note that the object O in the sequence diagram j of an application Ai is represented by OA_{ij} where the sequence diagram j is relative to functionality j.

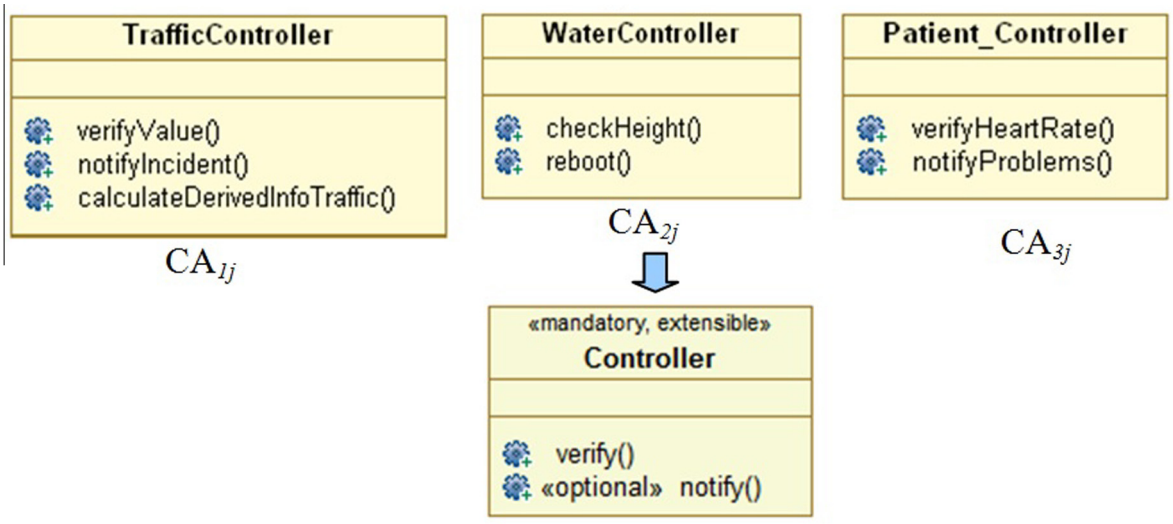

Figure 5 Example of classes representing a variation of a concept.

- N_var($OA_{1j},\ldots,OA_{nj}$) means that the names of the relative to the objects ($OA_{1j},\ldots,OA_{nj}$) are a variation of a concept.
- N_dist($O_{A1j},\ldots,O_{Anj}$) means that none of the above relations holds.

The message comparison criteria use the following two relationships:

- N_equiv($MA_{1j},\ldots,MA_{nj}$) means that the messages have either identical or synonymous names.
- N_dist($MA_{1j},\ldots,MA_{nj}$) means that messages have distinct names.

The design of a RT design pattern sequence diagram is guided by the following unification rules:

UR7. If there is a set of objects $\{OA_{1j},\ldots,OA_{nj}\}$ instantiating equivalent classes N_equiv($OA_{1j},\ldots,OA_{nj}$), then an object is added to the pattern sequence diagram as a fundamental element and it must be stereotyped ⟪***mandatory***⟫.
UR8. If there is a set of objects $\{OA_{1j},\ldots,OA_{nj}\}$ instantiating different classes playing the same role of a domain concept, then an object is added as a fundamental element stereotyped ⟪***mandatory***⟫ whose type corresponds to the domain concept.
UR9. If there is a set of objects $\{OA_{1j},\ldots,OA_{nj}\}$ whose types are a variation of a general concept N_Var($OA_{1j},\ldots,OA_{nj}$), then there are two cases:
Case 1: If these objects are instances of a class C in the pattern class diagram, then an object is added as a fundamental element stereotyped ⟪***mandatory***⟫ whose type corresponds to the class C.
Case 2: If these objects are instances of several subclasses inheriting from a general class and each of which has its own operations in the pattern class diagram, then the objects corresponding to these specialized classes are added to the pattern sequence diagram as optional items that are stereotyped ⟪***optional***⟫.
UR10. If there is a set of objects which are not common to all applications, they have equivalent types and they are pertinent to the RT domain, then an object is added to the pattern as an optional element stereotype ⟪***optional***⟫.
UR11. If there is a set of messages $\{MA_{1j},\ldots,MA_{nj}\}$ having equivalent names N_equiv($MA_{1j},\ldots,MA_{nj}$) for which the sender objects and receiver objects have been already transferred in the pattern sequence diagram, then there are two cases:
Case 1: If these messages are common to all applications, then a fundamental message is added to the pattern sequence diagram.
Case 2: If these messages are not common to all applications but they are pertinent to RT domain, then an optional representative message is added to the pattern. This message belongs to an optional combined fragment having *opt* operator.
UR12. If there is a set of messages $\{MA_{1j},\ldots,MA_{nj}\}$ sent by objects ($OA_{1j},\ldots,OA_{nj}$) to other objects ($OA'_{1j},\ldots,OA'_{nj}$) such that N_Var($OA_{1j},\ldots,OA_{nj}$) or N_Var($OA'_{1j},\ldots,OA'_{nj}$), then two cases are possible:
Case 1: If the sending objects (respectively receiving objects) are transferred according to the rule UR9 (case 1), then one message is added to the pattern as fundamental message if it is common to all applications. Else, if this message is not common to all applications but it is pertinent to RT domain, then it is added to the pattern sequence diagram in an optional combined fragment.
Case 2: If the sending objects (respectively receiving objects) are transferred according to the rule UR9 (case 2), then the pertinent messages are added to the pattern in a combined fragment with the alternative operator *alt*.

3.5. Pattern validation

The last step of the pattern development process checks if all the requirements and constraints are fulfilled by the generated patterns, in order to validate them. This validation is performed in two steps. First, the obtained RT patterns are instantiated and confronted with the original application fragments. Second, the completeness of the patterns is verified by checking if all the requirements and constraints of the different domain functionalities are fulfilled by the obtained patterns.

In the RT domain, the pattern validation phase is very important since it defines additional constraints and dependencies expressed with OCL. It allows also to add stereotypes modeling RT features as well as properties supporting Quality of Services (QoS) constraints. In this paper, we define the following five Validation Rules (VRi) to guide the RT pattern developers, when adding the RT stereotypes:

VR1. Each property that provides information about non functional requirements (e.g., throughput, delays or scheduling policies) has to be stereotyped ⟪***Nfp***⟫.

VR2. Each RT method whose execution must be achieved before a deadline has to be stereotyped ⟪***rtFeature***⟫.

VR3. Each passive class that includes shared RT data has to be stereotyped ⟪***ppUnit***⟫.

VR4. Each active class having its own execution resources and handling simultaneously various messages has to be stereotyped ⟪***rtUnit***⟫.

VR5. Each method belonging to a class stereotyped ⟪***ppUnit***⟫ or ⟪***rtUnit***⟫ and that has timing constraints to satisfy requires to be stereotyped ⟪***rtService***⟫.

4. The instantiation process for RT design patterns

We present, in this section, the instantiation process for RT design patterns and its associated tool based on model transformation. This process has as input the RT design patterns generated with our development process and it generates the class diagram of one applications instantiating the RT patterns, as shown in Fig. 6. The instantiation process consists of two steps: the mapping step and the transformation step. The first step defines the mapping model that links design pattern elements to application elements and, then, validates the application model against the pattern model. The transforma-

tion step generates automatically the application model deploying the instantiated patterns. Thus, the main objective of this proposal is to use model transformation engineering techniques in order to create a specific application model by instantiating RT design patterns while respecting the constraints implicitly defined by the stereotypes ⟪***mandatory***⟫ and ⟪***extensible***⟫. The RT constraints defined by RT stereotypes, such as ⟪***rtFeature***⟫ and ⟪***rtUnit***⟫, are transferred automatically to the RT application models instantiating the RT patterns. The verification of these constraints can be done only at runtime. For instance, we can verify that an operation deadline will not be missed, only when running the RT system.

4.1. Description and implementation of the mapping step

4.1.1. Mapping step description

The instantiation of a desired design pattern begins with an interactive mapping step that allows the application designer to instantiate the mandatory classes of the pattern as well as their attributes and operations. Then, the application designer has to choose the appropriate optional elements. Besides, the designer may rename the mapped pattern elements and may also add new application specific elements, such as attributes, operations, classes and so on. The result of this step is a mapping model that matches the elements of a pattern to the appli-

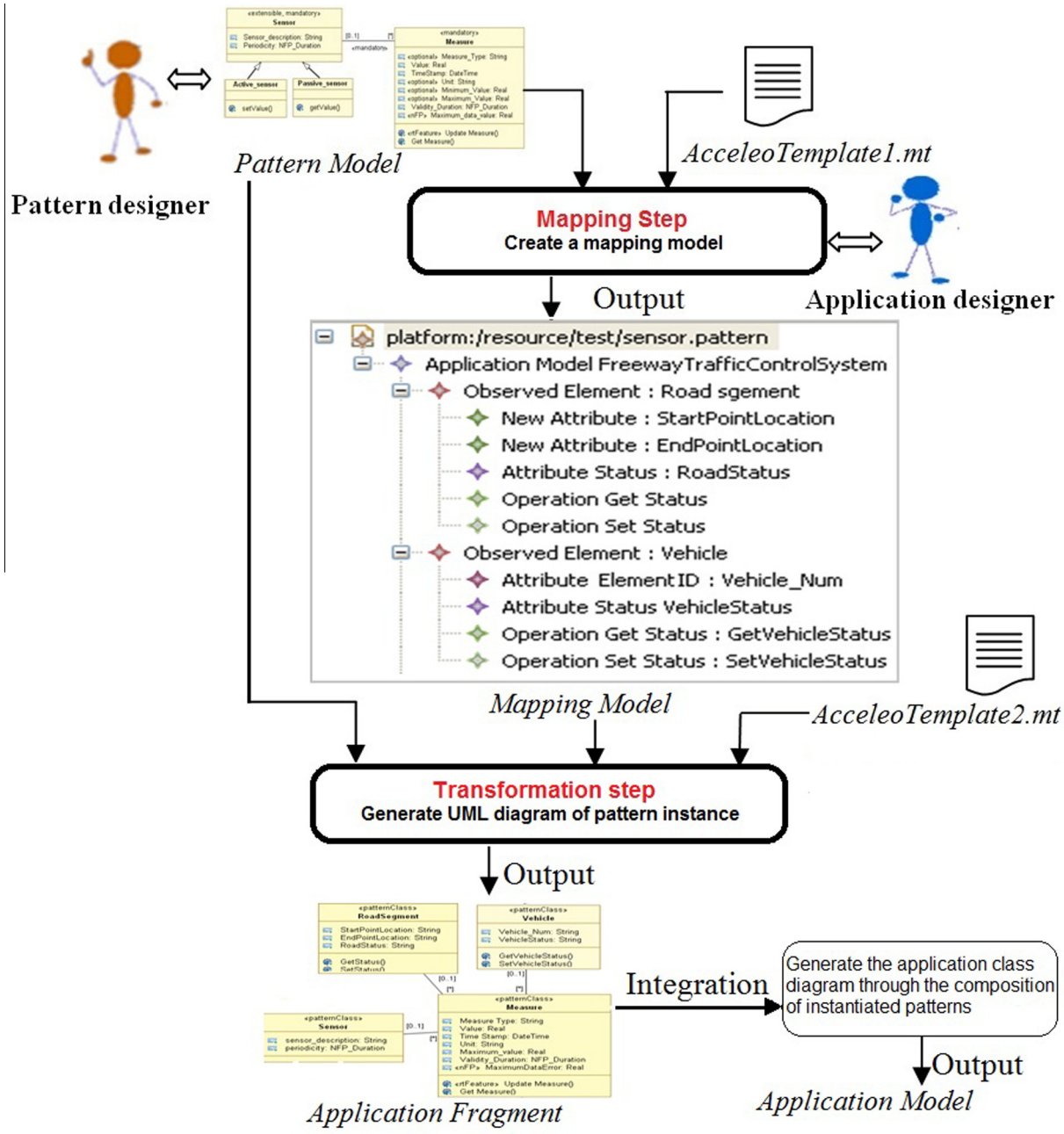

Figure 6 Pattern instantiation process.

cation model elements. The relational elements are transferred automatically when their associated pattern elements are mapped.

In the mapping model example, presented in Fig. 6, the *ObservedElement* class of the RT controller pattern is mapped twice in the application model (*RoadSegment* and *Vehicle* classes). This example shows also the addition of two new attributes (*StartPointLocation* and *EndPointLocation*).

4.1.2. Implementation of mapping step

The mapping model is implemented using Acceleo and EMF (Steinberg et al., 2008). We use Acceleo generator in order to transform each RT design pattern represented with UML to an Ecore model. This transformation is performed through the definition of the Acceleo template "*UMLtoEcore.mt*". The generated Ecore model will be the input model of the EMF framework that allows the automatic generation of an editor to instantiate the desired design pattern.

The generated editor shows a list of all classes corresponding to the selected pattern model. Then, it shows the attributes and the operations relevant for each selected pattern class. Thus, to create an instance of the desired pattern, the application designer chooses the appropriate elements from the list and fills in their properties (such as the name and the type) in the property window.

Note that, we have not chosen to use the automatic generation of an Ecore model from a UML model, which is provided by EMF, for two reasons:

- The generated Ecore model provided by EMF cannot allow adding application specific elements (i.e., new attributes, new operations, new classes) that do not instantiate pattern elements. Thus, it prevents application designers from adding new attributes or methods to a class which is stereotyped ⟪***extensible***⟫.
- The association concept of UML is treated differently in the Ecore meta-models (Kim and Carrington, 2006). In UML, an association is an independent concept, while in Ecore it is represented using references and opposites. As a result, the creation of an Ecore Model from pattern class diagram using EMF does not allow the generation of an editor which lists all the classes defined in the pattern model, especially those linked with UML associations and not with UML composition relationship.

4.1.3. Validation of the mapping model

The validity of the pattern instantiation is checked using EMF validation framework. This framework verifies, on the one hand, if all mandatory pattern elements (such as classes, attributes and operations) are mapped to application model elements. In addition, it verifies that each mapped pattern class, which is not stereotyped ⟪***extensible***⟫, does not have new attributes or new operations added by the application designer. On the other hand, it verifies if each element in the mapping model has a name and if each attribute has a type.

As a result, the validation of the mapping model is checked and if any constraint is violated, the designer is informed with a message displayed in the *Problems* tab, as shown the mapping model example presented in Fig. 7. In this example the *Measure* fundamental class belonging to the RT sensor pattern is not instantiated.

4.2. Description and implementation of the transformation step

Once a mapping model is created, EMF generates an XMI output file and places it in a repository containing all instantiated patterns. This file and the original pattern model file are the inputs of the model transformation engine, which generates automatically the class diagram of the pattern instance. The model transformation is based on the definition of the Acceleo template "*UMLtoUML.mt*" that specifies how to deploy a pattern in a specific application. This template imports java services that allow creating a copy of the selected pattern elements with their corresponding names specified in the mapping model, and the copy of the new specified application elements.

Then, the relationships between the mapped elements are defined based on the relationships between their corresponding elements in the original pattern model. This means that, for each relational element R_P that links the elements E_{P1} and E_{P2} in the pattern, if there are two elements E_{A1} and E_{A2} belonging to the application and instantiating respectively E_{P1} and E_{P2}, then the relational element R_P is copied to link the elements E_{A1} and E_{A2}. Moreover, if the element R_P has a name, then the same name is kept for the corresponding relational element in the application.

Besides, once a pattern class is mapped to an application class, then the ⟪***patternClass***⟫ stereotype is automatically associated to this application class in the transformation step. This stereotype allows easy retrieval of the pattern-related information used to check the existence of any conflicts with pattern properties, if any modification is performed after the application model generation. Also, the stereotypes modeling RT aspects (⟪***Nfp***⟫, ⟪***rtFeature***⟫, ⟪***rtUnit***⟫,...) are automatically associated with their corresponding application attributes and methods.

5. Evaluation

To determine the performance of our pattern development process, two questions arise: do the obtained patterns have a good design quality and also do they cover the essence of the RT domain? For this purpose, we proposed some projects to experts, while dividing them into two groups, one group models RT applications using our RT patterns, and the other will not use them. Then, we evaluate the design quality of the proposed RT patterns thanks to reuse metrics, reusability metrics and CK metrics proposed by Chidamber and Kemerer (1994).

We have chosen ten case studies. The ten collected case studies used in our experimental evaluation contain about 110 classes, 142 methods and 120 associations. They involve 30 different RT pattern occurrences.

5.1. Evaluation metrics

We evaluate the quality of the proposed RT patterns in terms of domain coverage through two kinds of metrics: reuse metrics and reusability metrics. Reuse metrics calculate the percentage of reuse of RT pattern elements in software design,

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

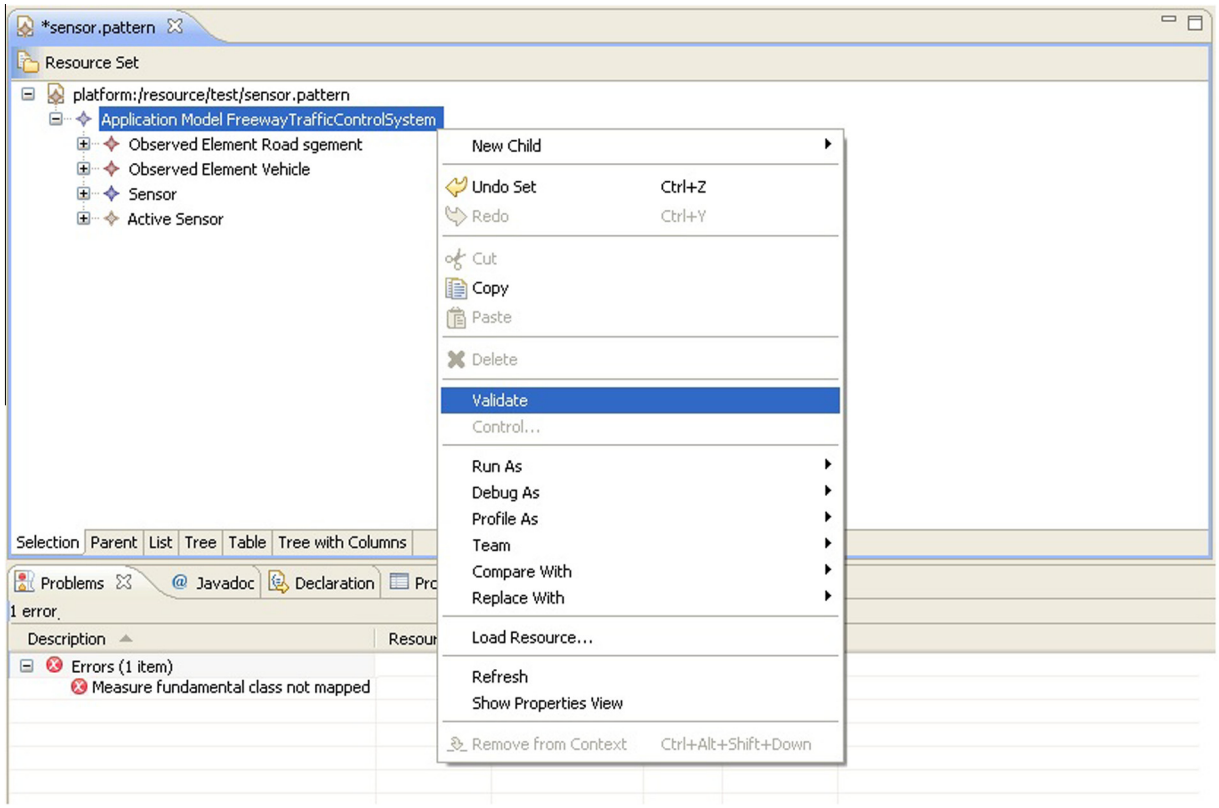

Figure 7 Mapping model editor showing an instantiation error.

whereas reusability metrics evaluate the possibility that a design pattern can be reused.

Frakes et al. (2009) stipulated that reuse metrics aim to determine how much reuse is present within a given system. There are many ways to implement these metrics. For example, the amount of code reuse is defined as the ratio between the number of reused lines of code in a system and the total lines of code in a system. Aggarwal et al. (2005) proposed two metrics for measuring the amount of reuse in object oriented software using generic programing in the form of templates. The first metric, called Class Template Factor (CTF) is defined as a ratio between the number of classes using class templates and the total number of classes in a source code. The second metric, called Function Template Factor (FTF) is defined as a ratio between the number of functions using function templates and the total number of functions. These works are focused on different reuse metrics, aiming to measure the amount of reuse of software components and to determine the portion of the new or modified code and the portion of the reused code. These metrics only deal with source code which is typically available at the later stages of the software life cycle, failing to address the importance of the software artifacts produced during earlier stages such as analysis and design. Thus, we intend hereafter to adapt existing reuse metrics CTF and FTF defined in Aggarwal et al. (2005) to measure, respectively, the amount of pattern classe reuse and operation reuse in a given designed with UML. Moreover, we will add another metric to compute the amount of attribute reuse.

On the other hand, reusability metrics indicate the possibility that a component is reusable and enable to identify a good quality of a component for reuse, but, they don't provide a measurement of how many components are reused. Different studies are based on the definition of reusability metrics. Gill and Sikka (2011) have proposed new metrics which can be computed from inheritance hierarchies: Breadth of Inheritance Tree (BIT), Method Reuse Per Inheritance Relation (MRPIR), Attribute Reuse Per Inheritance Relation (ARPIR),.... BIT measures the breadth of the whole inheritance tree. MRPIR and ARPIR metrics give a clearer picture of reuse since they consider inheritance. MRPIR metric (respectively ARPIR metric) computes the average number of reused methods (respectively attributes) in an inheritance hierarchy and not in all classes. Subedha and Sridhar (2012) have used reuse utility percent and reuse frequency metrics as the assessment attributes for reusability of the software component in context level. These metrics determine which components have high reuse potential from a set of standard components in an existing environment. The reusability metrics proposed in these works indicate whether or not the components are reusable in the future. But, they do not answer an essential question: Do the reusable components represent the specificities of a particular domain? In order to fill this lack, we propose in this paper new metrics for reusability assessment of domain specific design patterns. The aim of these metrics is to show if these patterns are well-defined by checking the presence of pattern elements in a system designed without the usage of patterns. In our context, these metrics are intended to measure if the

RT patterns cover the RT domain specificities. They calculate the number of elements identified as pattern elements in the applications that do not reuse RT patterns divided by the number of pattern elements in an application reusing the pattern. When this ratio is close to one, this means that the majority of the classes, attributes and operations of RT design patterns are identified as similar to the classes, attributes and operations of the applications designed by experts without using patterns.

5.1.1. Reuse metrics

Three reuse metrics are proposed, they are: *Class Reuse Level* (*CRL*), *Attribute Reuse Level* (*ARL*) and *Operation Reuse Level* (*ORL*). Their values range from 0 to 1. When the reuse metrics value is close to one, this indicates a high reuse level, while when it is close to zero, this indicates that none of the pattern elements is reused. Reuse metrics' high values indicate that the application designers need only to add some elements specific to the domain (classes, attributes and operations) since the majority of application elements are reused from the RT patterns. This means that our RT patterns cover the essence of the domain and that their adaptation does not necessitate a great effort.

5.1.1.1. Class Reuse Level (CRL). This metric is defined as the ratio between the number of reused pattern classes (RPC) and the total number of classes in the model corresponding to an instance of a pattern P as shown in (1).

Let us consider a model corresponding to an instance of a pattern P with n classes $C_1, C_2, \ldots, C_n$

$$CRL_p = \frac{\sum_{i=1}^{n} RPC(C_i)}{n} \quad (1)$$

where,

$$RPC(C_i) = \begin{cases} 1 & \text{if the class is reused from a pattern;} \\ 0 & \text{otherwise} \end{cases}$$

5.1.1.2. Attribute Reuse Level (ARL). This metric is defined as the ratio between the number of reused attributes (RAT) of pattern classes and the total number of attributes in the model corresponding to an instance of a pattern P as shown in (2).

Let us consider a model corresponding to an instance of a pattern P and having n classes $C_1, C_2, \ldots, C_n$ and mi attributes $a_1, a_2, \ldots, a_{mi}$ for each class C_i.

$$ARL_p = \frac{\sum_{i=1}^{n} \sum_{j=1}^{mi} RAT(a_{ij})}{\sum_{i=1}^{n} mi} \quad (2)$$

where,

$$RAT(a_{ij}) = \begin{cases} 1 & \text{if the attribute is reused from a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

$$IAT(a_{ij}) = \begin{cases} 1 & \text{if the attribute is identified as an attribute of a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

5.1.1.3. Operation Reuse Level (ORL). This metric is defined as the ratio between the number of reused Operations (ROP) of pattern classes and the total number of operations in the model corresponding to an instance of a pattern P as shown in (3).

Let us consider a model corresponding to an instance of a pattern P and having n classes $C_1, C_2, \ldots, C_n$ and qi operations $op_1, op_2, \ldots, op_{qi}$ for each class C_i.

$$ORL_p = \frac{\sum_{i=1}^{n} \sum_{k=1}^{qi} ROP(op_{ik})}{\sum_{i=1}^{n} qi} \quad (3)$$

where,

$$ROP(op_{ik}) = \begin{cases} 1 & \text{if the operation is reused from a pattern} \\ 0 & \text{otherwise} \end{cases}$$

5.1.2. Reusability metrics

Three reusability metrics are proposed, they are: *Class Reusability* (*CR*), *Attribute Reusability* (*AR*) and *Operation Reusability* (*OR*). These metrics indicate whether the RT patterns allow designing the specificities of the RT domain or not. They are calculated from two releases of each application. Release 1 is designed without using any pattern. Release 2 is designed using design patterns.

5.1.2.1. Class Reusability (CR). The metric CR is defined as the ratio between the number of identified pattern classes (IPC) in a model designed without using patterns and the number of reused pattern classes (RPC) in this model when designed using RT patterns as shown in (4).

Let us consider a model with n classes $C_1, C_2, \ldots, C_n$.

$$CR_p = \frac{\sum_{i=1}^{n} IPC(C_i)}{\sum_{i=1}^{n} RPC(C_i)} \quad (4)$$

where,

$$IPC(C_i) = \begin{cases} 1 & \text{if the class is identified as a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

$$RPC(C_i) = \begin{cases} 1 & \text{if the class is reused from a pattern;} \\ 0 & \text{otherwise} \end{cases}$$

5.1.2.2. Attribute Reusability (AR). The metric AR is defined as the ratio between the number of identified attributes (IAT) of pattern classes in a model designed without using patterns and the number of reused attributes (RAT) of pattern classes in this model when designed using patterns as shown (5).

Let us consider a model with n classes $C_1, C_2, \ldots, C_n$ and mi attributes $a_1, a_2, \ldots, a_{mi}$ for each class C_i.

$$AR_p = \frac{\sum_{i=1}^{n} \sum_{j=1}^{mi} IAT(a_{ij})}{\sum_{i=1}^{n} \sum_{j=1}^{mi} RAT(a_{ij})} \quad (5)$$

where,

$$\mathrm{RAT}(a_{ij}) = \begin{cases} 1 & \text{if the attribute is reused from a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

5.1.2.3. Operation Reusability (OR). The metric OR is defined as the ratio between the number of identified operations (IOP) of pattern classes in a model designed without using patterns and the number of reused operations (ROP) of pattern classes in this model when designed using patterns as shown in (6).

Let us consider a model with n classes $C_1, C_2, \ldots, C_n$ and qi operations $op_1, op_2, \ldots, op_{qi}$ for each class C_i.

$$\mathrm{OR}_p = \frac{\sum_{i=1}^{n}\sum_{k=1}^{qi}\mathrm{IOP}(op_{ik})}{\sum_{i=1}^{n}\sum_{k=1}^{qi}\mathrm{ROP}(op_{ik})} \quad (6)$$

where,

$$\mathrm{IOP}(op_{ik}) = \begin{cases} 1 & \text{if the operation is identified as an operation of a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

$$\mathrm{ROP}(op_{ik}) = \begin{cases} 1 & \text{if the operation is reused from a pattern class;} \\ 0 & \text{otherwise} \end{cases}$$

5.1.3. CK metrics

Chidamber and Kemerer (1994) present a state of the art of object-oriented metrics, called CK metrics, and classify them essentially into four categories: coupling, cohesion, complexity and inheritance. Next, we explain each category and we present its associated metrics.

- Complexity: complexity measures the simplicity and understandability of a design. Many complexity measures have been proposed in the literature, we present the most useful ones which are Number of attributes (NAtt) and Weighted Methods per Class (WMC).
- Cohesion: cohesion is a measure of how strongly-related and focused the various responsibilities of a class. A cohesive class is a class which all its methods are tightly related to the attributes defined locally. The cohesion should be maximized to get a design with a good quality. The essential metric measuring the cohesion is Lack of Cohesion in Methods (LCOM).
- Inheritance: inheritance measures the tree of inheritance and the number of children. In this category, we find Depth of Inheritance (DIT) and Number of Children (NOC).
- Coupling: coupling measures the degree of interdependency between classes/objects. Two objects are coupled if and only if at least one of them acts upon the other. The coupling could, essentially, be measured with the Coupling between Objects (CBO) and Response for Call (RFC) metrics. Note that, the CBO value should be minimized, first, because, when it increases, the sensibility to changes is higher and therefore maintenance is more difficult (Chidamber and Kemerer, 1994).

5.2. Evaluation results and interpretation

5.2.1. Evaluation of RT design patterns based on reuse and reusability metrics

Table 1 shows the values of reuse metrics and reusability metrics obtained for the proposed RT design patterns (i.e., sensor, controller and actuator patterns) which are used for modeling ten different RT applications that we reference A1, A2,..., A10.

On one hand, the values obtained for reuse metrics show that more than 60% of the classes, the attributes and the operations of RT applications are instantiated from the proposed RT patterns in all cases, as shown in Fig. 8. Then, a few numbers of applications specific elements are added by designers. For example, the values of reuse metrics obtained in Table 1 for the application A1 show that 85% of classes (CRL = 0.85), 73% of attributes (ARL = 0.73) and 83% operations (ORL = 0.83) are instantiated from RT patterns. There are even cases (applications A7 and A8) where all applications classes are instances of pattern classes (CRL = 1). Thus, we deduce a good reuse level of RT design patterns elements in the modeling of RT applications.

On the other hand, the values obtained for reusability metrics indicate that the degree of reusability of classes and attributes is better than the reusability of operations, as shown in Fig. 9. Indeed, we identified all the classes of the RT patterns (CR = 1) in three cases of real-time applications modeled without reusing this pattern. In addition, we have identified the majority of the attributes reused from the pattern classes. For example, the values of reusability metrics obtained in Table 1 for application A1 show that 83% of RT patterns classes are identified (CR = 0.83), 72% of the attributes are identified (AR = 0.72) and 60% of operations are identified (OR = 0.6). This means that the reuse of these patterns is interesting in the RT domain because they are well-defined and they represent adequately the specificities of the considered domain.

5.2.2. Evaluation of RT design patterns based on CK metrics

Table 2 shows the values of CK metrics obtained for the different applications reusing the proposed RT design patterns. For each application, we measure the minimum value and the maximum value of each CK metric except LCOM metric (cohesion) which is not measured since it can be calculated only at code level and not on the design level. We present the result of each metric as an interval [min, max] and we compare it to the defined threshold (Chandra et al., 2010) presented in Table 3.

We should caution that, in the software engineering field, in general, there is not yet a precise guideline for how to fix thresholds. The work proposed by Chandra et al. (2010) suggest a threshold value of 6 for DIT and NOC, as shown in

Table 1 Results for reuse and reusability metrics calculated for RT design patterns.

		A1	A2	A3	A4	A5	A6	A7	A8	A9	A10
Reuse metrics	CRL	0.85	0.70	0.66	0.8	0.75	0.81	1	1	0.87	0.8
	ARL	0.73	0.66	0.62	0.67	0.8	0.75	0.8	0.82	0.81	0.71
	ORL	0.83	0.66	0.76	0.76	0.9	0.76	0.83	0.84	0.75	0.89
Reusability metrics	CR	0.83	0.71	0.81	0.87	1	0.66	1	1	0.85	0.83
	AR	0.72	0.75	0.68	0.71	0.75	0.71	0.75	0.71	0.72	0.60
	OR	0.6	0.5	0.56	0.6	0.55	0.56	0.8	0.72	0.6	0.70

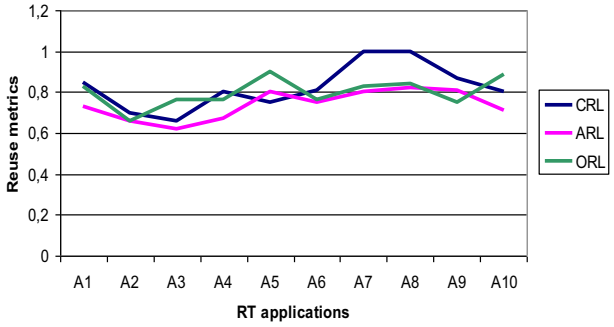

Figure 8 Evaluation of RT design patterns based on reuse metrics.

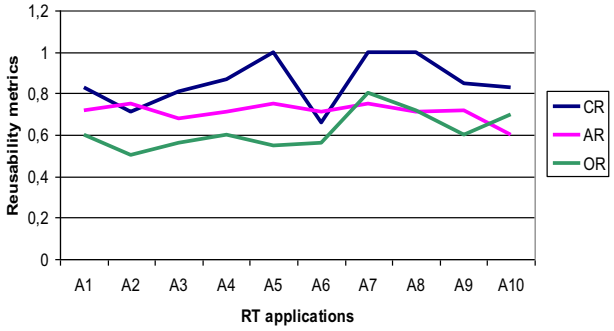

Figure 9 Evaluation of RT design patterns based on reusability metrics.

Table 3. The threshold limit for CBO metric is set to 8 per class. For the RFC metric the threshold limit is set to 35 per class, finally the threshold limit for the LCOM metric is set to 1 per class.

Examining the evaluation results, we see that the obtained metrics' values for the RT design patterns are in perfect agreement with the thresholds of Table 3. For example, we find that the DIT values of all classes are low (in the interval [0, 1]) in all application models. This means that the maximum length from the node to the root of the inheritance tree of a class is equal to 1. We find also that the NOC values are not very high in all cases. That is the maximum number of immediate sub-classes subordinated to a class in the class hierarchy does not exceed 4 classes (NOC value is in the interval [0, 4] for the application A1).

In addition, the values obtained for CBO metric are low in some cases (in the interval [1, 3] for the applications A5, A7 and A8) and slightly high in other cases (in the interval [1, 6] for the applications A1 and A3) but they do not exceed the thresholds in all cases. Indeed, the CBO high values are obtained in two case studies (A1 and A3) where the classes playing the role of *Measure* in the sensor pattern are used by many others classes (a maximum of six classes in the applications A1 and A3). Finally, the values obtained for WMC metrics are low compared to the defined thresholds. Thereby, according to the different values obtained for CK metrics, we can deduce that the applications models reusing RT patterns have a good design quality.

Table 2 Results for CK metrics calculated for RT design patterns.

	Metrics					
	DIT	NOC	CBO	RFC	WMC	NAtt
A1	[0, 1]	[0, 4]	[1, 6]	[0, 8]	[0, 5]	[0, 4]
A2	0	0	[1, 4]	[0, 6]	[0, 4]	[0, 6]
A3	[0, 1]	[0, 2]	[1, 6]	[0, 9]	[0, 7]	[0, 5]
A4	[0, 1]	[0, 2]	[1, 4]	[0, 6]	[0, 4]	[0, 5]
A5	0	0	[1, 3]	[0, 5]	[0, 4]	[0, 4]
A6	[0, 1]	[0, 3]	[1, 4]	[0, 8]	[0, 5]	[0, 6]
A7	0	0	[1, 3]	[0, 6]	[0, 4]	[0, 5]
A8	[0, 1]	[0, 2]	[1, 3]	[0, 6]	[0, 3]	[0, 5]
A9	[0, 1]	[0, 2]	[1, 4]	[0, 9]	[0, 7]	[0, 6]
A10	[0, 1]	[0, 3]	[1, 5]	[0, 7]	[0, 5]	[0, 5]

Table 3 Threshold values for the CK metric suite.

Metric	Threshold
WMC	0–15
DIT	0–6
NOC	0–6
CBO	0–8
RFC	0–35
LCOM	0–1

6. Conclusion

In this paper, we have proposed a pattern development process that integrates top-down and bottom-up approaches. This process is based on a set of unification rules identifying the commonalities and differences between the applications models in a specific domain and deriving the fundamental, optional and extensible elements of patterns. The proposed unification rules guide the development of domain-specific patterns, in general, and it is particularly adapted to the design patterns specific to real-time domain with intensive data. Moreover, we have presented a RT pattern instantiation process to provide assistance

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004

for an application designer when reusing a pattern. We have implemented this process using an existing modeling framework. We have provided a plug-in tool that supports the creation and validation of pattern instances and prevents the application designer from the violation of pattern constraints. Finally, we have proposed two kinds of metrics for evaluation purpose. The first one aims to assess the reuse level of pattern participants. The second category focuses on predicting the reusability of domain specific design patterns. This kind of metrics checks the presence of pattern elements in a system designed without the usage of patterns.

Our approach was illustrated through the creation of the Controller pattern. In addition, it was quantitatively evaluated on ten case studies in the RT domain. The evaluation has demonstrated the potential of the proposed approach, even though further refinements are necessary. For instance, it is essential that our pattern development tool keeps automatically the consistency between the class and sequence diagrams of the proposed patterns. It has to take into account the impact of the selection of a variable element in the static view and dynamic view when instantiating RT patterns.

Currently, we are examining how to experiment the approach on other RT applications. Moreover, we are interested in defining an ontology specific to RT domain to fill the gaps we encountered using WordNet and to reduce the intervention of designers when generating RT patterns.

Appendix A. Illustration of the pattern development process

In order to illustrate our pattern development process, we present, in the following, the construction of the RT controller pattern. To illustrate the applications decomposition and the unification of the applications fragments, three different RT applications are briefly shown and explained: a freeway traffic management system (Compass, 2010), an industrial control system which allows the control of tanks water levels (Reinhartz-Berger and Sturm, 2009) and a medical telesurveillance system (Baldinger et al., 2004).

The first and last applications were designed by two different UML experimented designers (not belonging to the authors team) while the third application relative to the system which controls water level in tanks is presented in Reinhartz-Berger and Sturm (2009). The freeway traffic management system and the medical telesurveillance system were designed manually since the design diagrams of these systems are not available; only their textual description is presented. In an ideal case, the design of the applications could be obtained by reverse engineering if source code is available.

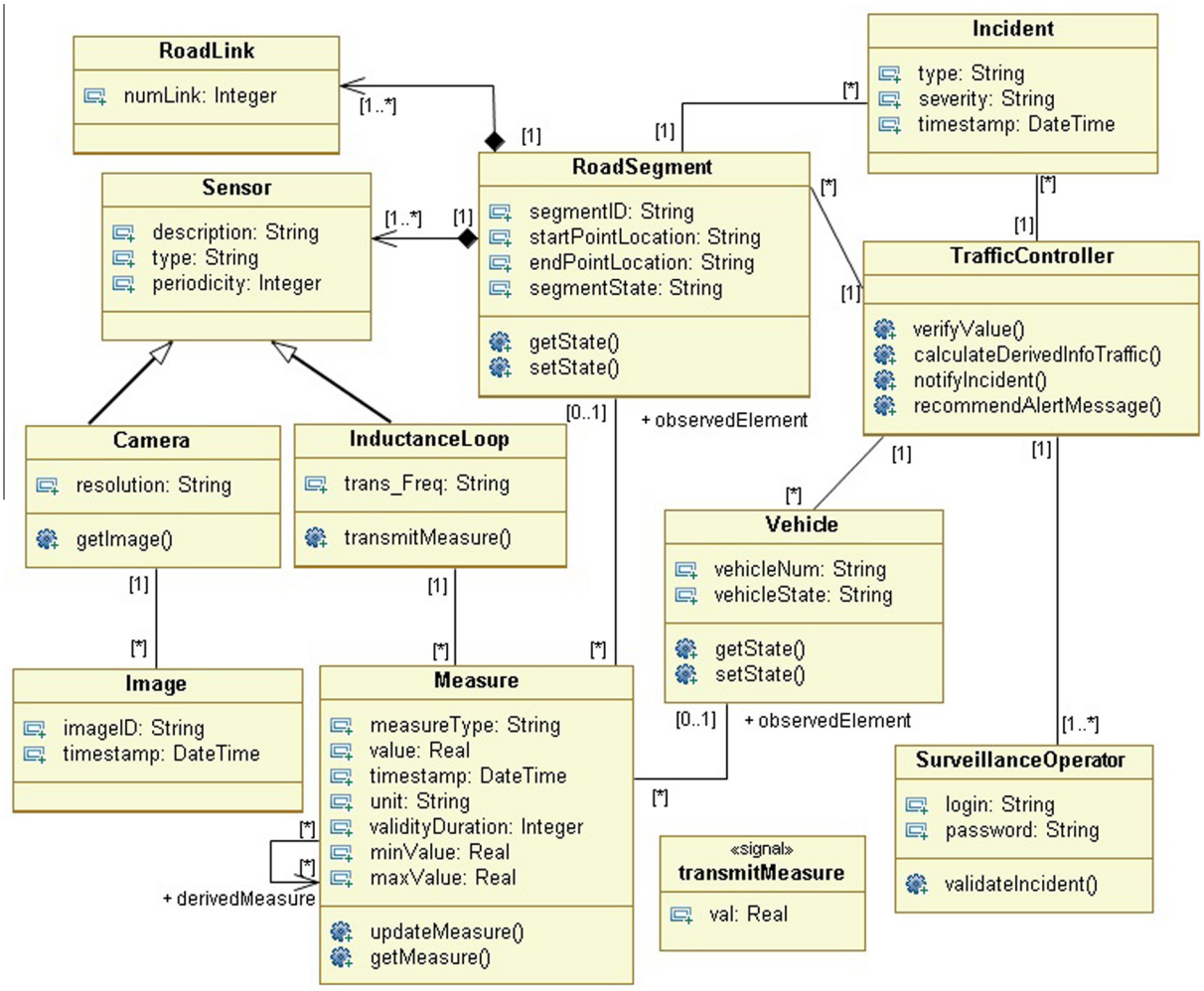

Figure A1 Class diagram of COMPASS system.

On the other hand, the class diagram of the system that controls tank water level presented in Fig. A2 is slightly modified from the original one presented by Reinhartz-Berger and Sturm (2009) since we consider that the history of the measured water heights is out of our scope.

A.1. Description of RT applications

A.1.1. Freeway traffic management system

COMPASS system (Compass, 2010) is regarded as one of the most important traffic management systems in North America. It uses inductance loop detectors to measure speed of each vehicle, number of vehicles in a road segment (i.e., traffic density) and road segment occupancy. As illustrated in Fig. A1, these measures can be related to the *Vehicle* class or the *RoadSegment* class. The *Vehicle* and the *RoadSegment* classes represent the physical elements observed by the active sensors. All the acquired measures are analyzed by the traffic controller in order to detect incidents on the highway and to notify the operators of any detected events.

A.1.2. Industrial control system

The purpose of the water level control of an industrial regulation system is to monitor and control the water levels in tanks, ensuring that the actual water level of tank *i* is always in the closed range [Low-level, High-level] (Reinhartz-Berger and Sturm, 2009).

Fig. A2 illustrates the design of a water level control system. It indicates that this system controls one type of element (a tank) and acquires one type of measure (the water height in tank), through the boundary stick passive sensors. This system uses a water controller to deal with both opening and closing faucets if the water height in a tank reaches its low or high limit.

A.1.3. Medical telesurveillance system

The MEDIVILLE system (Baldinger et al., 2004) for telesurveillance of patients at home continuously records patient's attitude, acquired from actimetry sensor, and its heart rate, acquired from pulse sensor. The acquired data are transmitted to the base station at regular intervals (every 30 s). In case of patient's falls and heart problems, the local system communicates with the central server and transmits both the alarm and the latest available data to notify the medical surveillance team of any detected event. The surveillance operator reports the detected problems to the appropriate practitioner. The class diagram of MEDIVILLE system is shown in Fig. A3.

A.2. Class diagram construction of RT controller pattern

The objective of RT controller pattern is to model both the control of data acquired from the environment and the initialization of corrective action(s) in case of constraint violation. The RT data control sub-problem is decomposed into six elementary functions (cf. Section 3.2). Table A1 shows the concepts and domain constraints related to each identified elementary function.

After the requirements identification phase, the classes belonging to each application and playing the role of *Controller*, *Observed-Element* and *Surveillance Operator* must be identified in order to determine the applications fragments related to RT data control functionality, as illustrated in Table A2.

The application fragments obtained after decomposition step are:

F_{12} = {TrafficController, Vehicle, RoadSegment, SurveillanceOperator, RoadLink, Incident}.

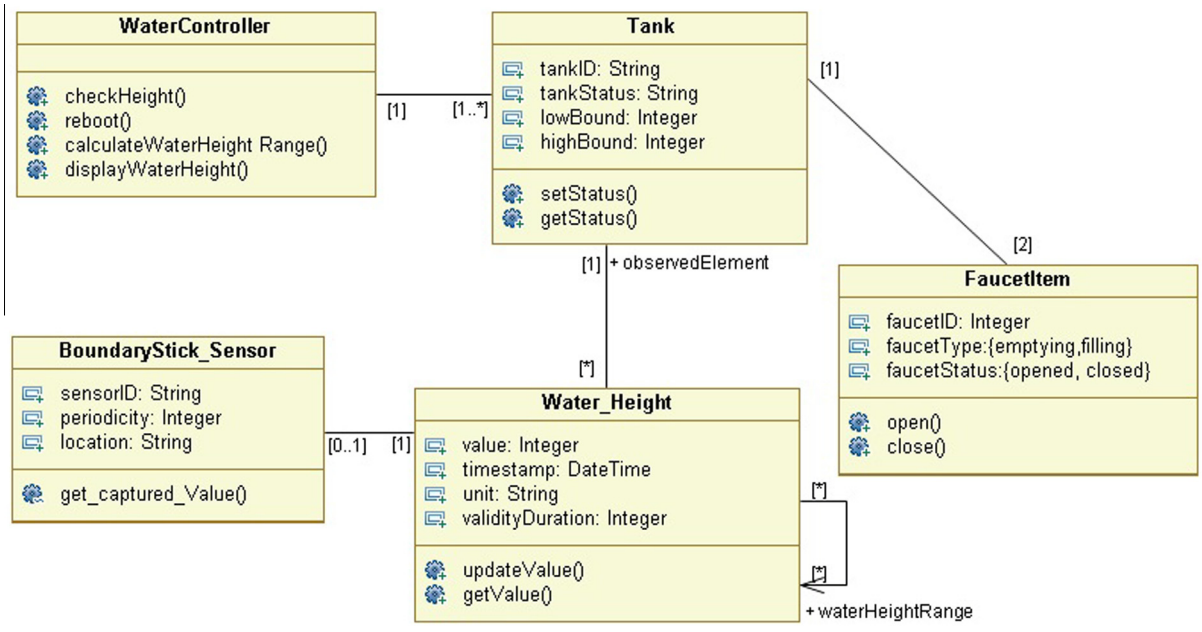

Figure A2 Class diagram of water level control application.

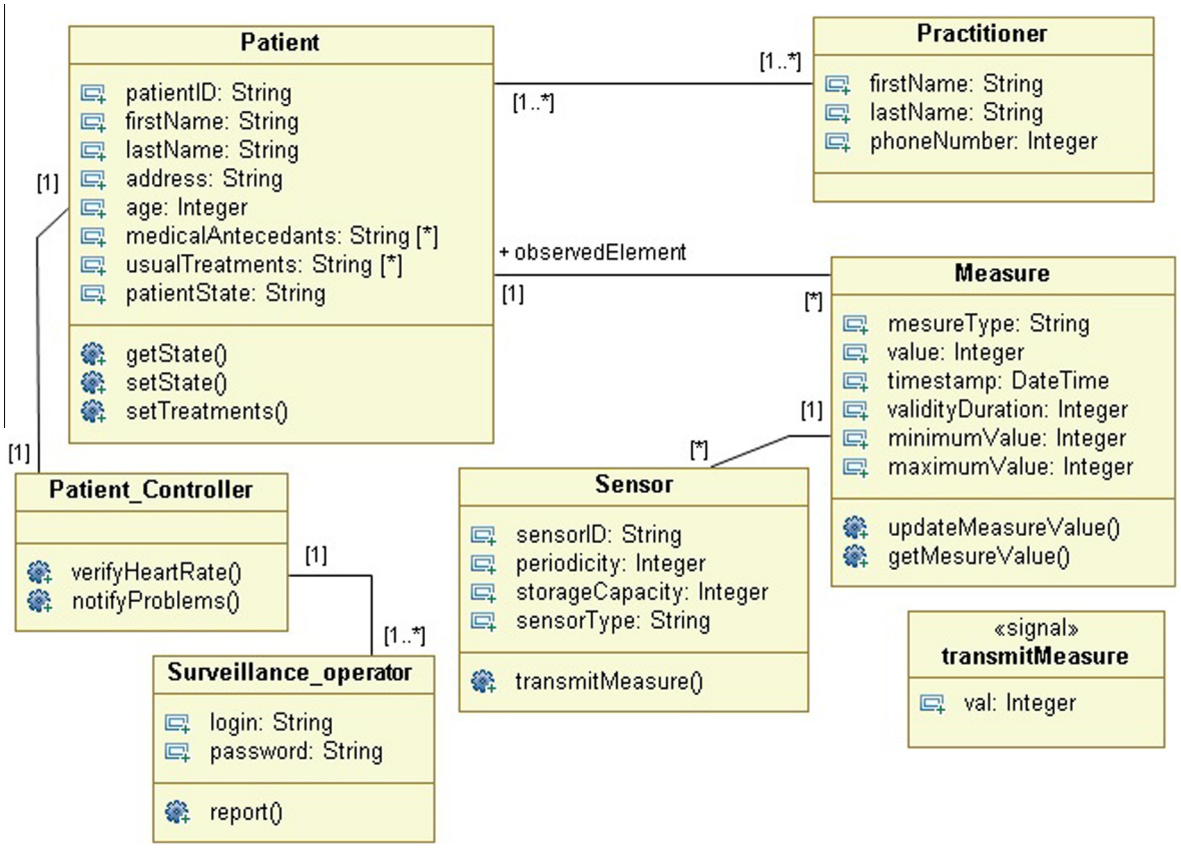

Figure A3 Medical telesurveillance application model.

Table A1 Requirement identification of RT data control functionality.

Domain functionality	Elementary function	Concept and constraint
RT data control functionality	A controller monitors the state of each observed element	Concepts: controller, observed element
	A controller calculates derived data from captured data	Concepts: controller
	A controller checks if an action misses its deadline	Concepts: controller. Constraint: The completion time of RT method must be achieved before the deadline
	A controller checks if a captured value does not exceed the predefined minimum value and maximum values	Concepts: controller. Constraint: Measure's value must not exceed the predefined minimum value and maximum value
	A controller initiates corrective actions when it detects abnormal situation	Concepts: controller
	A controller notifies the surveillance operator of any detected anomaly	Concepts: controller, surveillance operator

Table A2 Relation between concepts related to RT data control functionality and application classes.

Concepts	Classes	Applications examples
Controller	TrafficController	*a*1. Freeway traffic management application
Observed element	Vehicle, road segment	
Surveillance operator	SurveillanceOperator	
Controller	WaterController	*a*2. Water level control application
Observed element	Tank	
Surveillance operator		
Controller	Patient_Controller	*a*3. Medical telesurveillance application
Observed element	Patient	
Surveillance operator	Surveillance_Operator	

F_{22} = {WaterController, Tank}.
F_{32} = {Patient_Controller, Patient, Surveillance_Operator, Practitioner}.

The unification of these fragments allows the generation of RT controller pattern class diagram presented in Fig. A4. This

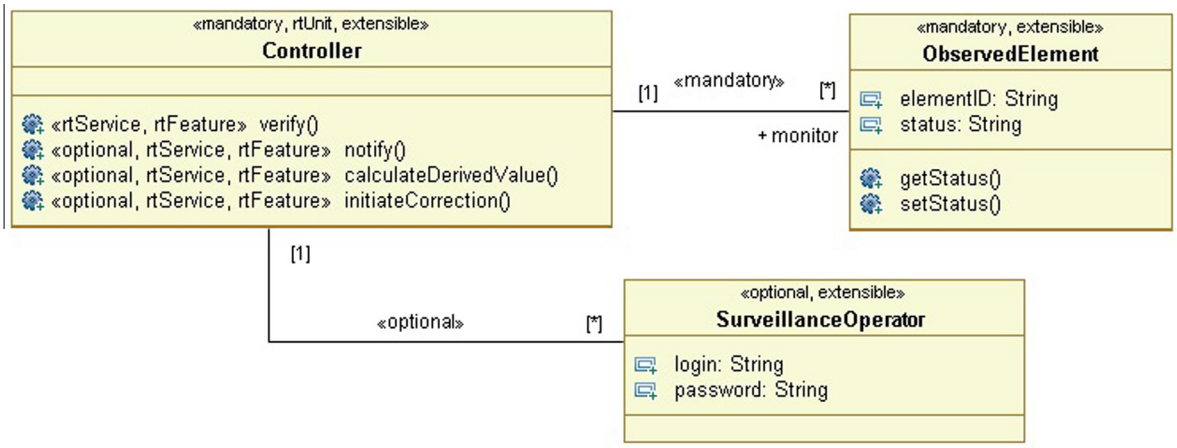

Figure A4 Class diagram of RT controller pattern.

pattern has been obtained by applying the unification rules (cf. Section 3.4.1) as follows:

- N_Var (TrafficControllerA$_{12}$, WaterControllerA$_{22}$, PatientControllerA$_{32}$) and Op_int (TrafficControllerA$_{12}$, WaterControllerA$_{22}$, PatientControllerA$_{32}$): rule *UR2* (case 2) is applied and a mandatory class *Controller* is added to the pattern with the mandatory method *verify( )* and the optional method *notify( )*. This class is stereotyped ⟪***extensible***⟫.
- N_dist (RoadSegmentA$_{11}$, VehicleA$_{11}$, TankA$_{21}$, PatientA$_{31}$), Att_int (RoadSegmentA$_{11}$, VehicleA$_{11}$, TankA$_{21}$, PatientA$_{31}$) and Op_equiv (RoadSegmentA$_{11}$, VehicleA$_{11}$, TankA$_{21}$, PatientA$_{31}$): rule *UR2* (case 3) is applied and a fundamental class *observedElement* is added to the pattern with two attributes (*elementID* and *status*) and two methods (*getStatus( )* and *setStatus( )*).
- N_equiv (Surveillance_OperatorA$_{12}$, Surveillance_OperatorA$_{32}$) and Att_equiv (Surveillance_OperatorA$_{12}$, Surveillance_OperatorA$_{32}$): rule *UR3* is applied and an optional class *SurveillanceOperator* is added to the pattern because it is present in only two application examples.
- Rule *UR6* is applied to transfer the relations between classes in the pattern.

In the pattern validation step, ⟪***rtFeature***⟫ stereotype is added to the *Verify( )* method of *Controller* class since it is essential to check that each measured value is in the closed

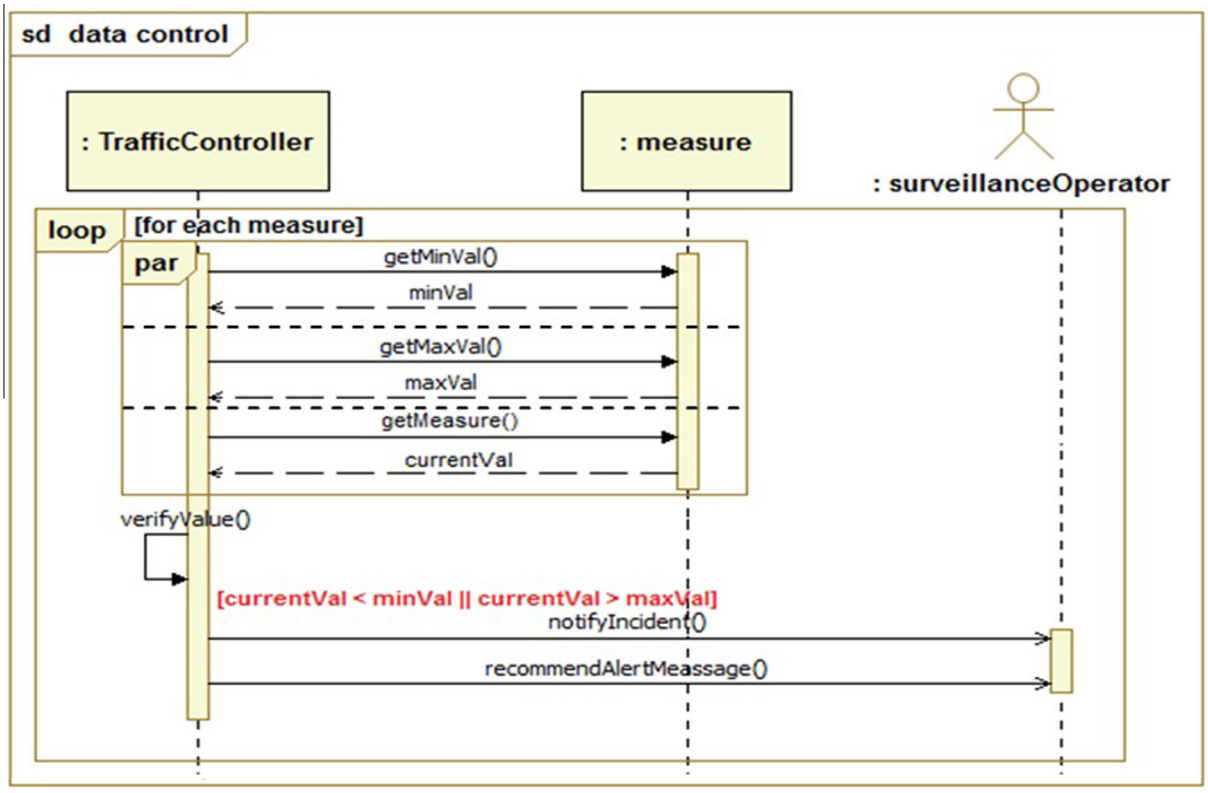

Figure A5 Sequence diagram relative to data control functionality of COMPASS system.

range [Minimum-Value, Maximum-value] before a deadline; otherwise the initialization of corrective actions may have no effect. For the same reason, the method *notify( )* is stereotyped ⟪***rtFeature***⟫ according to the rule *VR*2. Besides, the Controller represents an active entity responsible for checking the system state. Therefore, the rule *VR4* is applied and the *Controller* class is stereotyped ⟪***rtUnit***⟫.

In the resulting pattern, the *calculateDerivedInfoTraffic( )* method is not present because it belongs to one application (COMPASS). However, in some cases it is essential to have a method that calculates derived data, in order to fulfill RT domain functional requirements (cf. Ef_2 of RT data control functionality). In this case, the *CalculateDerivedValue( )* method is added by the designer as an optional element. Sim-

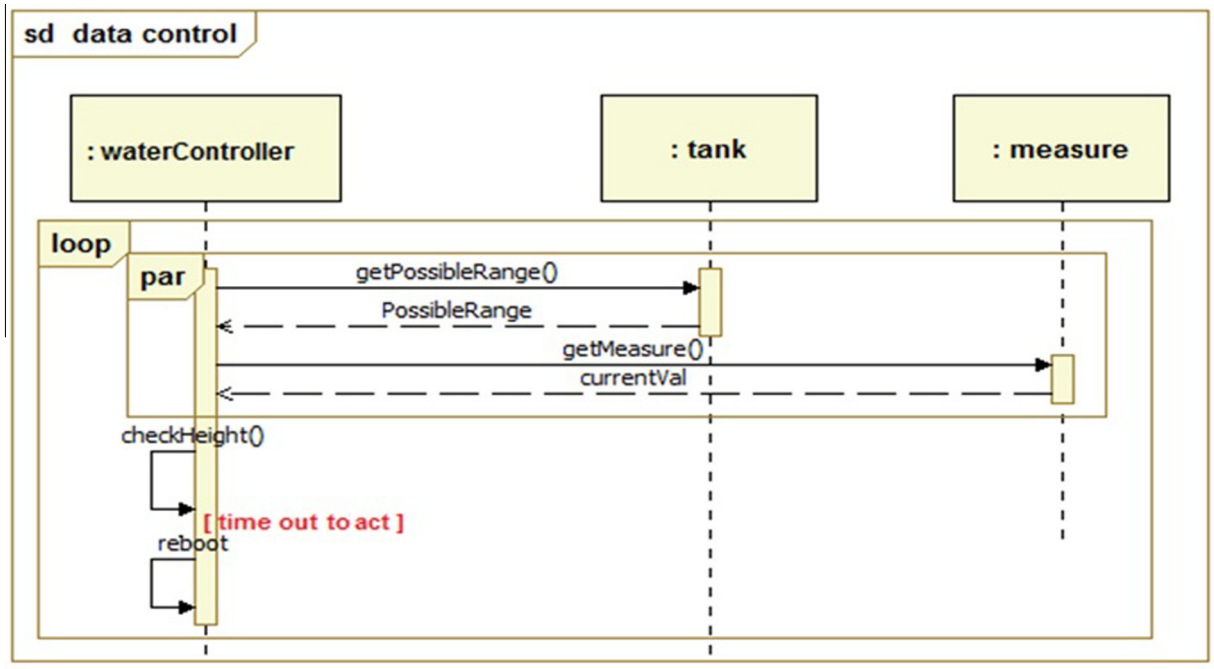

Figure A6 Sequence diagram relative to data control functionality of industrial control system.

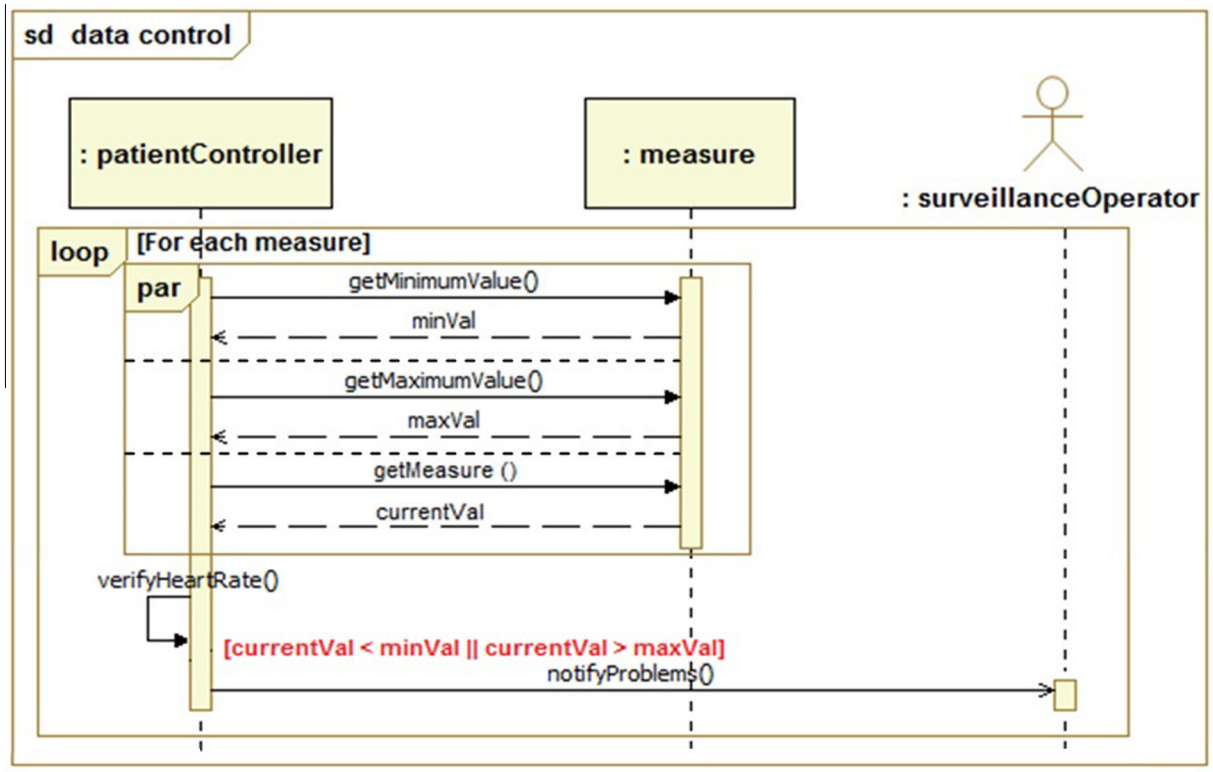

Figure A7 Sequence diagram relative to data control functionality of MEDIVILLE system.

ilarly, the *InitiateCorrection( )* method is added to the controller pattern as an optional method.

A.3. Sequence diagram construction of RT controller pattern

In order to illustrate the generation of RT controller pattern sequence diagram, we firstly present the sequence diagrams relative to the RT data control functionality of RT applications described in the Figs. A5–A7.

The sequence diagram of RT controller pattern is obtained through the application of the unification rules (cf. Section 3.4.2) as follows:

- N_var (trafficControllerA$_{12}$, waterControllerA$_{22}$, patientControllerA$_{32}$): the rule UR9 case 1 is applied and a fundamental object *Controller* is added to the pattern sequence diagram since the objects *trafficControllerA$_{11}$*, *waterControllerA$_{21}$* and *patientControllerA$_{31}$* are instances of one class named *controller* in the pattern class diagram.
- N_equiv (measureA$_{12}$, measureA$_{22}$, measureA$_{32}$): the rule UR7 is applied and a fundamental object *measure* is added to the pattern sequence diagram.
- The rule UR12 case 1 is applied and the messages *getMinimumValue( )*, *getMaximumValue( )* and *getMeasure( )* are added to the pattern sequence diagram since these messages have equivalent names and their sender objects represent a variation of a concept and they are instances of a unique class which is *Controller*. The messages *getMinimumValue ( )* and *getMaximumValue( )* are not common to all applications and they are placed in an optional combined fragment. However, the message *getMeasure( )* is added as fundamental element. These messages belong to parallel combined fragment in order to fulfill the application requirements.
- The rule UR12 case 1 is also applied to add the message *verify( )* as fundamental element and the message *notify( )* as optional element.

Note that the unification rules of actors are similar to those of objects. Therefore, the actor *surveillanceOperator* is added to the pattern as optional actor.

Similarly to the validation step of RT controller pattern class diagram, the validation of sequence diagram allow to add the message *initiateCorrection( )* in an optional combined fragment as shown in Fig. A8.

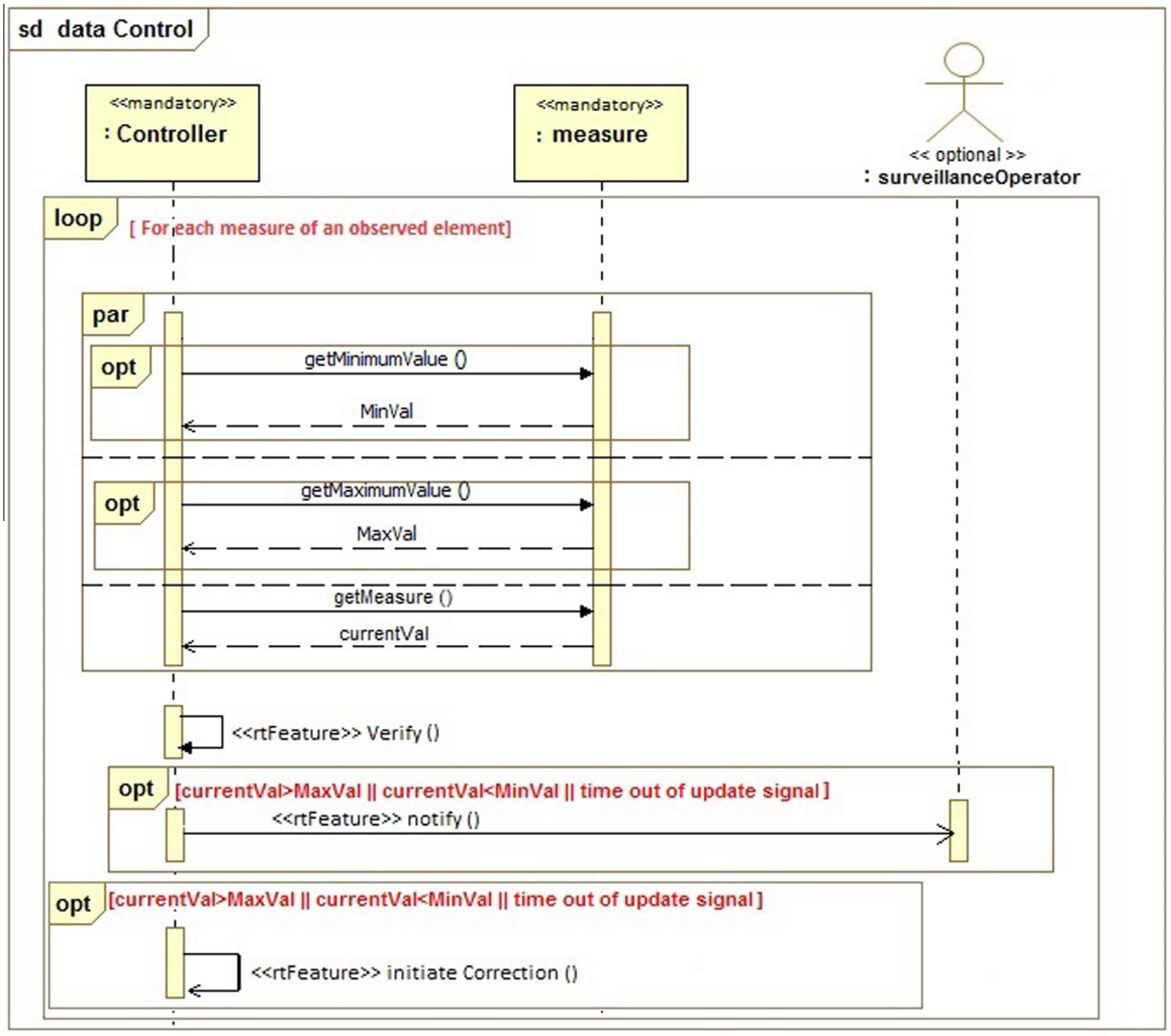

Figure A8 Sequence diagram of RT controller pattern.

A.4. RT controller pattern representation

Name: controller pattern.

Context: This pattern is applicable during the data control phase of RT applications once RT data are acquired and stored in the database.

Intention: The pattern aims to model RT data control and to express temporal constraints.

Solution:

Static view: The class diagram of RT controller pattern is presented in Fig. A4.

Participants:

- Controller: A controller has to monitor physical elements to respond to conditions that might violate safety. It takes periodically the value captured for each observed element as well as the minimum value and the maximum value that define the interval for which the controller does not detect an anomaly. If a captured value does not verify the boundary constraint, then the controller initiates some corrective actions, such as a reboot, a reset and a shut-down, or notifies an operator.
- On the other hand, the controller receives periodically a signal to be notified about the data that must be updated for each observed element. In this case, the controller is waiting for a signal. If this signal does not arrive on time, then the controller performs appropriate recovery actions.
- ObservedElement: This class represents the description of a physical element that is supervised by the controller. It can be an aircraft, a car, a road segment, and so on. One or more measure types (i.e., Temperature, Pressure, etc) of each observed element could determinate its evolution.
- Operator: The operators supervise the alarm signals sent by the controller. They provide decisions to validate reported incidents in case the controller only reports errors and does not have the responsibility of initiating corrective actions; or in case the confirmation of an operator is needed to achieve the correction.

Dynamic view: The sequence diagram of controller pattern is presented in Fig. A8.

References

Aggarwal, K.K., Singh, Y., Kaur, A., Malhotra, R., 2005. Software reuse metrics for object-oriented systems. In: Proceedings of the third ACIS International Conference on Software Engineering Research, Management and Applications (SERA'05). IEEE computer society, pp. 48–54.

Alnusair, A., Zhao, T., 2010. Using ontology reasoning for reverse engineering design patterns. In: Models Software Eng.. LNCS 6002, 344–358.

Baldinger, J.L., Boudy, J., Dorizzi, B., Levrey, J.P., Andreao, R., Perpère, C., Delavault, F., Rocaries, F., Dietrich, C., Lacombe, A., 2004. Tele-surveillance system for patient at home: the MEDIVILLE system. Book Chapter in Computers Helping People with Special Needs. LNCS, 3118. Springer, Berlin.

Ben-Abdallah, H., Bouassida, N., Gargouri, F., Ben Hamadou, A., 2004. A UML based framework design method. In: J. Object Technol. 3 (8), 97–120.

Bouassida, N., Ben-Abdallah, H., Issaoui, I., 2013. Evaluation of an automated multi-phase approach for patterns discovery. Int. J. Software Eng. Knowl. Eng. 23 (10), 1367–1398.

Boukhelfa, K., Belala, F., 2015. Towards a formalization of RT patterns based designs. In: Proc. Comput. Sci. Appl.. 624, 635.

Caeiro, M., Llamas, M., Anido, L., 2004. E-learning patterns: an approach to facilitate the design of E-learning materials. In: 7th IberoAmerican Congress on Computers in Education.

Chandra, E., Edith, Linda P., 2010. Class break point determination using CK metrics thresholds. Global J. Comput. Sci. Technol. 10 (14), 73–77.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Software Eng. 20 (6), 476–493.

COMPASS Website, Available from: <http://www.mto.gov.on.ca/english/traveller/compass/main.htm>, Last modified June 17, 2010.

Douglass, B., 2004. Real Time UML, Third Edition: Advances in The UML for Real-Time Systems. Pearson Education Inc., 0-321-16076-2.

Frakes, W.B., Anguswamy, R., Sarpotdar, S., 2009. Reuse ratio metrics RL and RF. In: 11th International Conference on Software Reuse, Falls Church, VA, USA.

Gamma, E., Helm, R., Johnson, R.E., Vlissides, J., 1994. Design patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Edition.

Gill, S.N., Sikka, S., 2011. Inheritance hierarchy based reuse & reusability metrics in oosd. Int. J. Comput. Sci. Eng. (IJCSE) 3 (6), 2300–2309.

Graeme, S., 2000. The object-Z specification language. Advances in Formal Methods Series. Kluwer Academic Publishers, ISBN 0-7923-8684-1.

Hammouda, I., Ruokonen, A., Siikarla, M., Santos, A.L., Koskimies, K., Systä, T., 2009. Design profiles: toward unified tool support for design patterns and UML profiles. J. Software Pract. Exp. 39 (4), 331–354.

Kajsa, P., 2013. Semantics and model driven design patterns instantiation. Inf. Sci. Technol. Bull. ACM Slovakia 5 (1), 44–52.

Kim, D.K., 2007. The role-based meta-modeling language for specifying design patterns. In: Design Pattern Formalization Techniques. Idea Group Inc., pp. 183–205.

Kim, S.K., Carrington, D., 2006. A tool for a formal pattern modeling language. In: 8th International Conference on Formal Engineering Methods (ICFEM 2006), LNCS 4260, Macao, China 1–3 November 2006, pp. 568–587.

Kim, D.K., France, R., Ghosh, S., 2004. A UML-based language for specifying domain specific patterns. J. Visual Lang. Comput. 15, 265–289.

Koskinen, J., Kettunen, M., Systä, T., 2010. Behavioral profiles a way tomodel and validate program behavior. J. Software Pract. Exp. 40 (8), 701–733.

Lanusse, A., Gérard, S., Terrier, F., 1999. Real-time modeling with UML: the ACCORD approach. In: Bézivin, J., Muller, P.-A. (Eds.), The Unified Modeling Language, UML'98- Beyond the Notation, First International Workshop, LNCS 1618. Springer, pp. 319–335.

Liamwiset, C., Vatanawood, W., 2013. Detection of design patterns in software design model using graph. In: Proceedings, The 2nd International Conference on Information Technology and Management Innovation (ICITMI 2013), July 23–24, 2013.

OMG, A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, OMG document number: ptc/2008-06-09, 2008.

Port, D., 1998. Derivation of Domain Specific Design Patterns, USC-CSE Annual Research Review and Technology Week Presentations and Binder Materials.

Raminhos, R., Pantoquilho, M., Araújo, J., Moreira, A., 2006. A systematic analysis patterns specification. In: Proceedings of ICEIS, pp. 453–456.

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

Rekhis, S., Bouassida, N., Duvallet, C., Bouaziz, R., Sadeg, B., 2010. A process to derive domain-specific patterns: application to the real time domain. In: Proceedings of 14th International Conference on Advances in Databases and Information Systems (ADBIS'2010). LNCS 6295, September 2010, pp. 475–489.

Rekhis, S., Bouassida, N., Bouaziz, R., Duvallet, C., Sadeg, B., 2013. Modeling real-time design patterns with UML-RTDP profile. Book Entitled: Domain Engineering: Product Lines, Languages, and Conceptual Models. Springer (2013 edition), May 31, 2013, pp. 59–81. ISBN 978-3-642-36653-6.

Robles, K., Fraga, A., Morato, J., Liorens, J., 2012. Towards an ontology-based retrieval of UML class diagrams. Inf. Software Technol. 54, 72–86.

Satvinij, N., Vatanawood, W., 2011. Detection of behavioral design patterns. In: Proceedings, The 15th International Annual Symposium on Computational Science and Engineering (ANSCSE15), March 30–April 1.

Steinberg, D., Budinsky, F., Paternostro, M., Merks, E., 2008. EMF: Eclipse Modeling Framework, second ed. Addison-Wesley Professional.

Subedha, V., Sridhar, S., 2012. Design of a conceptual reference framework for reusable software components based on context level. Int. J. Comput. Sci. Issues (IJCSI) 9 (3), 26–31.

Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T., 2006. Design pattern detection using similarity scoring. IEEE Trans. Software Eng. 32 (11), 896–909.

Please cite this article in press as: Rekhis, S. et al., A new method for constructing and reusing domain specific design patterns: Application to RT domain. Journal of King Saud University – Computer and Information Sciences (2016), http://dx.doi.org/10.1016/j.jksuci.2016.04.004